DATABASE MANAGEMENT ALGORITHMS FOR ADVANCED BMD APPLICATIONS

Principal Investigator: Wesley W. Chu
Researchers: M. T. Lan, K. K. Leung, J. M. An

April 1985
CSD-850018

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

Database Management Algorithms for Advanced BMD Applications

5. TYPE OF REPORT & PERIOD COVERED

Final Report for the period: Feb. 1, 1984 - Jan. 31, 1985

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

W. W. Chu, M. T. Lan, K. K. Leung, J. M. An

8. CONTRACT OR GRANT NUMBER(s)

DASG 60-83-C-0019

9. PERFORMING ORGANIZATION NAME AND ADDRESS

University of California, Los Angeles
Computer Science Department
405 Hilgard Ave., Los Angeles, CA. 90024

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS

Ballistic Missile Defense Advanced Technology Center (BMDATC)
P. O. Box 1500, Huntsville, AL 35807

12. REPORT DATE

13. NUMBER OF PAGES

114

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)

Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution limited to U. S. Government Agencies only, Test and Evaluation. Other requests for this document must be referred to BMD Program Manager, ATTN: BMDSC-AU, P. O. Box 1500, Huntsville, AL 35807

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Concurrency, Database Management Algorithm, Distributed Systems, Intermodule Communication (IMC), Locking, Task Assignment, Interprocessor Communication (IPC), Fault Tolerant Locking, Resilient Commit Protocol, Task Response Model, Task Control-Flow Graph, Precedence Relation, Module Scheduling.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DATABASE MANAGEMENT ALGORITHMS FOR ADVANCED BMD APPLICATIONS

FINAL REPORT FOR THE PERIOD

FROM: February 1, 1984

TO: January 31, 1985

Contract No. DASG 60-83-C-0019

Prepared For:

Ballistic Missile Defense Advanced Technology Center

Huntsville, Alabama 35807

April 30, 1985

University of California, Los Angeles

Wesley W. Chu, Principal Investigator

Researchers: M. T. Lan, K. K. Leung, and J. M. An

CONTENTS

CHAPTER I

INTRODUCTION AND SUMMARY

I. INTRODUCTION AND SUMMARY

During the past year, we have been emphasizing our studies on distributed processing systems for the following three areas: task response time model, algorithm for task assignments for distributed real-time systems, and Fault Tolerant Locking protocol. We shall briefly summarize our findings in the following.

We have developed an analytical model for estimating the average response time for loosely coupled distributed systems. The model provides a good estimate of task response time as compared with simulations. For example, we are able to estimate the port-to-port time for the DPAD system and obtain results comparable to that generated by the DPAD simulator. Our analytical model not only provides us more insight, but also is far less time-consuming. The model allows us to study the performance of various scheduling algorithms, data base management algorithms, and module and file assignment. Currently, we are extending our model for the tightly coupled distributed systems.

Task assignment is one of the important problems in distributed systems. The three key parameters are: accumulative execution time (AET), interprocessor communication (IMC), and module precedence relation. During the past year, we have emphasized our investigation on the module precedence relationship area. We have performed simulation experiments as well as analytical studies and discovered that the module execution time is a key parameter in determining whether a module pair should be allocated on the same processor. In general, if a module with small execution time

precedes a module with large execution time, then they should be allocated on the same processor. Otherwise, they should be allocated on different processors. We have incorporated these rules in our allocation algorithm developed in the previous year that considers AET and IMC. We noticed that precedence relationship consideration could provide substantial response time improvements for some application tasks.

Fault tolerance is an important design issue for distributed systems. Fault Tolerant Locking (FTL) provides a resilient locking protocol for performing updates in primary and shadow file copies in a tightly coupled distributed systems. Techniques are developed to assure data consistency and recovery in cast of processor, memory, or communication path failures. FTL has been implemented on the SDC BMD testbed at Huntsville. Experimental results characterize the FTL behavior, operating overhead, and performance. The testbed results conclude that FTL is feasible (in terms of response time) for BMD applications. Currently, we are developing an analytical model for FTL to study the interrelationship of such parameters as the memory access conflict, number of lock retries (time-out), and response time. Such study should provide us with insight about selecting the parameter values for achieving optimal performance.

CHAPTER II

TASK RESPONSE TIME MODEL & ITS APPLICATIONS FOR REAL-TIME DISTRIBUTED PROCESSING SYSTEMS

TASK RESPONSE TIME MODEL & ITS APPLICATIONS FOR REAL-TIME DISTRIBUTED PROCESSING SYSTEMS*

Wesley W. Chu and Kin K. Leung

Computer Science Department
University of California, Los Angeles
California 90024

Abstract

Response time is an important system performance measure for real-time distributed processing systems. This paper introduces an analytic model to estimate the task response time for loosely coupled distributed systems. The model considers such factors as the precedence relationships among software modules, interprocessor communication, interconnection network delay, module scheduling policy, and assignment of modules to computers. Simulation experiments are used to validate the assumptions of the analytic model. Applications of the model to the study of such design issues for distributed systems as module assignment, precedence relationships, module scheduling policies, and database management algorithms are discussed.

1. INTRODUCTION

With the advent of low-cost VLSI and communication technologies, distributed processing (DP) has become an economically and technologically attractive computer architecture. The DP system considered in this paper consists of multiple computers, each with its own memory and peripherals, connected by an interconnection network.

In a DP system, an application task 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 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 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)[1]. 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 re-

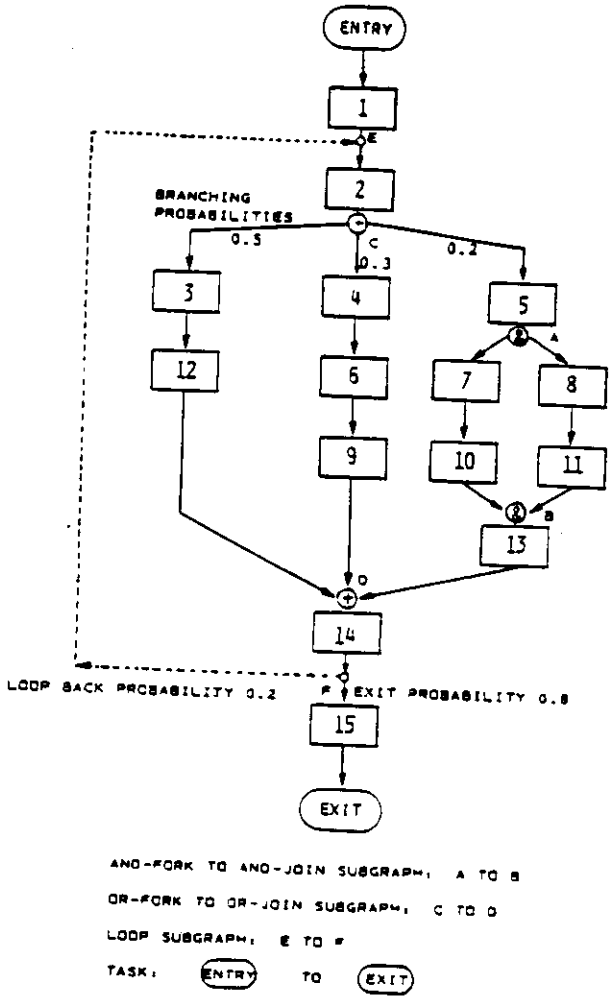


Figure 1a. A Sample Task Control-Flow Graph

*This work was supported by the Ballistic Missile Defense Advanced Technological Center under Contract DASG60-83-C-0019 and the U.C. MICRO Grant P-5607-N-84.

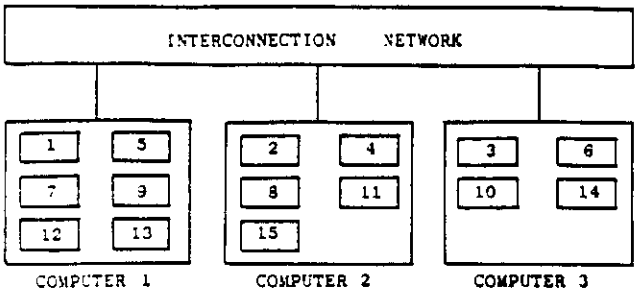


Figure 1b. Assignment of Modules to Computers

quires 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, 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 (e.g., locking, timestamp, exclusive-writer protocol). Therefore, planning a DP system is complicated by many such complex and interdependent design issues as module and file assignment[2], module scheduling policy, database management algorithm, etc. Presently, there is no systematic methodology for designing DP systems. Existing system designs use ad hoc methods which result in a trial-and-error approach. Further, since DP systems 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 DP systems. The model can be used as a unified approach for studying various DP 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. A TASK RESPONSE TIME MODEL

Queueing networks [1-4] are commonly used to model DP 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 DP 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 present 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.

2.1 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* for module executions.

Consider a computer that has n 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 $i=1,2,...,n$. For a bulk invocation that invokes a set M of distinct modules (referred to as *module bulk*), the corresponding service requirement is $U_i^*(s) = \prod_{m \in M} X_m^*(s)$, where $X_m^*(s)$ is the L.T. of the service time of module m.

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 $\lambda = \sum_{i=1}^{n} \lambda_i$, and the L.T. of service time for each invocation arrival is $U^*(s) = \sum_{i=1}^{n} \frac{\lambda_i}{\lambda} U_i^*(s)$. 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

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

and

$$\overline{w^2} = 2(\bar{w})^2 + \frac{\sum_{i=1}^{n} \lambda_i \overline{u_i^3}}{3(1-\rho)} \qquad (2)$$

where:
$\overline{u_i^n}$ = n^{th} moment of service time for i^{th} module invocation,
ρ = server utilization $= \sum_{i=1}^{n} \lambda_i \overline{u_i}$,
$\bar{w}$ = average module bulk waiting time.
From Eqs.(1) and (2), we obtained the variance of module bulk waiting time as

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

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 $j_1, j_2, ..., j_k, ..., j_m$. The response time (a random variable) for module j_k is

$$t(j_k) = w + \sum_{i=1}^{k} x(j_i) \qquad (4)$$

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.(4). We have

$$T(j_k) = \bar{w} + \sum_{i=1}^{k} \bar{x}(j_i) \qquad (5)$$

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.(4). Hence

$$\sigma_t^2(j_k) = \sigma_w^2 + \sum_{i=1}^{k} \sigma_x^2(j_i) \qquad (6)$$

where $\sigma_x^2(j_k)$ is the variance of execution time for module j_k and σ_w^2 is given in Eq.(3). For the case of a single module invocation, there will be only a single module in the execution sequence.

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). 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** 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.

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

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

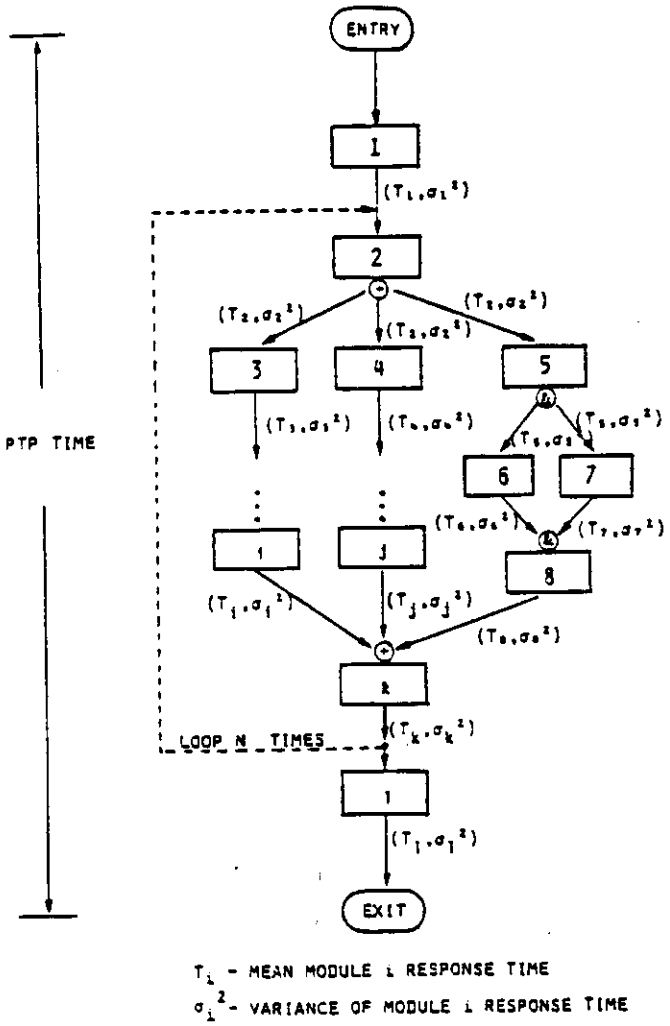


Figure 2. Weighted Control-Flow Graph for Response Time Estimations

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 3 to 6). A task control-flow graph may contain a combination of these basic logical relationships 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.

2.2.1 Sequential Thread Subgraph

Sequential thread subgraph (Figure 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.

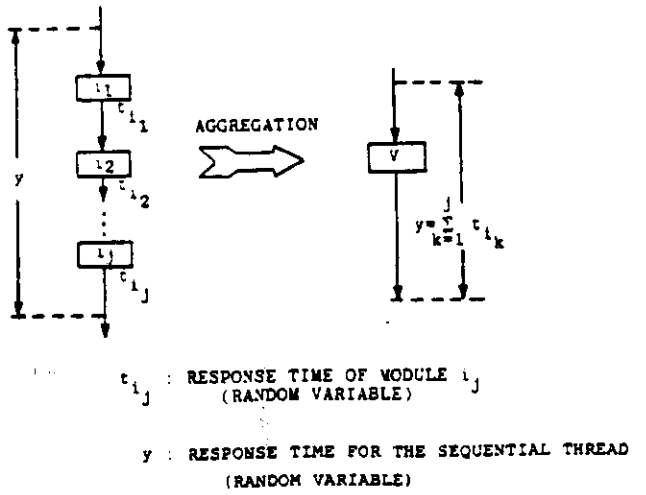


Figure 3. Sequential Thread

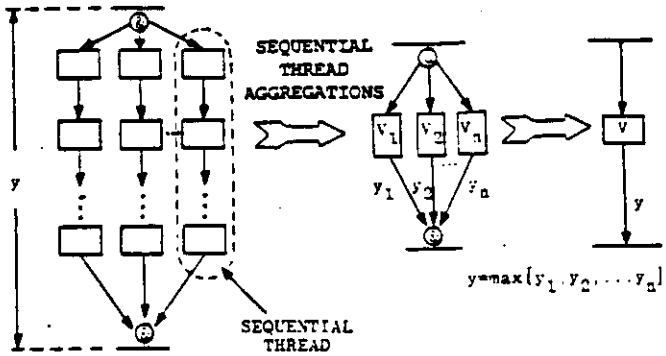


Figure 4. And-Fork To And-Join Subgraph

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 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 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 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 approxi-

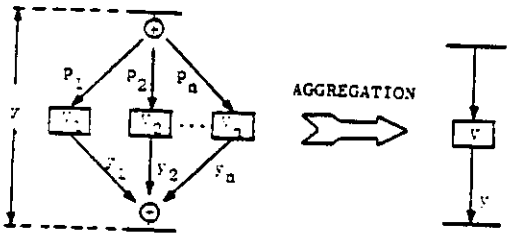
mated by either Erlangian or hyper-exponential distribution functions[6]. Assuming that y_i's are independent, the joint PDF for y_i's can be computed. Thus the mean and variance of the response time for the subgraph can be obtained.

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 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 weighted response times of all these threads.

2.2.4 Loop Subgraph

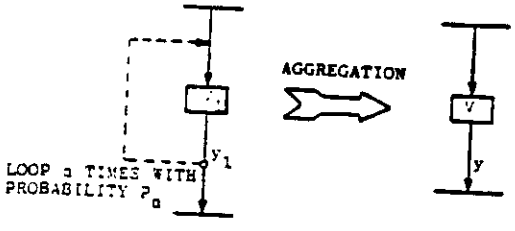
Loops are often contained in a task control-flow graph for repeatedly processing a set of modules for a task invocation. A loop may contain any of the aforementioned subgraphs. After aggregating these subgraphs, a loop may be represented by a single cyclic node graph as shown in Figure 6. The arc weight 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 required to execute a single loop.



P_i BRANCHING PROBABILITY (TO ENABLE MODULE V_i)

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

Figure 5. Or-Fork To Or-Join Subgraph



y_1 RESPONSE TIME FOR A SINGLE LOOP

y RESPONSE TIME FOR THE LOOP SUBGRAPH

Figure 6. Loop Subgraph

2.3 Module Response Times With Dependent Module Invocations

In Section 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.

2.3.1 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 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 8). Figure 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 8f) and the control branches to a module residing on another computer, then the execution terminates at the or-fork.

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)

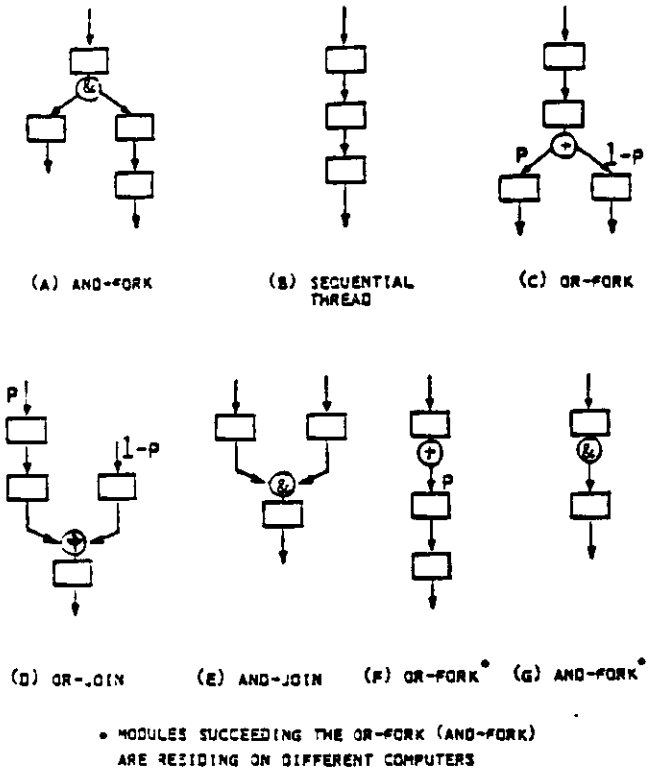


Figure 7. Examples of Control-Flow Subgraphs that Allocated on a Computer

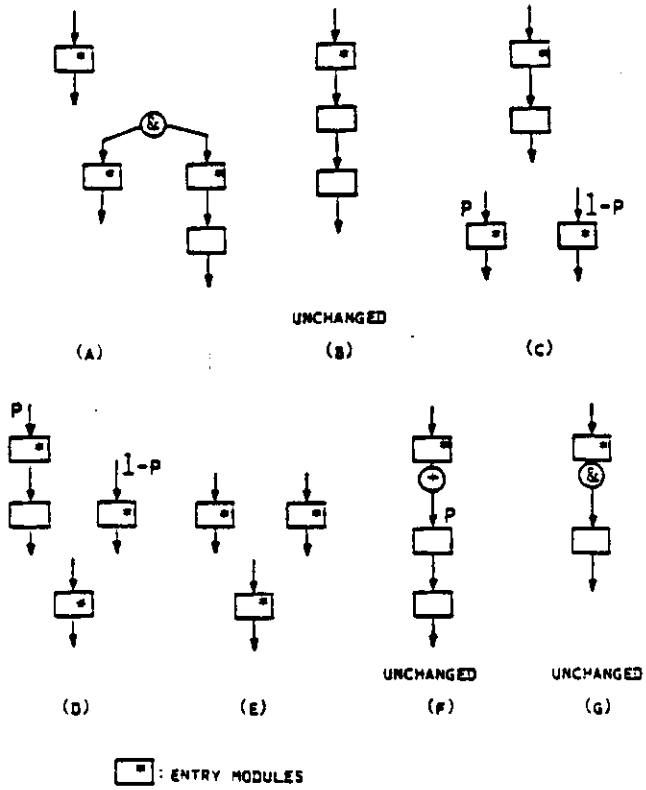


Figure 8. Partitioned Control-Flow Subgraphs of Figure 7

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 mean module response times* 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.(4)).

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

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. 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.

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 [7], and a simulator of the Distributed Processing Architecture Design (DPAD) System [8] 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. Further, each simulation experiment is repeated five times with different initial random numbers to reduce the statistical fluctuation. In this paper, let us consider the sample task control-flow graph in Figure 1a with its parameters given in Table 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 9 to 12 present the mean response times for these subgraphs and the whole task for the module assignments (Table 2). We aggregate the response 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. 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!

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

Table 1. Module Execution Times for the Sample Control-Flow Graph (Figure 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

Table 2. Module Assignments & Computer Processing Load for the Sample Control-Flow Graph

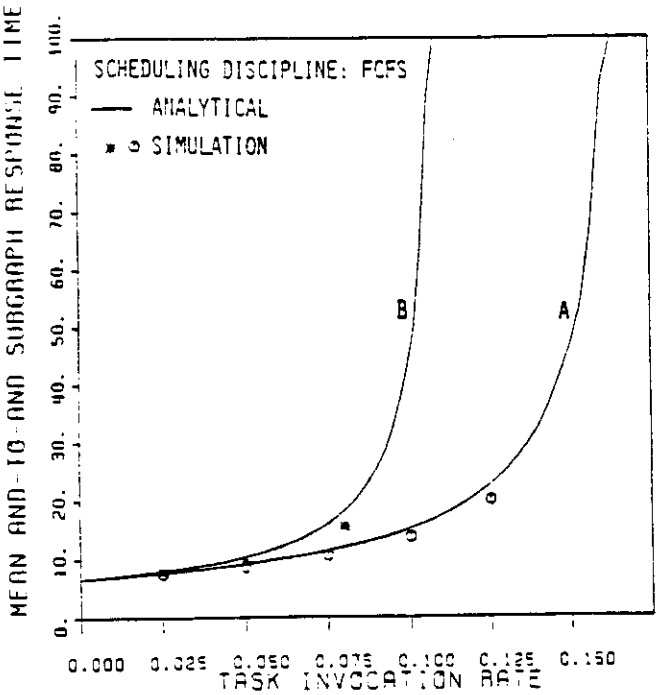


Figure 9. Mean Response Time for the And-Fork to And-Join Subgraph of the Sample Control-Flow Graph (Figure 1a)

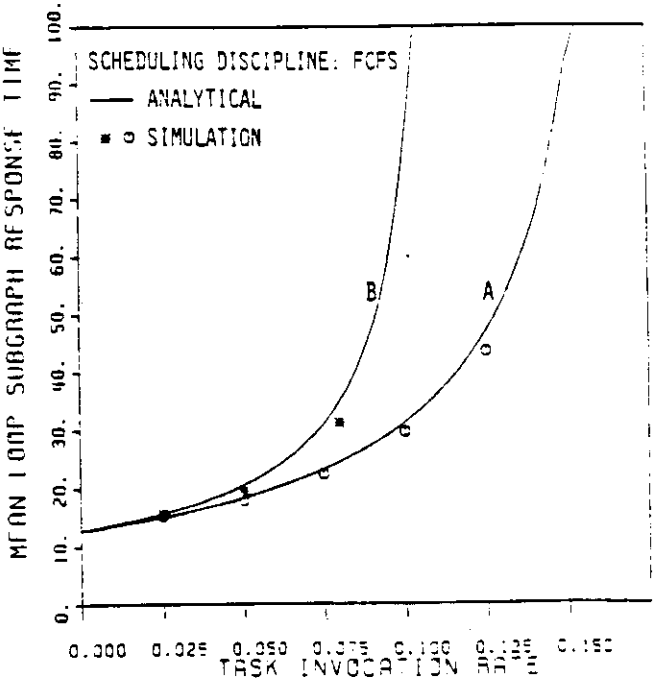


Figure 11. Mean Response Time for the Loop Subgraph of the Sample Control-Flow Graph

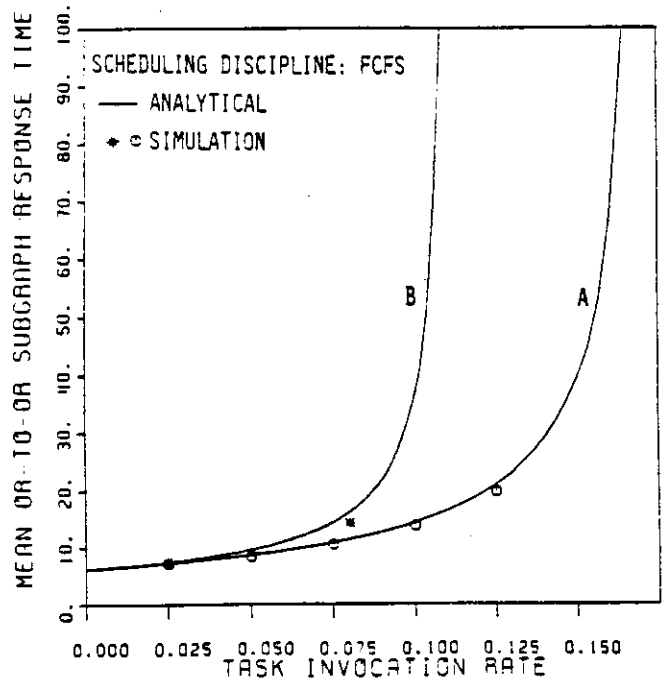


Figure 10. Mean Response Time for the Or-Fork to Or-Join Subgraph of the Sample Control-Flow Graph

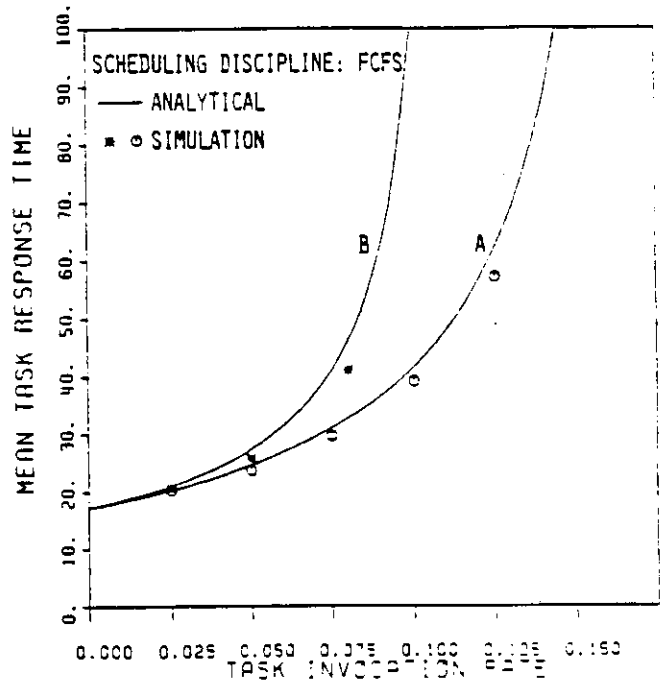


Figure 12. Mean Task Response Time for the Sample Control-Flow Graph

We now describe the model validation via the DPAD simulator*. The DPAD system is a real time DP system which processes radar return signals for space defense applications. The DPAD simulator provides detailed operating system operations for module scheduling and IPC message exchanges among computers. Further, non-Poisson task invocation arrivals are used. Its task control-flow graph is shown in Figure 13. The module assignment and module priorities are shown in Table 3. The processing thread for precision track function is indicated by shaded modules in Figure 13. For input data to the analytical model, we collected the IMC data, module execution times (Table 4) and invocation rates in every 100-msec time interval from the DPAD simulator. Since the DPAD System uses a priority module scheduling policy rather than FCFS, queueing formulas were derived to compute the module response times for this scheduling discipline. 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 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.

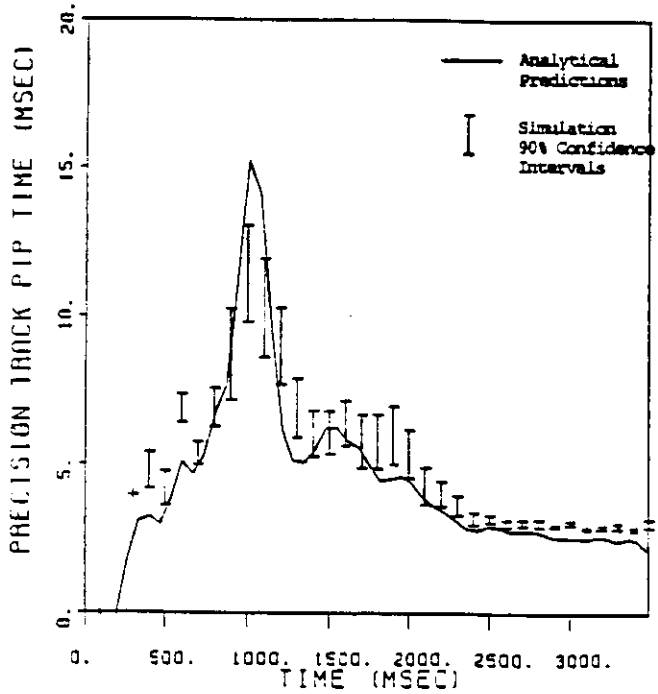


Figure 14. Comparing Analytical Predictions with the DPAD Simulation

*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.

