DISTRIBUTED DATA BASE MANAGEMENT FOR REAL-TIME BMD APPLICATIONS

Wesley W. Chu
K.K. Leung, C. M. Sit
R. C. Lee, H. S. Ngai
T. C. Shen, M. A. Merzbacher

July 1986

CSD-860040

DISTRIBUTED DATA BASE MANAGEMENT FOR REAL-TIME BMD APPLICATIONS

Principal Investigator: Wesley W. Chu

Researchers: K. K. Leung, C. M. Sit, R. C. Lee,

H. S. Ngai, T. C. Shen, M. A. Merzbacher

Abstract

This technical report summarizes the research progress on "Distributed Data Base Management for Real-Time BMD Applications" during the past year. The major goal of the research is to develop algorithms to reduce task response time and increase fault tolerance of the system. The research areas include: module allocation and scheduling for distributed systems, fault tolerant distributed database for both tightly coupled and loosely coupled systems, and knowledge based distributed database systems.

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)
"Distributed Data Base Management for Real-Time BMD Applications"

5. TYPE OF REPORT & PERIOD COVERED
Final Report for period: May 29, 1985 to May 28, 1986

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Wesley W. Chu, K. K. Leung, C. M. Sit, R. C. Lee, H. S. Ngai, T. C. Shen, M. A. Merzbacher

8. CONTRACT OR GRANT NUMBER(s)
DASG60-85-C-0059

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
US Army Strategic Defense Command
P.O. Box 1500
Huntsville, AL 35807

12. REPORT DATE
July 31, 1986

13. NUMBER OF PAGES
193

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)
Real-Time Systems, Distributed Processing Systems, Task Allocation, Batch Scheduling Policy with Time Out, Fault Tolerant Distributed Database Systems, Knowledge based Distributed Database Systems, Distributed Problem Solving, Query Processing with Semantic Reasoning

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
This technical report summarizes the research progress on "Distributed Data Base Management for Real-Time BMD Applications" during the past year. The major goal of the research is to develop algorithms to reduce task response time and increase fault tolerance of the system. The research areas include: module allocation and scheduling for distributed systems, fault tolerant distributed database for both tightly coupled and loosely coupled systems, and knowledge based distributed database systems.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-LF-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DISTRIBUTED DATA BASE MANAGEMENT FOR REAL-TIME BMD APPLICATIONS

FINAL REPORT FOR THE PERIOD

FROM: May 29, 1985

TO: May 28, 1986

Contract No. DASG60-85-C-0059

Prepared For:

US Army Strategic Defense Command

Huntsville, Alabama 35807

July 31, 1986

University of California, Los Angeles

Wesley W. Chu, Principal Investigator

Researchers: K. K. Leung, C. M. Sit, R. C. Lee,

H. S. Ngai, T. C. Shen, M. A. Merzbacher

Table of Contents

CHAPTER I

INTRODUCTION AND SUMMARY

INTRODUCTION AND SUMMARY

During the past year, we have concentrated our efforts in the following areas of distributed systems: module allocation and scheduling for distributed systems, fault tolerance for distributed database systems for both tightly-coupled and loosely-coupled systems, and knowledge based distributed database systems.

1. MODULE ALLOCATION AND SCHEDULING FOR DISTRIBUTED SYSTEMS

Module allocation and scheduling are two key issues that effect system performance in distributed systems. Response time is intimately related with module allocation strategy and processing scheduling policy. Therefore, we shall use response time as a performance measure in our investigation.

1.1 Module Replication and Assignment for Real Time Systems

An analytic model is developed to estimate the task response time of distributed systems. The model considers such factors as interprocessor communications, module precedence relationship, module scheduling, interconnection network delay, and assignment of modules and files to computers. A heuristic algorithm for module assignment is developed to iteratively search for module assignments which provide shorter task response times. Assigning replicated modules may reduce task response time. Therefore, the algorithm also considers module replications. Using the sum of task response time and penalty delay for the violations of specified thread response time requirements as the objective function, an "optimal" module multiplicity and module allocation can be determined by the proposed algorithm. The detailed model and algorithm are presented in Chapter 2.

Our study revealed that the task response times for a given module assignment (with replications) generated by the algorithm compare closely with that of the simulation. A series of

experiments is also performed to characterize the behavior of the algorithm.

1.2 Scheduling Policies for Real-Time Distributed Processing Systems

Module scheduling and processing consist of two types of overhead, fixed and incremental. A new scheduling algorithm called Batch Service with Time-out (BST) is proposed. This new scheduling algorithm processes the invocations of a module in batch thus reducing the fixed scheduling overhead. Therefore, it is particularly suitable for those applications that require repeated module invocations and have high fixed scheduling overhead. An analytical model is developed in Chapter 3 which provides response time estimate. The analytical results compare very closely with that of simulations. In addition, the performance comparison between the BST and FCFS algorithms are given for systems operating under varied environments.

2. FAULT TOLERANT DATABASES

The survivability of distributed systems can be improved with multiple copies of files. When an update is performed on a copy, the update should be written on all other file copies. If the computer that is handling the update fails during the update process, all the copies may not be updated, resulting in mutual inconsistency. We have proposed low cost (that is, short response time) techniques for maintaining mutual consistency among replicated file copies during update failures for both the tightly-coupled and loosely-coupled systems.

2.1 Fault-Tolerant File Updates for Tightly-Coupled Systems

For tightly-coupled systems, we have proposed the *fault-tolerant locking protocol (FTL)* that maintains mutual and internal consistency for updating replicated file copies in the event of computer and/or shared memory failures. The FTL protocol was implemented on the SDC multi-microcomputer testbed and a set of experiments was conducted to assess the performance of the FTL protocol. The testbed experiment results reveal that the FTL protocol increases the

CPU utilization by about 20%, and the port-to-port time constraints are met for a typical threat scenario. The testbed results confirm that the FTL protocol is feasible for BM/C^3 systems. Further, a computer failure can be effectively recovered by the FTL protocol. We have also developed an analytical model to characterize the behavior of FTL and estimate the lock retry period effect on FTL throughput. The detailed model is presented in Chapter 4. Our results indicate that the performance of the FTL is a function of lock conflict and memory conflict which in turn depend on the record size, number of records in the system, probability of record reference and system architecture (in terms of number of processors and basic structure and number of memory modules). G. Barnett of SDC at Huntsville is currently performing experiments on FTL with the CMS 2 crossbar switch system. We are working closely with him to validate our analytical model and predictions.

2.2 Fault-Tolerant File Updates for Loosely-Coupled Systems

When an update is performed at a site, this update should be delivered to those sites that have a replicated file. All file copies should be identical. This is referred to as *mutual consistency*. File copies may, however, differ temporarily during update propagation. If a site fails during an update broadcast, only a part of the file copies may be updated, resulting in mutual inconsistency.

Posting a file update in a system is known as a *commit;* which implies that the posted update will not be backed out. The site that broadcasts an update to other sites is known as the *coordinator* and the receiving sites of the update are the *participants*.

Two-phase commit is a well known method which ensures mutual consistency among file copies in case of failures. It is also called a *blocking commit* because when a coordinator site fails during an update broadcast, other sites are blocked from using the file copies until the failed coordinator recovers and completes the unfinished commit work. The cost of two-phase commit

is too high and is not suitable for real time applications. Therefore, we have proposed a low cost resilient commit protocol that is *non-blocking* and *tolerant to multiple failures* [2]. We have studied how to incorporate this commit protocol into existing concurrency control techniques such as EWP [3] and PSL [4] as well as the procedure for site recovery.

A simulation model is presented in Chapter 5 for studying the behavior of this resilient commit protocol and to estimate the response time for the commit protocols. We plan to study such key factors as network protocols, queuing and service delays due to CPU execution requirements, and interrupt and message processings. We plan to use this model to study the suitability of applying this resilient commit protocol to the SDI BM/C^3 environment.

3. KNOWLEDGE BASED DISTRIBUTED DATABASE SYSTEMS

3.1 Optimal Query Processing with Semantic Reasoning

Query processing is a key consideration in database management systems. For real-time systems query response time is usually used as a system performance measure. Many approaches have been proposed to improve the performance of query processing. Most of them are based on "syntactic" considerations; that is, the original query expression is transformed into a set of algebraically equivalent forms and the lowest cost query processing strategy among this set of expressions is selected as the optimal query processing policy.

In order to speed up query processing in the SDI BM/C^3 environment, new knowledge representations are needed for representing and organizing the vast amount of data. An alternative approach (Chapter 6) is to use the domain knowledge to transform the given query into a semantically equivalent but more efficient form for processing. This is known as Semantic Query Optimization. We are in the process of implementing a Semantic Query Processor (SQP) on top of a relational database. The rule based knowledge will be represented in PROLOG. The model can be used to develop a methodology for acquiring and incorporating useful knowledge

in Semantic Query Optimization and to study the query response time improvement from different types of knowledge. Initially we shall test out the concept via a relational naval database, and eventually apply the concept to the SDC SDI Battle Management database.

3.2 Multiple Agent Problem Solving for Distributed Systems

In a distributed system such as Distributed Processing Systems or Distributed Database Systems, most problem solving involves multi-agents (computers). A typical multi-agent problem solving system consists of a collection of agents, each with such various skills as sensing, communication, planning and acting. The group as a whole has a set of assigned tasks or a global goal. There are three main steps for problem solving in a multi-agent environment. The first step is to decompose the global goal into subgoals and assign them to appropriate agents. Each agent then designs its own plan to accomplish its assigned subgoals. Further negotiation is often required among these subgoals to accomplish the global goal. The final step is to carry out the global plan.

Contract Net protocol [5] provides a way to decompose the global goal into subgoals and assign them to appropriate agents and assuming that these subgoals are assigned to the set of agents are independent. However, subgoals usually are interdependent. The dependency of the subgoals among agents prevents each agent from devising its own plan. There are two types of inter-dependency among agents: cooperation and conflict. The goal in multi-agent planning with inter-dependent subgoals is to minimize conflict and maximize cooperation among agents. An agent has to devise its own plan while at the same time incorporating the plans of other agents into it. The problem becomes even more complex when the inter-dependency between two agents, A and B, is so strong that A cannot proceed without knowing B's plan first and vice versa. Thus, thus we reach a deadlock situation. We propose to use a two phase algorithm to generate a plan for a multi-agent environment to resolve such situations (see Chapter 7). In the first phase, each agent generates its own individual plan to accomplish its own subgoal without

considering subgoals from other agents. In the second phase, the agents compare and modify their plans with each other to avoid conflict and improve cooperation. We repeat the second phase process until the global requirements are satisfied.

Multi-agent problem solving can be applied to a variety of applications such as view integration for distributed database systems and Battle Management in SDI environment. The database at each platform may have a different view of the object. Multi-agent problem solving may be used for integrating the view of an object at different platforms into a single integrated view. We may also view each platform as an agent, based on the intergrated input scenario, battle requirements, and available resources, multi-agent problem solving technique may be used to solve for the BM strategy. We plan to use the SDC SDI BM/C^3 database to experiment various integration techniques and study their cost/performance tradeoffs.

References

[1] Wesley W. Chu and Jung Min An, "Fault Tolerant Locking (FTL) for Tightly Coupled Systems," *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems,* Los Angeles, California, January 13-15, 1986.

[2] Jung Min An and Wesley W. Chu, "A Resilient Commit Protocol for Real Time Systems," *Proceedings of the 1985 Real Time Systems Symposium,* San Diego, California, December, 1985.

[3] Wesley W. Chu and Joseph Hellerstein, "The Exclusive Writer Approach to Updating Replicated Files in Distributed Processing Systems," *IEEE Transactions on Computers,* pp. 489-500, June 1985.

[4] Michael Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," *IEEE* Transactions on Software Engineering, Vol. SE-5, No. 3, pp. 188-194, May 1979.

[5] Davis, R. and Smith R. G., "Negotiation as a Metaphor for Distributed Problem Solving," AI Memo 624, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1981.

CHAPTER II

MODULE REPLICATION AND ASSIGNMENT FOR REAL-TIME DISTRIBUTED PROCESSING SYSTEMS

MODULE REPLICATION AND ASSIGNMENT FOR REAL-TIME DISTRIBUTED PROCESSING SYSTEMS

1. INTRODUCTION

Computer systems with real-time applications (e.g., process control and space defense) have many functions that must finish within a specified time period if the systems are to perform properly. Distributed processing is a cost-effective technique for meeting these performance requirements while providing such features as incremental system growth, potential for improved system availability, and grace performance degradation in case of failures. In this chapter, we consider a class of real-time distributed processing systems (RTDPS) in which there is a single application task and where message passing is used for communication between processors.

The application task of a real-time system is often partitioned into a set of software modules (or simply, *modules).* The assignment of those modules to processors affects system response time, throughput and reliability. Several approaches for module assignment in distributed processing systems have been proposed. These techniques include graph-theoretic, mathematical programming, and heuristic approaches [CHU80]. The key parameters considered in these approaches are module execution times and communication times. The goal in module assignment is to balance the processing load among the processors such that either the total system time is minimized or computer loads are well balanced.

[STON77] and [RAO79] use graph-theoretic algorithms which are tractable only for systems with two computers. Algorithms proposed in [MA82], [EFE82], [CHOU82] and [SHEN85] balance the workload on computers but neglect the impact of module precedence relationships. Further, they assume that the application task is invoked only once. As a result, the queueing effect from multiple invocations which is an important portion of the the response time is ig-

nored. Moreover, the interconnection network delay is not considered in the module assignment methods.

Another important issue in sharing the processing workload among processors is to selectively *replicate* modules on different processors according to the loading conditions. An invocation for a replicated module is routed to the appropriate module's resident processors for execution. As a result, module replications may improve system load balancing, response time, system reliability and availability.

The current module assignment algorithms neglect: repeated task invocations, queueing effects, module precedence relationships, interconnection network delays, and module replications. The Algorithm proposed in this paper remedies these shortcomings.

Task response time [1] in a RTDPS is the time from the invocation of an application task to the completion of the task execution. Often the application task consists of several sequences of modules which are referred to as *threads*. For some applications, the response times of individual threads rather than the entire task, are of interest. Alternatively, task response time may be defined as the sum of thread response times weighted by certain factors according to the application requirements. Since task response time is an important performance measure for RTDPS, minimizing the response time is the major goal of module assignment. Key factors (parameters) that affect task response time include IPC, processor loading, module precedence relationships and interconnection network delay. In a previous paper [CHU84b], a task response time model was introduced which considers these key parameters. Here we shall use this analytic model for estimating task response times to perform module assignment.

In this chapter, we shall describe the task response time model. Then a module assignment algorithm based on the model is introduced. The algorithm considers both the module re-

[1] Mean *task response time* refers to the *mean* response time unless otherwise stated.

plication and assignment simultaneously. Next, we present the algorithm validations by applying to a simple distributed system and a real-time distributed system for space defense applications. A series of experiments to characterize the behavior of the algorithm is also presented.

2. TASK RESPONSE TIME MODEL

The application task in a RTDPS is partitioned into a set of modules. The logical structure and precedence relationships among the modules may be represented by a task control-flow graph, as shown in Figure 1. The task is repeatedly invoked in accordance with the application requirements. 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 send messages to update shared data files. Such message exchanges among modules are referred to as *intermodule communication (IMC)* [CHU84a]. The overhead for communication among modules that reside on the same computer is usually small. If, however, messages are sent between modules that reside on different computers, the messages are called *interprocessor communication* (IPC). IPC requires extra processing such as communication protocol and management of the distributed data files. IPC also incurs interconnection network delay. Therefore IPC has a more significant impact on system performance than IMC.

Simulation techniques may be used to estimate the response time for RTDPS, but such approaches are time-consuming and expensive. Queueing networks [BASK75, HEID82, LAZO84] are commonly used to model distributed processing systems. In such models, computers are represented as servers, modules' invocations as customers, and task invocations correspond to external arrivals. Customers are routed for service in accordance with the task control-flow graph and the module assignment. In distributed systems, a module may enable more than one module (referred to as an *and-fork* in the control-flow graph). Alternatively, a module may have several immediate predecessor modules which must complete their executions before the succeeding module can be executed (referred to as an *and-join).* When a control-flow

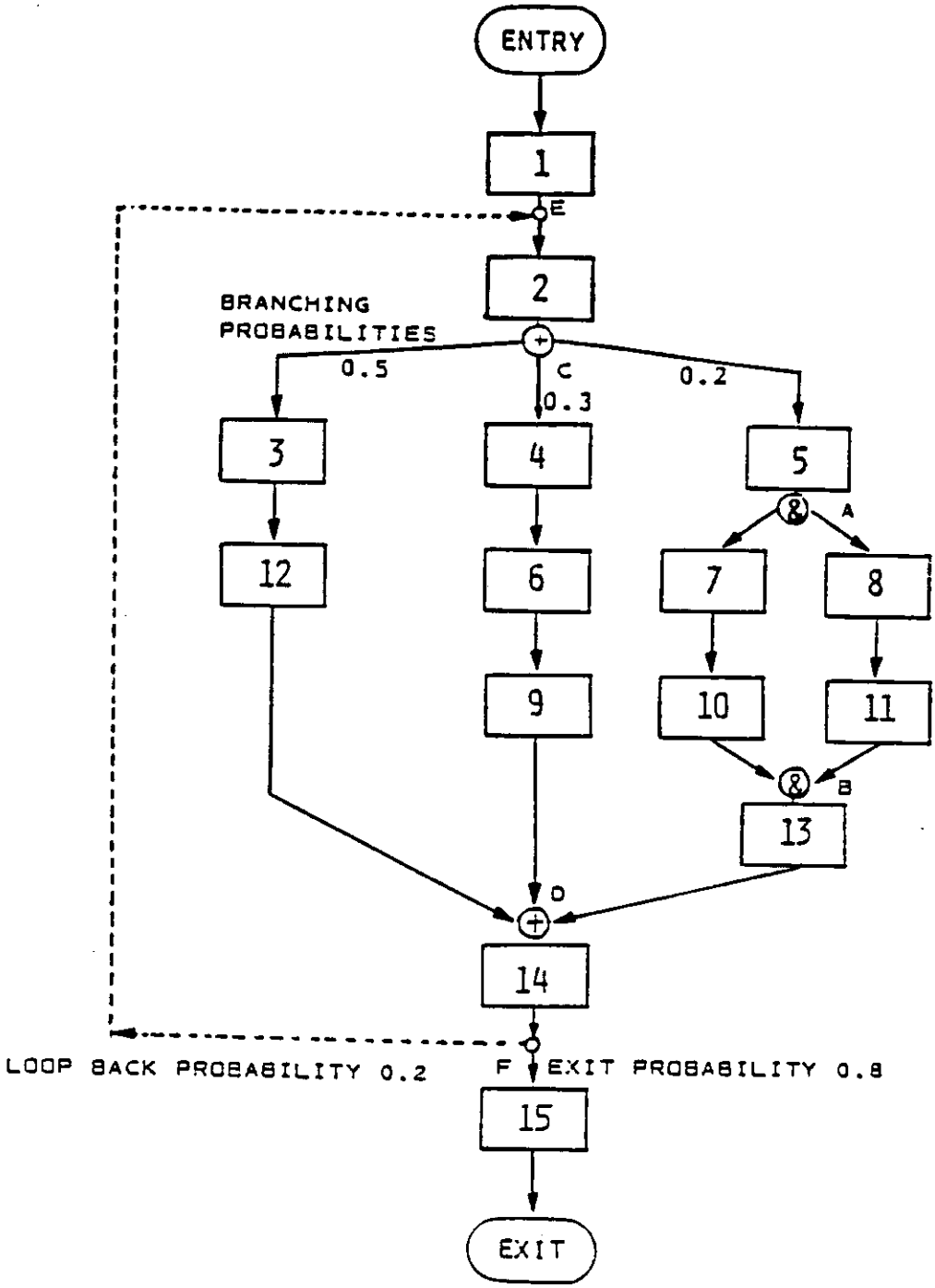


Figure 1. A Sample Task Control-Flow Graph

graph consists of these forks and joins, the routing scheme in the queueing network model becomes inadequate to represent the logical relationships among modules. Thus, the system cannot be represented by a tractable queueing network model. Therefore, we have introduced a new model to estimate the task response time.

Since a task may be repeatedly invoked and modules are enabled in accordance with the sequence 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 to the start of its execution on a computer. This waiting time is the time spent waiting for module executions and IPC processings. *Module execution time* is the sum of a module's execution time and its output IPC (processing) time. *Module response time,* on the other hand, refers to the sum of a module's waiting time and its execution time. The *precedence waiting time* is the intermodule synchronization delay resulting from the precedence relationships among modules. Our task response time model consists of two sub-models: the *module response time model* and the *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, this model is used to compute module response times on each computer. 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. These IPC's are transmitted over the interconnection network, and eventually arrive at their destinations. On the destination computers these input IPC's can be viewed as a special module which also contends for processing. Based on the module assignment and IMC among modules, IPC times can be obtained. Let module execution times be characterized by probability distribution functions. Then each computer can be modeled as a queueing system with its resident modules (customers of different types) of specified service dis-

tributions. Based on the module assignment, 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 forms a bulk module invocation.

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

Consider a computer that has h distinct module invocations (single or bulk invocations). Let the arrival rate for the i^{th} module invocation be λ_i and the Laplace Transform (L.T.) of the service requirement be $Y_i^*(s)$ for i=1,2,...h. One of these h module invocations (say the c^{th}) represents all the input IPC on the computer. Then λ_c and $Y_c^*(s)$ become the arrival rate and L.T. of processing time for the input IPC. For the i^{th} invocation of a set S_i of distinct module(s), the corresponding service requirement is $Y_i^*(s) = \prod_{j \varepsilon S_i} X_j^*(s)$, where $X_j^*(s)$ is the L.T. of the service time of module j. In case the i^{th} invocation just invokes a single module, S_i has one element.

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 $\lambda = \sum_{i=1}^{h} \lambda_i$. The L.T. of service time for each invocation arrival is $Y^*(s) = \sum_{i=1}^{h} \frac{\lambda_i}{\lambda} Y_i^*(s)$. For the M/G/1 queue, the first two moments of the module invocation waiting time the period from the invocation arrival to the start of its first module execution, are:

[1] The model can also be applied to other module scheduling policies by using the corresponding queueing delay equations.

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

and

$$\overline{w^2} = 2(\bar{w})^2 + \frac{\sum_{i=1}^{h} \lambda_i \overline{y_i^3}}{3(1-\rho)} \tag{2}$$

Where:

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

ρ = server utilization $= \sum_{i=1}^{h} \lambda_i \overline{y_i^1}$,

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

From Eqs.(1) and (2), we obtain the variance of module invocation waiting time:

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

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

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

where:

w = module invocation waiting time (independent of module invocations as FCFS is used),

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

Note that the first two terms on the right side of Eq.(4) correspond to the module waiting time

for module j_k. The average response time $T(j_k)$ for module j_k can be calculated from the expected values of Eq.(4). Thus, we have

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

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,

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

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

Our previous experiments show that because of the random and asynchronous operations on computers, the assumptions used in the task response time model are acceptable and generate accurate module response time estimations [1].

2.2 Weighted Control-Flow Graph Model

The next step in computing task response time is to consider precedence waiting times. Our general approach is to classify the types of precedence relationships and to show how precedence waiting can be computed by mapping the mean and variance of 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 weight for all arcs emerging from module i in the control-flow graph. If module i has executed and enables module j (on a different computer), the module enablement message is transmitted via the interconnection network. Assuming the network delay is independent of module response times, the mean and vari-

[1] In case the independent module invocation assumption generate results that are not accurate enough, a model that considers dependent invocation arrivals may be used to estimate the modules' response times [CHU84b].

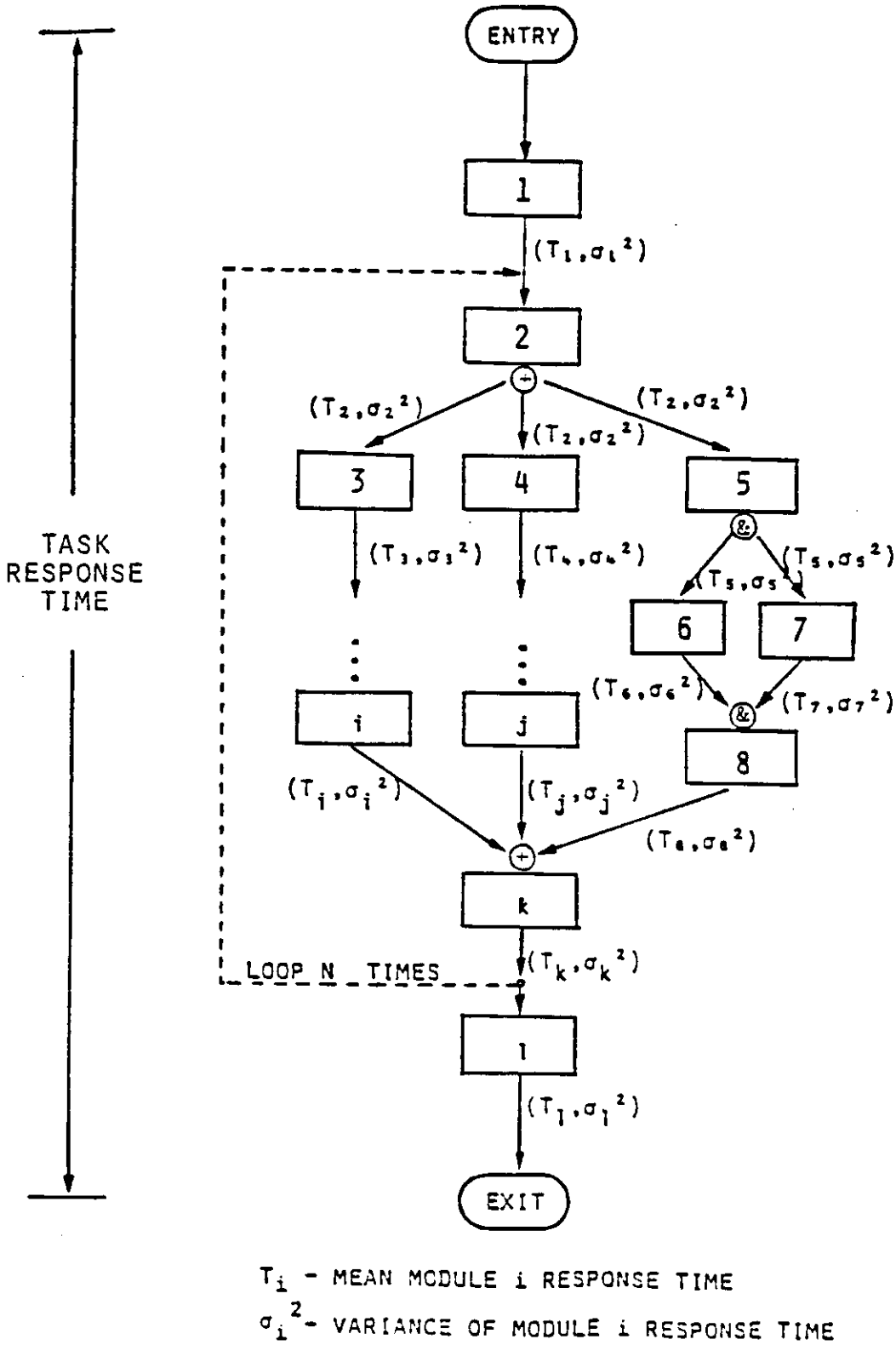


Figure 2. Weighted Control-Flow Graph for Response Time Estimations

ance of network delay[2] can be added to the weight of the arc from module i to j. The task response time can be estimated from this weighted control-flow graph model.

There are four common types of control-flow subgraphs: *sequential thread, and-fork to and-join, or-fork to or-join,* and *loop* that are based on the logical structures and precedence relationships among modules (Figures 3 to 6). A task control-flow graph may contain a set of subgraphs which are a combination of these basic logical relationships among modules. Each of these subgraphs can be reduced to a single node graph. Successive graph reductions yield the estimation of response time for the complete task.

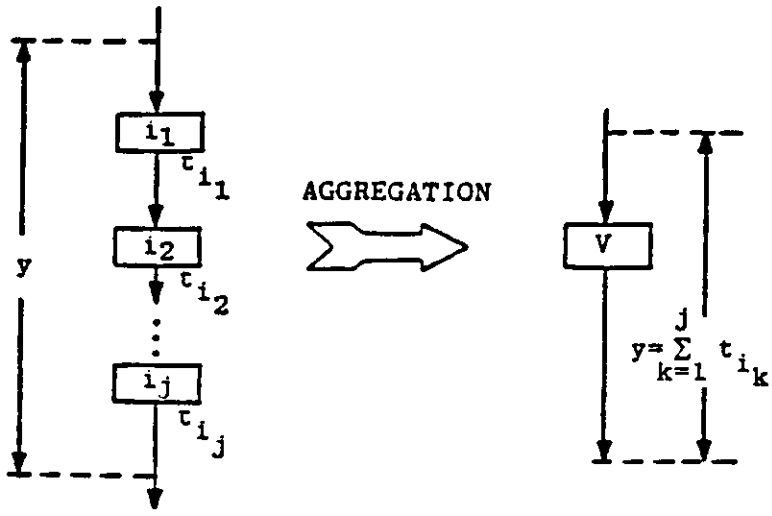
2.2.1 Sequential Thread Subgraph

The sequential thread subgraph (Figure 3) is a sequence of modules connected in series where 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 the arc weights, are random variables, the total response time of the sequence thread is the sum of the arc weights of all modules in the thread.

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 $i=1,2,..n$ (Figure 4). Because of the and-join function, the response time of the subgraph is

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



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

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

Figure 3 Sequential Thread

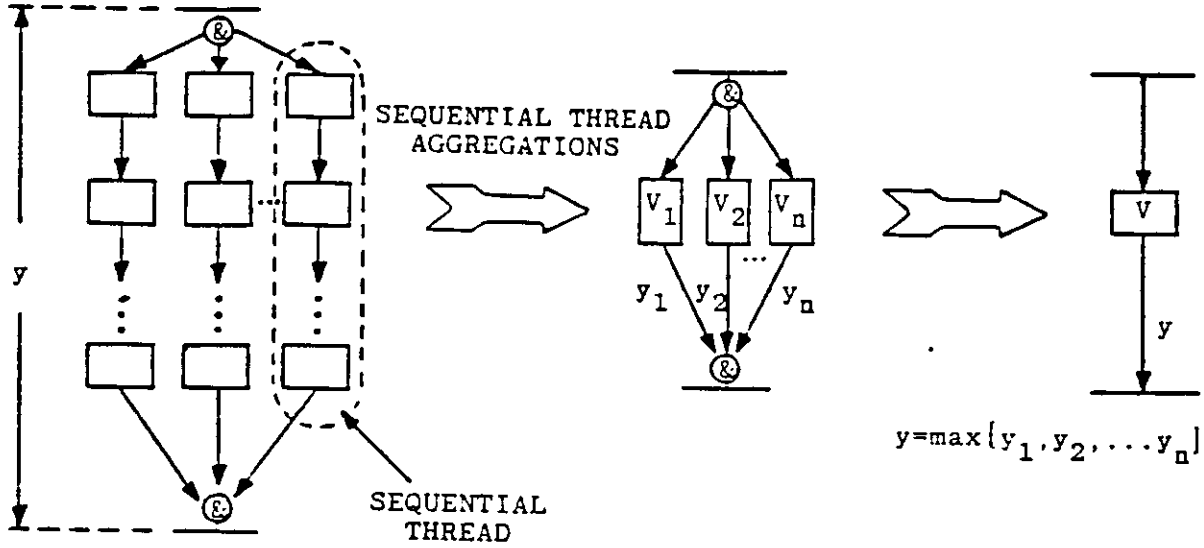


Figure 4 And-Fork to And-Join Subgraph

the maximum of y_i's.