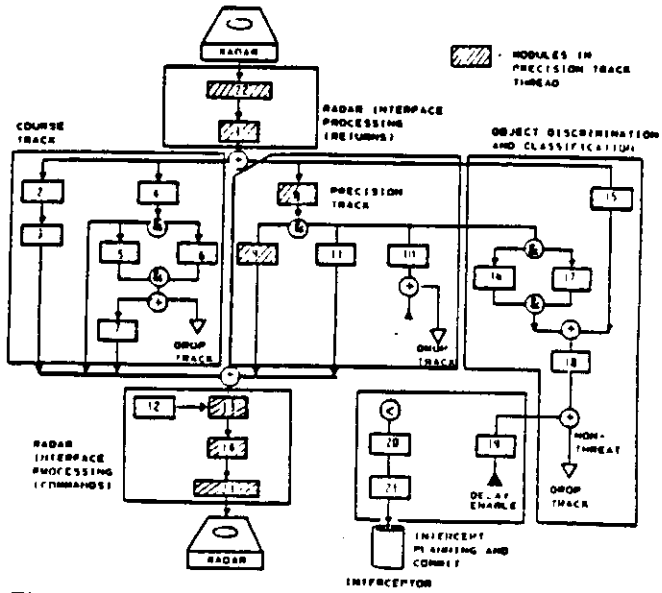


Figure 13. The Task Control-Flow Graph for the DPAD System

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 3. A Module Assignment for the DPAD System

MODULES	MEAN EXECUTION TIME (ms)	COEF. OF VARIATION
1	0.157043	0.300161
2	0.313522	0.323414
3	0.297477	0.300066
4	0.422061	0.385611
5	0.379137	0.300042
6	0.321826	0.252115
7	0.325322	0.1789-2
8	1.128163	0.307536
9	0.659989	0.300000
10	0.535785	0.000226
11	0.300000	0.000000
12	0.300000	0.300000
13	0.334381	0.302868
14	0.331086	0.301647
15	0.300000	0.000000
16	0.717703	0.300090
17	1.317110	0.300000
18	0.656880	0.302384
19	3.339637	0.300003
20	6.695341	0.000012
21	0.730000	0.300000
22	0.380173	0.317631
23	0.162269	0.305265

Table 4. Module Execution Times (Including Output IPC) for the DPAD System Averaged over 35 100-msec Time Intervals

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.

4.1 Module Assignment and Precedence Relationships

The assignment of modules to computers is an important problem in DP 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 [9-12]. 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, a module scheduling policy and a data management algorithm, the task response time can be estimated from the proposed model. The proposed model can be used to investigate the performance in terms of task response time of module assignment strategies with various module precedence relationships at different operating environments. This study provides insight into the interrelationship among precedence relationships, module assignment and task response time.

4.2 Module Scheduling Disciplines

For a given module assignment, the scheduling policy for module execution affects the response time. For example, to reduce operating system overhead, invocations for the same module may be delayed until a predefined number of module invocations have arrived and are executed in a batch manner. To avoid excessive waiting, a time-out mechanism may be used in conjunction with this scheduling policy. The proposed model can be employed to study the relationships among time-out constant, batch size, operating system overhead and the PTP time for this scheduling algorithm. Further, it can investigate the optimal assignment of module priorities for head-of-line priority scheduling policy and the relationships among scheduling policy, module assignment and the PTP time.

4.3 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. Commonly used techniques for consistency controls are locking, timestamp, and exclusive-writer protocol [13,14]. In the proposed model, the effect of IPC can be included 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. Thus the model can be used to study the overhead in terms of PTP 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.

5. CONCLUSIONS

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.

Acknowledgement

The authors wish to thank Min-Tsung Lan of UCLA and Joseph Hellerstein of IBM Thomas J. Watson Research Center (formerly of UCLA) for their effort in collecting various statistics for our DPAD model validation.

REFERENCES

[1] 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, Aug. 1984, pp.691-699.

[2] W.W. Chu, L. Holloway, M.T. Lan and Kemal Efe, "Task Allocation in Distributed Processing," *IEEE Computer*, Nov. 1980, pp. 57-69.

[3] 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, April 1975, pp. 248-260.

[4] P. Heidelberger, and K.S. Trivedi, "Queueing Network Models for Parallel Processing with Asynchronous Tasks," *IEEE Trans. on Computers*, Nov. 1982, pp. 1099-1109.

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

[6] C.H. Sauer and K.M. Chandy, *Computer Systems Performance Modeling*, Prentice-Hall, New Jersey (1981).

[7] R. Berry, K.M. Chandy, J. Misra, and D. Neuse, *PAWS 2.0 Performance Analyst's Workbench System: User Manual*, Information Research Associates, Austin, Texas, Dec. 1982.

[8] 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," *Special Report* 35010-79-A005, TRW Defense and Space Systems Group, May 15, 1980.

[9] H.S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. on Software Eng.*, Vol.SE-3, No.1, Jan. 1977, pp.85-93.

[10] G.S. Rao, H.S. Stone and T.C. Hu, "Assignment of Tasks in a Distributed Processor System with Limited Memory," *IEEE Trans. on Computers*, Vol.C-28, No.4, April 1979, pp.291-299.

[11] P.Y.R. Ma, E.Y.S. Lee and M. Tsuchiya, "A Task Allocation Model for Distributed Computing System," *IEEE Trans. on Computers*, Vol.C-31, No.1, Jan. 1982, pp.41-47.

[12] T.C.K. Chou and J.A. Abraham, "Load Balancing in Distributed Systems," *IEEE Trans. on Software Eng.*, Vol.SE-8, No.8, July 1982, pp.401-412.

[13] P.A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *Computing Surveys* 13, 2, June 1981, pp. 185-221.

[14] W.W. Chu and J. Hellerstein, "The Exclusive-Writer Approach to Updating Replicated Files in Distributed Processing Systems," to appear in *IEEE Trans. on Computer*, 1985.

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

Appendix

Computation of 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 8a). Assume that the i^{th} subgraph consists of b_i sequential threads, and the j^{th} sequential thread comprises $d_i(j)$ modules for $j=1,2,\ldots,b_i$. 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 $i=1,2,\ldots,v$, $j=1,2,\ldots,b_i$, and $1 \le k \le t \le d_i(j)$. 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,

$$P_i(j;k,k) = 1 \qquad \text{(A.1)}$$

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

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

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, the mean waiting time for $M_i(1;1)$ (i.e., an entry module) for all $i=1,2,\ldots,v$ 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

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

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_i = mean residual module execution time
$= \sum_{r=1}^{v} \sum_{s=1}^{b_r} \sum_{t=1}^{d_r(s)} \frac{1}{2} \lambda_r(s;t)\, \bar{x}^2_r(s;t)$,

$\bar{x}^2_r(s;t)$ = 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[18] (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.(A.3), we have

$$W_r(1;1) = W_r + \sum_{r=1}^{v} \sum_{s=1}^{b_r} \sum_{t=1}^{d_r(s)} \rho_r(s;t)\, W_r(s;t) \qquad (A.4)$$

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 $M_r(j;1)$ before module $M_r(s;1)$ for $j<s$. Thus the mean waiting time $W_r(j;1)$ is

$$W_r(j;1) = W_r(1;1) + \sum_{s=1}^{j-1} \bar{x}_r(s;1) \qquad (A.5)$$

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

Mean Waiting Time for Non-Entry Modules

Let us consider the waiting time for the non-entry modules $M_r(j;k)$ (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, $W1_r(j;k)$, 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 $M_r(j;1)$. The second component, $W2_r(j;k)$, is due to the waiting for the module executions invoked by the bulk module invocations (i.e., $M_r(s;1)$, $s = 1, \cdots b_r$ and $s \neq j$). The last component, $W3_r(j;k)$, 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,

$$W1_r(j;k) = \sum_{r=1}^{v} \sum_{s=1}^{b_r} \sum_{t=1}^{d_r(s)-k+1} \bar{n}_r(s;t)\, P_r(s;t,t+k-1)\, \bar{x}_r(s;t+k-1)$$

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

The first term of Eq.(A.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 $M_r(j;1)$. 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 $M_r(j;1)$ arrives.

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

$$\lambda_r(s;t+k-1) = \lambda_r(s;t)\, P_r(s;t,t+k-1) \qquad (A.7)$$

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

$$W1_r(j;k) = \sum_{r=1}^{v} \sum_{s=1}^{b_r} \sum_{t=1}^{d_r(s)-k+1} \rho_r(s;t+k-1)\{W_r(s;t) + \bar{x}_r(s;t)\} \qquad (A.8)$$

The second component of $W_r(j;k)$ is

$$W2_r(j;k) = \sum_{s=1}^{j-1} P_r(s;1,k)\, \bar{x}_r(s;k) + \sum_{s=j+1}^{b_r} P_r(s;1,k-1)\, \bar{x}_r(s;k-1) \qquad (A.9)$$

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

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

$$W3_r(j;k) = [W_r(j;k-1) + \bar{x}_r(j;k-1)] \sum_{r=1}^{v} \sum_{s=1}^{b_r} \lambda_r(s;1)\, \bar{x}_r(s;1)$$

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

$$+ \quad \vdots$$

$$+ [W_r(j;1) + \bar{x}_r(j;1)] \sum_{r=1}^{v} \sum_{\substack{s=1 \\ d_r(s) \geq k-1}}^{b_r} \lambda_r(s;1)\, P_r(s;1,k-1)\, \bar{x}_r(s;k-1) \qquad (A.10)$$

Due to the first-come-first-serve scheduling, those module invocations from other computers arriving during the response time (waiting plus execution) of $M_r(j;k-1)$ contribute a part of the waiting time for $M_r(j;k)$. Thus the first term of Eq.(A.10) is the total time for executing those module invocations arriving during the response time of $M_r(j;k-1)$. Likewise, the remaining terms of Eq.(A.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 $\lambda_r(s;t) = \lambda_r(s;1)\, P_r(s;1,t)$ and $\rho_r(s;t) = \lambda_r(s;t)\, \bar{x}_r(s;t)$ into Eq.(A.10), and simplifying, we have

$$W3_r(j;k) = \sum_{t=1}^{k-1} \left\{ [W_r(j;t) + \bar{x}_r(j;t)] \sum_{r=1}^{v} \sum_{\substack{s=1 \\ d_r(s) \geq k-t}}^{b_r} \rho_r(s;k-t) \right\} \qquad (A.11)$$

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

$$W_r(j;k) = W1_r(j;k) + W2_r(j;k) + W3_r(j;k) \qquad (A.12)$$

for $i=1,2,\ldots v$, $j=1,2,\ldots b_r$, and $d_r(j) \geq k \geq 2$.

The mean module waiting time for each module is expressed in Eq.(A.4), (A.5) or (A.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.

CHAPTER III

PRECEDENCE RELATIONS AND TASK ALLOCATION FOR DISTRIBUTED REAL-TIME SYSTEMS

PRECEDENCE RELATIONS AND TASK ALLOCATION FOR DISTRIBUTED REAL-TIME SYSTEMS

1. INTRODUCTION

Although computer speed has increased several orders of magnitude during the past decades, the demand for computing capacity increases more rapidly. Many real-time applications require speed capability not achievable by a single processor. One approach to this problem is via distributed data processing (DDP) that concurrently processes an application on multiple processors. If properly designed and planned, DDP provides a more economical and reliable approach than the centralized processing with a single high-speed processor.

Task partitioning and task allocation are two major steps in the design of DDP systems [CHU80]. If these steps are not done properly, an increase in the number of processors in a system may actually result in a decrease of the total throughput. Assuming the software for an application (a task) has been partitioned into a set of program *modules* (or, subroutines), we study how to properly allocate (assign) these modules to the set of processors in the DDP system.

We shall first present the three important input parameters for task allocation: intermodule communication (IMC), accumulative execution time (AET) of each module, and precedence relations (PR) among program modules. Next, we propose an objective function for task allocation that is based on IMC and AET. A task-allocation algorithm based on that objective function is then proposed. The PR states that a program module should not be enabled before all its predecessor(s) finish execution. Simulation and analytical results are shown and they reveal that the *program-size ratio* between a module and its predecessor module plays an important role in task allocation, in terms of task response time. An improved

task-allocation algorithm, based on PR, IMC, and AET, is then proposed. Examples are given to illustrate the performance improvement when PR is considered in the task allocation.

2. A NEW OBJECTIVE FUNCTION FOR TASK ALLOCATION

2.1 Key Parameters

The three parameters that play important roles in module assignment are intermodule communication (IMC), accumulative execution time (AET) of each module, and precedence relations (PR) among program modules. The AET for a module M_j during a time interval (t_k, t_{k+1}) is the total execution time incurred for this module during that time interval, i.e.,

$$T_j(t_k, t_{k+1}) = N_j(t_k, t_{k+1}) y_j(t_k, t_{k+1})$$

where $N_j(t_k, t_{k+1})$ = number of times module M_j executes during (t_k, t_{k+1}), and $y_j(t_k, t_{k+1})$ = average execution time of M_j during (t_k, t_{k+1}). Both the y_j and AET can be expressed in machine-language instructions (MLI) executed. Although the execution time of a machine-language instruction varies from instruction to instruction, we can use the *mean* instruction execution time given the mix ratios for various different instructions. Our study reveals that both the number of module executions and the AET are almost independent of module assignments when the load offered to the system is *fixed*. Fig. 1 shows that the AETs produced by five different assignments for a module in a space-defense distributed system [1] are almost identical.

IMC is the communication between program modules through shared files. When a module on a processor writes to or reads from a shared file on *another* processor, such IMC becomes IPC (interprocessor communication) and requires processing overhead. The importance of IPC minimization has been recognized by many researchers [CHU78, GENT78,

[1] This system, the Distributed Processing Architecture Design (DPAD) system, will be used as an example in Section 3. A portion of its control-and-data-flow graph is given in Fig. 2.

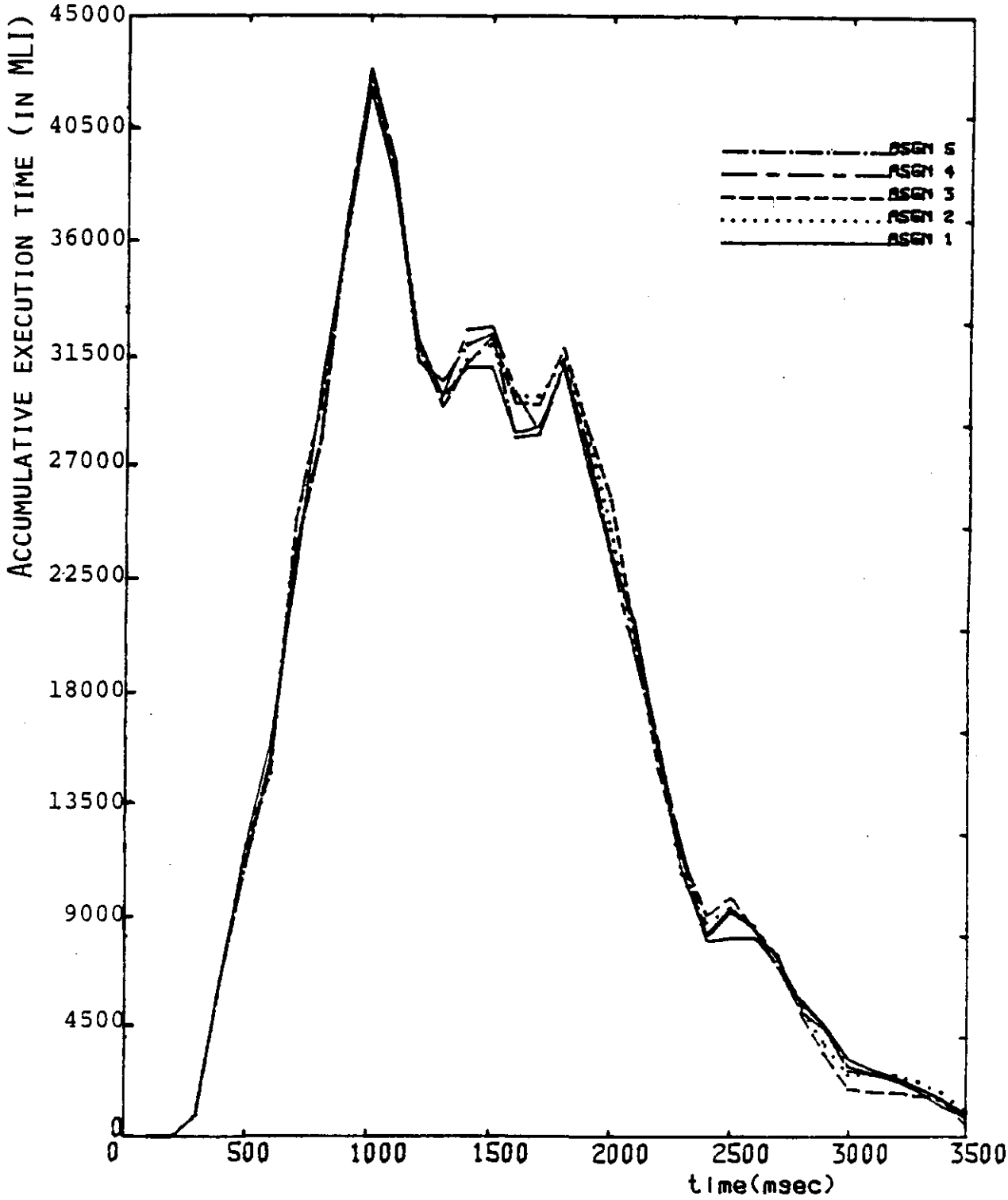


FIG. 1. ACCUMULATIVE EXECUTION TIME FOR MODULE M_8, $T_8(T, T+100\text{MSEC})$

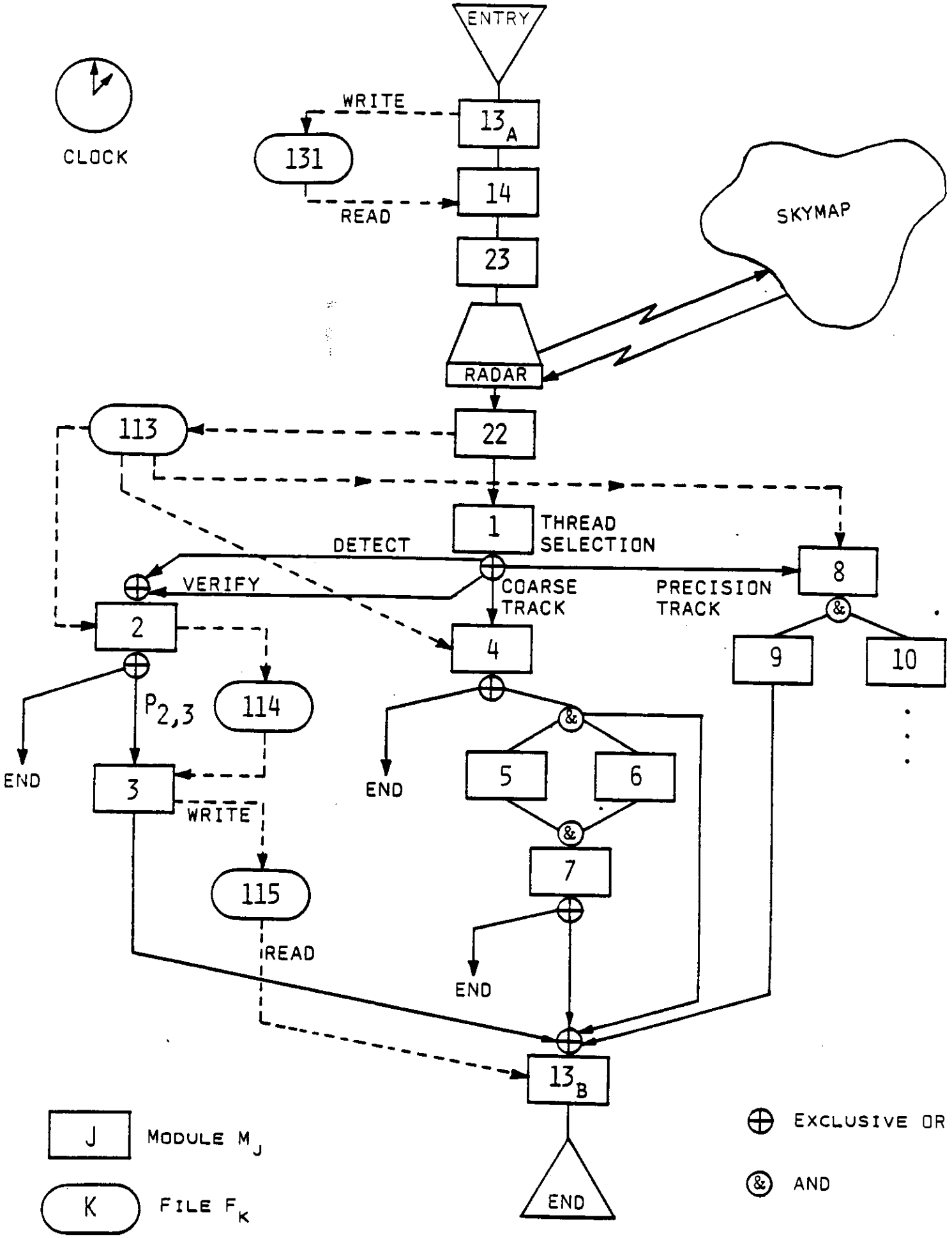


FIG. 2. EXAMPLE 1: A CONTROL-AND-DATA-FLOW GRAPH FOR A SPACE-DEFENSE TASK

IRAN82, WU84, CHU84b]. IPC can be reduced by assigning a pair of heavily communicating modules to the same processor. IMC can also be assumed to be independent of module assignments [LAN85]. A method for estimating both IMC and AET has been reported in [CHU84b].

IPC varies with module assignments because the occurrence of IPC between two communicating modules depends on whether these two modules are assigned to different processors. For example, if two modules communicate through a *replicated* shared file and reside on different processors, then the file is replicated on each processor. When a module updates the file, it updates the copy on its local processor. It then sends the updates to the remote processor, resulting in IPC which requires processing load on both the sending and receiving processors. Such IPC is eliminated if the two modules are assigned to the same processor since both modules are sharing the same local file copy.

2.2 The Objective Function

Since each module can be assigned to any of the S processors, there are S^J different ways to assign J modules to S processors, assuming that each module is assigned to one and only one processor. This can be represented by an assignment tree with S^J leaves, each leaf corresponding to a possible assignment. This tree has J levels, each representing a module. At each non-leaf node there are S downward branches, each representing the choice of a processor to host the particular module. An example with $J = 23$ and $S = 3$ is shown in Fig. 3.

An *exhaustive search* approach for module assignment is to search every leaf of the assignment tree. The optimal module assignment is the one that minimizes (or maximizes, e.g., throughput) the given objective function. Exhaustive search is usually undesirable because of the enormous amount of time required. For example, if the computation time for a leaf is 250 μs on a computer system, then the enumeration for a tree with 3^{20} leaves requires about 10

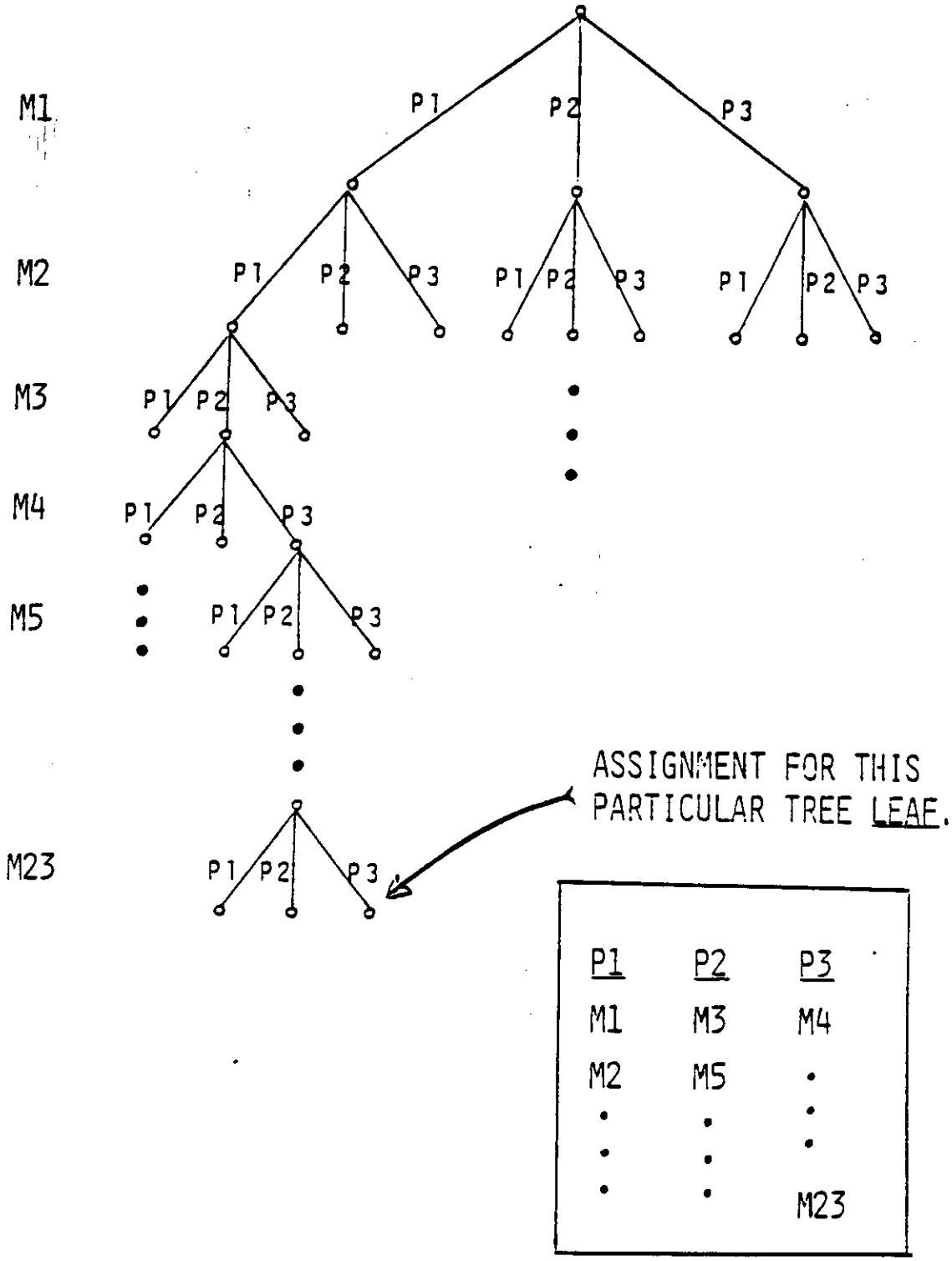


FIG. 3. AN ASSIGNMENT TREE

days of processing time which is prohibitive.

Existing approaches to task allocation can be divided into three categories: graph-theoretic [JENN77, STON77, STON78a, STON78b, RAO79, BOKH79, BOKH81, CHOU82], integer 0-1 programming approach [CHU69, CHU80, MA82], and heuristic approach [GYLY76, EFE82]. Many of these methods try to minimize a job's total cost which is defined as the sum, across all processors, of both processing cost and IPC cost dedicated to that job. This might be acceptable for a distributed system *shared* by multiple simultaneous non-real-time applications (jobs), each having program modules running on some or all of the multiple processors. Such applications attempt to maximize the total throughput. For a distributed system with identical processors, their formulation is equivalent to the minimization of IPC since the processing cost is fixed.

For real-time systems, response time is the most important performance measure. A computer system is designated solely for a specific application, i.e., the system is *not* shared by any other application. The system is required to finish a certain task within a specified time limit. Merely minimizing IPC alone may not produce a good assignment. In fact, a minimum-IPC assignment will assign all program modules to a single processor which will saturate the system and thus yield poor response time.

The processor with the heaviest loading in a distributed system is the one that causes the *bottleneck*. For instance, for a system with three processors, an assignment requiring 58%, 60%, and 61% of processor utilizations might have a better response time than a second assignment with 20%, 40% and 90% utilizations. This is mainly due to the fact that the second assignment has a *bottleneck processor* more heavily loaded than the first assignment, and queuing delay is a non-linear function that rises rapidly with the level of bottleneck (processor load).

The *processor load* consists of the loads due to 1) program module execution and 2) IPC. Therefore, both AET and IPC play important roles in module assignment and thus influence task response time. AET is usually represented in MLI (see Section 2.1). The number of transferred IPC words can be converted into the MLI's spent by both processors that send and receive the IPC.