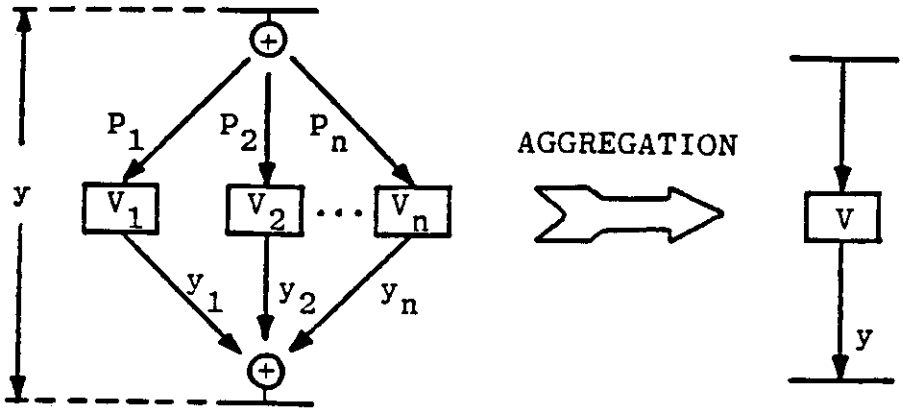
Computing the response time for this subgraph requires the knowledge of the probability distribution functions for y_i's, which is rather complicated. In this study, we shall emphasize the *average* task response time, which can usually be determined by the first two moments of module response time. Therefore, these moments are derived from the module response time model. They can be approximated by either Erlangian or hyper-exponential distribution functions [SAUE81]. according to the coefficients of variation of y_i's. Assuming that y_i's are independent, the joint distribution function 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. At the or-join, the succeeding module can be enabled by any one of its preceding modules. This type of subgraph facilitates the system to process one of several threads based on certain selection criteria. The branching probability of each thread's execution can be measured or estimated from the IMC data. The response time for the subgraph is the sum of the thread response times weighted by their invocation probabilities.

2.2.4 Loop Subgraph

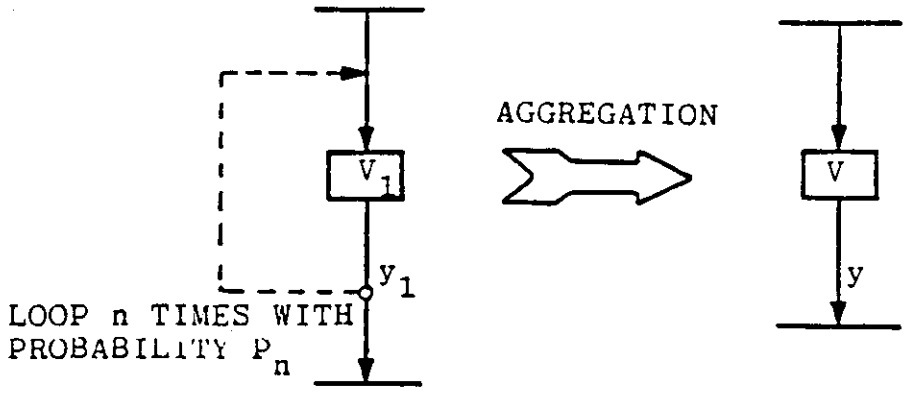
Loops are often contained in a task control-flow graph for repeatedly processing a set of modules for a task invocation. A loop may contain any of the aforementioned subgraphs. After aggregating these subgraphs, a loop may be represented by a single cyclic node graph, as shown in Figure 6. The arc weight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

Simulation experiments have been conducted to validate the task response time model. Our results reveal that the model yields accurate task response time estimations for different module assignments with non-exponential module invocation rates under various loading environments.

3. AN ALGORITHM FOR MODULE ASSIGNMENT WITH MODULE REPLICATIONS

In a distributed system *replicating* a module on several processors instead of allocating each module to only one processor, provides sharing of the workload among processors. Each invocation for these replicated modules is routed to and executed on one of their resident processors; a special algorithm is required to route the invocations to the appropriate computers. The routing algorithm assumes that it does not cause system bottleneck. A simple strategy is to route invocations for a replicated module to its resident processors in a round-robin fashion.

The replication and assignment of modules to computers in a distributed system is referred to as the *replicated module assignment problem* (RMAP). A RMAP consists of two key problems: determining the optimal module multiplicities (i.e., number of copies for each module), and allocating those module copies to computers such that system performance specifications can be satisfied. Since both module multiplicities and assignment of module copies to computers affect system performance, the problems are considered jointly. For simplicity, we refer to the replication and assignment of modules to computers as *module assignment.*

The response time requirements for specific threads of an application task are usually specified by users. Module replications provide us with more flexibility in system design. The key question is which modules require replications, and how many copies of them are needed to minimize task response time and satisfy the thread response time specifications as well.

3.1 Assumptions

Let us make the following assumptions for the RTDPS:

1. There is no memory space constraint at any processor.
2. Data files are stored at a processor where its resident modules read and/or update the files.
3. The scheduling discipline for module executions (e.g., first-come first-serve, head-of-line priorities) at each processor is given.
4. All processors in the system are identical. Thus, the execution time for each module is the same.
5. The network delay is independent of module assignment. Under this assumption, although different module assignments may generate different volume of IPC traffic in the network, we assume the network has sufficient bandwidth and the delay remains unchanged.

Assumptions 4 and 5 can be relaxed by adjusting the modules' execution times according to the processing speed and the interconnection network delay for each module assignment.

3.2 A New Objective Function

To search for optimal module assignment, we need to establish an objective criterion (function). The RMAP for RTDPS has two objectives: (1) to minimize task response time, and (2) to satisfy response time specifications for the threads. We shall combine these into a single objective function as follows.

$$T_{obj}(A) = \begin{cases} T(A) & \text{if } \textit{all thread response time requirements are met} \\ T(A) + aT_{pd}(A) & \textit{otherwise} \end{cases} \quad (7)$$

where:

$T(A)$ = task response time for module assignment A,

$T_{pd}(A)$ = a positive-valued penalty delay function for module assignment A,

a = a positive scaling constant to account for the impact of violating thread response time requirements.

This new objective function is the sum of task response time, T, and the possible *penalty delay*, aT_{pd}. Both T and T_{pd} depend on module assignment. For a given module assignment, the penalty delay may be added to the objective function to "penalize" violations of thread response time requirements. Clearly, if the penalty delay scaling constant a is chosen such that aT_{pd} is sufficiently large when compared with T, T_{obj} will yield too large of an increase when some threads violate their response time specifications. Any algorithm for the RMAP searches for a module assignment with the minimum value of T_{obj}. The algorithm implicitly avoids those solutions which yield unsatisfactory thread response times.

Let us define the following system parameters:

n = total number of processors in the system,

m = total number of modules in the application task,

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

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

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

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

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

D_{net} = average network delay;

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

λ = task invocation rate.

Upon the completion of a module execution, a module may communicate with other modules. The processing time required for sending a message from module i to module j is referred to as *IMC time* for the module pair. If the communicating modules are allocated on two different processors, then extra processing overhead is required on both the transmitting and receiving computers. The processing time for the IPC (interprocessor communication) at each processor is equal to the IMC time plus the protocol processing overhead. Let us define the following:

$\bar{t}_c(i,j)$ = average IMC time for the IMC from module i to j;

$\sigma_c^2(i,j)$ = variance of IMC time for the IMC from module i to j;

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

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

The module assignment matrix $A = [A_{ij}]$ is an indicating function such that

$$A_{ij} = \begin{cases} 1 & \text{if } \textit{module } j \textit{ resides on processor } i,\ i \in [1,n],\ j \in [1,m] \\ 0 & \textit{otherwise} \end{cases}$$

Given these parameters, the task response time for a module assignment A can be computed by the task response time model. [1] Let us use a function F to denote the task response time model. Then, the task response time of the distributed system for module assignment A can be expressed as

$$T(A) = F[\, G,\ A,\ X,\ \sigma^2(x),\ T_c,\ \sigma^2(c),\ D_{net},\ \sigma^2{}_{net},\ \lambda,\ m,\ n].$$

3.3 Penalty Delay Function

Assume a task consists of k distinct threads. Let $t_i(A)$ be the average response time for thread i for module assignment A, and R_i be the average response time requirement for thread i. The response time overrun of thread i for module assignment A is defined as

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

where $i \in [1,k]$. We can express the thread response time overruns for all threads for module assignment A as a vector $D(A) = [d_1(A), d_2(A), \cdots d_k(A)]$. For a given module assignment A, if the response time specification for thread i is violated; that is, $t_i(A) > R_i$, then $d_i(A)$ represents the discrepancy between that thread's response time and its requirement. Let $\bar{w}_i$ be the *required*

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

average module waiting time for each module in thread i which consists of n_i modules (denoted as a set S_i), then

$$\overline{w}_i = \frac{R_i - \sum_{j \epsilon S_i} \overline{x}_j}{n_i} \tag{8}$$

where $\overline{x}_j$ = mean execution time for module j. We can express the required mean module waiting times for all the threads as a vector $W_R = [\overline{w}_1, \overline{w}_2, \cdots \overline{w}_k]$. For a given R_i, the numerator in Eq.(8) is the maximum allowable sum of waiting times for all modules of thread i. Thus, $\overline{w}_i$ represents the required average waiting time for each module in the thread i. The smaller the value of $\overline{w}_i$ is, the faster response the thread i requires from the processor.

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

Property 1:

The penalty delay increases as a thread response time overrun increases; that is $T_{pd}(D(A), W_R)$ is an increasing function of $d_i(A)$ for all $i \in [1,k]$.

Property 2:

The penalty delay is inversely related to the required mean module waiting times for all threads. That is, if two threads have the same thread response time overrun, then the thread with the stricter response time requirement (i.e., with a smaller $\overline{w}_i$) yields a higher penalty delay.

Based on these properties of the penalty delay function, the objective function, T_{obj}, in Eq.(7) can help us to search for module assignments that reduce the thread response time overruns. The search process attempts to first satisfy those threads that have stricter response time requirements. Therefore, we define the penalty delay function according to these properties as

follows:

$$T_{pd}(D(A),W_R) = \sum_{i=1}^{k} l_i \, d_i(A) \tag{9}$$

where $l_i = \dfrac{\max\{ \overline{w}_1, \overline{w}_2, ..., \overline{w}_k \}}{\overline{w}_i}$ for all $i \in [1,k]$.

Note that Eq.(9) satisfies Property 1. If $\overline{w}_i > \overline{w}_j$, then $l_i < l_j$. Thus Property 2 holds. By the definition of $d_i(A)$'s, $T_{pd}(D(A),W_R)$ is equal to zero if all thread response time requirements are satisfied. Substituting Eq.(9) into (7), we obtain the objective function for the RMAP as

$$T_{obj}(A) = T(A) + a \sum_{i=1}^{k} l_i \, d_i(A). \tag{10}$$

Given a module assignment for the distributed system, each computer processes a set of assigned modules. Based on the task invocation rate and the routing algorithm for replicated module invocations, the module invocation rates at each computer can be determined. From other system parameters, the task response time and thread response times can be computed by the task response time model. The penalty delay function can be calculated from the thread response times and their requirements.

3.4 Search Algorithm for the RMAP

The RMAP for the distributed system is to find module assignment A such that the objective function is minimized; that is,

To minimize $T_{obj}(A) = T(A) + a \, T_{pd}(D(A),W_R)$

$$= F(G,A,X,\sigma^2(x),T_c,\sigma^2(c),D_{net},\sigma^2_{net},\lambda,m,n) + a \sum_{i=1}^{k} l_i \, d_i(A) \tag{11}$$

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

The constraint inequalities [1] indicate that each module must be allocated to at least one processor or may be replicated onto as many as all processors in the system.

The application task for the RTDPS requires repeated task invocations and the tasks also consist of various logical and precedence relations among modules. Therefore, the module assignment problem is more complicated than the multiprocessor scheduling problems [GARE79] which have been proved to be NP-complete. The common methods of tackling such combinatorial optimization problems include approximation algorithms, probabilistic algorithms, branch-and-bound and local search techniques [PAPA82]. However, due to the complexity and characteristics of the RMAP, we propose an algorithm that searches for the sub-optimal solutions and then selects the solution from this set of local optimals.

The RMAP algorithm consists of three major components:

1) Module Relocations from Longest-Wait Processor to Shortest-Wait Processor

For a given module assignment, let the processor in the system that has the longest waiting (queueing) time, and the one with the shortest waiting (queueing) time be denoted by LWP and SWP respectively. To reduce T_{obj}, modules may be relocated, one at a time, from the LWP to the SWP (without changing module multiplicities) until no further improvement can be made by such module relocation.

2) Further Module Replications on SWP

[1] To increase system reliability the lower bounds of the inequalities may be changed to force modules to be replicated on more than one processor. The upper bounds may be smaller than n which may be dependent on the application requirements.

After module relocations from the LWP to the SWP have reached a local optimum, the algorithm attempts to balance the processing workload by further replicating certain modules onto the SWP. In case certain thread response time requirements are violated, the candidates for further replications on the SWP are the modules of those threads that violate their response time requirements. If all thread response time specifications are satisfied, those modules currently residing on the LWP become the candidates for further replication onto the SWP. After one of these candidate modules is replicated onto the SWP, the processor loading will be altered and some modules may require relocation from the new LWP to SWP to minimize T_{obj}. If T_{obj} is improved by these replications, the module replication on the SWP that yields the minimum T_{obj} is finalized. Note that T_{obj} may not always be improved by these module replications on the SWP because it may increase IPC and/or violate of thread response time requirements.

3) Module Deletions from LWP

If further module replication on the SWP does not improve T_{obj}, the algorithm deletes certain modules from the LWP. This is because deleting modules may reduce IPC in the system and/or deleting some replicated modules of threads with less stringent response time requirements may improve T_{obj}. The algorithm also takes a greedy step to finalize a module deletion from the LWP that yields the lowest T_{obj}.

The RMAP Algorithm is given in the following:

REPLICATED MODULE ASSIGNMENT ALGORITHM

1. Determine initial module multiplicities (see Sec.3.5 for details), or use the module multiplicities of the previous local optimal assignment as initial module multiplicities for this iteration.

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

3. Relocate module(s) from LWP to SWP without changing module multiplicities until reaching a local optimal assignment:

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

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

   3.3 Invoke the task response time model:

      3.3.1 Compute $T_{obj}(A_o)$ for assignment A_o and

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

   3.4 Let S_L be the set of modules residing on $LWP(A_o)$ but not residing on $SWP(A_o)$. For each module $j \in S_L$, perform

      3.4.1 Temporarily relocate module j from $LWP(A_o)$ to $SWP(A_o)$ and form a new assignment A_j;

      3.4.2 Compute the assignment dependent parameters and processor utilization factors for assignment A_j (As Steps 3.1 and 3.2 do);

      3.4.3 If any computer(s) is saturated, set $T_{obj}(A_j) = \infty$; otherwise, invoke task response time model to compute and record $T_{obj}(A_j)$, $LWP(A_j)$ and $SWP(A_j)$ (As Step 3.3 does).