For a given assignment X, the work load $L(r;X)$ on a given processor r is

$$L(r;X) = \sum_{j=1}^{J} x_{jr} T_j + \sum_{\substack{s=1 \\ s \neq r}}^{S} \Big[IPC(r,s;X) + IPC(s,r;X) \Big] \tag{1}$$

where $X=[x_{jr}]$ is the assignment matrix in which $x_{jr} = 1$ indicates that module M_j is assigned to processor r. The first term in the equation is the AET for all modules assigned to processor r. The second term is IPC overhead which consists of two parts: overhead due to the IPC originated from processor r to other processors, and incoming messages to processor r from other processors. For a system whose file-update messages dominate the IPC traffic, we can ignore other types of IPC such as module enablement messages and system control messages. The total overhead due to outgoing IPC at processor r is

$$\sum_{\substack{s=1 \\ s \neq r}}^{S} IPC(r,s;X) = w \sum_{j=1}^{J} x_{jr} \sum_{k=1}^{K} V_{jk} \sum_{\substack{s=1 \\ s \neq r}}^{S} \delta_{ks} \tag{2}$$

where K is the number of files used in the distributed system; V_{jk} is the IMC message volume sent from M_j to update the replicated file F_k; δ_{ks} indicates whether a replicated copy of F_k resides at processor s; the term $\sum_{\substack{s=1 \\ s \neq r}}^{S} \delta_{ks}$ gives the number of remote copies of F_k that must be updated; and w is a weighting constant for converting the message volume into MLI's. For a system with message-broadcasting capability, a file update need only be sent out once; thus the term $\sum_{\substack{s=1 \\ s \neq r}}^{S} \delta_{ks}$ in eq. (2) should be replaced by the constant *one*.

Similarly, the total overhead at processor r for incoming IPC from all remote sites is

$$\sum_{\substack{s=1 \\ s \neq r}}^{S} IPC(s,r;X) = w \sum_{\substack{s=1 \\ s \neq r}}^{S} \sum_{j=1}^{J} x_{js} \sum_{k=1}^{K} V_{jk}\, \delta_{kr} \tag{3}$$

Based on our previous discussion, we propose to use the ***work load of the bottleneck processor*** (in unit of MLI) as the objective function for module assignment, i.e.,

$$Bottleneck(X) = \max_{r=1}^{S} \left\{ L(r;X) \right\} \tag{4}$$

We want to find the assignment that yields the ***minimum bottleneck*** [CHU84a] among all possible assignments in the assignment tree, i.e.,

$$\min_{X} \left\{ Bottleneck(X) \right\} \tag{5}$$

Substituting eqs. (1) and (4) into eq. (5) yields

$$\min_{X} \left\{ \max_{r=1}^{S} \left[AET(r) + IPC(r) \right] \right\} \tag{6}$$

where $AET(r)$ and $IPC(r)$ are the total module execution time and total IPC overhead incurred at processor r.

A good assignment can be obtained from minimizing IPC and balancing processor loads among the set of processors. A minimum-bottleneck assignment generally has low IPC and fairly balanced processor loads because:

1. If the loads were not fairly balanced for an assignment, the bottleneck (highest load among all processors) would be high and this assignment would not be a minimum-bottleneck assignment.

2. If a given assignment had high IPC, the sum of processor loads over all processors would be

IPC during Phase I, heavily communicating modules are combined into groups if the AET's of the resulting groups are not too large. Each group is a *set* of modules which will be assigned *as a single unit* to a processor during Phase II. The computation required for Phase I is small because this phase is a linear-time algorithm. Phases II assigns the module groups to available processors such that the bottleneck (in the most heavily utilized processor) is minimized. Our algorithm assumes that

1. there are J modules, M_1, M_2, . . ., M_J, and S processors;

2. the average AET (over the peak-load period), T_j, for all modules M_j ($j = 1,...,J$) are given;

3. the average IMC between any module pair M_i and M_j, $IMC_{i,j}$, ($i = 1,...J$; $j = 1,...J$) is given.

ALGORITHM I-A:

Phase I: Combine modules with large IMC into groups to reduce total system load.

1.1 Initially list all module pairs (M_i, M_j) in the *descending* order of IMC volume.

Calculate average AET and average processor load:

$$\overline{AET} \leftarrow \sum_{j=1}^{J} T_j / J$$

$$\overline{PL} \leftarrow \sum_{j=1}^{J} T_j / S$$

Set threshold values for IMC volume and for processor load:

$$\theta_{IMC} \leftarrow \overline{AET} \times \alpha\ \%$$

$$\theta_{PL} \leftarrow \overline{PL} \times \beta\ \%$$

Let each program module form a distinct group (a set):
$G_j \leftarrow \{M_j\} \quad j = 1,...,J$

1.2 If no more pairs exist in the module-pair list
go to Phase II.
Pick the next pair of modules, M_i and M_j, and delete this pair from the list.

1.3 If $IMC_{i,j} \leq \theta_{IMC}$
go to Phase II.

1.4 Find the group G_s that contains M_i, and the group G_t that contains M_j (i.e., $M_i \in G_s$, $M_j \in G_t$).
If $s = t$ (i.e., if M_i and M_j are already in the same group)
go to Step 1.2.

1.5 If $T_s + T_t > \theta_{PL}$
go to Step 1.2.

1.6 Combine the two groups G_s and G_t into a single one:
$G_s \leftarrow G_s \cup G_t$
$G_t \leftarrow \emptyset$
$T_s \leftarrow T_s + T_t$
$T_t \leftarrow 0$

1.7 Go to Step 1.2.

Phase II: Assign module groups to processors.

2.1 Perform an exhaustive search through the new assignment tree for the assignment that has the smallest bottleneck.

2.2 Stop.

Note Phase I reduces J modules to G groups, $G < J$, which corresponds to a much smaller assignment tree. Let us now discuss the rationale of Steps 1.3 and 1.5, respectively.

Step 1.3: For a pair of modules whose IMC is smaller than the *IMC threshold* θ_{IMC} (α % of $\overline{AET}$), merging them gives little benefit in terms of the IPC saved. Our experience reveals that α should range between 1% to 10%.

Step 1.5: Our assignment algorithm reduces large IPC. However, when merging two groups into one, we should leave some processing capacity in the resulting new group for accommodating the remaining IPC and the possible grouping with some other module groups. If two groups were combined and formed a group that was too large, we could not obtain a balanced-load assignment during Phase II. Therefore, the *processor-load threshold* θ_{PL} is limited to β % of the average processor load $\overline{PL}$.

3.2 Example 1: the DPAD System

In this section we demonstrate the performance of both the proposed objective function and Algorithm I-A by applying them to an example system, the Distributed Processing Architecture Design (DPAD) system. The DPAD system was developed to manage the data processing and radar resources for a space-defense application [GREE80, HOFF80]. A portion of its control-and-data-flow graph is given in Fig. 2. The twenty (20) modules are to be assigned to three processors.

3.2.1 Performance of the Proposed Objective Function

The average AET and IMC during the peak-load period for all modules of the DPAD system are given in Table 1. The identified peak-load period is from 1.0 s to 2.0 s of mission time. For example, $T_8 = 32{,}055$ MLI is the average of ten measured AET values within the period, at each increment of 100 ms. Column 3 shows the file(s) updated by the write-module. Each IMC value in column 4 is the total file-update volume for 100 ms, written to the file in column 3 by the write-module in column 1; like AET, this IMC value is an average of ten values in the peak-load period. Column 5 lists all the modules which read the updated file. If a read-module for a file and the associated write-module are on different processors, both processors would have a copy of the file and IPC occurs for updating the replicated file copy.

A FORTRAN program was developed to compute the proposed objective function for *every* assignment in the assignment tree. When an assignment (corresponding to a tree leaf) yields a bottleneck value smaller than the smallest bottleneck obtained so far, that assignment is saved. The last ten saved assignments, denoted as Assignment #1 through #10 (Fig. 4), were simulated with the DPAD simulator and their performance compared. Figs. 5a shows the CPU utilization for the minimum-bottleneck Assignment #1. We note that the loads for the three processors are quite balanced during the peak-load period. The processor loads for

Write Module	AET (in MLI)	File Updated	IMC Size (in MLI)	Read Modules
M1	8865	none		
M2	2700	F114	124	M3
M3	1590	F115	144	M13
M4	10410	F116	112	M13
		F117	314	M5,M6,M7
M5	1860	F119	68	M7
M6	1950	F121	68	M7
M7	1680	F122	67	M13
M8	32055	F120	62	M13
		F123	1568	M6,M10,M16 M9,M17,M19,M20
		F124	6387	M6,M10,M16
M9	18600	F125	806	M13
M10	3360	F127	1	M8
		F134	1	M18
(M11)	0	Module Not Implemented		
(M12)	0	Module Not Implemented		
M13	25305	F131	30371	M14
M14	16860	F147	1800	M13
		F132	5019	M23
(M15)	0	Module Not Implemented		
M16	4170	F135	100	M18
M17	6240	F136	100	M18
M18	3975	F137	229	M19
		F138	36	M8
		F139	244	M20
M19	9705	F139	599	M20
M20	2010	F140	62	M21
M21	195	F141	32	Radar
M22	16410	F142	242	M23
		F113	4593	M1,M2,M4,M8
M23	17025	F112	5112	Radar
Radar		F111	14737	M22

Each Transferred IPC Word Has Been Converted To 3 MLIs.

Each AET and IMC Value Is an Average of Ten Values Over the Peak-Load Period Between 1.0 and 2.0 Seconds.

TABLE 1. AET AND FILE-UPDATE IMC (IN MLI) PER 100 MS

FIG. 4 ENUMERATION RESULTS: 10 GOOD ASSIGNMENTS

	ASSIGNMENT	LOAD-1	LOAD-2	LOAD-3	BOTTLENECK	TOTAL LOAD
10th	11111 11121 00330 12322 123	75612	75546	70420	75612	221578
9th	11111 11121 00330 12322 323	75323	75546	70709	75546	221578
8th	11112 11121 00330 13222 223	75413	75352	73643	75413	224408
7th	11112 12131 00220 13231 132	75178	74275	73829	75178	223282
6th	11112 12131 00220 13231 232	75013	74564	73829	75013	223406
5th	11112 32333 00220 33211 112	74414	74275	74023	74414	222712
4th	11112 32333 00220 33211 312	74249	74275	74312	74312	222836
3rd	12123 23212 00330 21322 113	74308	73873	74275	74308	222456
2nd	12123 23212 00330 21322 213	74019	74038	74275	74275	222332
MINIM. BOTTLE-NECK	12213 13121 00330 11322 223	74004	73805	74275	74275	222084

NOTE: 1. LOAD-i IS EACH PROCESSOR'S LOAD PER 100 MSEC (IN UNIT OF MLI).

2. AN ASSIGNMENT WITH THE MINIMUM TOTAL LOAD
IS NOT THE ASSIGNMENT WITH THE MINIMUM BOTTLENECK.

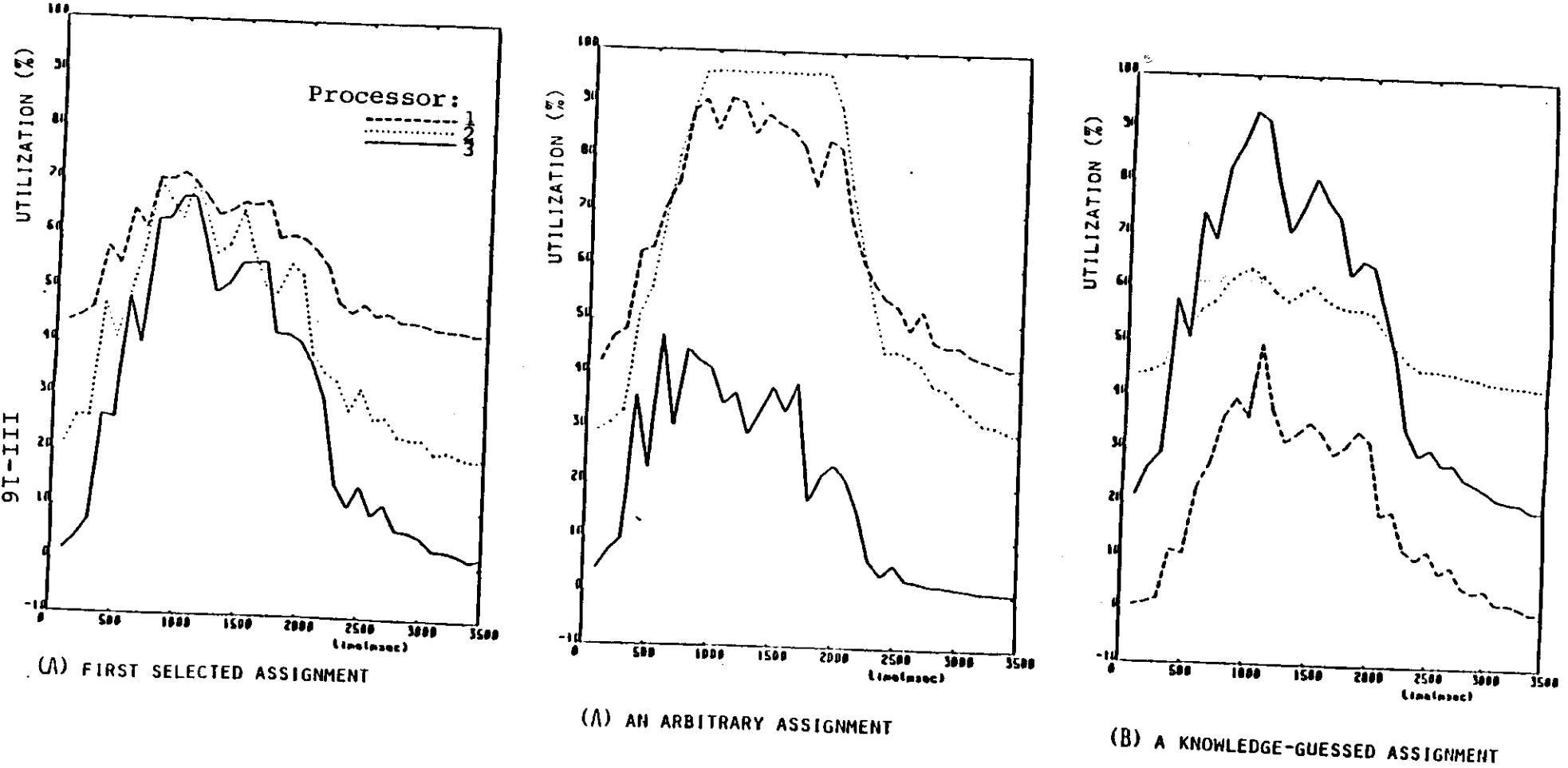


FIG. 5. PROCESSOR UTILIZATION FOR A) THE BEST MODULE ASSIGNMENT SELECTED BY EXHAUSTIVE SEARCH, B) AN ARBITRARY ASSIGNMENT, AND C) A KNOWLEDGE-GUESSED ASSIGNMENT

Assignments #2 through #10 are also fairly balanced. This verifies our conjecture that the minimum-bottleneck objective function provides *balanced* loads. For comparison, Figs. 5b and 5c show the processor loads for an arbitrary assignment and a manually generated knowledge-guessed [1] assignment [HOLL82]; they are less balanced and their bottleneck loads are much higher than that of Assignment #1.

Fig. 6 shows the Precision-Tracking port-to-port time (the DPAD terminology for response time for a task thread) for Assignments #1 through #9 as well as the above arbitrary assignment. The arbitrary assignment has a poor performance because two of its three processors are saturated. Note that the performance difference between a good and a bad assignment can be substantial. Poor assignments yield poor response time.

Fig. 7 compares the port-to-port time for the Precision Tracking thread between the knowledge-guessed assignment and Assignments #1 through #9. Similar results were obtained for other tactical threads. The experiments reveal that our proposed objective function generates good module assignments.

3.2.2 Application of Algorithm I-A To the DPAD System

Let us now apply Algorithm I-A to the DPAD module assignment problem. Table 1 shows the IMC between a module and each file it accesses (updates or reads). For Phase I of Algorithm I-A, this table is reorganized into Table 2 which provides IMC size between all module pairs. (Phase II uses Table 1). Fig. 8 shows the merging process of Phase I where 5% and 75% are used for the α and β respectively. Column 1 in Fig. 8 lists the IMC values in descending order, column 2 displays the modules combined into a group, and column 3 calculates the total AET for all modules in the group.

---

[1] The knowledge-guessed assignment was obtained by a combination of intuitive insight and trial-and-error. It was one of the best assignments known to the authors in terms of port-to-port time for the DPAD system.

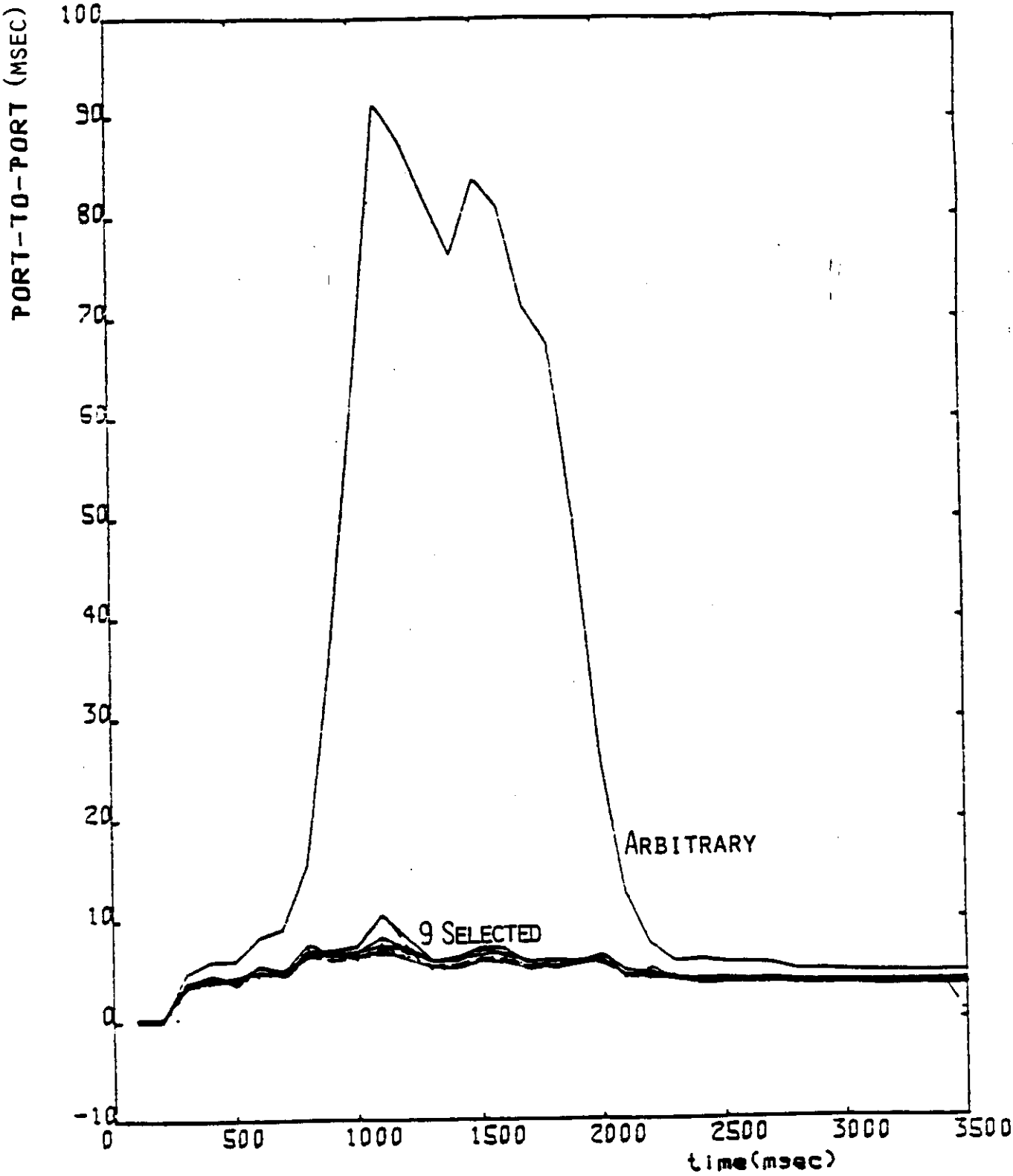


FIG. 6. PTP TIME FOR PRECISION-TRACKING THREAD --- COMPARE AN ARBITRARY ASSIGNMENT AND TOP NINE ASSIGNMENTS SELECTED FROM EXHAUSTIVE SEARCH

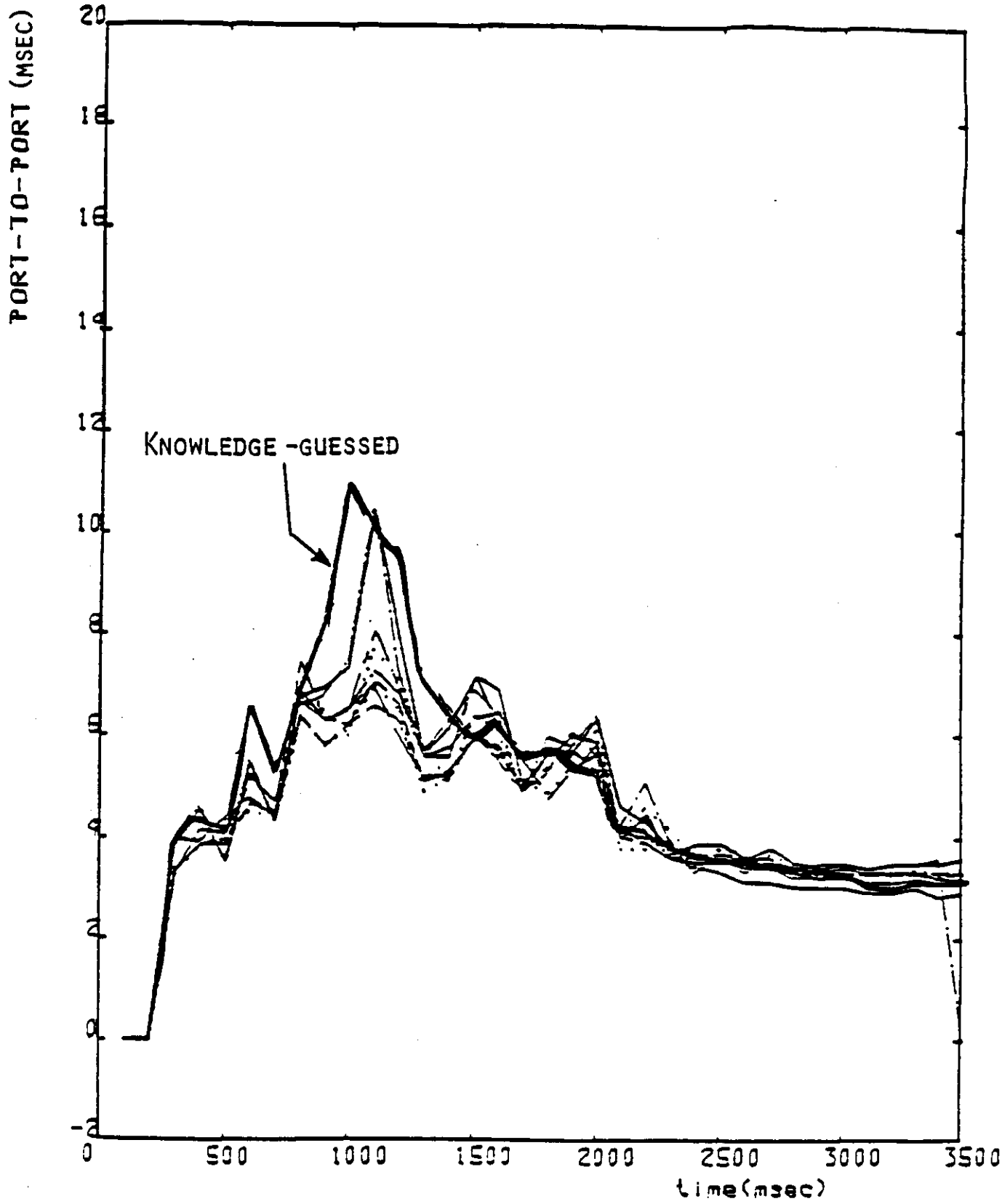


FIG. 7. PTP TIME FOR PRECISION-TRACKING THREAD --- COMPARE THE KNOWLEDGE-GUESSED ASSIGNMENT AND TOP NINE ASSIGNMENTS FROM EXHAUSTIVE SEARCH

Write Module	Files Involved	IMC Size (in MLI)	Read Module
M2	F114	124	M3
M3	F115	144	M13
M4	F117	314	M5
M4	F117	314	M6
M4	F117	314	M7
M4	F116	112	M13
M5	F119	68	M7
M6	F121	68	M7
M7	F122	67	M13
M8	F123, F124	7955	M6
M8	F123	1568	M9
M8	F123, F124	7955	M10
M8	F120	62	M13
M8	F123, F124	7955	M16
M8	F123	1568	M17
M8	F123	1568	M19
M8	F123	1568	M20
M9	F125	806	M13
M10	F127	1	M8
M10	F134	1	M18
M13	F131	30371	M14
M14	F147	1800	M13
M14	F132	5019	M23
M16	F135	100	M18
M17	F136	100	M18
M18	F138	36	M8
M18	F137	229	M19
M18	F139	244	M20
M19	F139	599	M20
M20	F140	62	M21
M21	F141	32	Radar
M22	F113	4593	M1
M22	F113	4593	M2
M22	F113	4593	M4
M22	F113	4593	M8
M22	F142	242	M23
M23	F112	5112	Radar
Radar	F111	14737	M22

Each Transferred IPC Word Has Been Converted To 3 MLIs.

Each IMC Value Is an Average of Ten Values
Over the Peak-Load Period Between 1.0 and 2.0 Seconds.

TABLE 2. TOTAL FILE-UPDATE IMC PER 100 MS FOR MODULE PAIRS FOR EXAMPLE 1

IMC(i,j) (in MLI)	Modules in the Merged Group	Exec. Time of the Merged Group
IMC(13,14) = 30371	13-14	25305+16860=42165
IMC(8, 6) = 7955	6-8	1950+32055=34005
IMC(8,10) = 7955	6-8-10	34005+3360=37365
IMC(8,16) = 7955	6-8-10-16	37365+4170=41535
IMC(14,23) = 5019	Can't group 13-14-23	Note 1
IMC(22, 1) = 4593	1-22	8865+16410=25275
IMC(22, 2) = 4593	1-2-22	25275+2700=27975
IMC(22, 4) = 4593	1-2-4-22	27975+10410=38385
IMC(22, 8) = 4593	Can't group 1-2-4-6-8-10-16-22	Note 1
IMC(14,13) = 1800	Note 2	
IMC(8, 9) = 1568	Can't group 6-8-9-10-16	Note 1
IMC(8,17) = 1568	Can't group 6-8-10-16-17	Note 1
IMC(8,19) = 1568	Can't group 6-8-10-16-19	Note 1
IMC(8,20) = 1568	6-8-10-16-20	41535+2010=43545
IMC(9,13) = 806	Can't group 9-13-14	Note 1
IMC(19,20) = 599	Can't group 6-8-10-16-19-20	Note 1
IMC(4, 5) = 314		

(Phase I finishes because IMC(4,5) = 314 < θ_{IMC})

Note 1: Otherwise, the merged group would have a total AET greater than θ_{PL}.

Note 2: M_{13} and M_{14} are already in the same group.

FIG. 8. PHASE I OF ALGORITHM I-A FOR DPAD EXAMPLE

For this example, $\overline{AET} = \sum_{j=1}^{23} T_j / 20 = 8{,}933$ MLI and $\overline{PL} = \sum_{j=1}^{23} T_j / 3 = 59{,}555$ MLI. Thus $\theta_{IMC} = 446.7$ MLI and $\theta_{PL} = 44{,}666.3$ MLI. Phase I finishes when $IMC_{4,5} = 314$ MLI is considered since 314 is smaller than θ_{IMC}. The resultant groups are:

Group	Modules
1	1,2,4,22
2	3
3	5
4	6,8,10,16,20
5	7
6	9
7	13,14
8	17
9	18
10	19
11	21
12	23,(11,12,15)*

* Modules M_{11}, M_{12}, and M_{15} are not implemented in the DPAD and thus have zero AET.

We have merged 20 modules into 12 groups. This implies a reduction from 3^{20} possible module assignments to 3^{12} possible group assignments which reduces the computation time from 10 days on a VAX-11/780 to two minutes.

To evaluate the effectiveness of our Algorithm I-A, the best assignment obtained from the algorithm is compared with that from the exhaustive search (Fig. 9). We note that the module assignment generated by Algorithm I-A provides response-time performance comparable with that from the exhaustive search. We have also used our DPAD simulator to simulate the four assignments (Assignments A-1 through A-4) reported in [MA82] which minimizes the sum of AET and IPC and thus, does usually not generate balanced-load assignments (see Section 2.2). Therefore the assignment generated by our algorithm performs better than that of [MA82], as shown in Fig. 10.

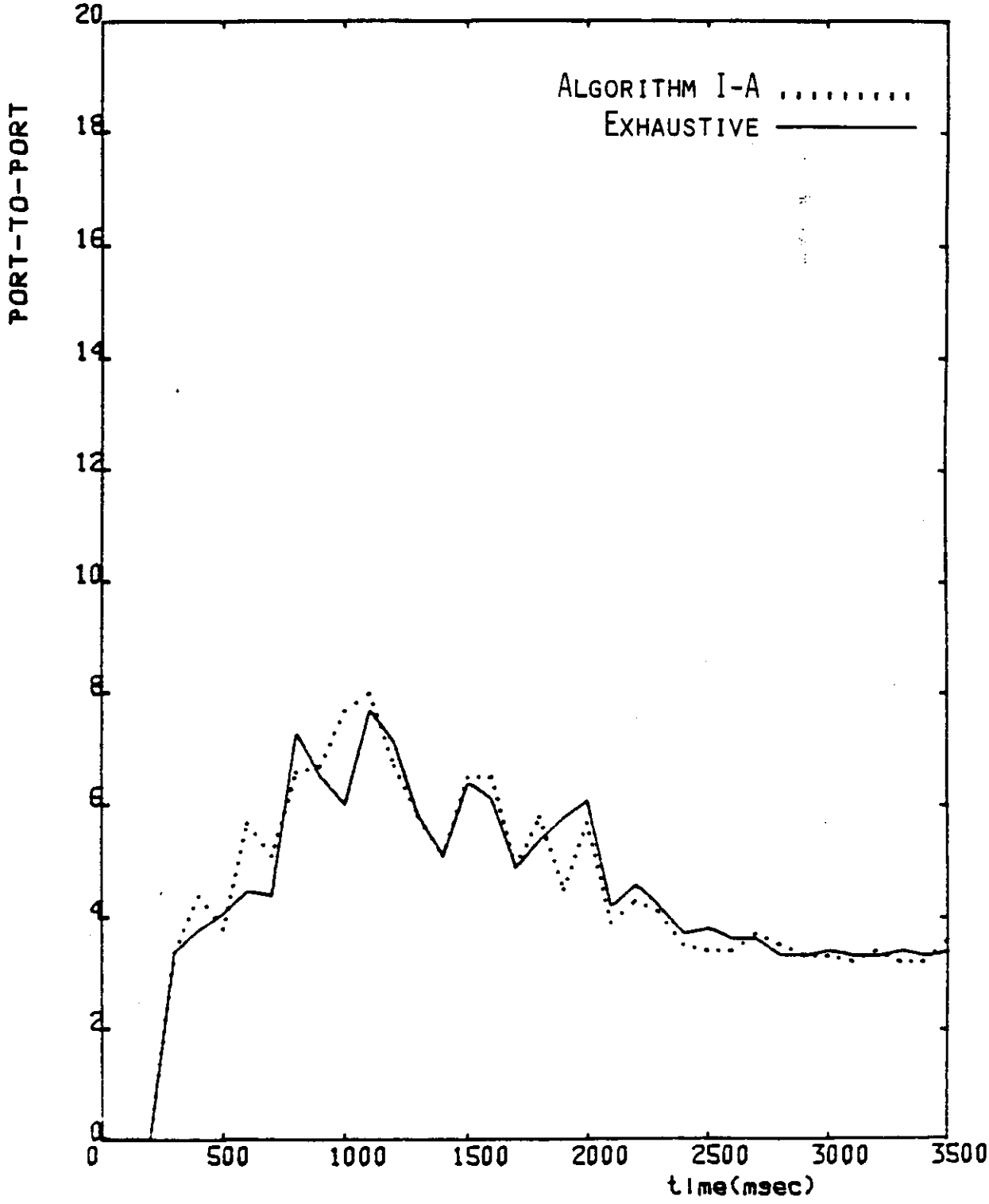


FIG. 9. PERFORMANCE COMPARISON OF PRECISION-TRACKING-THREAD BETWEEN THE BEST ASSIGNMENT FROM HEURISTIC ALGORITHM AND THAT SELECTED FROM THE EXHAUSTIVE SEARCH

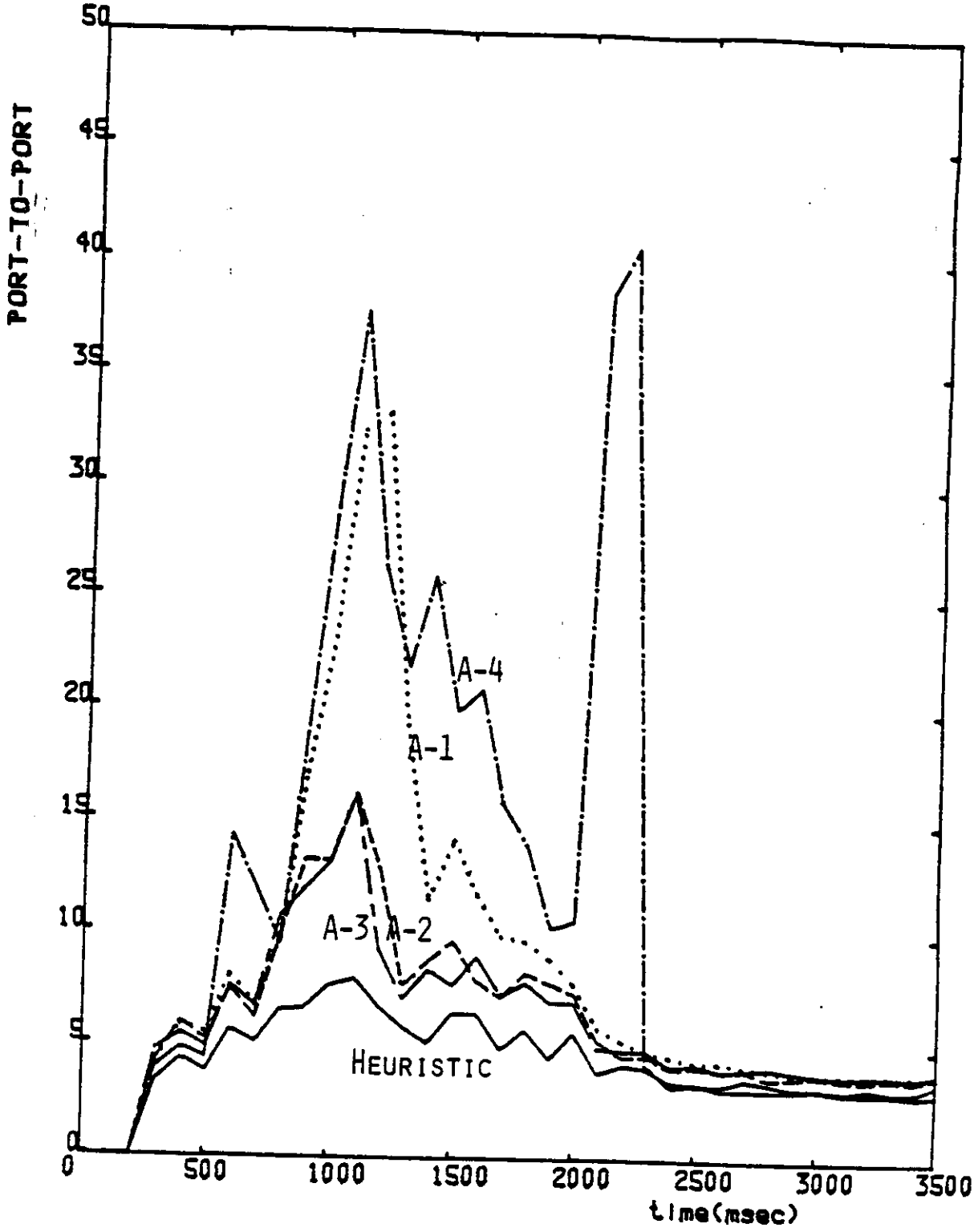


FIG. 10. PTP TIME FOR PRECISION-TRACK THREAD --- COMPARE THE BEST ASSIGNMENT FROM ALGORITHM I-A WITH THE FOUR ASSIGNMENTS FROM [MA82]

4. PRECEDENCE RELATIONSHIP AND MODULE ASSIGNMENT

Another important factor that needs to be considered in task allocation is precedence relation (PR) among program modules. In this section we describe several experiments on PR and study its impact on response time. In Experiment No. 1, we compare three assignments of assigning nine (9) modules to three processors (Fig. 11). PR exists in the control-flow graph from one module to another. Assume the job arrival is a Poisson process with arrival rate λ and each job *enables* module M_1 which is placed in the *ready queue* of M_1's residence processor waiting to be processed. Upon the completion of execution, a module enables the succeeding module in the control-flow graph; the enabled module is placed in the ready queue of its residence processor. The execution time of every module is assumed to be *constant* (i.e., deterministic service time) and equal to *one* time unit. To simplify our analysis and to isolate the PR effect, we further assume there is no IMC between modules and thus no IPC overhead between processors. The three load-balanced assignments in Fig. 11 are simulated with PAWS simulator [BERR82], using FCFS queuing discipline. Simulation results (Fig. 12) reveal a significant difference in response time among these assignments. The pipeline assignment (#2) yields the best response time. Vertical bars in the figure represent 90%-confidence intervals for each simulation point. Since all assignments yield balanced loads and there is no IPC overhead, the response-time discrepancy among these assignments is due solely to PR among modules.

In Experiment No. 2, each module has an *exponential,* instead of deterministic, execution time with a *mean* of one time unit. All other parameters remain the same as those used in Experiment No. 1. Simulation results (Fig. 12) exhibit that the response times for all three assignments are about the same. This is because when every execution time (service time) is exponentially distributed, each processor in the system can be treated individually as an M/M/1 queue. Since all modules have the same service time distribution and the same

ASSIGNMENT #1 (SEQUENTIAL)

COMPUTER	1	2	3
MODULES #	1	4	7
	2	5	8
	3	6	9

ASSIGNMENT #2 (PIPELINED)

COMPUTER	1	2	3
MODULES #	1	2	3
	4	5	6
	7	8	9

ASSIGNMENT #3 (SKEWED)

COMPUTER	1	2	3
MODULES #	1	2	3
	5	6	4
	9	7	8

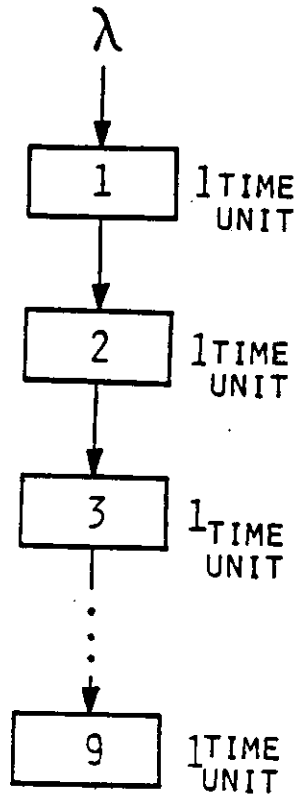


FIG. 11. PRECEDENCE RELATIONSHIP EXPERIMENT NO. 1: TASK CONTROL-FLOW GRAPH AND THREE MODULE ASSIGNMENTS

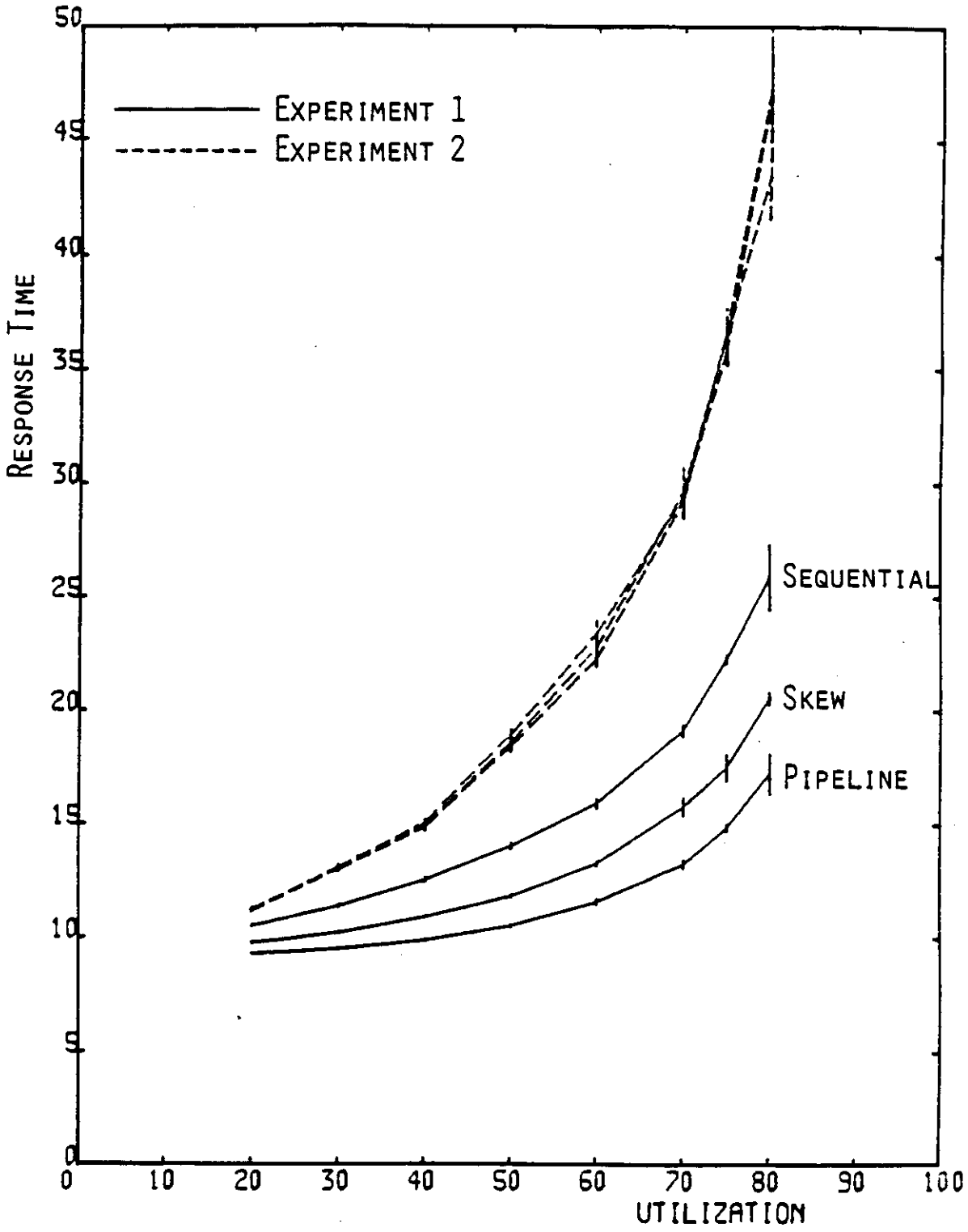


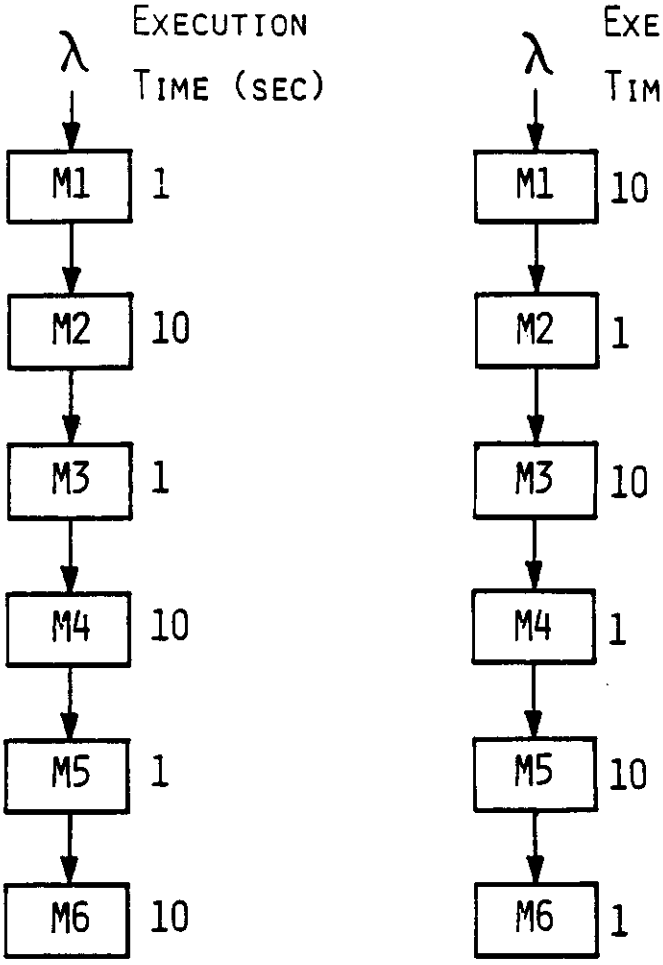
FIG. 12. PRECEDENCE RELATIONSHIP EXPERIMENTS: COMPARE THE RESPONSE TIME OF THREE MODULE ASSIGNMENTS. EXPERIMENT NO. 1 USES DETERMINISTIC EXECUTION TIME. EXPERIMENT NO. 2 USES EXPONENTIAL EXECUTION TIME

invocation (arrival) rate in this experiment, all load-balanced processors are treated as identical M/M/1 queues, and thus all modules have identical wait-time. Experiment No. 1 reveals that precedence relationship does have an impact on task response time. Experiment No. 2 reveals that module execution-time distributions alter the PR's effect on response time.

Experiment No. 3 is for testing the effect of module size in precedence relationship. Modules are assumed to have exponential execution times with the mean values shown in Fig. 13a. The three assignments in Fig. 13b are simulated and the experimental results (Fig. 14) reveal that assigning two consecutive modules to the same processor yields good response times *if the execution time of the second module is much larger than that of the first module.* We denote this as our PR rule #1. For example, in Assignment #1, M_1 and M_2 are assigned to the same processor. *If the second module is much smaller than the first one, it is better to separate two consecutive modules and assign them to distinct processors.* This is our PR rule #2. In Assignment #1, M_2 and M_3 are assigned to different processors. Because Assignment #1 satisfies PR rules for all pairs of consecutive modules, it yields the best response time. Assignment #2 is the worst because it violates PR rules for all module pairs. Assignment #3 violates PR rule #1 for some module pairs (e.g., separation of M_1 from M_2) and satisfies PR rule #2 for some other pairs (e.g., separation of M_2 from M_3), therefore its performance lies between Assignments #1 and #2.

Experiment No. 4 is similar to No. 3 except with different execution times as shown in Fig. 13c. The same three assignments in Fig. 13b are used in the experiment. From Fig. 14 we note that Assignment #2 yields the best performance because it satisfies PR rules for all pairs of consecutive modules. Assignment #1 is the worst since it violates PR rules for all module pairs. We repeated these experiments with deterministic execution times and obtained the same results. The intuitive reasons for the PR rules are as follows:

1. If the arrival process for a module is highly random such as Poisson, there are occasions



Experiment No. 3

(A)

Experiment No. 4

(C)

Assignment	CPU 1	CPU 2	CPU 3
1	M1,M2	M3,M4	M5,M6
2	M1,M6	M2,M3	M4,M5
3	M1,M4	M2,M5	M3,M6

(B)

Fig. 13. Task Control-Flow Graph and Module Assignments for PR Experiments 3 (solid curves) and 4 (dashed curves)

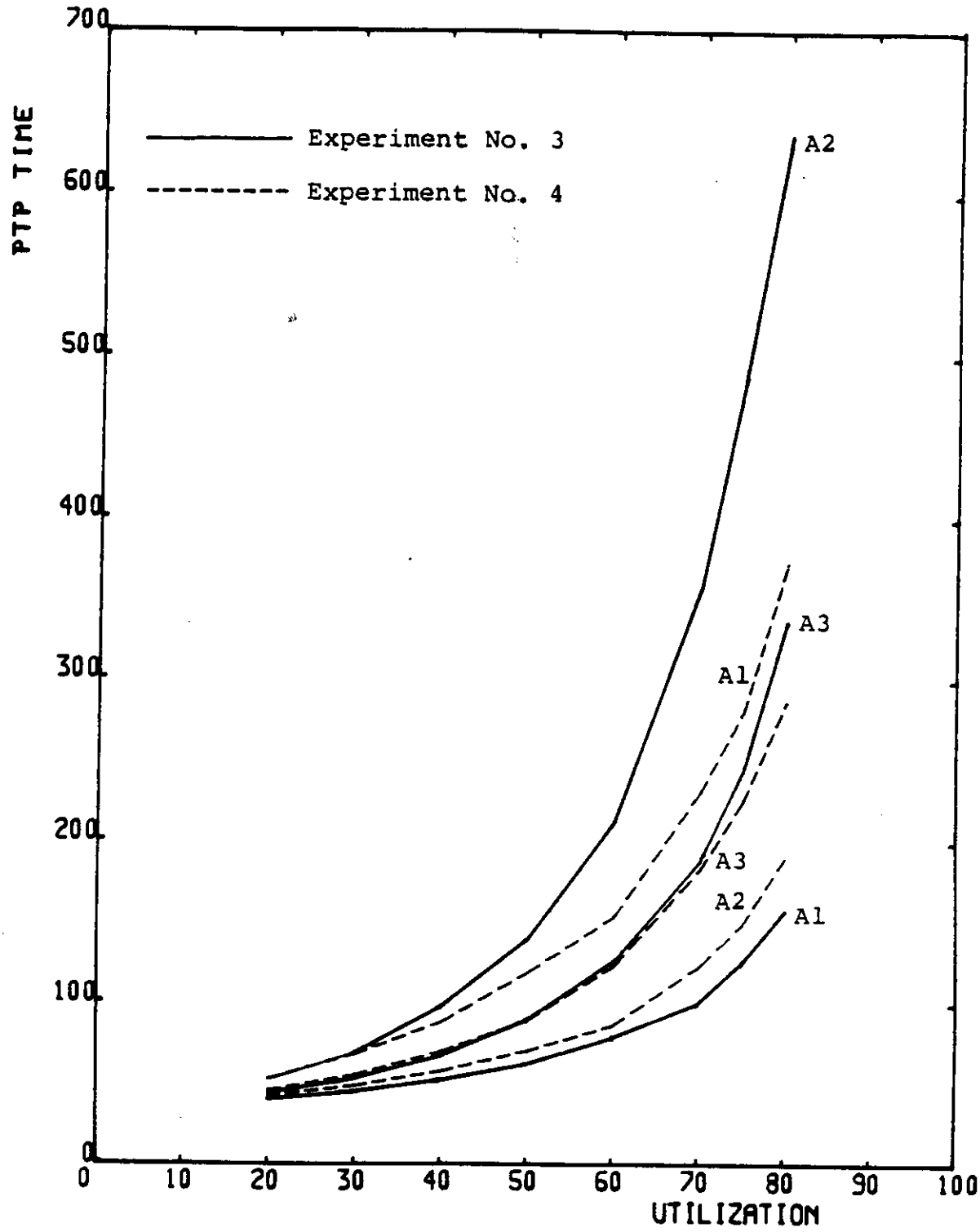


FIG. 14. PERFORMANCE OF THREE MODULE ALLOCATIONS FOR PR EXPERIMENTS 3 AND 4

for bursty arrivals. On the other hand, if the job arrival process is deterministic, the work load is *evenly spread* over the time; therefore the average queue-length at every processor as well as the average module wait-time should be smaller than that of a Poisson arrival process.

2. If two consecutive modules are assigned to the same processor and if the second module is much larger than the first one, the second one will act as a *regulating valve* which regulates the task flow into the next processor. For instance, in Assignment #1 of Experiment No. 3, although there are bursty arrivals for M_1, the invocation arrivals for M_3 at Processor 2 are not bursty and are fairly evenly spread over time because M_2 at Processor 1 is a large module. As a result, the queue (containing invocations for M_3 and M_4) at Processor 2 is short and yields short wait-time for M_3 and M_4. In the same manner, M_4 acts as a regulating valve for the task flow into Processor 3.

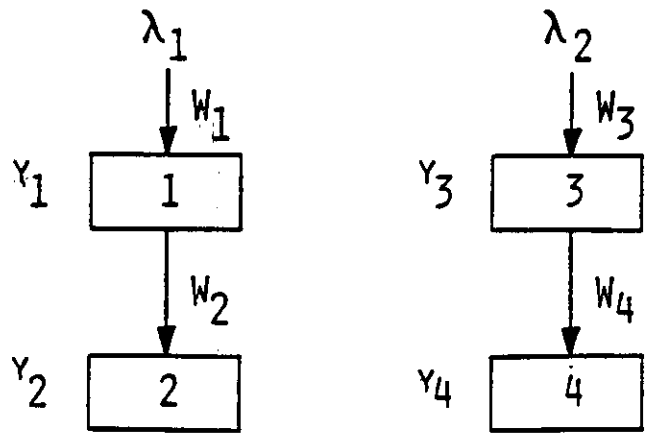
3. The reason that Assignment #2 of Experiment No. 3 yields poor response time is mainly due to the fact that the size of M_1 is small. Bursty invocation arrivals for M_1 result in bursty arrivals for M_2 at Processor 2 (i.e., there is no regulating valve between Processors 1 and 2). As a result, there is a high probability to have several arrivals for M_2 in Processor 2's ready queue. A newly invoked arrival for M_2 at Processor 2 may find a previously arrived M_2 in execution and possibly some other M_2's waiting in the queue. After the first (the oldest) M_2 in the queue finishes its execution, it invokes an M_3 arrival and places that arrival at the end of the queue. Thus, M_3 invocations will experience a long wait-time because of the large M_2's in front of them. Later on, there will be multiple M_3's *next to each other* which will then quickly finish their execution (because the M_3 size is small) and generate bursty invocations for M_4 at Processor 3.

Having noted that the *module-size ratios* of consecutive modules influence response time, we need to determine when two consecutive modules M_1 (of module size y_1) and M_2 (of module

size y_2) should be located in the same processor.

Consider the control-flow graph in Fig. 15 where all modules have deterministic execution times. Let $y_1 = y_2$, $y_3 = y_4$ (thus, the module-size ratio $r_{1,2} = y_2/y_1 = r_{3,4} = y_4/y_3$), and job arrival rates $\lambda_1 = \lambda_2$. Both Assignments #1 and #2 result in balanced processor loads. We like to use the module-size ratio $r_{1,2}$ ($= r_{3,4}$) as a key parameter to determine if M_1 and M_2 (also, M_3 and M_4) should be co-located. That is, if $r_{1,2}$ is greater than some threshold value, M_1 and M_2 should be assigned to the same processor; otherwise they should be separated.