   3.5 If there exists $T_{obj}(A_j) \leq T_{obj}(A_o)$ for any $j \in S_L$ tested in Step 3.4, then perform

      3.5.1 Set $A_o = A_j$, $T_{obj}(A_o) = T_{obj}(A_j)$, $LWP(A_o) = LWP(A_j)$ and $SWP(A_o) = SWP(A_j)$ where $T_{obj}(A_j) = \min_{i \in S_L} \{ T_{obj}(A_i) \}$. (Finalize the single module relocation from LWP to SWP -- A greedy step!)

      3.5.2 Go to Step 3.4.

   3.6 Otherwise, continue Step 4. (Reach a local optimum with respect to module relocation)

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

5. If there exists $d_i(A_o) > 0$ for any $i \in [1,k]$, then let S_R be the set of modules of all threads where $d_i(A_o) > 0$ for all $i \in [1,k]$ (Some thread response time requirements violated); Otherwise, let S_R be the set of modules residing on $LWP(A_o)$. (All thread response time requirements satisfied)

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

   6.1 Temporarily replicate module j onto $SWP(A_o)$ and form a new assignment A_j;

   6.2 Compute $T_{obj}(A_j)$ and relocate modules from $LWP(A_j)$ to $SWP(A_j)$ until reaching a local optimal assignment A_{jo}. (As Step 3 does)

7. If there exists $T_{obj}(A_{jo}) < T_{obj}(A_o)$ for any $j \in S_R$ from Step 6, then

   7.1 Set $A_o = A_{jo}$, $T_{obj}(A_o) = T_{obj}(A_{jo})$, $LWP(A_o) = LWP(A_{jo})$, and $SWP(A_o) = SWP(A_{jo})$ where $T_{obj}(A_{jo}) = \min_{i \in S_R} \{ T_{obj}(A_{io}) \}$; (To finalize a single module replication on SWP)

   7.2 Go to Step 4.

8. Otherwise, let S_L be the set of modules residing on $LWP(A_o)$. For each module $j \in S_L$ which has more than one copy, perform

   8.1 Temporarily delete module j from $LWP(A_o)$ and form a new assignment A_j;

   8.2 Perform Step 6.2 to obtain the local optimal assignment A_{jo}.

9. If there exists $T_{obj}(A_{jo}) < T_{obj}(A_o)$ for any $j \in S_L$ from Step 8, then

   9.1 Set $A_o = A_{jo}$, $T_{obj}(A_o) = T_{obj}(A_{jo})$, $LWP(A_o) = LWP(A_{jo})$, and $SWP(A_o) = SWP(A_{jo})$ where $T_{obj}(A_{jo}) = \min_{i \in S_L} \{ T_{obj}(A_{io}) \}$; (To finalize a single module deletion from LWP)

   9.2 Go to Step 4.

10. Otherwise, Stop. (Reach the final local optimal assignment A_o)

3.5 Initial Module Multiplicities

During the execution of the algorithm, module multiplicities are changed due to the module replications and deletions from searching for better assignments. Further, to provide a good module assignment for a given distributed system, the algorithm is re-iterated with a

number of randomly selected assignments.[1] The final sub-optimal solution is chosen to be the local optimal assignment that yields the lowest T_{obj}. In addition, to explore different assignments with the same module multiplicities, the module multiplicities of a local optimal solution are used to generate the next random module assignment (Step 1 in the RMAP Algorithm). However, the algorithm should start with a set of feasible initial module multiplicities. Therefore, the initial module multiplicities should be properly determined.

There are many ways to select the initial module multiplicities. The basic requirement is that the processing requirement for each module copy does not saturate a processor; that is, the processor utilization from each module copy, which is the product of the invocation rate and the mean execution time of the module copy, on each processor should be less than its capacity. Further, it is desirable to select the initial module multiplicities so that the processing workload can be easily balanced among the processors. Based on these considerations, the following procedure is devised to determine the initial module multiplicities for the RMAP Algorithm:

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

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

3. Compute the initial multiplicity α_i for module i for $i \in [1,m]$:

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

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

Note that α_i should be less than the total number of processors in the system. Although the ini-

[1] A similar technique was used by [LIN65] for the traveling salesman problem.

tial module multiplicities determined by the above procedure may not provide satisfactory thread response times at the beginning, they are subsequently modified during the search for the module assignment that minimizes T_{obj}.

4. PERFORMANCE OF THE ALGORITHM

In order to validate the RMAP algorithm, we shall use the algorithm to generate sub-optimal assignments for a set of simple distributed systems. We shall compare the algorithm's optimal module assignments (with optimal module multiplicities) with those determined by an exhaustive search over all possible solutions. The first system consists of three identical processors and its task has three modules, all of them have constant processing times as shown in Figure 7(a). For simplicity, we assume there is no IMC among modules and no IPC among processors. Since there is no IPC, (to avoid the trivial solution of replicating each module on all processors), we assumed that M_2 cannot be replicated, and resides on only one of the processors. However, M_1 and M_3 can be freely replicated and allocated on all processors. The optimal module assignment for this system, as shown in Figure 7(b), is obtained by exhaustive search which balances the workload at each processor. Each invocation requires two units of processing at each processor. Since the task has a single sequential thread, the thread response time is equivalent to the task response time.

To apply the RMAP algorithm, we set the task invocation rate, $\lambda = 0.4$, $a = 10$, and the thread response time requirement to be 30 seconds. The procedure given in Section 3.5 is used to determine the initial module multiplicities: initially both M_1 and M_2 have a single copy while M_3 has two copies. The RMAP algorithm is then re-iterated with 50 random initial module assignments. Due to the possible processor saturation of the random initial assignment, 13 sub-optimal assignments are generated in the algorithm run. All these assignments can satisfy the thread response time of 30 seconds. More importantly, all these 13 feasible solutions turn out to be identical to the optimal module assignment by the exhaustive search! If we choose $a = 1000$

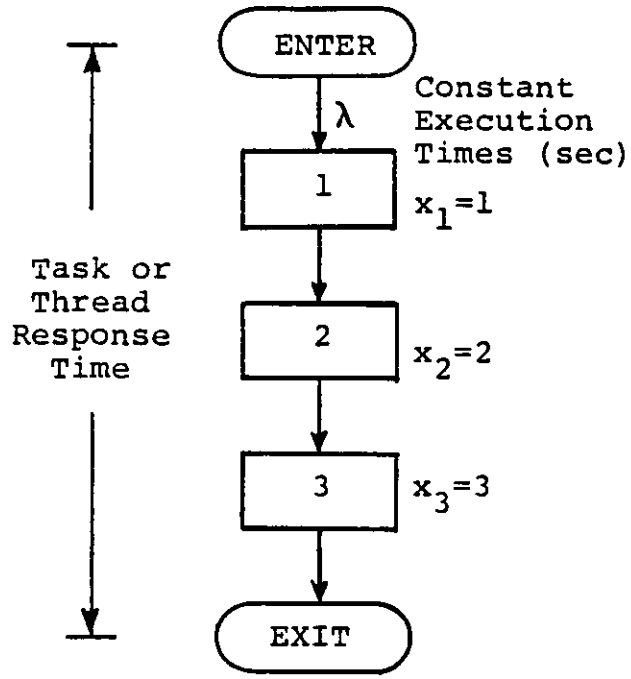


Fig.7(a) The Task Control-Flow Graph

CPU#1	CPU#2	CPU#3
1	2	1
3		3

Fig.7(b) Optimal Module Assignment

λ	Task Response Time (sec)
0.2	8.333
0.4	14.167

Fig.7(c) Task Response Times

and/or set the thread response time requirement to be 15 seconds, similar results have been achieved. We repeat the experiment for $\lambda = 0.2$. M_3 no longer saturates a processor in this case; all modules have a single copy initially. Since the processor load is light for this case, each iteration of the algorithm with a random initial assignment is able to generate the optimal solution. The task (thread) response times for the optimal assignment are given in Figure 7(c).

5. ALGORITHM APPLICATION: THE SENTRY SYSTEM

Next, we shall apply the RMAP Algorithm to a real-time distributed system, the Sentry System [TITA85] that processes radar signals for space defense applications. We shall first describe the characteristics of the Sentry System. Then we present the behavior of the RMAP algorithm obtained from a series of experiments. Simulation results reveal that the module allocation and replication generated by the proposed model meet the specified task response time.

5.1 The Characteristics of the Sentry System

The Sentry System is a loosely coupled distributed system which consists of six processors interconnected by a high-speed bus. The application task is comprised of 12 modules and its control-flow graph is given in Figure 8. Three of these modules, M_{10}, M_{11} and M_{12}, are periodically enabled by the system, while the rest of them are invoked according to the arrivals of radar return signals. When a return signal arrives, M_1 is invoked. When M_1 completes its execution, it is branched to process a particular thread in accordance with the type of the return signal received. The names of various threads are given in Figure 8. The response time for a thread is defined as the time from the arrival of a return signal at the system (i.e., M_1 is invoked) until the message sent by the last module of the thread to M_{10} is processed by the resident processor of M_{10}. Based on the thread response time and loading requirements, modules (except M_{10}) are selectively replicated on several processors. In addition, since M_{11} performs functions that are not directly related to the rest of the modules, it is not allocated to any of these six processors.

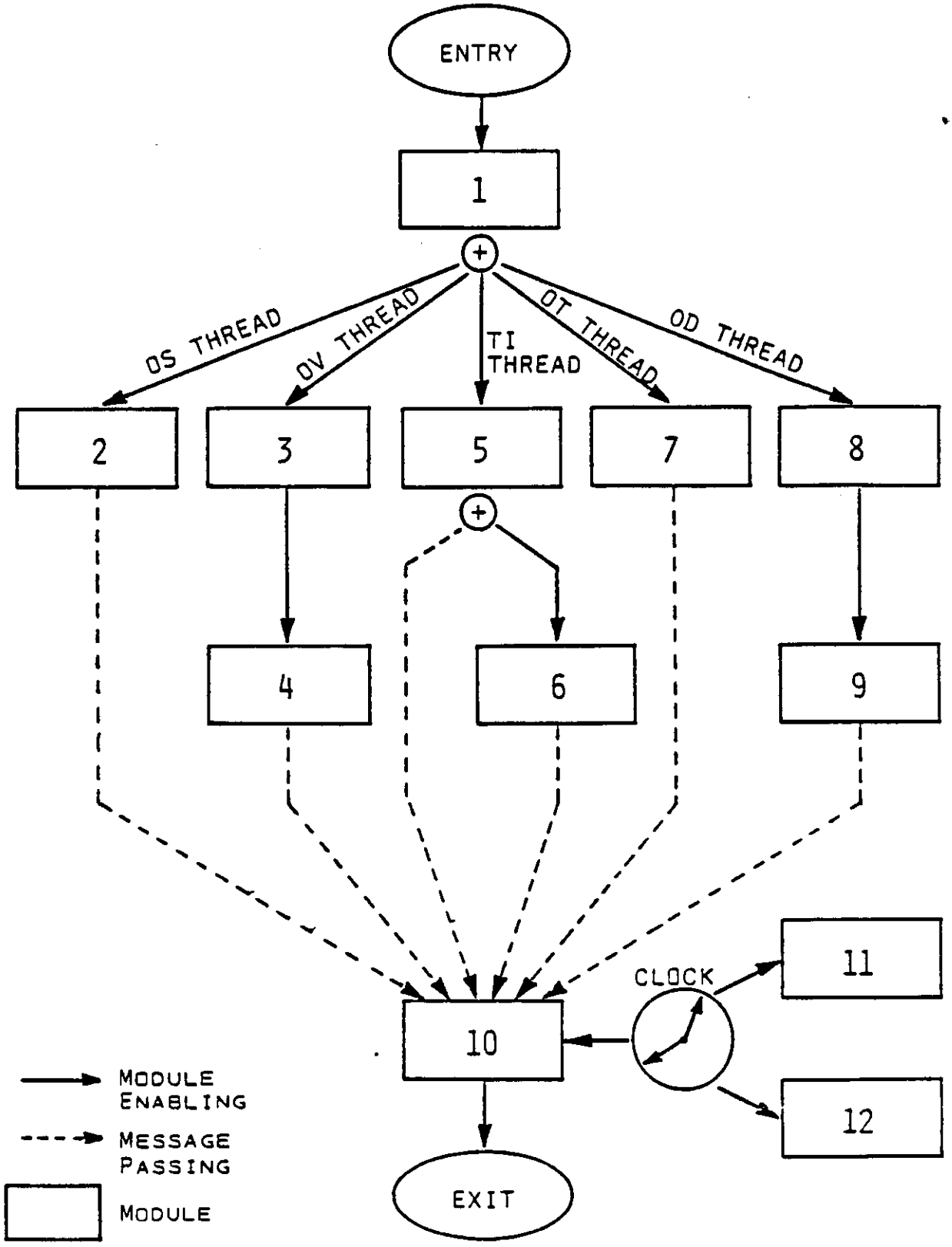


Figure 8 Task Control-Flow Graph for Sentry System

To consider the response times of those modules (i.e., M_{10} and M_{12}) which do not belong to any thread, the task response time for the system is defined as the weighted sum of the average module/thread response times; that is,

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

where:

λ_i = invocation rate for M_i,

λ_{tot} = total invocation rate for all modules,

T_i = mean response time for M_i (averaged over the response times for all copies when M_i has replicated copies).

p_{ij} = probability that M_i enables M_j,

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

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

The data flow of shared file access is presented in Figure 9. Each ellipse represents a data file. An arc pointing from a module to a file indicates a file-update, while one pointing from a file to a module designates a file-read. An arc with double arrows means that the module will both read and update to the file during the module execution.