Because of the symmetry in this control-flow graph and in loading on both processors, the two threads in the graph have the same response time, which is $w_1 + y_1 + w_2 + y_2$ ($= w_3 + y_3 + w_4 + y_4$), where w_i is the queuing wait-time for module M_i. Given any module assignment for any control-flow graph, a model developed in [CHU84c] can estimate the wait-time w_i's for all modules. Since y_1, y_2, y_3, and y_4 are constants (independent of module assignment), the ***wait-time ratio*** between Assignments A_1 and A_2, $R_w = R(A_1/A_2) = \frac{w_1(A_1) + w_2(A_1)}{w_1(A_2) + w_2(A_2)}$, can be used as a measure for selecting a good assignment. If $R_w < 1$, then Assignment #1 is better than Assignment #2, i.e., we should assign the consecutive modules M_1 and M_2 to one processor, and the other consecutive modules M_3 and M_4 to another processor. If $R_w > 1$, then Assignment #2 is better and consecutive modules should be run on different processors. Fig. 16 shows the wait-time ratio R_w for various module-size ratio $r_{1,2} = y_2/y_1$. The horizontal axis is processor utilization $\rho = \rho_1 + \rho_2$, where $\rho_1 = \lambda_1 y_1$ and $\rho_2 = \lambda_1 y_2$ are contributed by the execution of M_1 and M_2, respectively. Note that as $r_{1,2}$ decreases, R_w increases until reaching approximately to 1.7; then it reverses the trend and decreases. R_w varies only slightly with the processor utilization. Curves for wait-time ratio can be obtained in the same manner when the execution time of each module is changed from a



$Y_1 = Y_3 \quad Y_2 = Y_4 \quad \lambda_1 = \lambda_2$

Assignment	CPU 1	CPU 2
1	M1,M2	M3,M4
2	M1,M4	M2,M3

FIG. 15. TWO THREADS OF CONSECUTIVE MODULES FOR STUDYING WAIT-TIME RATIO BETWEEN ASSIGNMENTS 1 AND 2 AS A FUNCTION OF SIZE RATIO BETWEEN THE CONSECUTIVE MODULES

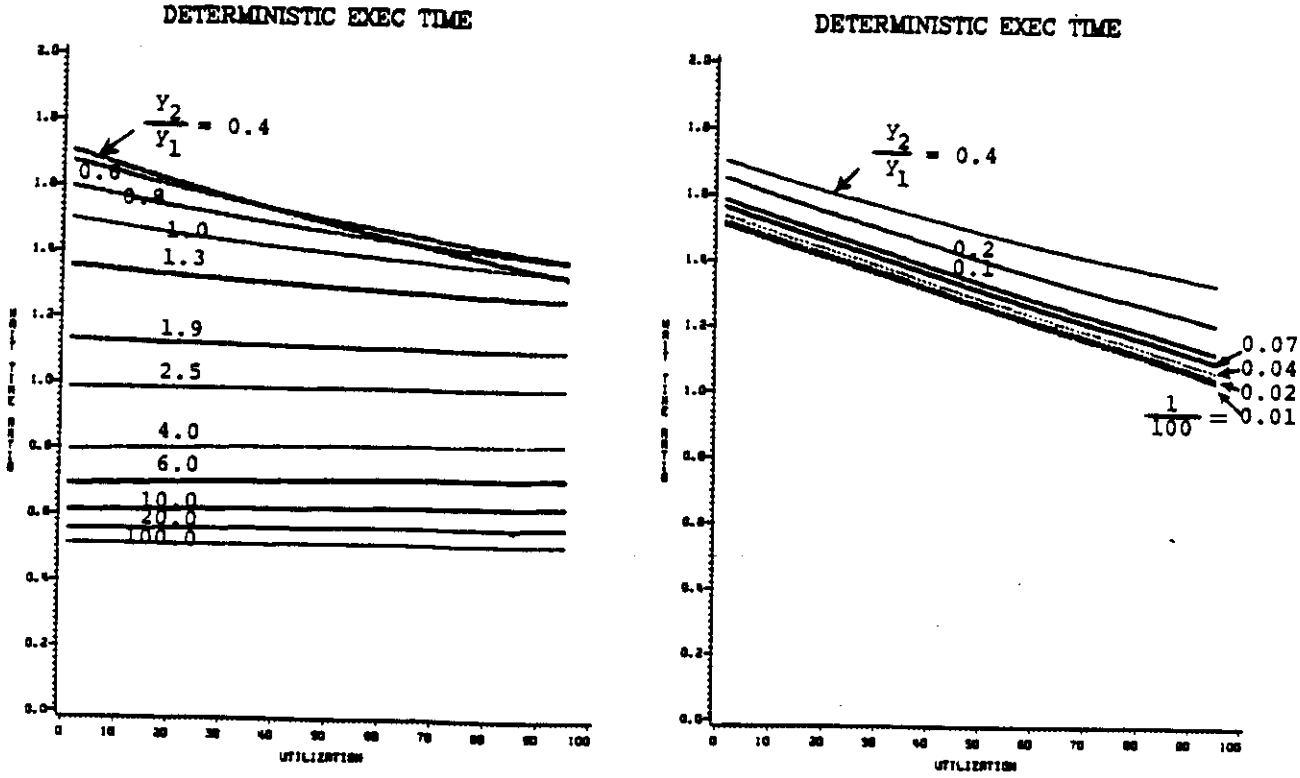
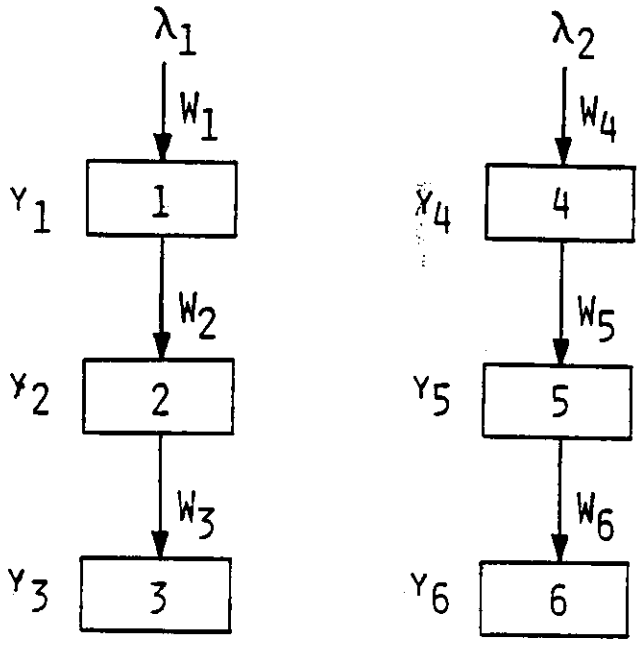


FIG. 16. WAIT-TIME PATIO BETWEEN TWO ASSIGNMENTS AS A FUNCTION OF MODULE-SIZE RATIO

deterministic value to an exponentially or hyperexponentially distributed one. Since the execution times of most program modules are more deterministic than exponentially or hyperexponentially distributed, the following discussions will be for deterministic execution time.

We shall now study the execution of *three* consecutive modules (Fig. 17), using the wait-time ratio $R_w = \dfrac{w_1(A_1) + w_2(A_1) + w_3(A_1)}{w_1(A_2) + w_2(A_2) + w_3(A_2)}$. Our analysis shows that if the size of M_1 is fixed (thus, $\rho_1 = \lambda_1 y_1$ is fixed), as the size ratio of M_3 to M_2 ($r_{2,3} = y_3/y_2 = \rho_3/\rho_2$) decreases, R_w increases to a certain point and then reverses the trend and decreases (Fig. 18). Likewise, fixing M_3 and varying the size ratio of M_2 to M_1, we observe similar results. These relations between R_w and $r_{i,j}$ are similar to the previous observations for the two-module threads as shown in Fig. 16. Similar relations are also observed for a control-flow graph consisting of four modules in each thread. When y_1, y_2, and y_3 (Fig. 17) vary simultaneously, the wait-time ratio is shown in a 3-dimension contour plot (Fig. 19). Note that when both size ratios $r_{1,2}$ and $r_{2,3}$ are large, the wait-time ratio R_w is the smallest. Thus assigning all three consecutive modules in a thread to the same processor (i.e., Assignment #1) yields better response time, which is consistent with our previous observations. If $r_{1,2}$ is large while $r_{2,3}$ is small, or vice versa, then the benefit from one module pair (e.g., M_1 and M_2) is canceled out by the penalty from another pair (e.g., M_2 and M_3). As a result, both assignments have similar wait-time (i.e., $R_w \approx 1$). If both $r_{2,3}$ and $r_{1,2}$ are small, then Assignment #2 is better than Assignment #1.

Our experimental observations reveal that in assigning modules to processors, each pair of consecutive modules in a control-flow graph can be treated independently, and using the PR rules on each individual pair of consecutive modules in task allocation yields good task response time.



$Y_1 = Y_4$ $Y_2 = Y_5$ $Y_3 = Y_6$ $\lambda_1 = \lambda_2$

Assignment	CPU 1	CPU 2
1	M1,M2,M3	M4,M5,M6
2	M1,M5,M3	M4,M2,M6

Fig. 17. Three Consecutive Modules in Each Control-Flow Thread for Studying Wait-Time Ratio Between Assignments 1 and 2 as a Function of Size Ratio Between Consecutive Modules

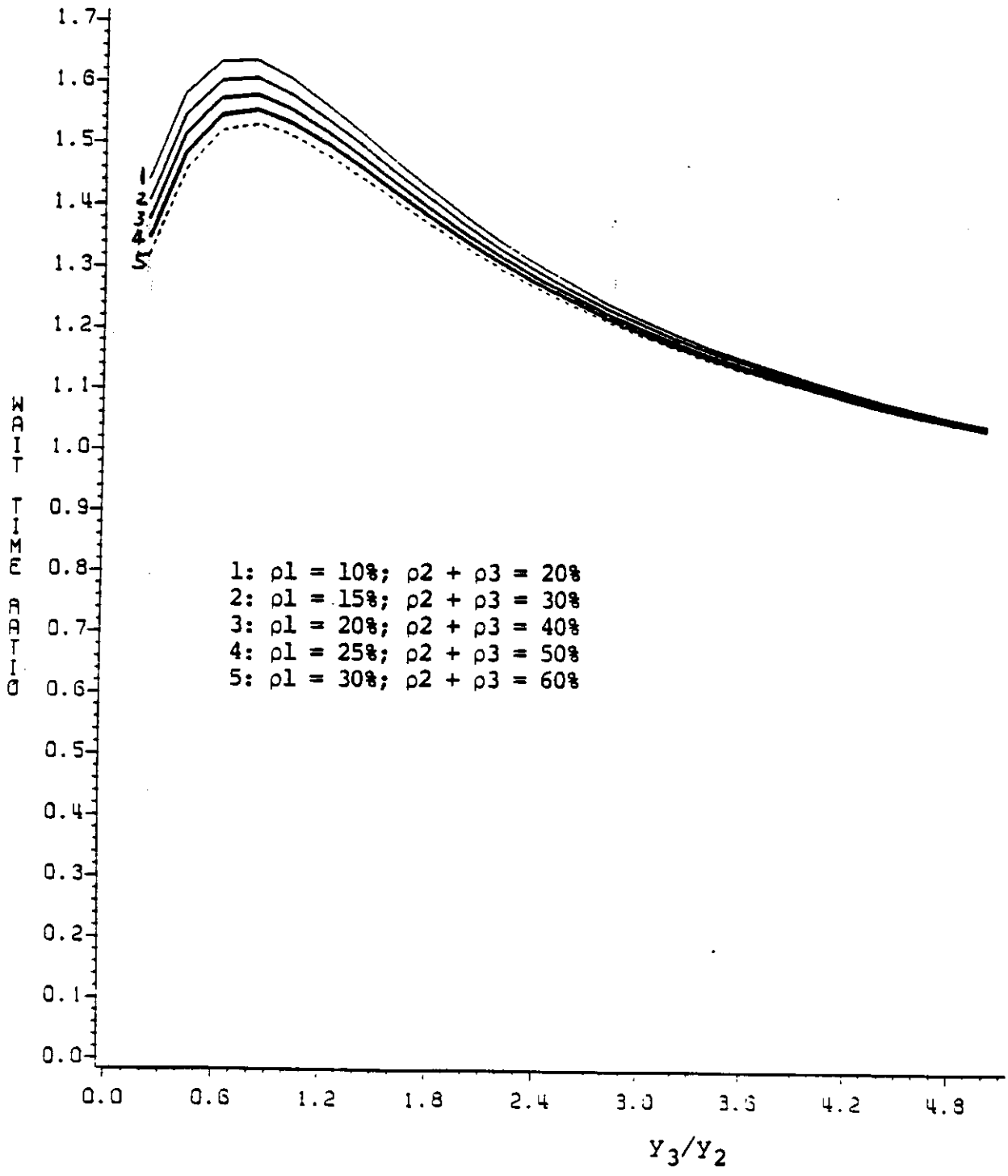


FIG. 18. WAIT-TIME RATIO AS A FUNCTION OF MODULE-SIZE RATIO Y_3/Y_2 (FOR VARIOUS PROCESSOR UTILIZATION) FOR THREADS WITH THREE CONSECUTIVE MODULES

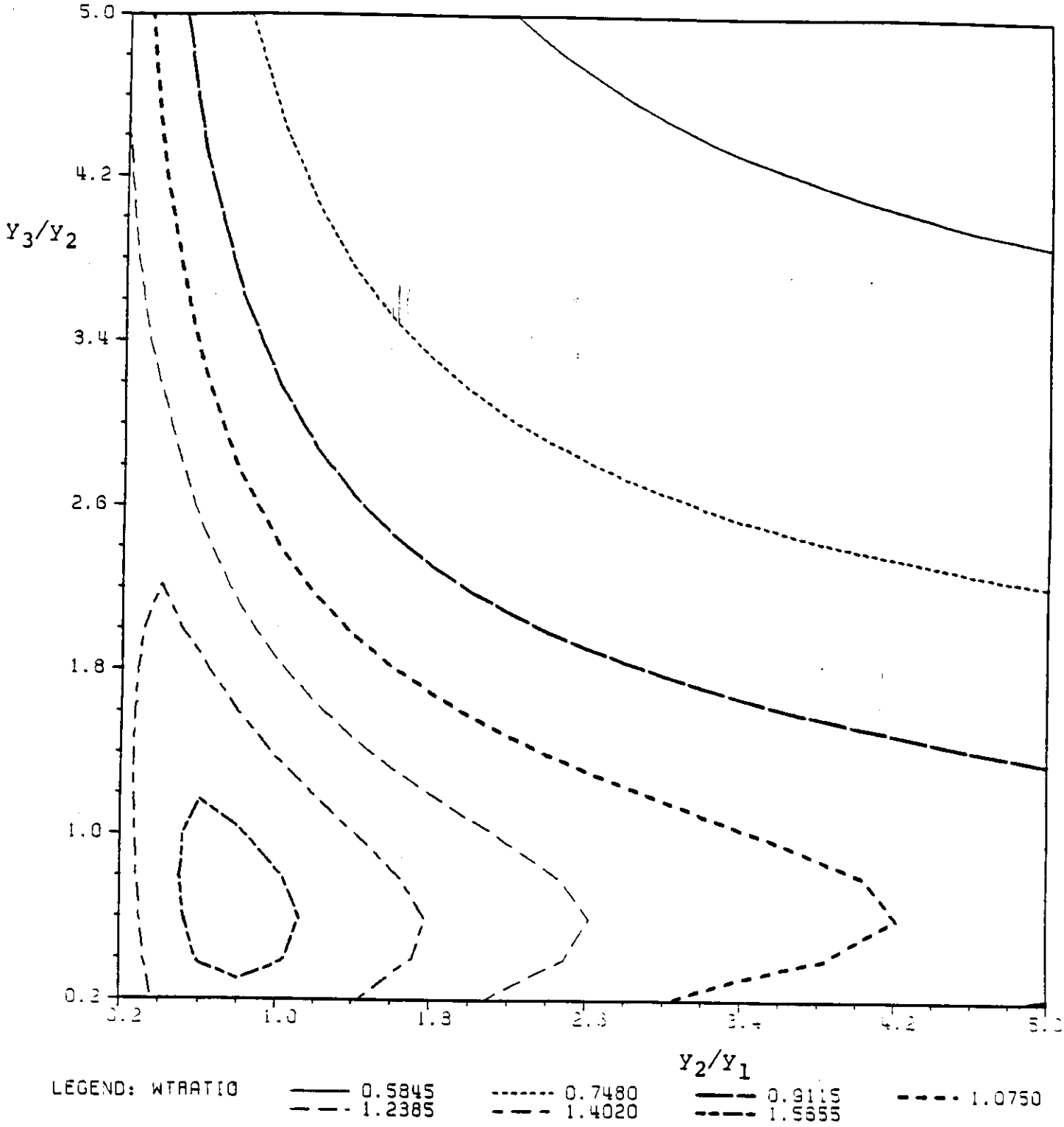


FIG. 19. 3-D CONTOUR PLOT FOR WAIT-TIME RATIO AS A FUNCTION OF MODULE-SIZE RATIOS y_2/y_1 AND y_3/y_2

5. MODULE ASSIGNMENT ALGORITHM CONSIDERING PR

5.1 Algorithm P-I-A

We shall now include the PR rules into our task-allocation algorithm. The decision on whether to group two consecutive modules should be based on the two possibly conflicting factors: IMC size and PR (i.e., module-size ratios). Therefore, *IMC index* and *PR index* are introduced. In Step 1.1 of Algorithm I-A, let us define the IMC index and PR index between modules M_i and M_j as follows:

$$\gamma_{IMC}(i,j) = \frac{IMC_{i,j}}{\theta_{IMC}} \qquad i = 1,...,J; \quad j = 1,...,J$$

$$\gamma_{PR}(i,j) = \frac{1 - R_w(r_{i,j})}{I_{PR}} \qquad i = 1,...,J; \quad j = 1,...,J$$

where R_w is a function of module-size ratio $r_{i,j}$ (see Fig. 16). Note that a R_w value on the Y-axis of Fig. 16 always lies in the range of [0, 2]. This value is translated into the PR index $\gamma_{PR}(i,j)$ — the condition $R_w < 1$ ($R_w > 1$) corresponds to a positive (negative) $\gamma_{PR}(i,j)$ which prescribes the grouping (separation) of modules M_i and M_j. For simplicity, we divide the range [0, 2] on the Y-axis into N_{PR} equal-size intervals for evaluating the PR index. The interval size is $I_{PR} = 2.0/N_{PR}$. At the break point between grouping or separating two consecutive modules, $R_w = 1$. Thus, the function $(1 - R_w)/I_{PR}$ gives the PR index $\gamma_{PR}(i,j)$ for any given module-size ratio $r_{i,j}$. For example, if we choose to have 20 PR levels within the range [0, 2], we have an interval size $I_{PR} = 2.0/20 = 0.1$. If $R_w = 1.4$, then $\gamma_{PR} = -4$, which is against the grouping of modules M_i and M_j. To introduce PR rules in our algorithm, we shall replace Step 1.3 of Algorithm I-A with the following:

1.3 If $\gamma_{IMC}(i,j) + \gamma_{PR}(i,j) \leq 0$

go to Step 1.2.

Let us denote this generalized algorithm as Algorithm P-I-A (adding the initial "P" for PR).

There exist three variables in Algorithm P-I-A — α, β, and N_{PR} (or, I_{PR}). For a given distributed system (e.g., the DPAD system), if N_{PR} is fixed, then all $\gamma_{PR}(i,j)$ values are uniquely determined. In that case, increasing the α value will reduce the IMC influence on module assignment (see new Steps 1.1 and 1.3), assuming a fixed β value. On the other hand, reducing the N_{PR} will reduce the PR influence. If we reduce N_{PR} by half and double the α value, then the minimum-bottleneck assignment generated by Algorithm P-I-A will remains unchanged because both $\gamma_{IMC}(i,j)$ and $\gamma_{PR}(i,j)$ are reduced by half, and thus the sum $\gamma_{IMC}(i,j) + \gamma_{PR}(i,j)$ does not changes its sign (positive or negative). Therefore, theoretically one of the two variables α and N_{PR} can be fixed. Table 3 contrasts γ_{PR}'s and γ_{IMC}'s (rounded to the nearest integers) for various values of α and N_{PR}. We summarize the heuristic task-allocation algorithm as follows:

```
Fix the number of PR intervals, N_PR;

Do α = α1% to α2%;
   Do β = β1% to β2%;
      Perform Algorithm P-I-A;
   end;
end;
```

The experimental results on DPAD and other systems reveal that using $N_{PR} = 20$ and α between 1% and 10% generates good assignments. A good range for β is between 60% and 120%. This is because too small a β would retard proper module grouping while too large a β makes it impossible to balance processor loads during Phase II.

5.2 PR Has No Effect on Example 1

Applying Algorithm P-I-A to the DPAD system produces the bottlenecks as shown in Fig. 20. Simulation reveals that the response-time performance of the assignment with a bottleneck of 74,985 MLI (generated by $\alpha = 3\%$ and $\beta = 60\%$) is slightly better than the one with a bottleneck of 74,312 MLI (for $\alpha = 4\%$ and $\beta = 70\%$). This shows that a smallest bottleneck does not necessarily yield the *best* response time. However, assignments with close

M_i From	M_j To	Number of PR intervals, N_{PR} 200	100	50	40	20	10	8	6	4	2
2	3	-32	-16	-8	-6	-3	-2	-1	-1	-1	0
5	7	-15	-8	-4	-3	-2	-1	-1	0	0	0
6	7	-15	-8	-4	-3	-2	-1	-1	0	0	0
8	9	-15	-8	-4	-3	-2	-1	-1	0	0	0
16	18	-15	-8	-4	-3	-2	-1	-1	0	0	0
17	18	-15	-8	-4	-3	-2	-1	-1	0	0	0
14	23	-37	-19	-9	-7	-4	-2	-1	-1	-1	0
20	21	-15	-8	-4	-3	-2	-1	-1	0	0	0

$\gamma_{PR}(i,j)$

M_i From	M_j To	α% 5	10	15	20	25	30	35	40
13	14	65	32	21	16	13	10	9	8
8	6	17	8	5	4	3	2	2	2
8	10	17	8	5	4	3	2	2	2
8	16	17	8	5	4	3	2	2	2
14	23	10	5	3	2	2	1	1	1
22	1	9	4	3	2	1	1	1	1
22	2	9	4	3	2	1	1	1	1
22	4	9	4	3	2	1	1	1	1
22	8	9	4	3	2	1	1	1	1
14	13	3	1	1	0	0	0	0	0
8	9	3	1	1	0	0	0	0	0
8	17	3	1	1	0	0	0	0	0
8	19	3	1	1	0	0	0	0	0
8	20	3	1	1	0	0	0	0	0
9	13	1	0	0	0	0	0	0	0
19	20	1	0	0	0	0	0	0	0
4	5	0	0	0	0	0	0	0	0
4	6	0	0	0	0	0	0	0	0
4	7	0	0	0	0	0	0	0	0
18	20	0	0	0	0	0	0	0	0
22	23	0	0	0	0	0	0	0	0
18	19	0	0	0	0	0	0	0	0
3	13	0	0	0	0	0	0	0	0
2	3	0	0	0	0	0	0	0	0
4	13	0	0	0	0	0	0	0	0
16	18	0	0	0	0	0	0	0	0
17	18	0	0	0	0	0	0	0	0
5	7	0	0	0	0	0	0	0	0
6	7	0	0	0	0	0	0	0	0
7	13	0	0	0	0	0	0	0	0
8	13	0	0	0	0	0	0	0	0
20	21	0	0	0	0	0	0	0	0
18	8	0	0	0	0	0	0	0	0
10	8	0	0	0	0	0	0	0	0
10	18	0	0	0	0	0	0	0	0

$\gamma_{IMC}(i,j)$

TABLE 3. COMPARE γ_{PR} AND γ_{IMC}

20 PR Levels

	1%	2%	3%	4%	5%	6%	7%	8%	9%	10%
60%	77739	77739	74985	75705	75705	75705	75705	75705	75705	75705
70%	79739	79739	76000	74312	74312	74312	74312	74312	74312	74312
80%	79739	79739	76000	74312	74312	74312	74312	74312	74312	74312
90%	80661	80661	76938	75938	75938	76938	76938	75938	76938	76938
100%	79739	79739	76000	74312	74312	74312	74312	74312	79491	79491
110%	79739	79739	76000	74312	74312	74312	74312	74312	79491	79491
120%	80661	80661	76938	75938	75938	75938	76938	75938	79491	79491

FIG. 20. MINIMUM BOTTLENECKS FOR DPAD GENERATED BY ALGORITHM P-I-A

bottleneck values always yield similar response times.

The above assignment with a bottleneck of 74,985 MLI performs only slightly better than the assignment generated by Algorithm I-A (Fig. 21). In fact, using the same parameters $\alpha = 5\%$ and $\beta = 75\%$ as were used in Algorithm I-A in Section 3.2.2, Algorithm P-I-A will generate *the same* assignment as Algorithm I-A. This can be seen by considering the column of γ_{PR} for $N_{PR} = 20$ in Table 3, and the column of γ_{IMC} for $\alpha = 5\%$. Those pairs of modules recommended to be grouped by Algorithm I-A (i.e., $IMC_{i,j} > \theta_{IMC}$) are also recommended by Algorithm P-I-A (i.e., $\gamma_{IMC}(i,j) + \gamma_{PR}(i,j) > 0$). Further, those module pairs not recommended to be grouped by Algorithm I-A (i.e., $IMC_{i,j} \leq \theta_{IMC}$) are also not recommended by Algorithm P-I-A (i.e., $\gamma_{IMC}(i,j) + \gamma_{PR}(i,j) \leq 0$). For example, $IMC_{14,23}$ = 5,019 MLIs > θ_{IMC} = 446.7 (see Section 3.2.2) and thus Algorithm I-A recommends grouping M_{14} and M_{23}. On the other hand, $\gamma_{IMC}(14,23) + \gamma_{PR}(14,23) = 10 + (-4) = 6 > 0$ and therefore, Algorithm P-I-A also recommends grouping M_{14} and M_{23}.

5.3 Example 2: PR Has Effect on Module Assignment and Response Time

An example is given in this section to show that a significant response-time improvement can be achieved when PR is considered in task allocation. Consider the control-flow graph shown in Fig. 22 where each program module has a deterministic execution time of either 100 or 1,000 μs. Thus the size ratio of each pair of consecutive modules is either 0.1 or 10 (with four exceptional pairs whose size ratios are 1.0). According to the PR rules derived in Section 4, we should assign M_4 and M_6 to the same processor, and M_9 on a different processor. Using the model of [CHU84b], we can estimate the AET for a specified time interval for each module. In this example let us assume a time interval of 100 job arrivals, the inter-arrival time is exponentially distributed, and *each* arrival invokes the entire control-flow graph *once*. The estimated AET's are shown in column 2 of Table 4. Let us further assume that the IMC sizes for all communicating module pairs are about equal, either 1,400 or 1,500 μs as shown in Table

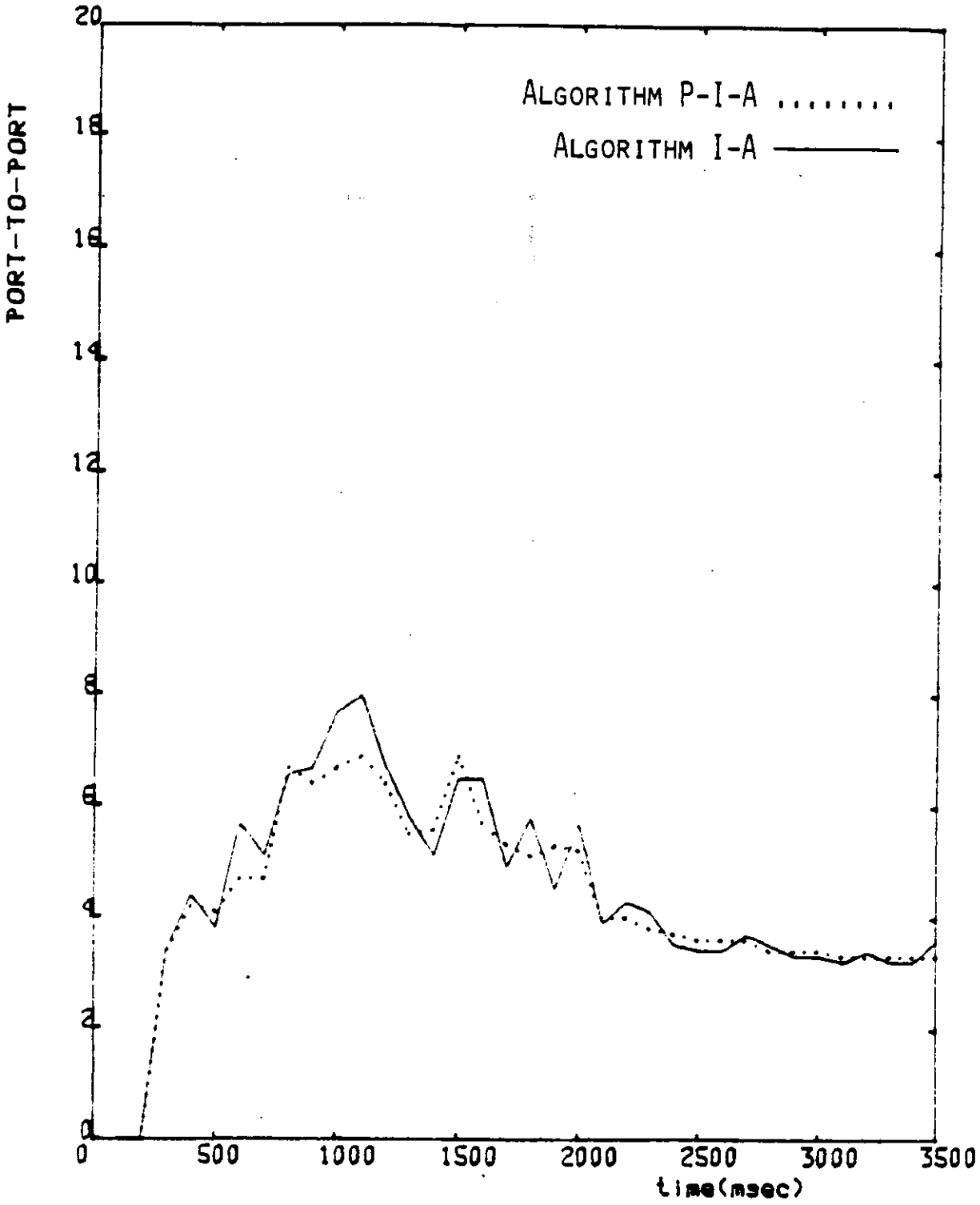


FIG. 21. PTP TIME FOR PRECISION-TRACK THREAD --- COMPARE ASSIGNMENTS WITH AND WITHOUT PR

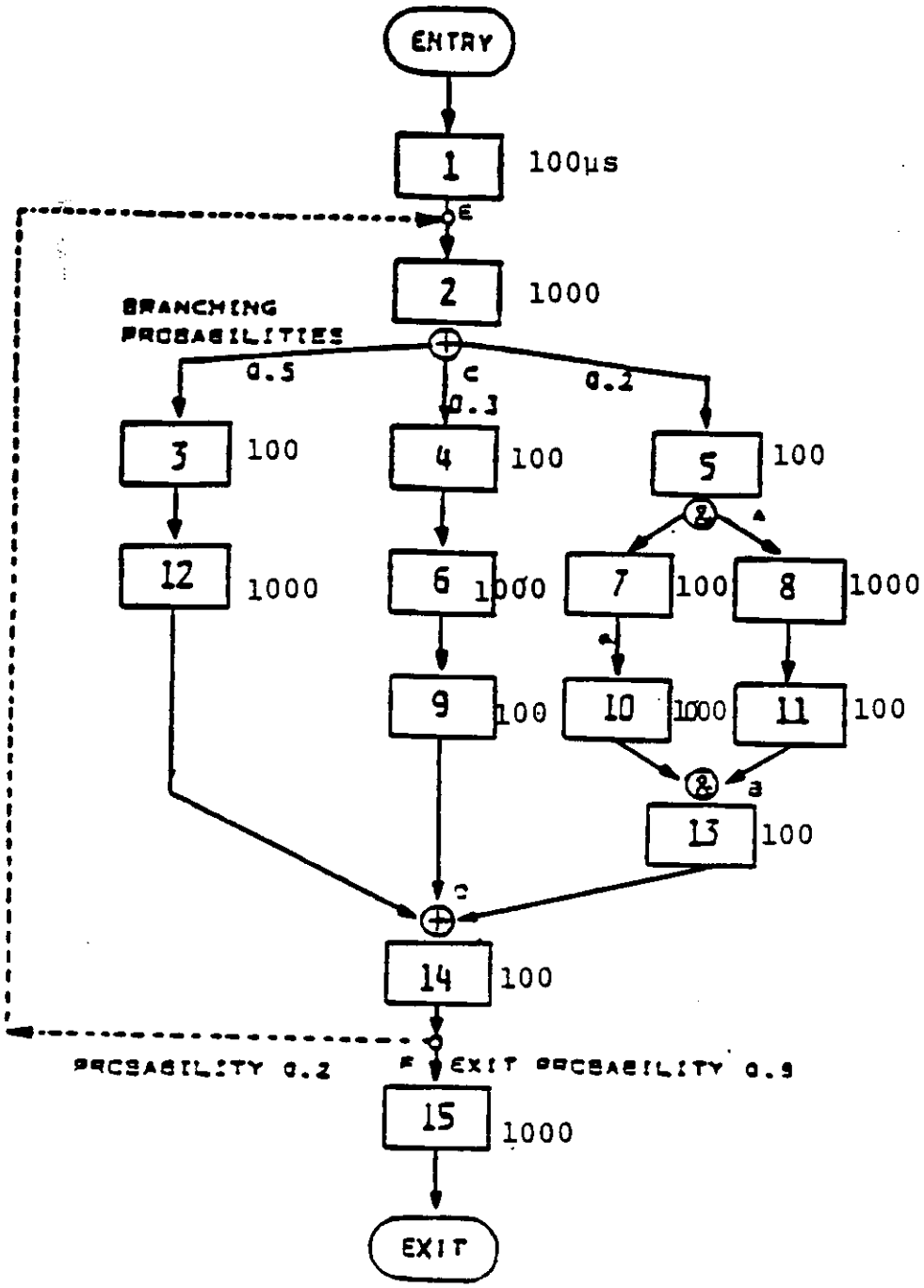


FIG. 22. A SAMPLE TASK CONTROL-FLOW GRAPH FOR EXAMPLE 2

Write Module	AET* (in μs)	File Updated	IMC* (in μs)	Read Modules
1	10,000	101	1400	2
2	125,000	102	1400	3,4,5
3	6,250	103	1400	12
4	3,750	104	1400	6
5	2,500	107	1400	7,8
6	37,500	106	1500	9
7	2,500	108	1400	10
8	25,000	109	1500	11
9	3,750	110	1400	14
10	25,000	111	1500	13
11	2,500	112	1500	13
12	62,500	105	1500	14
13	2,500	113	1400	14
14	12,500	114	1400	15
15	100,000	--	--	--

* AET and Total IMC during a 100-arrival period

TABLE 4. AET AND IMC FOR EXAMPLE 2

4 and Fig. 23, so that the IMC plays a much less important role than PR. Given these PR, IMC, and AET, the module assignments generated by Algorithms I-A and P-I-A are shown in Fig. 24. Both assignments yield fairly balanced processor loads with similar bottleneck values. Therefore, if they differ significantly in response time, it is due to the PR. Note that for the assignment generated by Algorithms P-I-A, most module pairs are assigned (either co-located or separated) according to our PR rules instead of by IMC size. For example, the module size ratio $r_{4,6}$ is $y_6/y_4 = 10$; thus, M_4 and M_6 are co-located on Processor 3. On the other hand, $r_{6,9} = 0.1$; thus, M_6 is separated from M_9 although $IMC_{6,9}$ is larger than $IMC_{4,6}$.

These two assignments are simulated via the PAWS simulator. The average response time for each job arrival is measured from when the job arrives at the system until it finishes the execution of M_{15}. Fig. 25 portrays the response time for the two assignments. Note that Algorithm P-I-A yields better response time than that of Algorithm I-A, with 10.8% improvement at processor utilization $\rho = 20\%$ and 25.7% improvement at $\rho = 80\%$.

6. CONCLUSIONS AND DISCUSSIONS

The three important parameters in task allocation are accumulative execution time (AET) of each module, intermodule communication (IMC), and precedence relations (PR) among program modules. AET always contributes to processor load; its contribution is independent of task allocation. IMC is the communication between program modules through shared files. When a module on a computer writes to or reads from a shared file on *another* computer, it requires extra processing and communication overhead known as IPC (interprocessor communication). Therefore, a task-allocation algorithm should try to minimize IPC by assigning a pair of heavily communicating modules to the same computer.

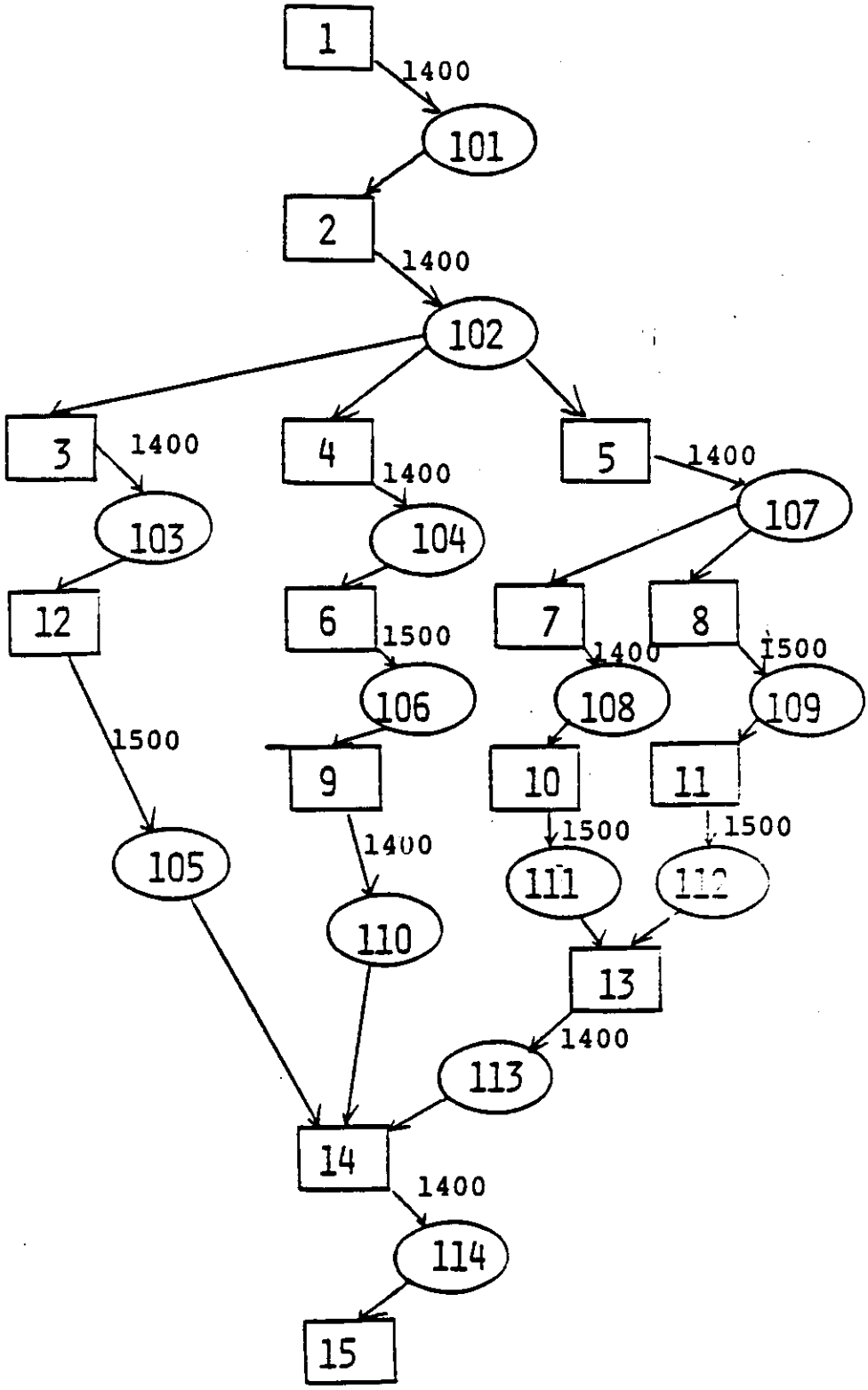


FIG. 23. CONTROL AND DATA-FLOW GRAPH FOR EXAMPLE 2

ASSIGNMENT #1 (W/O CONSIDERING PR)			ASSIGNMENT #2 (CONSIDERING PR)		
CPU1	CPU2	CPU3	CPU1	CPU2	CPU3
1	7	3	1	6	3
2	10	4	2	9	5
9	13	5	4	15	7
	14	6			8
	15	8			10
		11			11
		12			12
					13
					14

FIG. 24. MODULE ASSIGNMENTS FOR EXAMPLE 2

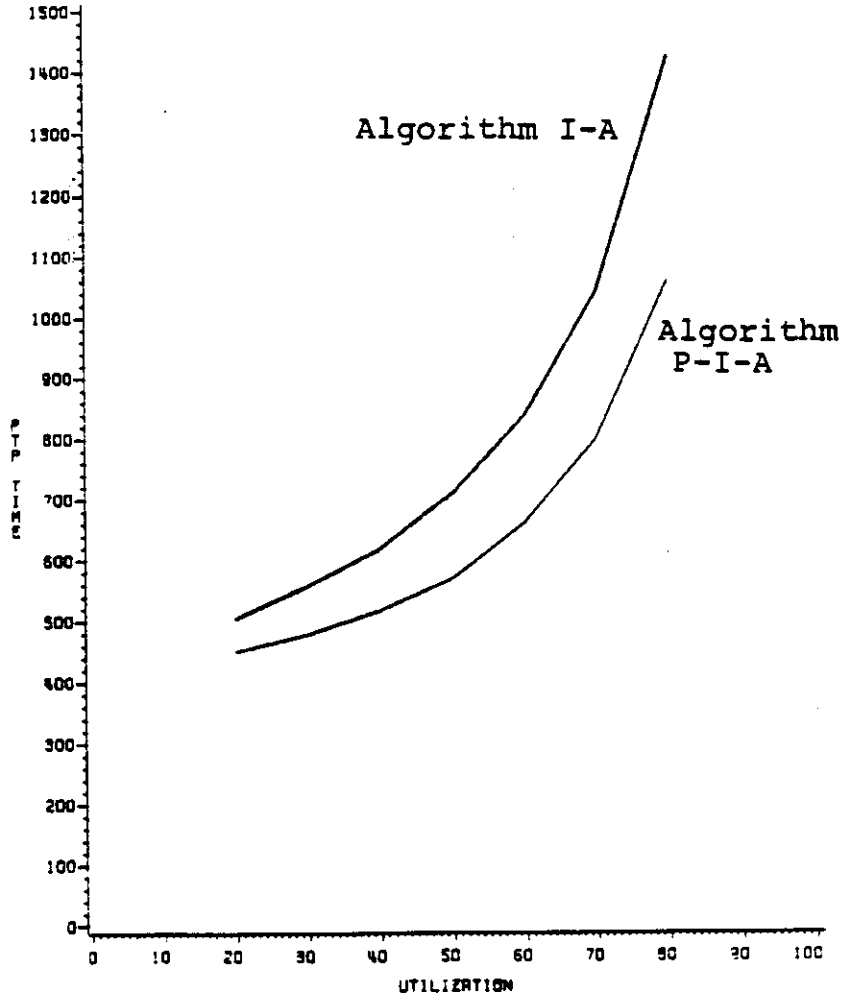


FIG. 25. TASK RESPONSE TIME COMPARISON FOR EXAMPLE 2 FOR ASSIGNMENTS WITH AND WITHOUT PR CONSIDERATION

In this paper, an objective function for the minimization of the bottleneck processor load (based on IMC and AET) is proposed for task allocation. An algorithm is developed, based on this objective function, for grouping module pairs to reduce both the IPC and the search space for good modules assignments. This algorithm is able to generate assignments comparable with that from the exhaustive search.

The third parameter for task allocation is the *precedence relationship (PR)* in which a program module can not be enabled before all its predecessor(s) finish execution. Simulation study and analysis revealed that the module-size ratio of two consecutive modules affects task response time. Two simple rules are: 1) Assigning two consecutive modules to a same processor yields good response times *if the execution time of the second module is much larger than that of the first module;* 2) If the second module is much smaller than the first one, it is better to separate two consecutive modules and assign them on two distinct processors. Allocating the modules according to IMC values and the PR rules yields performance improvement.

A heuristic algorithm that considers PR, IMC, and AET was developed for task allocation. The algorithm was applied to two example systems. The results revealed that module assignments considering PR may yield better response time than assignments without PR consideration.

6.1 Future Research Areas

Many related issues in task allocation still need further investigation.

a. Replication of files — A file-replication policy should be developed to decide how many copies of a replicated file are needed and where these copies should reside, for either access speed, fault-tolerance, or reduction of file-update message volume. Data consistency among the copies is a major concern that affects performance.

b. Replication of program modules — Some modules might be so frequently invoked that their processing requirement cannot be met by a single processor. It is desirable to assign identical copies of such a module on multiple computers with each processing a subset of invocations for that module. Techniques need to be developed to decide a) the needed number of copies for a program module, b) the file structure (centralized, replicated, or partitioned [CHU76]) for the files accessed by a replicated program module, c) the number of copies (and the sites) a file should be replicated and/or partitioned into, and d) the policy for distributing module invocations among all computers which run a copy of the invoked module.

c. Task scheduling policy — Scheduling policy plays an important role in real-time systems. Besides the FCFS discipline, there might be other scheduling policies more suitable for distributed real-time systems. One approach is to schedule multiple invocations of a module in a group and process them as a batch. As a result, some of the operating overhead (e.g., the initializing housekeeping code) can be shared by all invocations in the batch. Of course, this reduced overhead should be weighed against the increased overhead in task scheduling.

d. Branching probability vs. precedence relation — If the branching probability between two consecutive modules is very small, the effect of the PR on task response time will be small because the arrival processes of these two modules can be treated as independent from each other [CHU84c]. In this paper, a branching-probability cutoff point of 0.5 was arbitrarily chosen to determine whether the PR between two consecutive modules should be considered, or ignored. More studies need to be performed to decide on a better cutoff point for the branching probability.

REFERENCES

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.

BOKH79 S. H. Bokhari, "Dual processor scheduling with dynamic reassignment," *IEEE Trans. on Software Eng.*, vol. SE-5, no. 4, pp. 341-349, July 1979.

BOKH81 S. H. Bokhari, "On the mapping problem," *IEEE Trans. on Computers*, vol. C-30, no. 3, pp. 207-214, Mar. 1981.

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.

CHOW79 Y. C. Chow and W. H. Kohler, "Models for dynamic load balancing in a heterogeneous multiple processor system," *IEEE Trans. on Computers*, vol. C-28, no. 5, pp. 354-361, May 1979.

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.

CHU76 W. W. Chu, "Performance of file directory systems for distributed data bases," in *Proc. AFIPS National Computer Conf.*, vol. 45, pp. 577-587, 1976.

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.

CHU84a W. W. Chu, J. Hellerstein, M-T. Lan, J. M. An, and K. K. Leung, "Database management algorithms for advanced BMD applications," Dept. Computer Science, Report # UCLA-ENG-84-07 (CSD-840031), Univ. of California, Los Angeles, Apr. 1984.

CHU84b 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.

CHU84c W. W. Chu and K. K. Leung, "Task-response-time model & its applications for real-time distributed processing systems," in *Proc. 5th Real-Time Systems Symposium*, Austin, TX, Dec. 1984.

EFE82 K. Efe, "Heuristic models of task assignment scheduling in distributed systems," *Computer*, vol. 15, no. 6, pp. 50-56, June 1982.

GENT78 W. M. Gentlemen, "Some complexity results for matrix computations on parallel processors," *J. of ACM,* Jan. 1978, pp. 112-115.

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.

GYLY76 V. B. Gylys and J. A Edwards, "Optimal partitioning of workload for distributed systems," in *Proc. COMPCON Fall 76,* Sep. 1976, pp. 353-357.

HOFF80 R. H. Hoffman, R. W. Smith, and J. T. Ellis, "Simulation software development for the BMDATC DDP underlay experiment," in *Proc. 4th Intl. Computer Software and Applications Conf. (COMPSAC),* Oct. 1980, Chicago, pp. 569-577.

HOLL82 L. J. Holloway, "Task assignment in a resource limited distributed processing environment," Ph.D dissertation, Dept. Computer Science, Univ. of California, Los Angeles, 1982.

IRAN82 K. B. Irani and K-W. Chen, "Minimization of interprocessor communication for parallel computation," *IEEE Trans. on Computers,* vol. C-31, no. 11, pp. 1067-1075, Nov. 1982.

JENN77 C. J. Jenny, "Process partitioning in distributed systems," in *Proc. NTC 1977,* pp. 31:1-1 — 31:1-10.

LAN85 L. M-T. Lan, "Characterization of intermodule communications and heuristic task allocation for distributed real-time systems," Ph.D dissertation, Report No. CSD-850012, Univ. of California, Los Angeles, Mar. 1985.

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.

PRIC81 C. C. Price, "The assignment of computational tasks among processors in a distributed system," in *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.

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, Mar. 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.

STON78a H. S. Stone, "Critical load factors in two-processor distributed systems," *IEEE Trans. on Software Eng.*, vol. SE-4, no. 3, pp. 254-258, May 1978.

STON78b H. S. Stone, and S. H. Bokhari, "Control of distributed processes," *Computer*, vol. 11, no. 7, pp. 97-106, July 1978.

CHAPTER IV

FAULT TOLERANT LOCKING FOR TIGHTLY COUPLED SYSTEMS

FAULT TOLERANT LOCKING FOR TIGHTLY COUPLED SYSTEMS

1. INTRODUCTION

In a tightly coupled system, multiple copies of shared files are maintained in different shared memory modules to meet high survivability. Data updates should be applied to all file copies to keep *mutual consistency* among the copies. However, if a processor fails during an update process, some file copies may have been updated while others have not, resulting in mutual inconsistency. To recover from this type of failures, the *Fault Tolerant Locking (FTL)* protocol [1] is introduced on top of the conventional consistency-control protocols. FTL detects a processor failure, identifies and recovers inconsistent file copies, and releases the file lock so that other processors may lock and use the file again. FTL also prevents processors from reading and updating out-of-date or inconsistent file copies in the failure of shared memory modules and/or paths.

We shall first describe the FTL principal, its implementation and operations. Then we present the time relationship about the lock-holding time. Next, we discuss the FTL experimental results from SDC testbed which provide information about the performance and characteristics of FTL. Areas for further research and experimentation are identified.

2. PRINCIPAL OF FTL

In order to provide the status of a file, a word that indicates the current state (free, locked, update-initiated, or failed) is appended to each file copy in the shared memory modules. File copies are updated, one at a time. In this manner, if a processor fails during a file update, we can tell which copies of the file have been completely updated, which particular copy is *partially* updated, and which copies have not been updated at all. Other processors that attempt to lock this file in problem would detect the processor failure by a time-out

mechanism. Based on the status of all file copies, the inconsistent copy can be detected and recovered from any consistent copy.

To prohibit further accesses to failed copies, each processor maintains a file copy *status table* in its local memory. When a processor experiences a memory or path failure while accessing a file copy, it marks the failure on the local status table.

3. FTL OPERATIONS

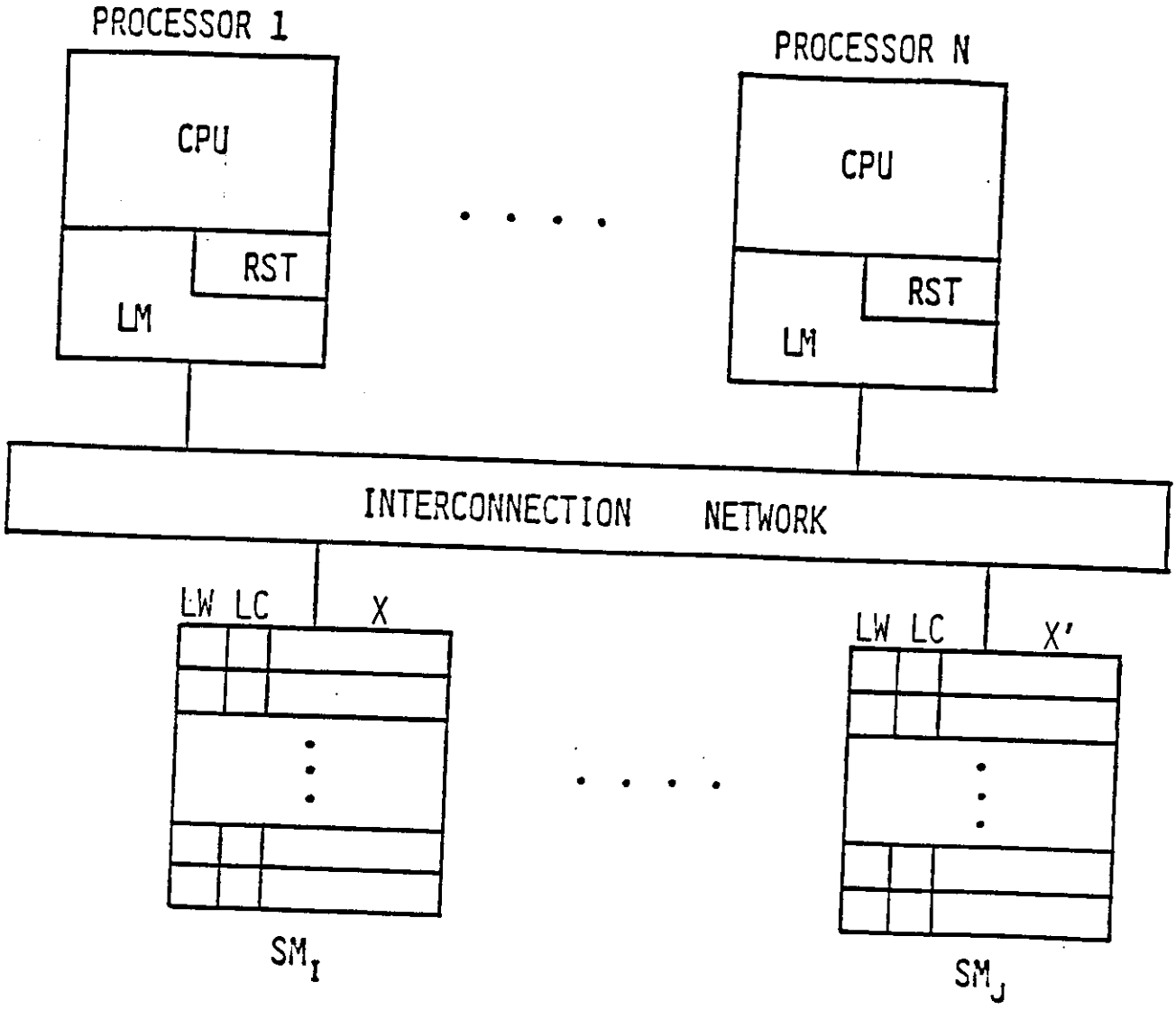
3.1 Implementation

To implement the FTL in a tightly coupled system for the BMD application, shared records [1] are duplicated in different shared memory modules (Fig. 1). Each record copy has a *Lock Word (LW)* that indicates one of the four possible states of the copy: free, locked, update-initiated, or failed (Fig. 2). To simplify our discussion, we assume each file has two copies: the *primary* copy and the *shadow* copy. Each processor maintains the *Record Status Table* (RST) in its local memory which indicates the status (good or failed) of each record copy in the shared memory modules.

Before accessing a record copy, a processor first checks the RST to determine if the copy is good. Then it reads the LW of the record copy. If the LW of the copy indicates 'failed', the processor marks "failed" on the RST and tries the other copy (this will be discussed in section 6). If the requested copy is being locked or update-initiated (by some other processor), the processor repeatedly checks the LW until the copy becomes free, or until a predetermined *time-out* period elapses. When the processor finds the copy is free, it locks the copy and does the same process for the second copy. Then it prepares updates in its local memory. When all updates to the record are ready, the processor marks 'update initiated' on

[1] For the BMD system, *records* rather than *files* are used as a unit of data items for locking and recovery.

FIGURE 1. A TIGHTLY COUPLED DISTRIBUTED SYSTEM WITH FTL



LM: LOCAL MEMORY

SM: SHARED MEMORY

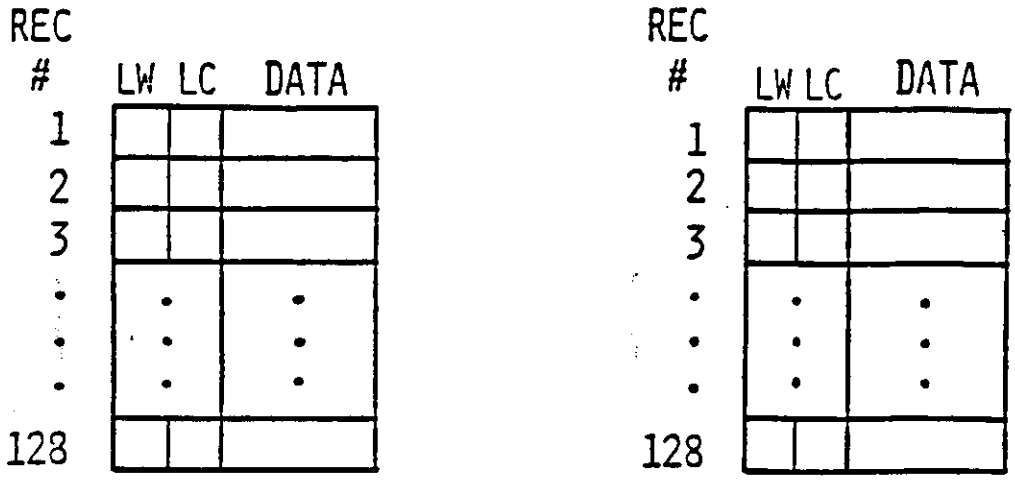
X': SHADOW COPY OF RECORD X

LW: LOCK WORD (FOUR POSSIBLE STATES)

LC: LOCK COUNTER

RST: RECORD STATUS TABLE

FIGURE 2. DUPLICATED RECORDS AND RECORD STATUS TABLE

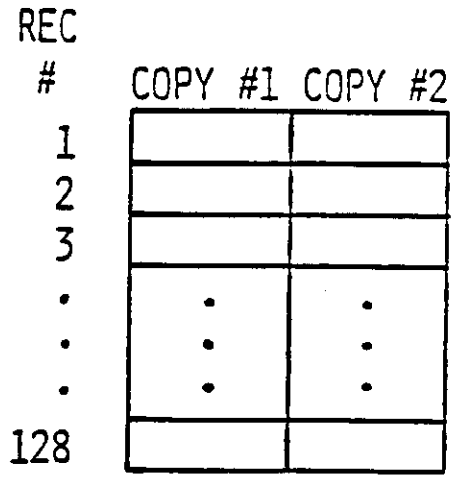


LW = LOCK WORD

LC = LOCK COUNTER

Each lock word indicates one of four states:
Free, Locked, Update-initiated, or *Failed.*

(a) Duplicated Records In Shared Memory Modules



Each entry indicates one of two states: *Good* or *Failed.*

(b) Record Status Table (RST) In Each Processor

the LW of the first copy and performs the update. After completing the update onto the second record copy in the same manner, the processor releases the lock for both copies. In normal operations with no failure, if there is no lock contention, the FTL update procedure can be shown in Figs. 3 and 4.

3.2 Detection of Processor Failure

When two or more processors request to lock the same record simultaneously, only one of them will obtain the lock grant. Other processors might experience time-out and initiate the recovery process undesirably. To prevent this from occurring, *Lock Counter (LC)* is introduced for each record copy. After a processor successfully locks a record copy, the LC of the copy is incremented by one. When a record copy is currently locked, a processor trying to lock the same copy will repeatedly request to lock the copy until it succeeds. When the processor finds that the LC of the requested record copy has been incremented while waiting for a lock grant (this implies that the record has been released by its holding processor and locked by some processor again), the processor resets the *time-out counter* and continues requesting for the lock-grant (Fig. 5). If the LC remains unchanged after a predetermined number of lock requests (i.e., time-out period), the processor currently holding the lock is considered failed (Fig. 6). The processor that detects the time-out then increments the LC of the record copy by one. This prevents other processors from detecting the same failure. To prevent *false* failure detection, the time-out period (determined by the number of repeated lock requests) must be larger than the lock-holding time of any application program.