The Sentry System has an operating system for module scheduling and IPC processing. Invocations for a replicated module are routed to and executed on one of its resident processors in a round-robin manner. Module execution times include the scheduling overheads and file access times, and are assumed to be deterministic. Module execution times and invocation rates are shown in Table 1. Modules communicate with other modules by sharing common data files and/or direct message exchanges. The processing time for the IMC from M_i to M_j is referred to as IMC time (IMC_{ij}) for the module pair. The IMC times for various module pairs are presented

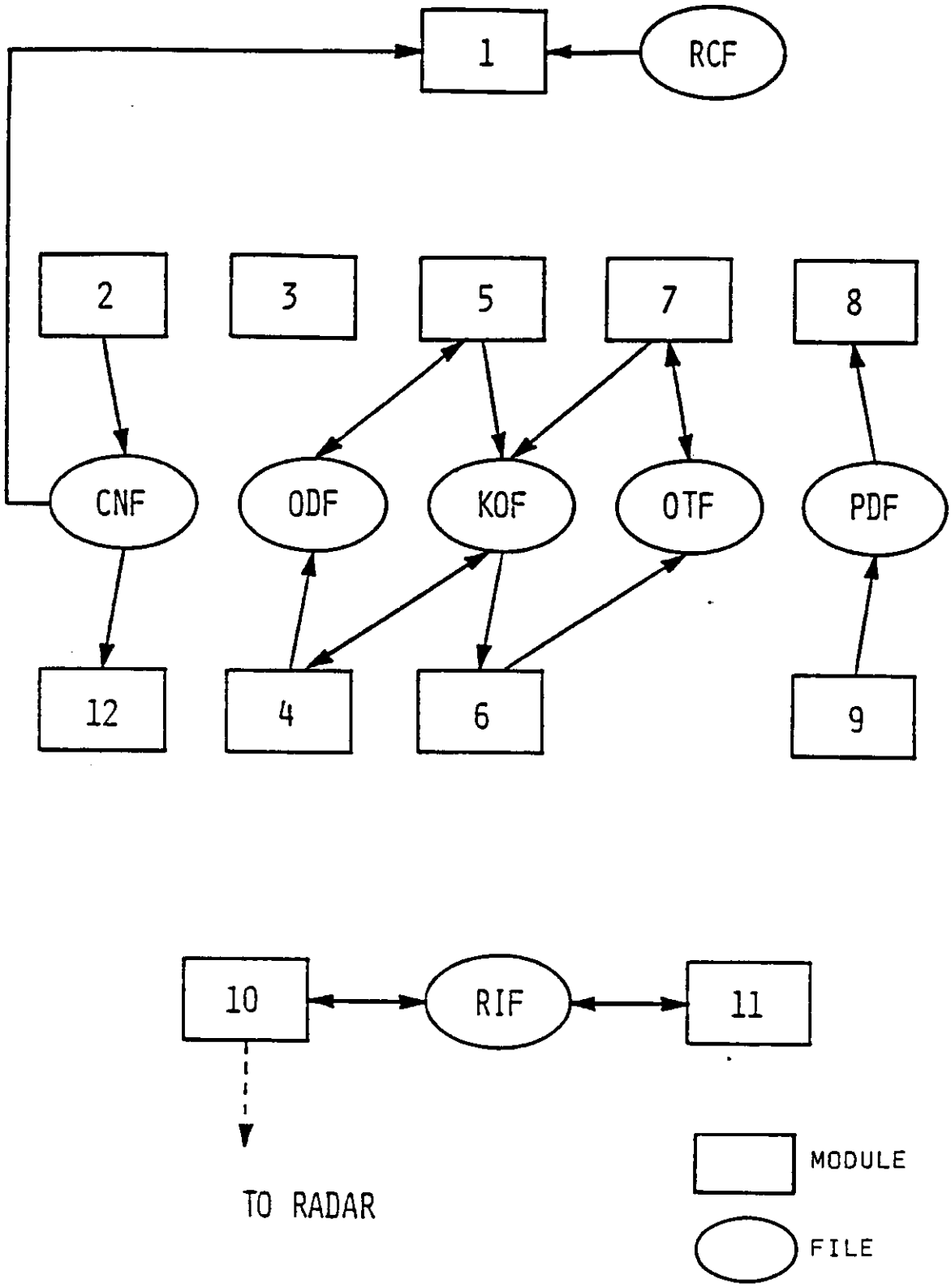


Figure 9 Shared Data Files in the Sentry System

Modules	Invocation Rates (No. of Invocations/ms)	Exec. + Scheduling Time	Read File/Time	Write File/Time	Total Exec. Time
1	1.58	138	RCF/5 CNF/63	-	206
2	0.57	199	-	CNF/98	297
3	0.1695	1144	-	-	1144
4	0.1695	286	KOF/66	ODF/149 KOF/139	639
5	0.6795	1049	ODF/64	KOF/138 ODF/149	1400
6	0.0075	355	KOF/66	OTF/149	570
7	0.1015	1406	OTF/64	OTF/149 KOF/133	1752
8	0.0595	1286	PDF/97	-	1383
9	0.0595	981	-	PDF/215	1196
10	0.2	660	RIF/16	RIF/94	770
11	0.01	1137	RIF/26	RIF/84	1247
12	0.2	269	CNF/102	-	371

Note: All times are in micro-second.

Table 1 Module Execution, File Access Times and Invocation Rates for the Sentry System

in Table 2. If the communicating module pair is located on distinct computers, the IMC becomes IPC which requires processing on both the transmitting and the receiving processor. The processing time for the IPC is called *IPC time* (IPC_{ij}). In the Sentry System, the IPC times on transmitting and receiving processors are different. $IPC_{ij} = 80$ μsec for the transmitting processor and the $IPC_{ij} = IMC_{ij}$ for the receiving processor. The interconnection network delay is the bus delay in the Sentry System. This delay depends on the message length, and ranges from 0.165 to 0.2 msec.

5.2 Characteristics of the RMAP Algorithm

To study the characteristics of the RMAP algorithm, we experiment with it under such different environments as varied thread response time requirements, initial module multiplicities and penalty delay scaling constant. Four selected sets of thread response time requirements, R_A, R_B, R_C and R_D, are used as shown in Table 3. Among these requirements, R_A is the least stringent, R_D is the strictest one, and R_B and R_C lie between those of R_A and R_D. Two different sets of initial module multiplicities, α_A and α_B (Table 4) are used in the experiments. α_A is generated in accordance with the procedure in Section 3.5. All modules except M_5 in α_B consist of only a single copy. Since the processor utilization for M_5 is 95%, M_5 is initially duplicated into two copies to avoid saturating a processor. Three penalty delay scaling constants, 1, 10, and 1000, are used in the experiments.

We have 11 experiments, each of which uses a different combination of thread response time requirements, initial module multiplicities and penalty scaling constants. Experiments #1 through #9 use α_A as initial module multiplicities, while Experiments #10 and #11 use α_B. The scaling constant for Experiment #1 is 1. Experiments #2 to #5 and #10 use 10, while Experiments #6 to #9 and #11 use 1000 as the scaling constant. In each experiment, the RMAP algorithm is iterated with a prespecified number of random initial module assignments (500 or 1000). For each initial assignment, the algorithm generates a sub-optimal assignment. Based on the

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

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

Table 2. IMC Times for Various Module Pairs

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

Table 3. Selected Sets of Thread Response Time Requirements for the Sentry System

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

Table 4. Two Sets of Initial Module Multiplicities for the Sentry System

thread response time requirements, only certain local optimal assignments can satisfy the specifications. The final module assignment is selected from this set of local optimums that yields the minimum T_{obj}. The experiment specifications, thread response times, and T_{obj} of the final module assignment generated from these experiments are presented in Table 5. The corresponding module assignments are shown in Table 6.

Since R_D is the most stringent thread response time requirement, no module assignment is generated (Experiments #5 and #9) to meet the thread response time requirements, even using 1000 randomly selected initial module assignments. The number of module assignments tested and CPU time used to obtain the final module assignment varies from one experiment to another. They ranged from 19,100 to 45,000 module assignments with various module multiplicities. And, the CPU times required for these experiments range from 1.48 to 3.48 hours on a VAX 11/780 machine as shown in Table 5. Further, we also observe the following characteristics of the algorithm:

(1) Effect of T_{obj} on the stringent thread response time requirements

From Experiments #2 to #4 and #6 to #8, we note that T_{obj} is higher[1] for the cases with stricter thread response time requirements. The modules in a thread with a strict response time requirements should be allocated to lightly loaded processors in order to avoid violating the stringent thread response time specifications. Modules with less stringent response time requirements may be allocated to the more heavily loaded processors. This restricts the freedom of the search algorithm for module relocations, replications, and/or deletions. However, if the threads have less stringent response time requirements, the algorithm has more flexibility in searching for alternative assignments. Thus, the final selected module assignments may yield a lower T_{obj}.

(2) Penalty delay scaling constant

[1]The decrease of T_{obj} from Experiment #3 to #4 is because Experiment #4 is iterated with 1000 instead of 500 random initial assignments as for Experiment #3.

Experiment No.		1	2	3	4 *	5 *	6	7	8 *	9 *	10	11
Thread Response Time Requirements		R_B	R_A	R_B	R_C	R_D	R_A	R_B	R_C	R_D	R_B	R_B
Initial Module Multiplicities		α_A	α_A	α_A	α_A	α_A	α_A	α_A	α_A	α_A	α_B	α_B
Penalty Scaling Constants		1	10	10	10	10	1000	1000	1000	1000	10	1000
Thread Response Times (msec)	OS	1.741	1.394	1.699	1.653	1.678	1.441	1.586	1.616	1.678	1.642	1.767
	OV	6.051	5.894	6.052	6.136	6.153	5.657	6.236	6.309	6.131	6.044	6.261
	TI	3.204	3.186	3.173	3.101	3.169	3.247	3.171	3.128	3.160	3.167	3.189
	OT	3.710	3.639	3.638	3.667	3.743	3.657	3.578	3.767	3.738	3.687	3.776
	OD	4.977	5.057	4.954	4.797	4.848	4.556	4.772	4.798	4.844	4.990	4.995
T_{obj}		1.138	1.058	1.148	1.119	1.837	1.062	1.106	1.151	61.59	1.131	1.151
% of satisfactory local optimal assignments		15.8%	99.8%	15.9%	0.44%	0%	99.8%	14.5%	0.33%	0%	13.2%	14.3%
No. of Module Assignments Tested		21241	23868	19621	43755	43764	22960	19155	43455	45045	19572	19648
CPU Time (Hour)		1.62	1.52	1.48	3.48	3.42	1.55	1.50	3.35	3.37	1.48	1.53

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

Table 5. The Thread Response Times of the Module Assignments Generated by the RMAP Algorithm

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

Table 6 Final Module Assignments Generated for the 11 Experiments

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

Table 6 (cont.)

For threads that do not require strict response time requirements, the scaling constant does not have much affect on module assignment. However, for stringent thread response time requirements, selection of the scaling constant is critical. Experiment #1 uses a very small scaling constant (a = 1). Note that the TI thread in the Experiment violates its response time requirement, yet the algorithm is not able to detect this violation. This is because the scaling constant, a, is so small that the final sub-optimal assignment yields the minimum T_{obj} in spite of slight violations of TI thread's response time requirement. Meanwhile, there exist many assignments which can meet the specifications as indicated in Experiments #3 and #7. Therefore, the penalty scaling constant should be chosen sufficiently large so that the assignment that violates the response time specifications can be reflected in T_{obj}. The scaling constant should be selected such that T_{obj} for the final assignment, which meets the thread response time requirements, is less than that of other assignment which has one or more threads violating their response time specifications. For example, T for the Sentry System is about one msec and thread response times are a few times larger than T. Experiment results indicate that using a scaling constant equal to or greater than 10 (one order of magnitude greater than T) is large enough to detect thread response time violations. Therefore, all our experiments (except for Experiment #1) use 10 or 1000 as the scaling constant. Similar results were also obtained by using $a = 50$. The experimental results reveal that the algorithm is insensitive to the selections of the penalty delay scaling constant as long as it is larger than a certain threshold value (e.g., $10 \times T$).

(3) Insensitivity of the initial module multiplicities

Experiments #3, #7, #10 and #11 have the same thread response time requirements, R_B. Experiments #3 and #7 use α_A as initial multiplicities whereas Experiments #10 and #11 use α_B as initial multiplicities. We note the response times for the assignments generated in Experiments #10 and #11 are similar to those of #3 and #7. This indicates that the RMAP algorithm is insensitive to the initial module multiplicities provided that no single module copy for the initial

multiplicities can saturate a processor.

5.3 Validation of the RMAP Results via Simulations

To assess the performance of the module assignments generated by the RMAP algorithm, we use simulation to obtain the thread response times for the assignments from Experiments #2, #3 and #4, and compare them with the analytical predictions. Our model assumes that arrival processes are Poisson processes. However, the interarrival times of radar return signals (i.e., task/thread invocations) for the Sentry System are not exponentially distributed. The thread invocation rates (referred to as the *non-Poisson* case in the following) are time variant as shown in Figure 10. In particular, the invocation rates for OT and OD threads are highly variable with time. To evaluate the response time performance of the assignments, both Poisson and non-Poisson thread invocations are simulated, as shown in Table 7. It is interesting to observe that the simulated response times for both Poisson and non-Poisson radar signal return cases are similar to the analytical predictions. The response times for the Poisson signal arrivals match more closely with analytical predictions than those of the non-Poisson case, especially for OT an OD threads. When the task invocation arrival processes differ significantly from a Poisson process, the module assignment generated by the RMAP algorithm may produce high deviation from the response time predictions (see the OD thread in Experiment #2 in Table 7). The system designers, therefore, should analyze the task invocation arrival patterns and make appropriate calibrations. In general, however, if the task invocation rates are fairly constant in the time period of interest, the RMAP algorithm with Poisson invocations should generate module assignments that satisfy the required thread response times.

6. CONCLUSIONS

An analytic model based on the module response time model and task control-flow graph has been introduced for estimating task and thread response times for loosely coupled distributed

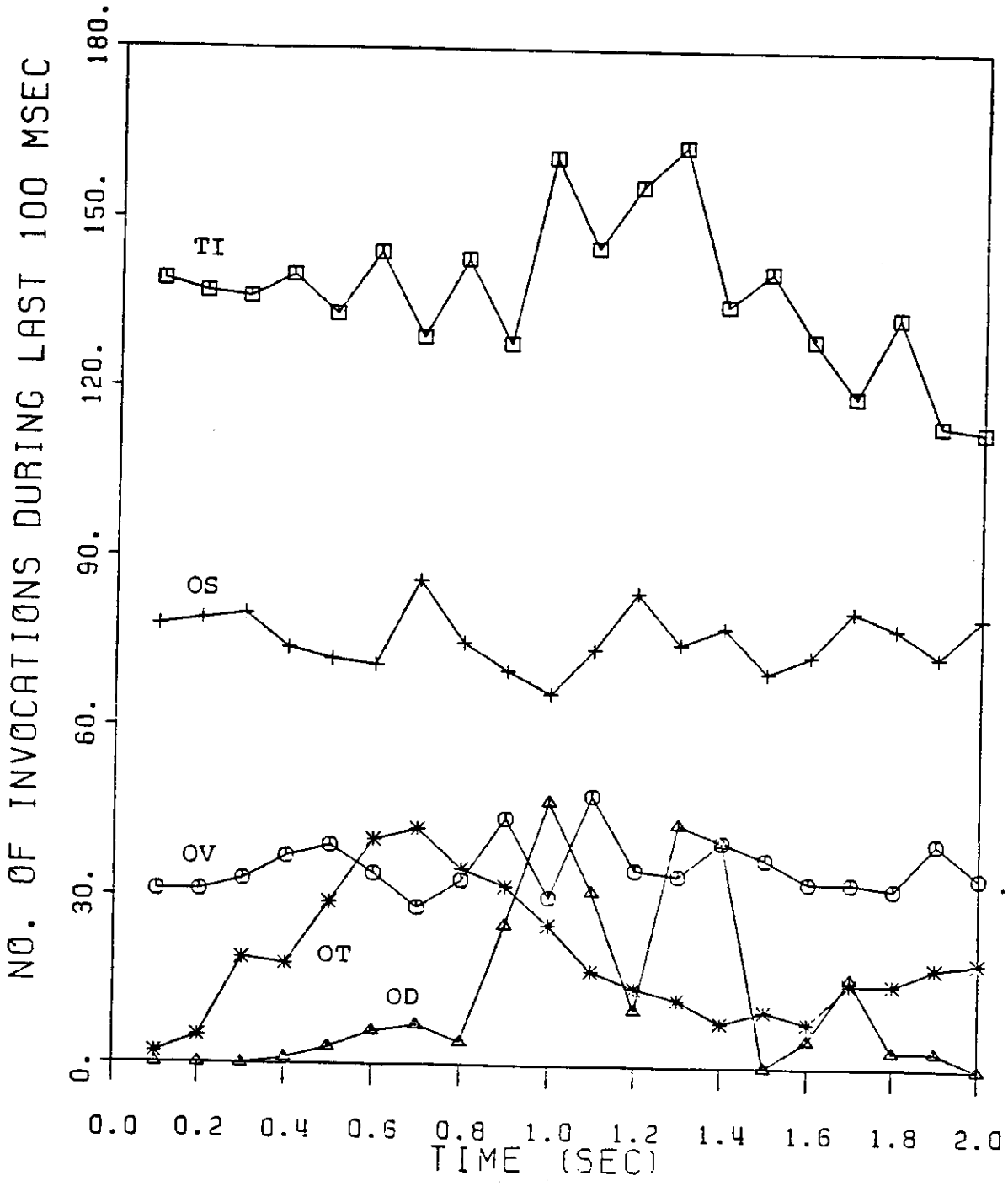


Figure 10 Thread Invocation Rates for the Sentry System

Response Time (ms) / Threads	Experiment #2				Experiment #3				Experiment #4			
	R_A	Anal. Pred.	Simulations		R_B	Anal. Pred.	Simulations		R_C	Anal. Pred.	Simulations	
			P	N-P			P	N-P			P	N-P
OS	2.5	1.39	1.03	1.01	1.8	1.70	1.66	1.84	1.75	1.65	0.98	1.04
OV	7.0	5.89	5.40	5.31	6.6	6.05	5.77	5.96	6.55	6.14	5.30	5.79
TI	4.0	3.19	3.03	3.62	3.2	3.17	3.05	3.31	3.15	3.10	2.93	3.13
OT	4.5	3.64	3.81	4.00	4.0	3.84	3.84	5.20	3.95	3.67	3.61	5.54
OD	5.5	5.06	4.63	19.56	5.0	4.95	4.94	6.22	4.95	4.80	4.57	5.07

R_A, R_B, R_C : Sets of Thread Response Time Requirements
Anal. Pred.: Analytical Predictions
P: Poisson Radar Signal Returns
N-P: Non-Poisson Radar Signal Returns

Table 7 Response Time Comparisons

systems. The model considers such factors as IPC, module precedence relationships, module scheduling, interconnection network delay, and assignment of the modules and files to computers. Based on this analytic model, we have developed a new search algorithm for module assignment for distributed systems. This algorithm uses the sum of task response time and penalty delay as the objective function, and optimizes it over all possible module assignments.