3.3 Technique for Reducing the Time-Out Period for Performance Improvement

To have quick failure detection, the time-out period should be short. However, a short time-out period may cause false failure detections. To avoid this, the processor that holds the lock can increase the LC periodically should it hold a lock longer than the pre-specified time-

FIGURE 3. FTL PROCEDURE FOR RECORD UPDATE

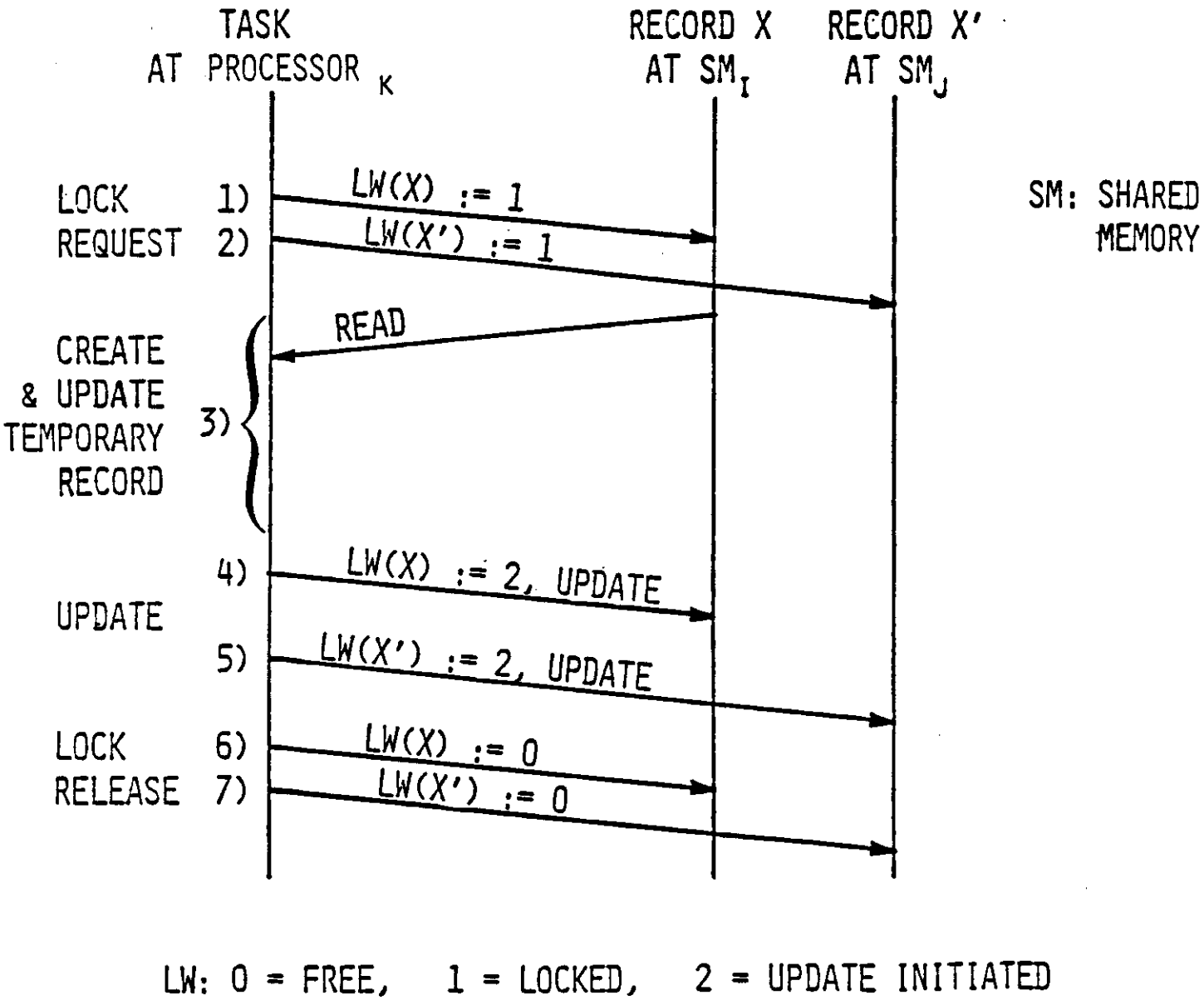


FIGURE 4. FTL PROTOCOL FOR RECORD UPDATE

1) Lock Record X (LW(X): = 1)

2) Lock Record X'(LW(X'): = 1)

3) Create and update temporary record in local memory

4) Mark lock word (LW(X): = 2) and update record X

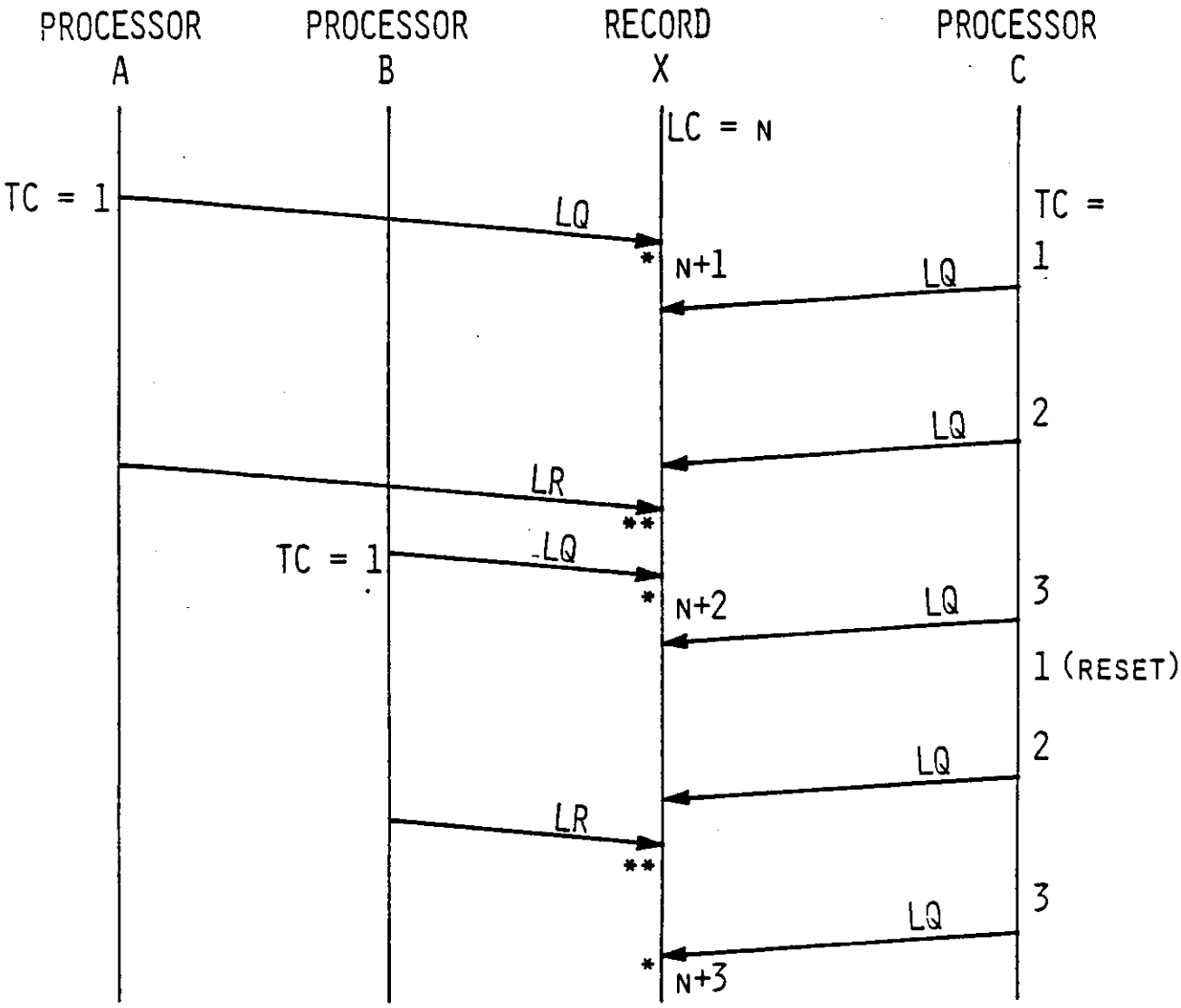
5) Mark lock word (LW(X'): = 2) and update record X'

6) Unlock record X (LW(X): = 0)

7) Unlock record X' (LW(X'): = 0)

LW: 0 = Free, 1 = Locked, 2 = Update-initiated

FIGURE 5. EXAMPLE OF THE FTL TIME-OUT MECHANISM: NO-FAILURE CASE



LC: LOCK-COUNTER

LQ: LOCK-REQUEST

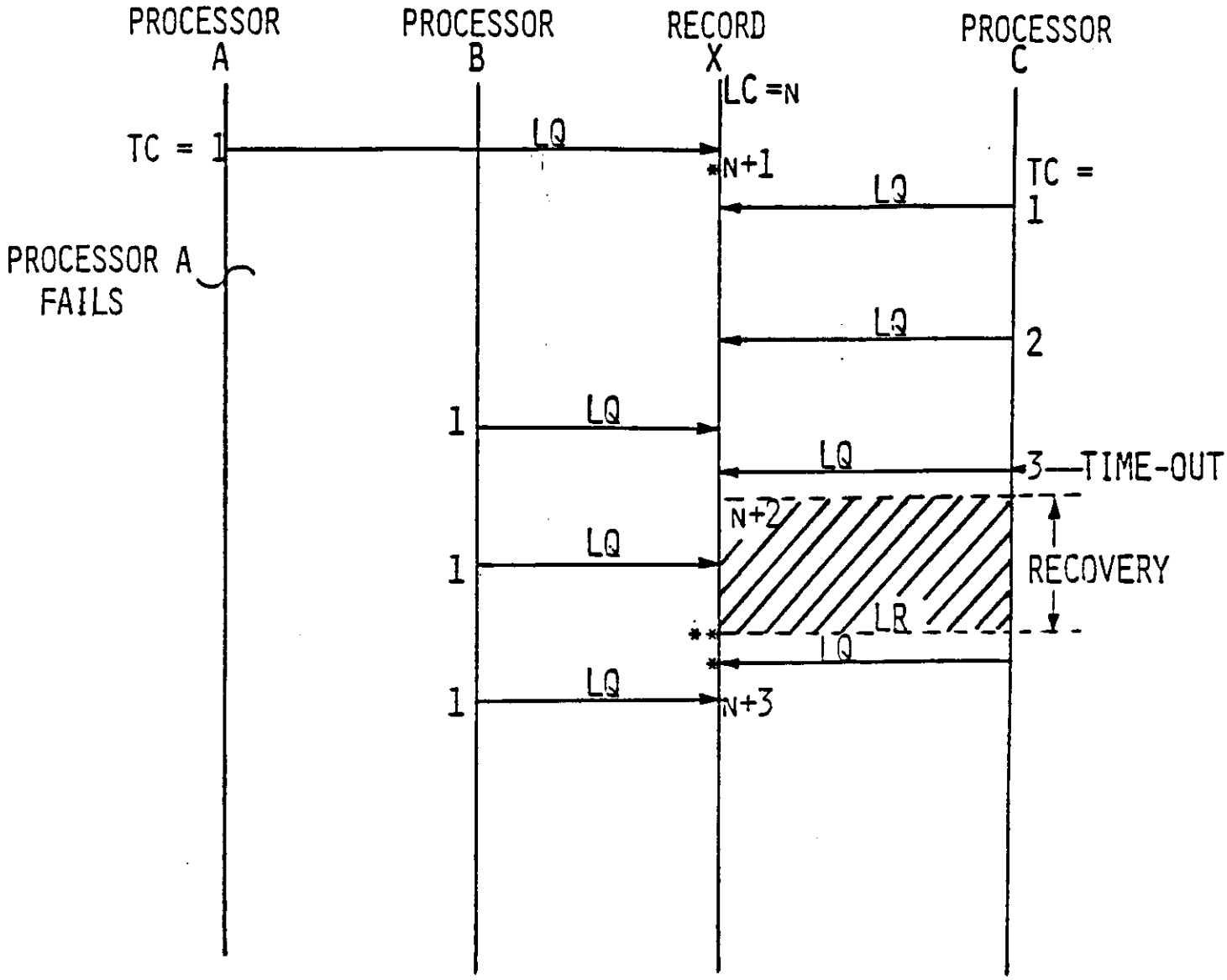
TC: TIME-OUT-COUNTER
TC(MAX) = 3

LR: LOCK-RELEASE

*: RECORD IS LOCKED

**: RECORD IS UNLOCKED

FIGURE 6. EXAMPLE OF THE FTL TIME-OUT MECHANISM: FAILURE CASE



LC: LOCK-COUNT

TC: TIME-OUT COUNTER
TC(MAX) = 3

LQ: LOCK-REQUEST

LR: LOCK-RELEASE

*: RECORD IS LOCKED

**: RECORD IS UNLOCKED

out period. This will prevent other processors from generating undesirable time-out signals.

3.4 Recovery from a Processor Failure

The processor reads the LWs of all copies of the requested record when it detects a time-out on a record. Based on the FTL status table as shown in Fig. 7, the processor takes the appropriate recovery action: either discarding the inconsistent record copy and operating in a degraded mode, or copying from the consistent record copy into the inconsistent one.

4. RECORD LOCK-HOLDING TIME

In this section, we present the record lock-holding time as a function of various system parameters. This may shed light on the interrelationship among the number of retries, the retry period for lock-request, FTL overhead, and response time.

The *lock-holding time*, T_L (diagramed in Figs. 8 and 9), is the sum of the time for locking record copy X', T_1, the execution time for an application process (including read and update of record copies), T_A, and the time for releasing record copy X, T_2. That is,

$$T_L = T_1 + T_A + T_2$$

Since the FTL requires reading a record copy (before copying the record into the local memory) and updating the primary and shadow copies, there are three memory accesses for each record update. Thus,

$$N_m = 3\ N_r$$

where

N_r = Record size in words,

N_m = Number of memory cycles for accessing a record

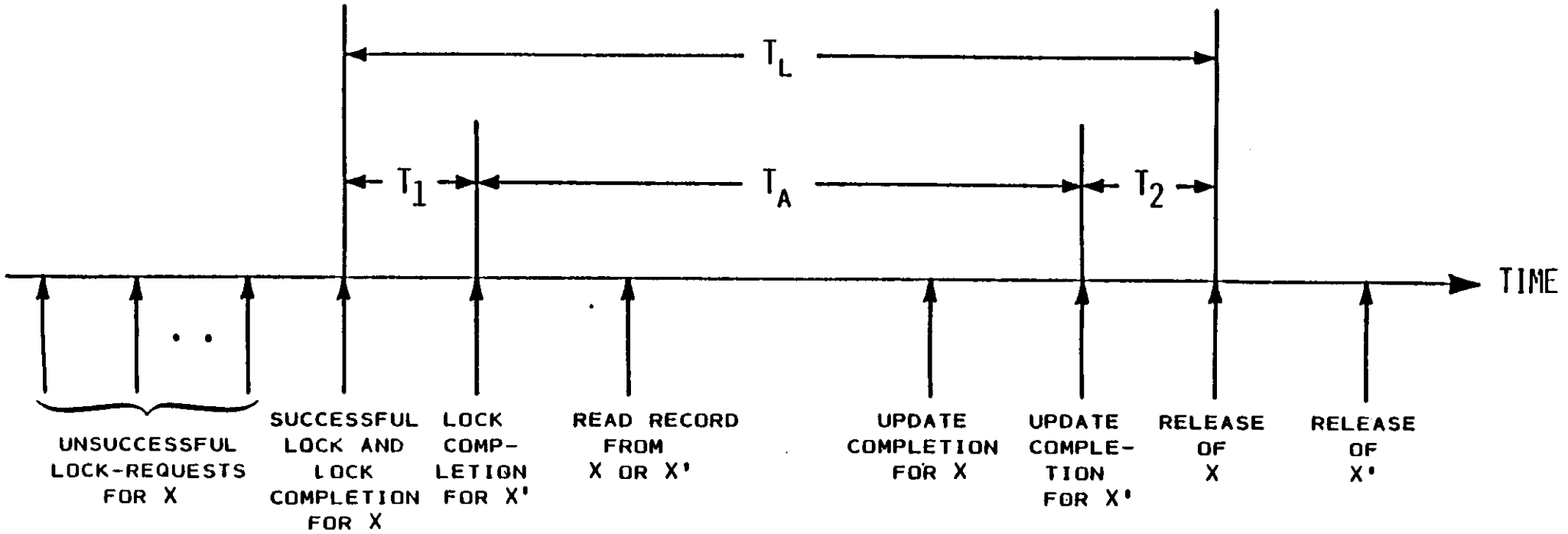
The execution time for an application process, T_A, is the sum of the execution time of that

FIGURE 7. FTL RECOVERY PROCEDURES FOR A PROCESSOR FAILURE

LOCK STATUS TABLE

LW (X)	LW (X')	RECOVERY REQUIRED?	INCONSISTENT RECORD COPY	UPDATE COMPLETED?
0	0	NO	N/A	NO
1	0	NO	N/A	NO
1	1	NO	N/A	NO
2	1	YES	X	NO
2	2	YES	X'	NO
0	2	NO	N/A	YES

FIGURE 8. FTL LOCK-HOLDING TIME FOR A RECORD



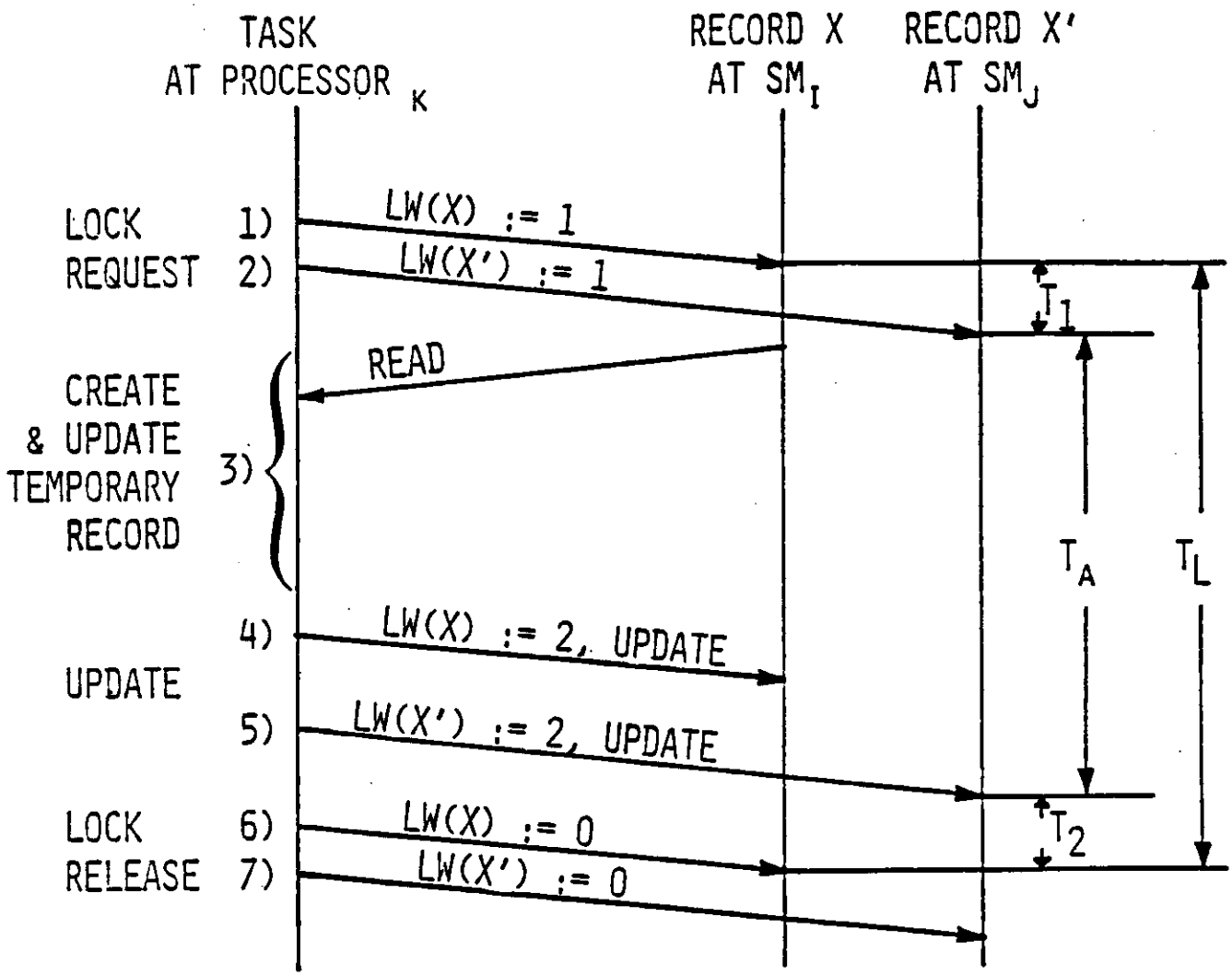
T_1: TIME FOR LOCKING RECORD COPY X'

T_2: TIME FOR RELEASING RECORD COPY X

T_A: EXECUTION TIME FOR AN APPLICATION PROCESS (INCLUDING READ AND UPDATE OF RECORD COPIES)

T_L: TOTAL LOCK-HOLDING TIME

FIGURE 9. TIMING DIAGRAM FOR FTL LOCK-HOLDING TIME



LW: 0 = FREE, 1 = LOCKED, 2 = UPDATE INITIATED

process without memory conflict and the additional delay due to memory conflict. Thus,

$$T_A = T_a + (1 + P_c)\, N_m = T_a + 3(1 + P_c)N_r$$

where

T_a = Execution time of a process without memory conflict, and

P_c = Probability of memory conflict for each memory cycle.

5. FTL EXPERIMENTAL RESULTS VIA THE SDC TESTBED

A set of experiments was performed for evaluating the feasibility of the FTL protocol via the BMDADC testbed. Three experiments characterize the FTL protocol under no-processor-failure situation, in terms of 1) overhead of the FTL protocol, 2) choice of lock-request retry period, and 3) choice of time-out period for processor failure detection. Another experiment studies the FTL protocol recovery time in the presence of processor failures. These experiments were performed at SDC [2,3] and we shall discuss the implications of the results.

5.1 Overhead of the FTL Protocol

For the management of distributed databases, a consistency-control protocol enhanced with the FTL protocol requires more processing than a baseline system without fault-tolerance. Fig. 10 compares the measured processor utilization for both systems. The system with the FTL protocol uses more processor resources since it requires additional lock and update to the second copy of each record. As a result, the lock-holding time of the FTL system is longer than the baseline system. Because of larger processor utilization, port-to-port times are also increased for the FTL system. Fig. 11 compares the port-to-port time for the Track Thread of the two systems. The experiment with 711 threat detections reveal that the load differences between the two systems for Track Initiate and Track are rather small and the FTL system still satisfies the port-to-port time requirements [2].