To improve load balancing and response time, certain modules may be replicated and processed on several computers. The algorithm iteratively searches for module assignments with appropriate module multiplicities which yield lower task response time yet satisfy the thread response time requirements. The search process is terminated if the objective function cannot be improved further. The search process is repeated with a prespecified number of random initial assignments. The final module assignment is then selected (based on the value of T_{obj}) from this set of feasible local optimal assignments.

The RMAP algorithm has been validated by applying it to a simple distributed system and a real-time distributed system for space defense applications. For simple distributed systems, with a very small number of initial module assignments, the local optimal assignment generated by the algorithm is equal to the optimal module assignment because of exponential growth in computation requirements. For complex systems, it is not feasible to generate the global optimal assignment by exhaustive search. Therefore, a series of experiments were performed to characterize the behavior of the algorithm. The experiments indicate that the final module assignment is rather insensitive to initial module multiplicities. The algorithm is quite robust over a wide range of the penalty delay scaling constants. Our experiments show that the proposed search algorithm is able to generate module assignments that yield satisfactory task response time yet meet the set of thread response time specifications. To assess the response time performance of the module assignments generated by the algorithm for the real-time distributed system, simulations have been performed for Poisson and non-Poisson task invocation cases.

Although the analytic model is based on Poisson arrivals, the simulation results reveal that the model in many instances can be used for approximating non-Poisson arrival cases. Few cases of deviation are noted where the input are significantly different from Poisson input arrivals. In these cases, simulation should be used to examine their response time performance. However, using the RMAP algorithm can greatly reduce the time needed to search for feasible module assignments, which is otherwise prohibitive. Therefore, the proposed algorithm is a valuable tool for module assignment with replication for distributed processing systems.

7. ACKNOWLEDGEMENT

The authors wish to thank Dr. James Huang and Mr. Dennis Townsend of Titan Systems, Inc., Los Angeles, for performing the simulation for the Sentry System.

8. REFERENCES

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

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.

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

CHU84a 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.

CHU84b W.W. Chu and K.K. Leung, "Task Response Time Model and Its Applications for Real-Time Distributed Processing Systems," *Proceedings of 5th Real-Time Symposium*, Texas, Dec. 1984, pp.225-236.

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

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

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

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

LIN65 S. Lin, "Computer Solutions of the Traveling Salesman Problem," *Bell Syst. Tech. Journal,* pp. 2245-2269, Dec. 1965.

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.

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

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, April 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, March 1985.

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

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

CHAPTER III

MODULE SCHEDULING POLICIES IN REAL-TIME DISTRIBUTED PROCESSING SYSTEMS

MODULE SCHEDULING POLICIES IN REAL-TIME DISTRIBUTED PROCESSING SYSTEMS

1 Introduction: A New Scheduling Algorithm - Batch Service with Time-out

In real-time processing systems, a task is divided into software modules. Since the task is repeatedly invoked, the modules are repeatedly invoked as well. Each processor in the distributed processing system handles repeated invocations of a set of modules.

Whenever a processor schedules a module invocation for processing, scheduling overheads are incurred. When the processor repeatedly executes invocations of a set of modules, the initialization overheads would be incurred for each invocation.

The scheduling overhead consists of process initialization and database access overheads. Among invocations of the same module, we expect locality in file and memory references, and a similar pattern in process initialization. Taking advantage of these properties, scheduling overheads can be reduced by processing invocations of a frequently invoked module in a batch. Since excessive delay can be incurred in waiting for the batch to be formed, a time-out mechanism should be used. The group of invocations of a module will be serviced when either a certain predetermined time-out period has been exceeded, or that the module invocations reach the specified batch size, whichever occurs first. This new scheduling algorithm is called *Batch Service with Time-out (BST).*

We shall compare the scheduling overhead improvement of the batch service with time-out (BST) scheduling algorithm with that of the individual invocation scheduling algorithm (e.g., first-come first-served).

2 Characteristics of BST

Let the initialization time for an single invocation of a given module be D and the average service requirement be X. Then the total processing requirement of a single invocation,

$$y = D + X. \tag{1}$$

The total processing time requirement for processing m invocations (if individually invoked and processed) of the same module is

$$my = m(D + X). \tag{2}$$

In order to analyze batch scheduling algorithm, the scheduling overhead needs to be further decomposed into two components: fixed and incremental. Fixed scheduling overhead is the portion of initialization overhead that is repeated for each invocation of a given module. For example, the process for the initial set up of a processor to execute an invocation of a module is likely to be the same. In this case, fixed scheduling overhead is that portion of initialization overhead which can be shared among processors if several invocations of the same module are batched and executed as a single unit.

Incremental scheduling overhead, on the other hand, is the portion of initialization overhead incurred by each invocation. A good example is the memory and file references performed during the execution of module invocations. Hence, incremental scheduling overhead is not affected by the BST scheduling algorithm.

Therefore, the batch scheduling overhead of m processes, D', is the sum of the fixed scheduling overhead, D_f, (independent of batch size) and the incremental overhead, D_v, that results from the scheduling of more than one invocation at a time. Hence,

$$D' = F + mV. \tag{3}$$

The total processing requirement of a batch of m invocations is

$$Y = mX' + D'. \tag{4}$$

The overhead improvement of batch service over individual invocation scheduling can be obtained from Eqs. (2) and (4).

$$\Delta y = my - Y = (mD - D') + m(X - X') \tag{5}$$

When m invocations are batched and processed together, due to the locality of data and file references, the batch service time, X′, and scheduling overhead, D′, should be much less than if the invocations are processed individually. That is, $0 \le X' \le X$ and $D \le D' \le mD$. Thus, we expect an improvement in response time from batch services with time out. However, the magnitude of overhead improvement depends on the relationship between D, D′, F and V.

Since $D' = F + mV$

then $D'(m=1) = F + V$ for m=1.

D′(m=1) may not equal D since two different scheduling algorithms may have different scheduling procedures. For example, D′(m=1) and D are related by a factor k:

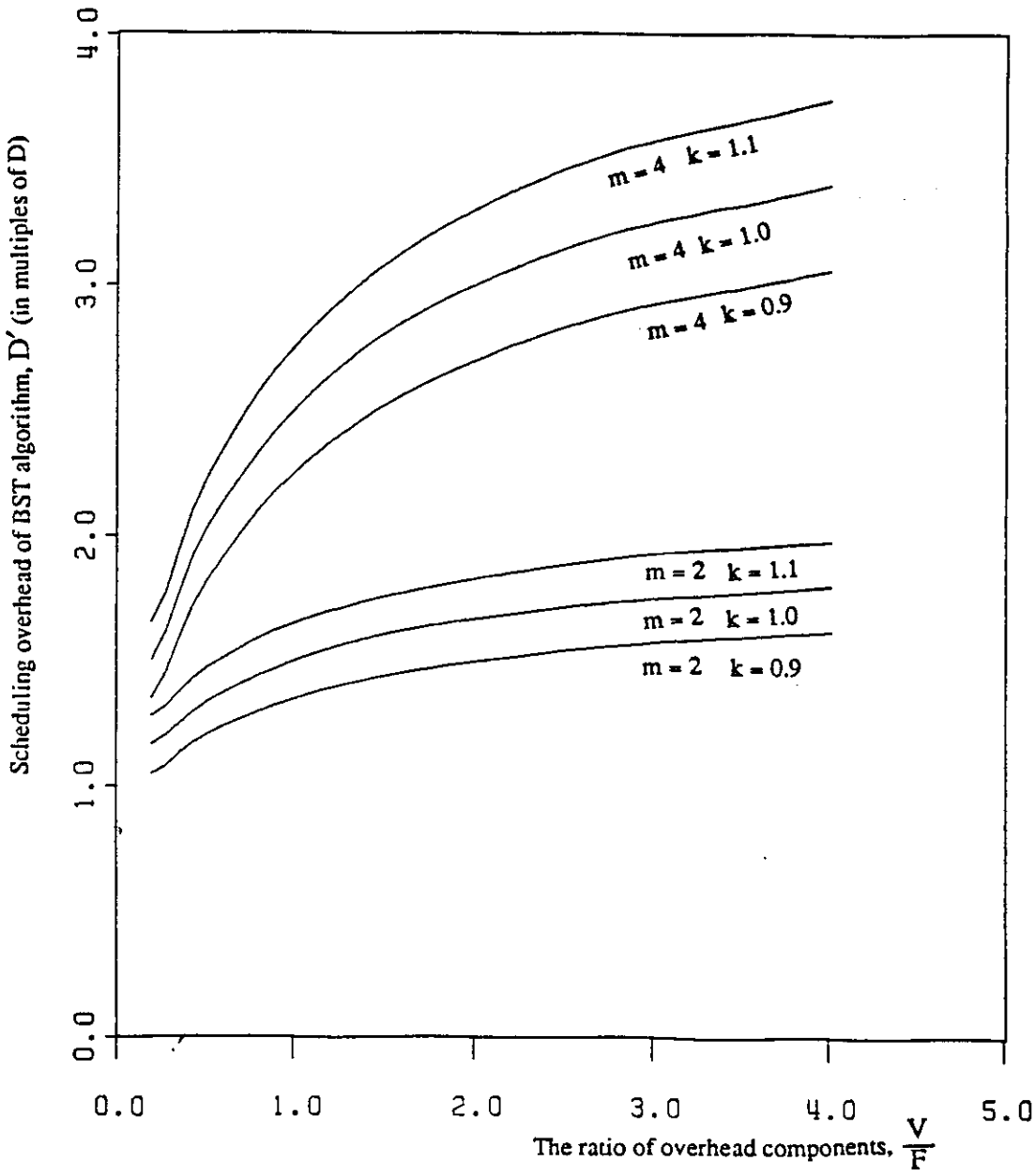
$$D'(m=1) = kD \tag{6}$$

The relationship between D′ in terms of D and the ratio $\frac{V}{F}$ is shown in figure 1. In the figure, D′(m=1) = kD is plotted for k = 0.9, 1.0, 1.1 versus the ratio $\frac{V}{F}$.

A small value of $\frac{V}{F}$ indicates a large portion of scheduling overhead results from fixed overhead. On the other hand, a large value of $\frac{V}{F}$ indicates that only a small portion of the scheduling overhead results from fixed overhead. Hence, there is more potential overhead improvement in the region of small $\frac{V}{F}$ ratio values. In the figure, the potential overhead improvement is the difference between a curve and the line D′ = mD. Notice that the gap closes up as $\frac{V}{F}$

gets larger. Also, the larger the m (m = 2,4 in this case), the gap which indicates potential overhead improvement becomes wider. That means more overhead improvement can be made by batching more invocations of modules with larger fixed scheduling overhead. However, the response time improvement of BST scheduling algorithm depends not only on overhead improvement, but also on the delay incurred in forming a batch of module invocations. Both factors must be considered in the analytical model of the scheduling algorithm.

Figure 1 The Relation Between Batching Scheduling Overhead and Ratio of the Overhead Components



3 Analytical Model

3.1 Task Response Time Model

A task response time model has been developed for loosely coupled real-time distributed processing systems with FCFS scheduling algorithm [CHU84]. The model has been validated by simulations and yields good response time estimates for several real systems. This model is extended to analyze the BST scheduling algorithm for such systems.

The task response time model consists of two submodels: the module response time model and the weighted control flow graph model. The first submodel computes the module response times, while the latter considers the precedence waiting times.

For a given module assignment, each processor 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 processing time. Based on the module assignment and IMC's among modules, IPC times can be obtained and included as part of the module response time. Hence, each processor can be modeled as a single server queueing system that have several modules (customers of different types) with specified service distributions [BASK75]. The invocation rate of each module can be obtained from the task invocation rate and the control flow graph.

The model assumes that 1) the module invocation arrival processes are independent of each other, 2) module invocation interarrival times are exponentially distributed, 3) FCFS scheduling policy for module invocation processing, 4) constant initialization cost. Based on these assumptions, each single server queueing system (processor) is an extension of the regular FCFS M/G/1 queue. Thus the mean and variance of module response times can be estimated [LAVE83].

The other component of task response time is precedence waiting time. Precedence waiting time is the intermodule synchronization delay resulting from the precedence relationships among modules. The model considers the precedence waiting times by mapping the mean and variance of the module response times onto the control flow graph as arc weights (figure 1). Since the interconnection network delay is independent of module response times, the mean and variance of network delay can be added to the arc weights.

According to the logical structures and precedence relationships among modules, there are different types of control flow subgraphs. Each of these subgraphs can be reduced to a single node graph. Such successive graph reduction yields the estimation of the task response time.

3.2 Model Extensions

In order to enhance the task response time model to represent BST scheduling algorithm, we need to add analytical representation of the batching and time-out mechanism. In addition, we need to further subdivide module processing time into module scheduling time and module execution time. We must be able to derive from the model the module response time as a function of maximum batch size, M, time-out constant, T_o, module invocation rate, module scheduling overheads and module execution requirement.

Since the precedence relationship among task modules is not affected by the new scheduling algorithm, the precedence waiting times remain unchanged. However, the new module response times are different under BST scheduling algorithm. Task response time estimate can be obtained by mapping these new estimates of module response time to the control flow graph model.

The new module response time model is shown in figure 2. The server in the model represents one of the processors in the distributed processing system. The processor (server)

consists of two parts: scheduling and execution. The scheduling part of the processor is used to model the overhead, D′, which is a function of F,V and m. While the execution part represents the module execution requirement, X′. In the model, there is a processor queue and a set of batch queues. The set of batch queues, numbered from 1 to k, models the batching of invocations of the k modules that are assigned to the processor.

The module invocations of the i^{th} module, λ_i, consist of external invocations λ_i' and internal invocations, λ_i''. The external invocations are generated from other processors in the system. The internal invocations are generated from modules residing in the same processor, which will be generated from the feedback loop. Invocations of i^{th} module arrive and queue at the i^{th} batch queue.