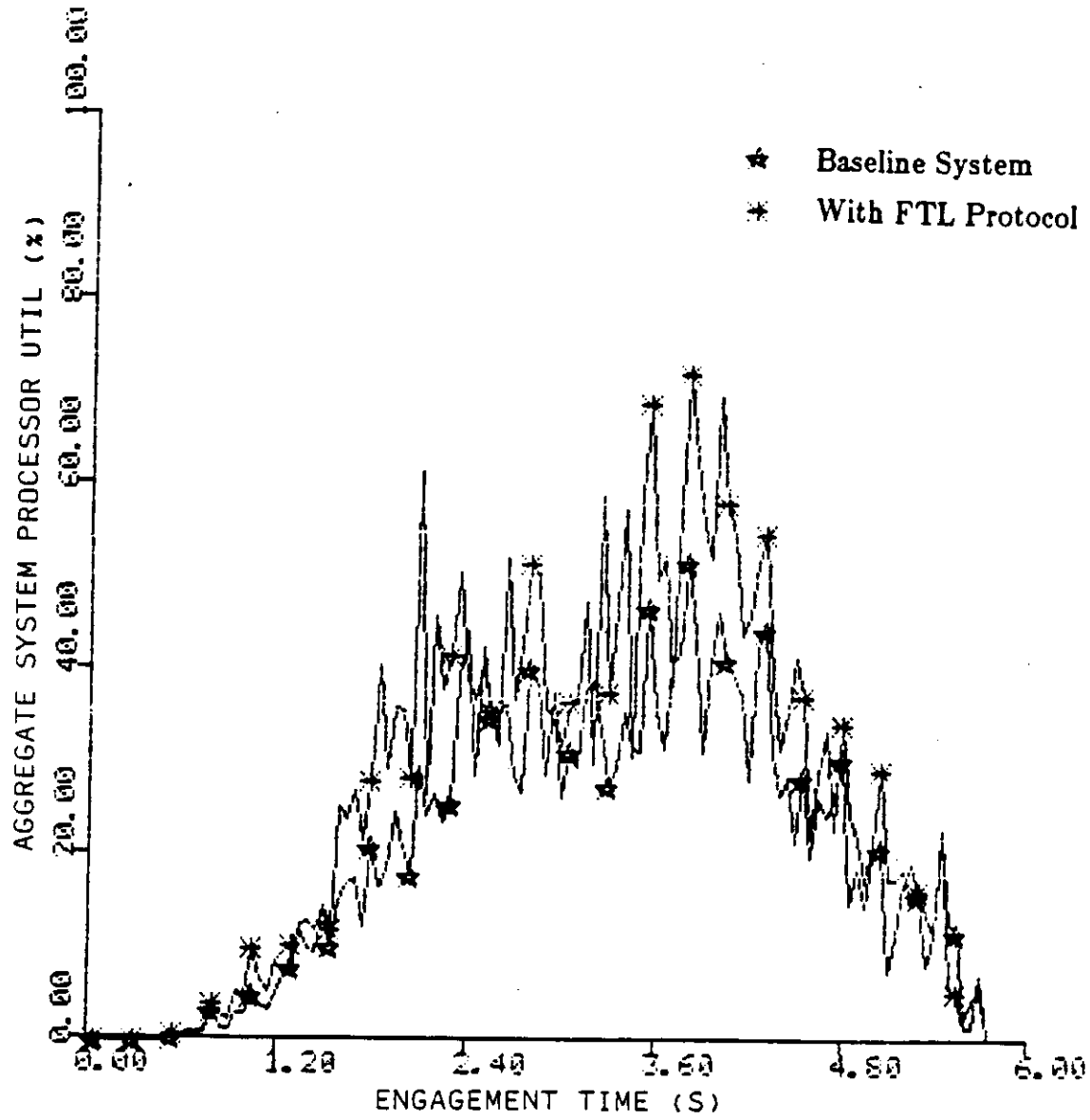


FIGURE 10. AGGREGATE SYSTEM PROCESSOR UTILIZATION

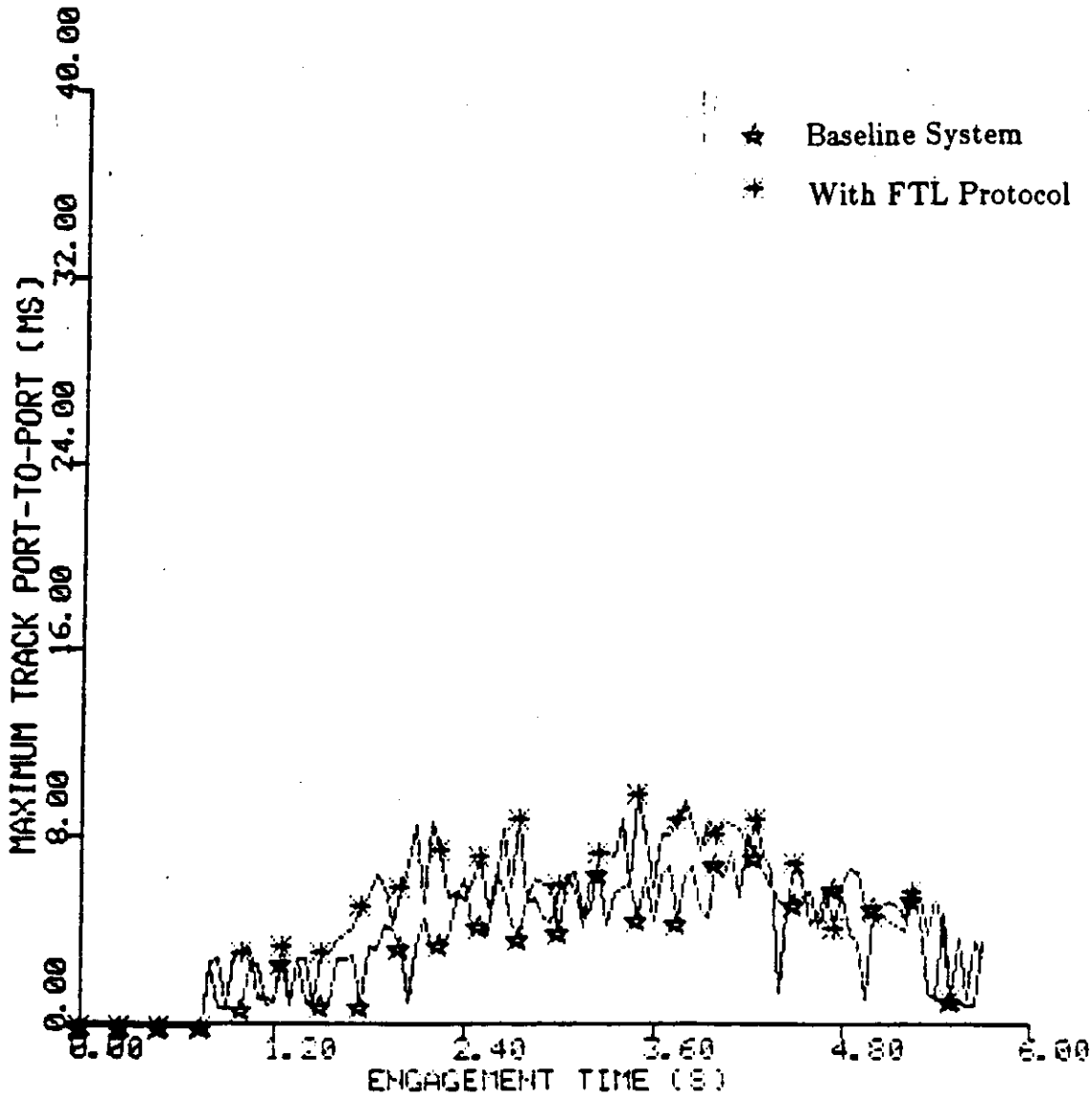


FIGURE 11. MAXIMUM PORT-TO-PORT TIME FOR THE TRACK THREAD

5.2 Choice of Lock-Request Retry Period

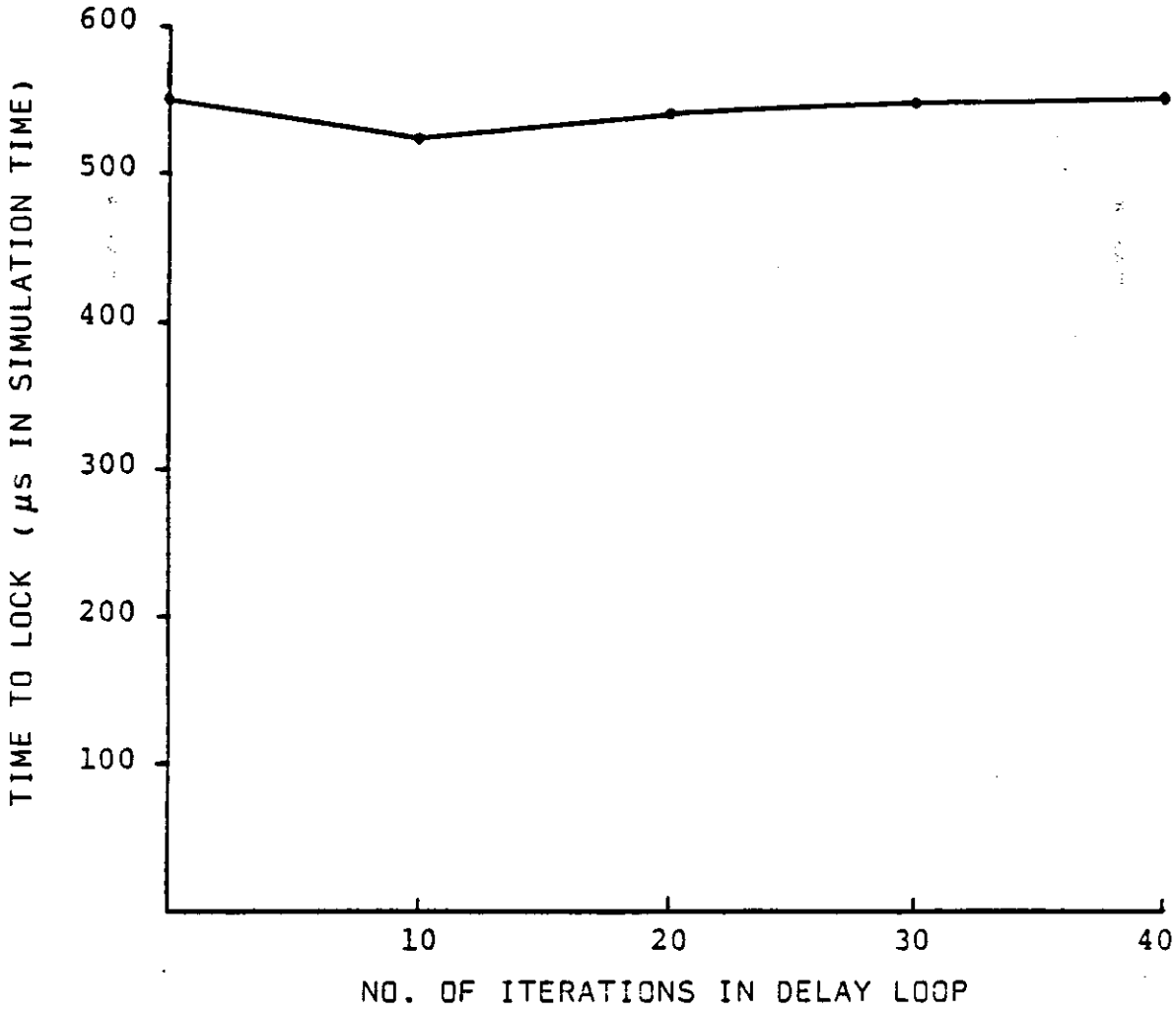
When a processor fails to obtain a lock grant for a record, it retries repeatedly until it receives a grant or reaches a time-out (i.e., detects a failure). If the period between retries (retry period) is too short, the number of shared memory conflicts increases. If the retry period is too long, the processor may be waiting for a lock even though the record is free. [1] Experiments were performed by inserting a delay loop in the lock procedure. The retry period can be controlled by varying the number of the loop iterations. Each loop (retry) iteration is about 0.95 μs. Fig. 12 displays the lock-grant time as a function of the retry period. We noted that the lock-grant time is rather insensitive to the retry period. A slightly lower average lock-grant time occurred at ten loop iterations (9.5 μs simulated time).

5.3 Choice of Time-Out Period for Processor Failure Detection

The time-out period for detecting a processor failure during a record update should be longer than the maximum lock-holding time for any task. The time-out period is measured as the maximum number of lock requests for a record. If the time-out period is too long, a processor would issue unnecessary requests for a lock that is held by a failed processor. On the other hand, if the time-out period is too short, the processor would initiate undesirable recovery processes. Again, the time-out period is implemented by repeating iterations of a loop. Each iteration runs for about 54 μsec. The experiment shows that 13 iterations (corresponding to 650 μs) is the lowest number that yields no false time-out detection as shown in Fig. 13.

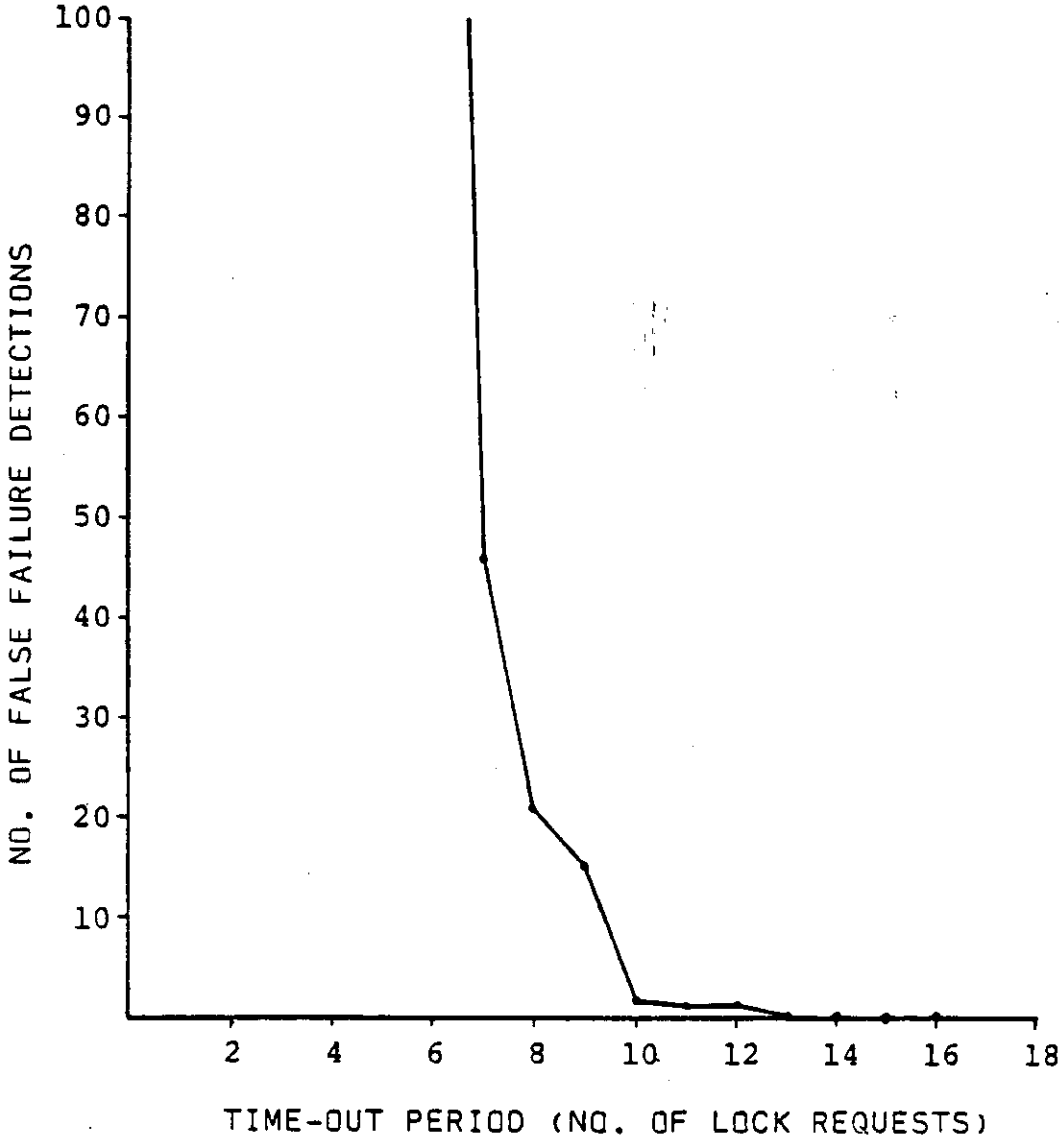
---

[1] Queuing of lock requests is not feasible since memory modules do not have enough intelligence for queuing handling. Queuing handling at the processor would be costly because of the required interprocessor communication (IPC) among processors.



10 ITERATIONS OF THE DELAY LOOP EQUALS ABOUT 9.5 μs SIMULATION TIME.

FIGURE 12. LOCK-REQUEST RETRY PERIOD VS. LOCK-GRANT TIME



TIME-OUT PERIOD WITH 13 LOCK REQUESTS YIELDS THE LOWEST NUMBER OF RETRIES THAT DOES NOT GENERATE FALSE FAILURE DETECTIONS.

FIGURE 13. TIME-OUT PERIOD VS. FALSE FAILURE DETECTION

5.4 Performance of the FTL Protocol With Processor Failures

This set of experiments study the time required to detect and to recover from a processor failure. To emulate a failure, the processor is forced into an infinite loop while it holds a lock. It was shown that the time for detecting the failure and completing the recovery is within 2 to 10 ms for a task with a maximum allowable of 40 ms port-to-port time.

The performance of FTL was also measured under heavy loads. The port-to-port times for processing 1153 object-detections are well under the maximum allowed port-to-port time, 50 ms.

6. FTL FOR MEMORY AND PATH FAILURES

6.1 Shared Memory Failures

When a processor detects a memory failure, it notifies all the other processors. We proposed previously to use the fault-tolerant message-passing technique for such notification. However, implementation of this technique on CMS-I results in a significant increase in system utilization and port-to-port time. This is in part due to the Kernel Operating System (KOS) for CMS-I that does not provide interrupts for messages received. As a result, whenever the processor accesses a record copy in a shared memory, it needs to check duplicated message boxes [2], which requires large processing overhead. Further, two or more processors may detect the same memory failure independently and may cause system thrashing.

Therefore, an alternative technique is introduced for handling memory failure that does not require message passing. When a processor requests a record from a shared memory module and detects a memory failure, it marks this fact on its local RST (without notifying the other processors). It also marks 'failed' on the LW of the record copy if the LW is still accessible. When a second processor finds that the LW on its requested record copy is marked

'failed', this second processor marks the failure of that record in its RST. If the LW of that record copy is inaccessible, this record copy will not be accessed by any processor anyway.

6.2 Path Failures

A single point failure in the crossbar switch may prevent a set of processors from accessing a particular memory module. This prevents updating the records in that memory module and it causes data inconsistency. However, this record may be accessed by other processors that are not blocked by that single point failure, and they might retrieve inconsistent data. This can be avoided by maintaining the status of the ul other record copy in the LW of each record copy. When a processor detects a copy failure, it marks the failure on its RST and on the LW of the other copy (the non-failed copy). Later, when a second processor accesses the non-failed copy, the LW will reflect the failure of the other copy. This second processor should then mark that information on its local RST in order to avoid further accesses to that record copy.

In the following, we summarize the technique for detecting memory and path failures that do not require message passing. The PDL codes for its implementation are given in the Appendix.

1) Records are duplicated on different shared memory modules and each record copy has a lock status word which indicates: a) the lock state of its record copy (free, locked, update-initiated, or failed) and b) the status of the other copy (good or failed).

2) Each processor is required to read the lock word before accessing the record copy and, if either of the record copies is failed, should mark the fact on its local record status table.

7. AREAS FOR FURTHER RESEARCH

Testbed results reveal that the FTL is capable of detecting processor failures and recovering from inconsistency between record copies in case of processor failure, yet satisfying the real-time requirements. To assure data consistency in case of shared memory and path failures, the FTL prohibits further accesses to those record copies that are inaccessible by other processors.

Testbed results reveal that using message passing technique on CMS-I to notify other processors about memory and path failures was too costly. This is in part due to the fact that the Kernel Operating System (KOS) does not provide interrupts for messages received. An alternative technique for handling memory and path failure without message passing has been proposed. The KOS for CMS-II does provide interrupts for message passing and thus eliminates much of the overhead involved with CMS-I. Thus, we recommend that both the message-passing and non-message-passing techniques be experimented on CMS-II to compare their performances.

Currently, the FTL protocol is implemented with one-mode locking. Under such locking protocol, a record is locked exclusively whenever there is a read or write to the record. Locking increases the probability of lock conflict and degrades performance. Therefore, for certain BMD threads, two-mode locking (exclusive lock for write and shared lock for read) or three-mode locking (reserve, upgrade, and exclusive lock [4]) may be used to reduce lock-grant time and port-to-port time. Further experimentation in this area should be performed.

8. REFERENCES

[1] W. W. Chu, et al., "Database Management Algorithms for Advanced BMD Applications," University of California, Los Angeles, CSD-840031, April 1984.

[2] G. Barnett, "Fault Tolerant Locking Protocol Experiment Results - Volume I," System Development Corporation, TM-HU-311/204/00, November 1984.

[3] G. Barnett, "Fault Tolerant Locking Protocol Experiment Results - Volume II," System Development Corporation, TM-HU-311/205/00, November 1984.

[4] W. W. Chu, et al., "Database Management Algorithms for Advanced BMD Applications," University of California, Los Angeles, CSD-830430, April 1983.

APPENDIX

A PDL IMPLEMENTATION OF THE FAULT-TOLERANT LOCKING PROTOCOL

TYPE DECLARATIONS FOR FTL

```
TYPE

TRACK_DATA = RECORD
             OBJECT_STATUS  : OB_STAT;
             PULSE_TYPE     : OB_STAT;
             RETURN_COUNT   : INTEGER;
             XTR_LIFE       : INTEGER;
             KOR_LIFE       : INTEGER;
             POD_LIFE       : INTEGER;
             IPP_LIFE       : INTEGER;
             RTE_LIFE       : INTEGER;
             STATE_TIME     : INTEGER;
             THREAD_TIME    : INTEGER;
             PULSE_TIME     : INTEGER;
      END;

LOCK_STAT = (FREE, LOCKED, UPD_INITED, FAILED);

REC_STAT = (GOOD, FAILED);

LOCK_WORD = RECORD
             LOCK_FLAG      : LOCK_STAT;
             LOCK_COUNT     : INTEGER;
             STAT_OTHER_COPY : REC_STAT;
      END;

TRACK_RECORD = RECORD
             REC_LOCK       : LOCK_WORD;
             REC_DATA       : TRACK_DATA;
      END;

TRACK_FILE = ARRAY [1..128] OF TRACK_RECORD;
```

VARIABLE DECLARATIONS AND INITIALIZATION OF THE RECORD STATUS TABLE

```
VAR

REC_STAT_TABLE  : ARRAY [1..2,1..128] OF REC_STAT;

TRACK_COPY      : ARRAY [1..2] OF ^TRACK_FILE;
                  {Pointers to duplicated TRACK file}

TRACK_REC       : TRACK_DATA;  {Local copy of a record}

REC_ID          : INTEGER;
RSTAT           : INTEGER;
I,J             : INTEGER;

BEGIN

{Initialization of Record Status Table}
LOOP FOR I:=1 TO 2:
   LOOP FOR J:=1 TO 128:
      REC_STAT_TABLE(I,J) := GOOD;
   ENDLOOP;
ENDLOOP;

{Initialization of pointers to Track File copies}
TRACK_COPY(1) :: INTEGER := MSGADDR^(1);
TRACK_COPY(2) :: INTEGER := MSGADDR^(2);

   . . . . . .

   . . . . . .
```

USE OF A RECORD OF THE TRACK FILE

```
{The following shows how to lock a Track Record, make a local copy,
 update locally, update shared memories, and unlock it.  The Track
 Record is identified by 'REC_ID'}

{Lock the record}
RECORD_LOCK (REC_ID,RSTAT);

{If both copies are inaccessible, then go to error routine}
IF RSTAT=-1 THEN ERR_ROUTINE;

{Make a local copy of the record}
TRACK-REC := TRACK_COPY(RSTAT)^(REC_ID).REC_DATA;

{Update the local copy}
WITH TRACK_REC DO
  << UPDATE THE LOCAL COPY >> ;
ENDWITH;

{Update copies in the shared memory}
RECORD_UPDATE(REC_ID,TRACK_REC);

{Unlock the record}
RECORD_UNLOCK(REC_ID);
```

Subroutine RECORD_LOCK

```
PROCEDURE RECORD_LOCK (REC_ID:INTEGER, VAR RSTAT:INTEGER);

 {RSTAT: return status
  1 or 2: successful lock and use copy#1 or copy#2
  -1    : unsuccessful because both copies are failed}

CONST
  MAX_TRY = 100;  {Time-out Period}

VAR
  I  : INTEGER;
  STAT: INTEGER;

BEGIN

WITH LW1 = TRACK_COPY(1)^(REC_ID).REC_LOCK, {Lock word of copy#1}
   LW2 = TRACK_COPY(2)^(REC_ID).REC_LOCK  {Lock word of copy#2}
 DO

START:

 IF REC_STAT_TABLE(1,REC_ID) = GOOD THEN
  IF REC_STAT_TABLE(2.REC_ID) = GOOD THEN

   {In case that both copies are accessible}

   START1: {Lock-request to copy #1}
     CUR_COUNT := LW1.LOCK_COUNT;

   LOOP1:
     LOOP FOR NO_LOOP:=1 TO MAX_TRY: {Try until time-out}

       IF LW2.STAT_OTHER_COPY = GOOD THEN
```

Subroutine RECORD_LOCK (Cont'd)

```
{When LW2 tells that copy#1 is good}

<< EXCLUSIVE ACCESS TO LW1.LOCK_FLAG>>;

CASE LW1.LOCK_FLAG OF

 FAILED:BEGIN {LW1 tells that copy#1 is failed}
   <<RELEASE OF EXCLUSIVE ACCESS>>;
   REC_STAT_TABLE(1,REC_ID) := FAILED;
   LW2.STAT_OTHER_COPY := FAILED;
        {Mark the failure of copy#1 on LW2}
   GO TO START;
   END;

 FREE: BEGIN {Successful Lock}
   LW1.LOCK_FLAG := LOCKED;
   <<RELEASE OF EXCLUSIVE ACCESS>>;
   LW1.LOCK_COUNT := CUR_COUNT+1;
   GO TO START2; {For the lock of copy#2}
   END;

 OTHERWISE: {Copy#1 is locked or being updated}
   <<RELEASE OF EXCLUSIVE ACCESS>>;

ENDCASE;

{When the lock-count has been changed, then start counting
 from the beginning}
IF CUR_COUNT <> LW1.LOCK_COUNT THEN
  GO TO START1;
ENDIF;
```

Subroutine RECORD_LOCK (Cont'd)

```
  ELSE
   {If LW2 tells that copy #1 is failed, then mark it on the
    local record status table and LW1}
   REC_STAT_TABLE(1,REC_ID) := FAILED;
   LW1.LOCK_FLAG := FAILED;
   GO TO START;
  ENDIF;

ENDLOOP;

{When time-out is detected, the lock-count is incremented
 to prohibit further detections by other computers}

<<EXCLUSIVE ACCESS TO LW1.LOCK_COUNT>>;
IF CUR_COUNT = LW1.LOCK_COUNT THEN
  LW1.LOCK_COUNT := LW1.LOCK_COUNT+1;
  <<RELEASE OF EXCLUSIVE LOCK>>;
  RECONF (REC_ID, STAT); {Recovery from failure}
  IF STAT=-1 THEN ERR_ROUTINE; ENDIF; {When recovery is unsuccessful}

ELSE {The failure is detected  and recovered by another computer}
  <<RELEASE OF EXCLUSIVE LOCK>>;
ENDIF;

GO TO START;
```

Subroutine RECORD_LOCK (Cont'd)

```
START2: {Locking of copy #2}

  IF LW1.STAT_OTHER_COPY = GOOD THEN

   {When LW1 tells that copy#2 is o.k.}

   CASE LW2.LOCK_FLAG OF

    FAILED:BEGIN {When copy#2 is already marked failed on its LW,
          then mark it on the local record status table and the LW1}
     REC_STAT_TABLE(2,REC_ID) := FAILED;
     LW1.STAT_OTHER_COPY := FAILED;
     END;

    OTHERWISE: {Lock copy#2}
     LW2.LOCK_FLAG := LOCKED;

   ENDCASE;

  {When LW1 tells that copy#2 is failed, then mark it on the
   local record status table and the LW2}

  ELSE
   REC_STAT_TABLE(2,REC_ID) := FAILED;
   LW2.LOCK_FLAG := FAILED;
  ENDIF;

  RSTAT := 1;
```

Subroutine RECORD_LOCK (Cont'd)

```
ELSE {The record status table tells that only copy#1 is available}

 START3:
   CUR_COUNT := LW1.LOCK_COUNT;

 LOOP3:
   LOOP FOR NO_LOOP:=1 TO MAX_TRY:
    <<EXCLUSIVE ACCESS TO LW1.LOCK_FLAG>>;

    CASE LW1.LOCK_FLAG OF

     FAILED: BEGIN
      <<RELEASE OF EXCLUSIVE ACCESS>>;
      REC_STAT_TABLE(1,REC_ID) := FAILED;
      RSTAT:=-1;
      ESCAPE LOOP3;
      END;

     FREE: BEGIN
      LW1.LOCK_FLAG := LOCKED;
      <<RELEASE OF EXCLUSIVE ACCESS>>;
      LW1.LOCK_COUNT := CUR_COUNT+1;
      RSTAT:=1;
      ESCAPE LOOP3;
      END;

     OTHERWISE: <<RELEASE OF EXCLUSIVE ACCESS>>;

    ENDCASE;

    IF CUR_COUNT = LW1.LOCK_COUNT THEN
     REC_STAT_TABLE(1,REC_ID) := FAILED;
     LW1.LOCK_FLAG := FAILED;
     RSTAT := -1;
    ELSE
     GO TO START3;
    ENDIF;

ENDIF;
```

Subroutine RECORD_LOCK (Cont'd)

```
ELSE IF REC_STAT_TABLE(2,REC_ID) = GOOD THEN
  {When the record status table tells that only copy#2 is available}

  START4:
    CUR_COUNT := LW2.LOCK_COUNT;
  LOOP4:
    LOOP FOR NO_LOOP:=1 TO MAX_TRY:
     <<EXCLUSIVE ACCESS TO LW2.LOCK_FLAG>>;
     CASE LW2.LOCK_FLAG OF
      FAILED: BEGIN
        <<RELEASE OF EXCLUSIVE ACCESS>>;
        REC_STAT_TABLE(2,REC_ID) := FAILED;
        RSTAT:=-1;
        ESCAPE LOOP4;
        END;
      FREE: BEGIN
        LW2.LOCK_FLAG := LOCKED;
        <<RELEASE OF EXCLUSIVE ACCESS>>;
        LW2.LOCK_COUNT := CUR_COUNT+1;
        RSTAT:=2;
        ESCAPE LOOP4;
        END;
      OTHERWISE: <<RELEASE OF EXCLUSIVE ACCESS>>;
     ENDCASE;

     IF CUR_COUNT = LW2.LOCK_COUNT THEN
      REC_STAT_TABLE(2,REC_ID) := FAILED;
      LW2.LOCK_FLAG := FAILED;
      RSTAT := -1;
     ELSE
      GO TO START4;
     ENDIF;

  ELSE {No copy is available}
   RSTAT := -1;
  ENDIF;
 ENDIF;

ENDWITH;
END;
```

Subroutine RECORD_UPDATE and RECORD_UNLOCK

```
PROCEDURE RECORD_UPDATE (REC_ID:INTEGER, TRACK_REC:TRACK_DATA);

VAR
  I: INTEGER;

BEGIN
  LOOP FOR I:=1 TO 2:
    IF REC_STAT_TABLE(I,REC_ID)=GOOD THEN
      WITH TRACK_COPY(I)^(REC_ID) DO
        REC_LOCK.LOCK_FLAG := UPD_INITED;
        REC_DATA := TRACK_REC;
      ENDWITH;
    ENDIF;
  ENDLOOP;
END;



PROCEDURE RECORD_UNLOCK (REC_ID:INTEGER);

VAR
  I: INTEGER;

BEGIN
  LOOP FOR I:=1 TO 2:
    IF REC_STAT_TABLE(I,REC_ID) = GOOD THEN
      TRACK_COPY(I)^(REC_ID).REC_LOCK.LOCK_FLAG := FREE;
    ENDIF;
  ENDLOOP;
END;
```

ACKNOWLEDGEMENTS

The authors would like to thank Joseph Bannister of UCLA for his comments and discussions of fault tolerant locking, and Laurel Cornachio for her secretarial and administrative support in preparing this report.

DISTRIBUTION LIST

1. Director
BMD Advanced Technology Center
ATTN: ATC-P
P. O. BOX 1500
Huntsville, AL 35807-3801

2. BMDPO
ATTN: DACS-BMT
P. O. Box 15280
Arlington, VA 22215-0150

3. Commander
Ballistic Missile Defense Systems
ATTN: BMDSC-AOLIB
P. O. Box 1500
Huntsville, AL 35807-3801

4. Defense Technical Information Center
Cameron Station
Alexandria, VA 22314

5. TRW, Incorporated
ATTN: Earl Swartzlander
One Space Park
Redondo Beach, CA 90278

6. General Research Corporation
ATTN: Dave Palmer
P. O. Box 6770
Santa Barbara, CA 93105

7. Stanford University
Stanford Electronics Laboratories
ATTN: Mike Flynn
Standford, CA 94305

8. McDonnell/Douglas Corporation
ATTN: Gale Schluter
5301 Bolsa Avenue
Huntington Beach, CA 92647

9. University of California/Berkeley
Dept. of Electrical Engineering and Computer Science
ATTN: C. V. Ramamoorthy
Berkeley, CA 94720

10. System Development Corporation
ATTN: SDC Library
4810 Bradford Blvd. NW
Huntsville, AL 35805

11. System Development Corporation
ATTN: W. C. McDonald
4810 Bradford Blvd, NW
Huntsville AL 35805

12. General Research Corporation
ATTN: Genry Minshew
307 Wynn Drive
Huntsville, AL 35805

13. Optimization Technology, Inc
ATTN: Paul McIntyre
20380 Town Center Lane
Suite 160
Cupertino, CA 95014

14. Auburn University
Dept. of Electrical Engineering
ATTN: Dr. Victor Nelson
207 Dunstan Hall
Auburn, AL 36830

15. University of South Florida
Computer Science Program - LIB 630
ATTN: K. H. Kim
Tampa, FL 33620

16. TRW, Incorporated
ATTN: Wayne Smith
213 Wynn Drive
Huntsville, AL 35805

17. Carnegie-Mellon University
Department of Computer Science
ATTN: Daniel P. Siewiorek
Scheneley Park
Pittsburgh, PA 15213

18. The University of Connecticut
Computer Science Department
ATTN: E. E. Balkovich
Storrs, CT 06268

19. Systems Control, Inc.
ATTN: Hank Fitzgibbon
555 Sparkman Drive, Suite 450
Huntsville, AL 35805

20. TRW, Incorporated
ATTN: Mack Alford
7702 Governor's Drive W
Huntsville, AL 35805

21. Carnegie-Mellon University
Department of Computer Science
ATTN: Zary Segall
Pittsburgh, PA 15213