The queues are serviced either when the queue size reaches a pre-determined maximum batch size for that module; or when the time elapsed since the first module invocation in the queue exceeds the time-out constant. If the first condition is true, the processor will serve M invocations. Otherwise, the processor will serve all the module invocations in the queue which will be less than M, the maximum batch size. Different modules (queues) may have different maximum queue size or time-out constants. The selection of M and T_o for different modules depends on the corresponding module invocation rate and module response time requirements.

The batches of module invocations departing from the batch queues are routed to the processor queue. Each batch of module invocations will be processed as a single unit by the processor. They are serviced in FCFS order. λ_i represents the arrival rate of batches of invocations of the i^{th} module to the processor queue.

In the task response time model, module response time is the sum of module waiting time and module processing time. While module processing time in this case is extended to include both the module scheduling and execution time, the module waiting time can be further subdi-

vided into module waiting time at processor queue. All the components of module response time are represented in figure 2. Because of the complexity of the queueing system model, an exact closed form analytical solution can not be obtained. Therefore, we shall impose certain assumptions to simplify the queueing system representation to make it mathematically tractable. This allows us to conveniently derive an approximate analytical solution for estimating the mean and variance of module response times for all modules. These results are then used to construct a weighted control flow graph for the estimation of task response time.

3.3 Components Of Module Response Time

In our new module response time model for the BST scheduling algorithm, module response time consists of four components. For the i^{th} module, these components are 1) waiting time at batch queue with time-out, (W_{Bi}); 2) waiting time at processor queue, (W_{Pi}); 3) scheduling time, (D'_i); 4) execution time, (X'_i).

The module response time of the i^{th} module, T_i, can be expressed as the sum of the four components:

$$T_i = W_{Bi} + W_{Pi} + D'_i + X'_i. \tag{7}$$

Since these components are independent of each other, mean module response time for module i is the sum of the mean value of each component.

$$E[T_i] = E[W_{Bi}] + E[W_{Pi}] + E[D'_i] + E[X'_i] \tag{8}$$

where i = 1, 2, ... , k for the k modules assigned to the processor. Thus we need to calculate the mean of each component. In addition, the construction of weighted control flow graph requires not only the mean but also the variance of the module response time. The variance of the module response time can be expressed as

$$Var[T_i] = Var[W_{Bi}] + Var[W_{Pi}] + Var[D'_i] + Var[X'_i] \tag{9}$$

for i = 1, 2, ... , k.

We shall use the following equality to obtain the variances :

$$Var[Y] = E[Y^2] - (E[Y])^2 \tag{10}$$

where Y is a random variable and $E[Y^2]$ denotes the second moment.

3.4 Waiting Time At Batch Queue With Time-out

For each module, invocations arrive and queue at the corresponding batch queue with time-out. The arrival processes of module invocations to the batch queues are non-Poisson. However, for mathematical tractability, we assume Poisson arrival processes in order to estimate the mean and second moment of waiting time.

In this section, we present the analytical result of invocation waiting time at a batch queue with time-out and Poisson arrival. Because we assume Poisson arrival processes, the result can apply to all batch queues in our model independently. Hence, the subscript i that has been used to index different modules will be dropped. We shall use our results to analytically approximate the waiting time of batch queue with non-Poisson arrival process.

List of notations :

M : Maximum batch size

T_o : Time-out constant

λ_B : Arrival rate to batch queue (assumed Poisson)

W_B : Waiting time at batch queue

$E[W_B]$: Expected waiting time at batch queue

$E[W_B^2]$: Second moment of waiting time at batch queue

S_B : Batch size

$E[S_B]$: Expected batch size

n_B : number of module invocations in the queue as seen by an arrival

3.4.1 Estimation of Mean Waiting Time at Batch Queue

Mean waiting time at batch queue with time-out can be expressed as the sum of two conditioned expected waiting times weighted with corresponding probabilities. The expected waiting time is conditioned on whether the queue is empty or nonempty as seen by an arrival.

$$E[W_B] = E[W_B \mid n_B=0]P[n_B=0] + E[W_B \mid n_B>0]P[n_B>0] \tag{11}$$

Case I : $n_B = 0$, arrival sees an empty batch queue (figure 3.3):

(a) $E[W_B \mid n_B=0]$

When an arrival sees an empty queue, it means that it is the first arrival to the queue. Its waiting time depends on whether a batch of maximum size is formed (figure 3.3a) or if the time-out constant is reached before that occurs. If a batch of maximum size is formed, the arrival has to wait for the time that it takes for (M-1) invocations to arrive; if time-out is exceeded (figure 3.3b), the arrival has to wait the whole time-out interval. The time interval required to collect multiple arrivals from a Poisson process can be calculated with the Erlangian probability density function [CHAU83].

$$E[W_B \mid n_B=0] = \int_{t=0}^{T_o} t\, P[t < \text{time of } (M-1) \text{ invocation arrivals} \le t+dt]$$

$$+ \int_{t=T_o}^{\infty} T_o\, P[t < \text{time of } (M-1) \text{ invocation arrivals} \le t+dt]$$

$$= \int_0^{T_o} t \frac{\lambda_B(\lambda_B t)^{M-2}}{(M-2)!} e^{-\lambda_B t} dt + \int_{T_o}^{\infty} T_o \frac{\lambda_B(\lambda_B t)^{M-2}}{(M-2)!} e^{-\lambda_B t} dt$$

$$= \frac{\lambda_B{}^{M-1}}{(M-2)!} \int_0^{T_o} t^{M-1} e^{-\lambda_B t} dt + \frac{\lambda_B{}^{M-1} T_o}{(M-2)!} \int_{T_o}^{\infty} t^{M-2} e^{-\lambda_B t} dt$$

$$= \frac{(M-1)}{\lambda_B} - \frac{(M-1)}{\lambda_B} \sum_{k=0}^{M-1} \frac{(\lambda_B T_o)^k}{k!} e^{-\lambda_B T_o} + T_o \sum_{k=0}^{M-2} \frac{(\lambda_B T_o)}{k!} e^{-\lambda_B T_o} \qquad (12)$$

(b) $P[n_B = 0]$

Since only the first invocation of a batch could arrive to find an empty batch queue, then,

$$P[n_B = 0] = \frac{1}{E[S_B]} \qquad (12)$$

where $$E[S_B] = \sum_{n=0}^{M} n\, P[S_B=n] = M\, P[S_B=M] + \sum_{n=0}^{M-1} n\, P[S_B=n]$$

i) $P[S_B = M] = P[$ at least $M-1$ invocations arrive in $T_o\ ] = \sum_{k=M-1}^{\infty} \frac{(\lambda_B T_o)^k}{k!} e^{-\lambda_B T_o}$

$$= 1 - \sum_{k=0}^{M-2} \frac{(\lambda_B T_o)^k}{k!} e^{-\lambda_B T_o}$$

ii) $P[S_B=n] = P[$ $n-1$ invocations arrive in $T_o\ ]$ for $n=1,2,\cdots,M-1$

$$= \frac{(\lambda_B T_o)^{n-1}}{(n-1)!} e^{-\lambda_B T_o}$$

Hence

$$E[S_B] = M\,[\,1 - \sum_{k=0}^{M-2} \frac{\lambda_B T_o{}^k}{k!} e^{-\lambda_B T_o}\,] + \sum_{n=1}^{M-1} n\, \frac{(\lambda_B T_o)^{n-1}}{(n-1)!} e^{-\lambda_B T_o}$$

$$= M[1-\sum_{k=0}^{M-2} \frac{(\lambda_B T_o)^k}{k!} e^{-\lambda_B T_o}] + \sum_{k=0}^{M-2} (k+1) \frac{(\lambda_B T_o)^k}{k!} e^{-\lambda_B T_o}$$

$$= M - \sum_{k=0}^{M-2} (M-k-1) \frac{(\lambda_B T_o)^k}{k!} e^{-\lambda_B T_o} \qquad (14)$$

From (3.7), we have :

$$P[n_B=0] = \left[M - \sum_{k=0}^{M-2} (M-k-1) \frac{(\lambda_B T_o)^k}{k!} e^{-\lambda_B T_o} \right]^{-1} \qquad (14)$$

Case II : $n_B > 0$, arrival sees a non-empty batch queue (figure 3.4):

(a) $E[W_B \mid n_B>0]$

The expected waiting time in this case depends upon :

- the number of invocations in the queue, which indicates how many arrival are needed in order to form a batch of maximum size;
- the time of the latest arrival, which indicates the maximum time interval allowed.

$$E[W_B \mid n_B>0] = \sum_{n=1}^{M-1} \int_{y=0}^{T_o} E[W_B \mid n_B=n, y< \hat{t} \leq y+dy] P[n_B=n, y< \hat{t} \leq y+dy \mid n_B>0] \qquad (15)$$

$\hat{t}$ = the instant the latest invocation arrives.

$E[W_B \mid n_B=n, y< \hat{t} \leq y+dy]$ = mean waiting time given an invocation sees n invocations in the queue upon arrival at $y< \hat{t} \leq y+dy$.

When an invocation arrives at the instant y and finds n already in the queue, it must wait for either M-n-1 invocations to arrive to form a batch of maximum size (figure 3.4a), or the maximum delay $T_o - y$ if time-out is exceeded (figure 3.4b).

$E[W_B \mid n_B=n, y< \hat{t} \leq y+dy]$

$$= \int_{t=0}^{T_o-y} t\, P[t < \text{time of } (M-n-1) \text{ invoation arrivals} \le t+dt]$$

$$+ \int_{t=T_o-y}^{\infty} (T_o-y) P[t < \text{time of } (M-n-1) \text{ invocation arrivals} \le t+dt]$$

$$= \int_{t=0}^{T_o-y} t \frac{\lambda_B(\lambda_B t)^{M-n-2}}{(M-n-2)!} e^{-\lambda_B t} dt + \int_{t=T_o-y}^{\infty} T_o \frac{\lambda_B(\lambda_B t)^{M-n-2}}{(M-n-2)!} e^{-\lambda_B t} dt$$

$$= \frac{(M-n-1)}{\lambda_B} - \frac{(M-n-1)}{\lambda_B} \sum_{k=0}^{M-n-1} \frac{[\lambda_B(T_o-y)]^k}{k!} e^{-\lambda_B(T_o-y)} + \frac{1}{\lambda_B} \sum_{k=0}^{M-n-2} \frac{[\lambda_B(T_o-y)]^{k+1}}{k!} e^{-\lambda_B(T_o-y} \quad (3.11)$$

When an invocation arrives at the instant y to find n already the queue, it becomes the $(n+1)^{th}$ arrival to the queue in time interval y. Accordingly, the probability is equivalent to that of having exactly n+1 arrivals in time interval y.

$$P[y< \hat{t} \le y+dy, n_B=n \mid n_B>0] = \frac{P[y< \text{time of n invocation arrivals} \le y+dy]}{C}$$

$$= \frac{\frac{\lambda_B(\lambda_B y)^{n-1}}{(n-1)!} e^{-\lambda_B y} dy}{C} \quad (3.12)$$

where C is the normalization constant and :

$$C = \sum_{n=1}^{M-1} \int_0^{T_o} \frac{\lambda_B(\lambda_B y)^{n-1}}{(n-1)!} e^{-\lambda_B y} dy$$

By substituting results of (3.11) and (3.12) into (3.10), we get:

$$E[W_B \mid n_B>0] = \frac{1}{C} \sum_{n=1}^{M-1} [\frac{(M-n-1)}{\lambda_B} - \frac{(M-n-1)}{\lambda_B} \sum_{k=0}^{n-1} \frac{\lambda_B T_o{}^k}{k!} e^{-\lambda_B T_o}$$

$$- \frac{(M-n-1)(\lambda_B T_o)^{n-1} T_o e^{-\lambda_B T_o}}{(n-1)!} \sum_{k=0}^{M-n-1} (\lambda_B T_o)^k \sum_{j=0}^{k} \frac{(-1)^j}{(k-j)!j!(n+j)}$$

$$+ \frac{(\lambda_B T_o)^{n-1} T_o e^{-\lambda_B T_o}}{(n-1)!} \sum_{k=0}^{M-n-2} (\lambda_B T_o)^{k+1} \sum_{j=0}^{k+1} \frac{(k+1)(-1)^j}{(k+1-j)!j!(n+j)} \;] \qquad (3.13)$$

(b) $P[\, n_B > 0 \,]$

The corresponding probability of an arrival finding a non-empty queue is simply the complement of the probability that an invocation arrives at empty queue (case I-(b)).

$$P[\, n_B > 0 \,] = 1 - P[\, n_B = 0 \,]$$

The expression for $E[W_B]$ could be obtained by substituting the results of cases I and II into equation (1). The resulting equation expresses expected waiting time at the batch queue with time-out in terms of the three parameters of the batch queue : arrival rate, maximum batch size, and time-out constant.

3.4.2 Estimation of Second Moment of Waiting Time at Batch Queue

In many real time situations where certain time constraints must be met, we need information about the variation among waiting times at batch queue in addition to the average value. Therefore, we need to estimate the second moment of the waiting time in order to obtain the variance. The same approach could be used to estimate the second moment of waiting time at batch queue. We start with the following expression similar to (1) :

$$E[W_B^2] = E[W_B^2 \mid n_B=0]P[n_B=0] + E[W_B^2 \mid n_B>0]P[n_B>0] \qquad (16)$$

Case I : $n_B = 0$

$$E[W_B^2 \mid n_B=0] = \int_{t=0}^{T_o} t^2 \, P[t < \text{time of } (M-1) \text{ invocation arrivals} \le t+dt]$$

$$+ \int_{t=T_o}^{\infty} T_o^2 \, P[t < \text{time of } (M-1) \text{ invocation arrivals} \le t+dt]$$

$$= \frac{M(M-1)}{\lambda_B^2} - \frac{M(M-1)}{\lambda_B^2} \sum_{k=0}^{M} \frac{(\lambda_B T_o)^k}{k!} e^{-\lambda_B T_o} + T_o^2 \sum_{k=0}^{M-2} \frac{(\lambda_B T_o)}{k!} e^{-\lambda_B T_o} \qquad (3.15)$$

Case II : $n_B > 0$

$$E[W_B^2 \mid n_B>0] = \sum_{n=1}^{M-1} \int_{y=0}^{T_o} E[W_B^2 \mid n_B=n, y< \hat{t} \le y+dy] P[n_B=n, y< \hat{t} \le y+dy \mid n_B>0] \qquad (3.16)$$

with

$$E[W_B^2 \mid n_B=n, y< \hat{t} \le y+dy] = \int_{t=0}^{T_o-y} t^2 \, P[t < \text{time of } (M-n-1) \text{ invoation arrivals} \le t+dt]$$

$$+ \int_{t=T_o-y}^{\infty} (T_o-y)^2 P[t < \text{time of } (M-n-1) \text{ invocation arrivals} \le t+dt]$$

$$= \frac{(M-n)(M-n-1)}{\lambda_B^2} - \frac{(M-n)(M-n-1)}{\lambda_B^2} \sum_{k=0}^{M-n} \frac{[\lambda_B (T_o-y)]^k}{k!} e^{-\lambda_B (T_o-y)}$$

$$+ \frac{1}{\lambda_B^2} \sum_{k=0}^{M-n-2} \frac{[\lambda_B (T_o-y)]^{k+2}}{k!} e^{-\lambda_B (T_o-y)} \qquad (3.17)$$

Substituting (3.17) into (3.16) and simplying, we have :

$$E[W_B^2 \mid n_B>0] = \frac{1}{C} \sum_{n=1}^{M-1} [\frac{(M-n)(M-n-1)}{\lambda_B^2} - \frac{(M-n)(M-n-1)}{\lambda_B^2} \sum_{k=0}^{n-1} \frac{\lambda_B T_o^k}{k!} e^{-\lambda_B T_o}$$

$$- \frac{(M-n)(M-n-1)(\lambda_B T_o)^{n-2} T_o^2 e^{-\lambda_B T_o}}{(n-1)!} \sum_{k=0}^{M-n} (\lambda_B T_o)^k \sum_{j=0}^{k} \frac{(-1)^j}{(k-j)!j!(n+j)}$$

$$+ \frac{(\lambda_B T_o)^n T_o^2 e^{-\lambda_B T_o}}{(n-1)!} \sum_{k=0}^{M-n-2} (\lambda_B T_o)^k \sum_{j=0}^{k+2} \frac{(k+2)(k+1)(-1)^j}{(k+2-j)!j!(n+j)} \Bigg] \qquad (3.18)$$

The expression for second moment of waiting time at batch queue could be obtained by substituting expressions (3.15) and (3.18) into (3.14).

3.4.3 Batch Queue Simulations

In order to check the validity of our analytical result, we have performed a set of simulations. The simulations are performed using a queueing network based simulation package PAWS. A queue with batching and time-out mechanism is simulated. The customer arrival process to the queue is specified to be Poisson with an average rate of 10 customer arrivals per simulation unit time ($\lambda = 0.1$). Simulations are performed with different sets of values for maximum batch size and time-out constant. Statistics on waiting time and batch size leaving the queue are collected from 20,000+ customers in each simulation. The results from both the simulations and our analysis are summarized in table 3.1. The analytical results are very close to the simulation results. Thus, our analysis is validated by the simulations.

3.5 Estimation Of Module Scheduling Time

Due to the random nature of module invocation arrival processes and the time-out mechanism, the batch size of invocations leaving the batch queue varies. Since batching does not begin until the arrival of the first module invocation, batch size takes on non-zero values. While the maximum batch size determines the maximum number of invocations a batch can have.

The scheduling time for a batch of size m (of module i) is given by,

$$D'_i(S_{Bi}=m) = F_i + mV_i \tag{17}$$

The average scheduling time for this batch is simply $\frac{F_i + mV_i}{m}$. Thus, the mean scheduling time for module i is the sum of average scheduling times of all possible batch sizes and each weighted with the corresponding probability of such a batch size occurring.

$$E[D'_i] = \sum_{m=1}^{M_i} \frac{(F_i + mV_i)P[S_{Bi}=m]}{m} \tag{18}$$

From the analysis of batch queue with time-out, the probability of the occurrence of batch size m is given by

$$P[S_{Bi}=M_i] = 1 - \sum_{k=0}^{M_i-2} \frac{(\lambda_{Bi}T_{oi})^k}{k!} e^{-\lambda_{Bi}T_{oi}} \qquad \text{for } m = M_i \tag{19}$$

$$P[S_{Bi}=m] = \frac{(\lambda_{Bi}T_{oi})^{m-1}}{(m-1)!} e^{-\lambda_{Bi}T_{oi}} \qquad \text{for } m = 1,2,....,M_i-1 \tag{20}$$

3.6 Estimation Of Mean Waiting Time At Processor Queue

Consider the model in figure 3.2. Batches of invocations of the k modules are assigned to the processor queue for processing. By viewing each batch of invocation as a job to the processor, we have a single server queueing system with multiple job classes with general arrival processes. The number of job classes is equal to k, the number of modules assigned to the processor for processing. We further assume that the job arrival process for each job class is an independent Poisson arrival process. The problem is reduced to a multiple job classes M/G/1 queueing system [BASK75]. The Pollaczek-Khinchin formula for the mean and variance of waiting time for M/G/1 queue [KLEI76] can be used to estimate the mean and variance of waiting time at the processor queue in the model.

$$E[W_P] = \frac{\lambda \overline{X^2_P}}{2(1-\lambda \overline{X_P})} \tag{21}$$

$$Var[W_P] = (E[W_P])^2 + \frac{\lambda \overline{X^3_P}}{3(1-\lambda \overline{X_P})} \tag{22}$$

where λ is the average total arrival rate to the processor queue; $\overline{X_P}$, $\overline{X^2_P}$, and $\overline{X^3_P}$ are the mean, second and third moments of the service time of the processor.

Average total arrival rate, λ, is equal to the sum of the average arrival of each independent Poisson arrival. In the model, the average arrival rate to the processor queue is equal to the average departure rate of batches from the batch queue.

$$\lambda = \sum_{i=1}^{k} \lambda_i \tag{23}$$

The three moments of the processor service time can be obtained from the transform of the service time distribution, $B^*(s)$. For a multiple job class queueing system, service time distribution can be obtained by summing the transform of service time distribution of each job class, weighted by the ratio of its job arrival rate to the total job arrival rate.

$$B^*(s) = \sum_{i=1}^{k} \frac{\lambda_i}{\lambda} Y^*_i(s) \tag{24}$$

where $Y^*_i(s)$ is the transform of the processing time distribution for batched invocation of module i.

As we have indicated, the transform of processing time distribution is made up of scheduling and execution time distributions. Both of which vary according to the batch size. Since we assume that the execution time distributions of modules in the task are given a priori, the batched invocations of the transform of processing time distribution for batched invocations of module i is

$$Y^*_i(s) = \sum_{m=1}^{M_i} [e^{-s(F_i+mV_i)}(X'^*_i(s))^m]P[S_{Bi}=m] \qquad (25)$$

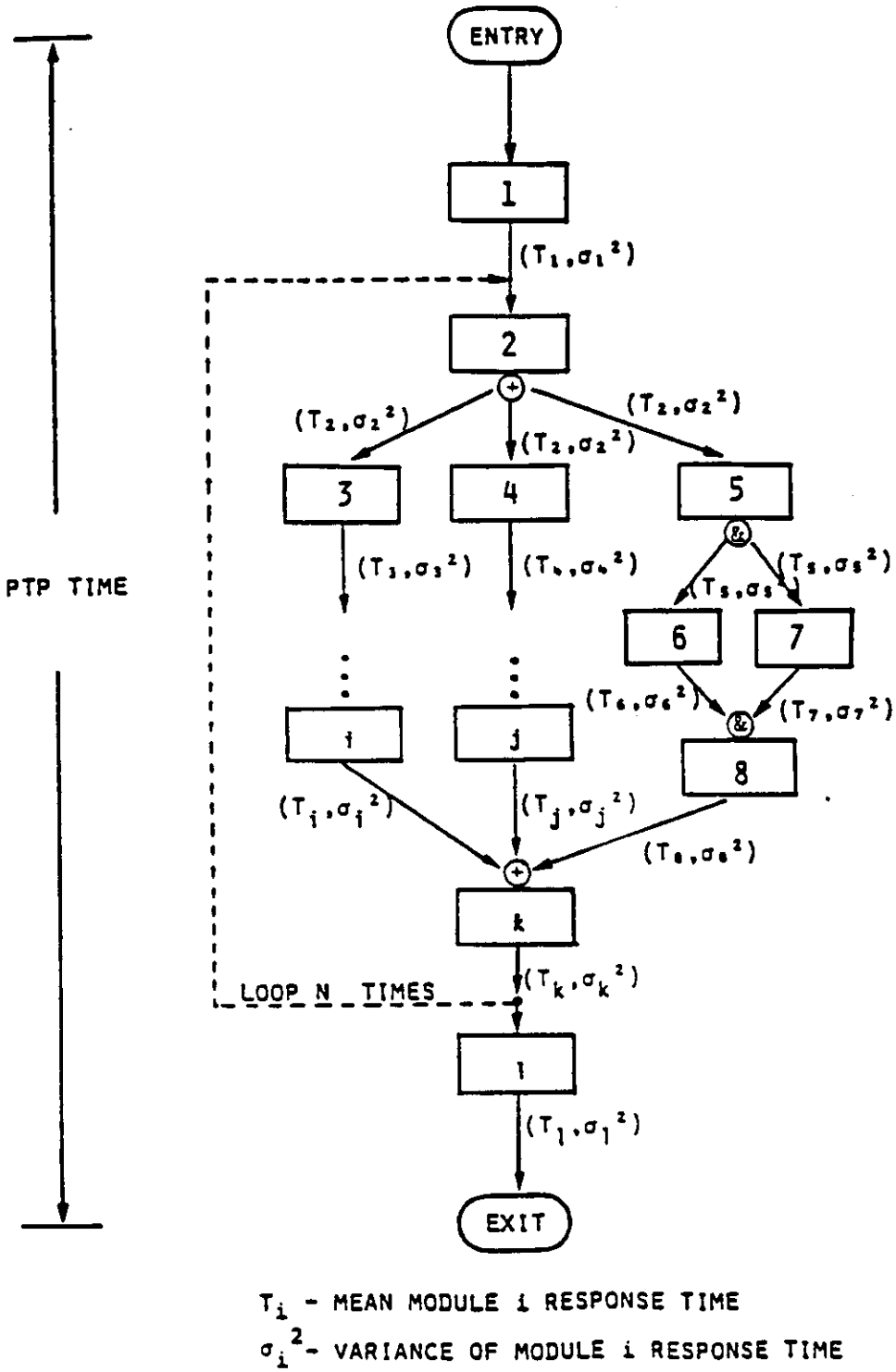


Figure 2 Weighted Control-Flow Graph for Response Time Estimation.

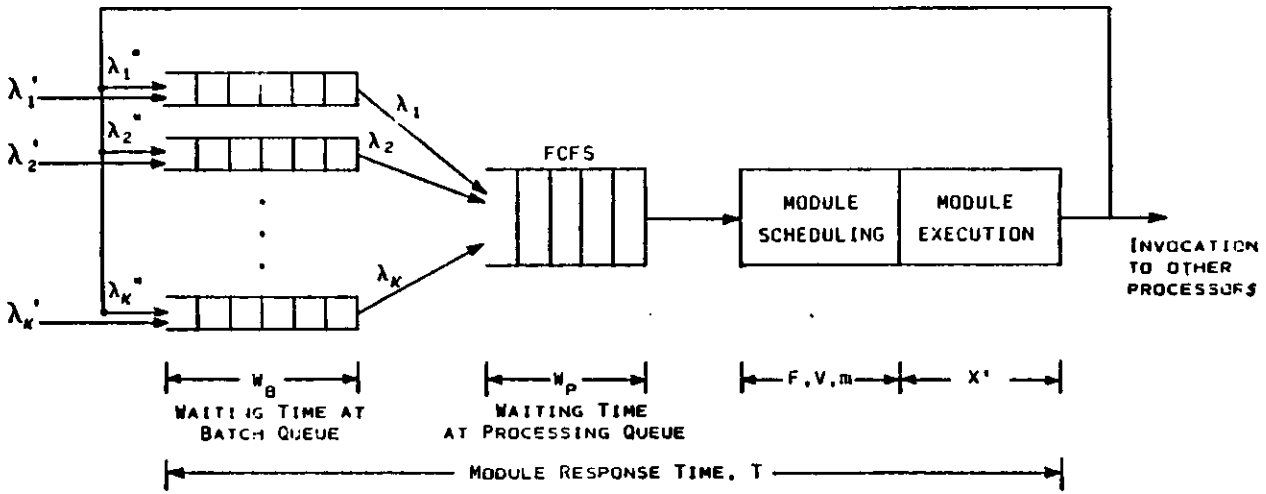
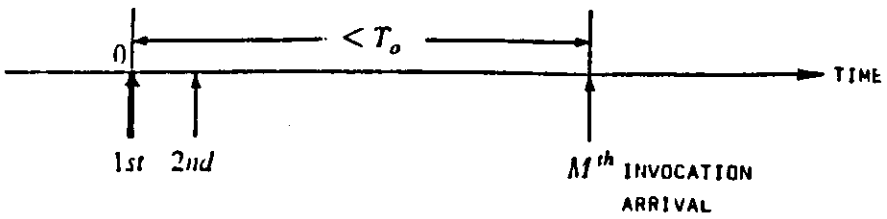


Figure 3 A Model for A Processor with k Modules that uses BST Scheduling Algorithm.

first case : $n_B = 0$, (1st arrival)

a) BATCH FORMED WITH MAXIMUM SIZE



b) BATCH FORMED WITH TIME-OUT EXCEEDED

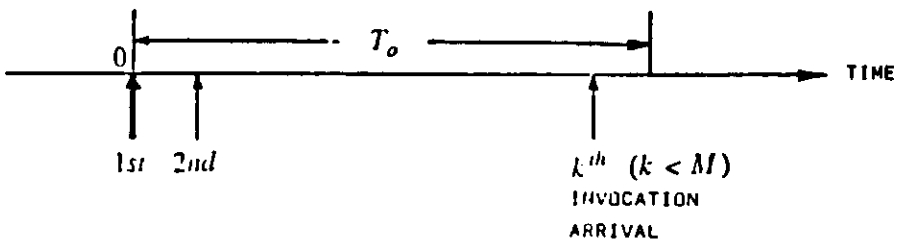
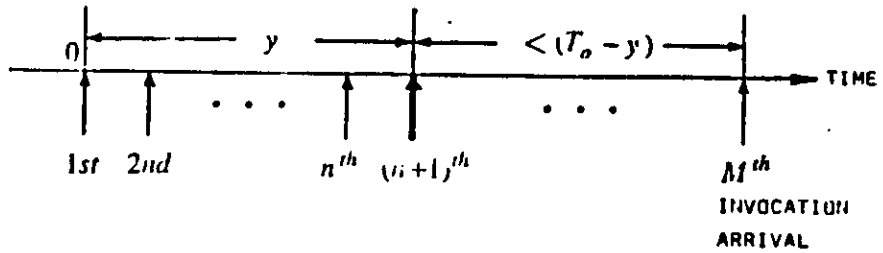


Figure 4 **Timing Diagram of Batch Queue Waiting Time of the First Invocation**

second case : $n_B = n > 0$

$n_B = 1,2, \ldots, M-1$

a) BATCH FORMED WITH MAXIMUM SIZE



b) BATCH FORMED WITH TIME-OUT EXCEEDED

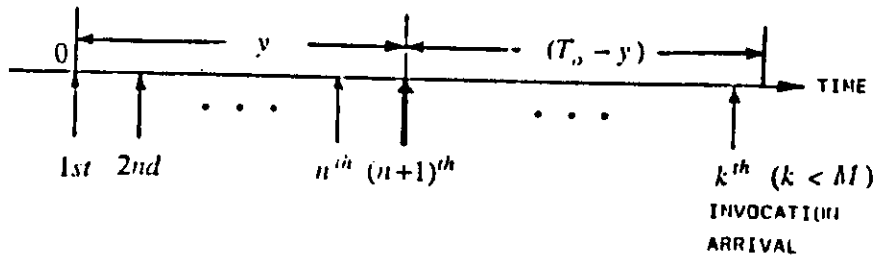


Figure 5 **Timing Diagram of Batch Queue Waiting Time of k^{th} invocation**

INVOCATION ARRIVAL RATE, $\lambda_B = 0.1$

		SIMULATION RESULTS		ANALYTICAL SOL.	
M	To	$E[S_B]$	$E[W_B]$	$E[S_B]$	$E[W_B]$
2	30	1.95	4.897	1.96	4.895
4	30	3.32	12.97	3.33	12.87
5	60	4.80	19.30	4.78	19.29
5	40	4.23	17.59	4.26	17.57
8	90	7.60	33.74	7.61	33.62
8	70	7.00	31.61	7.00	31.43
8	60	6.49	29.34	6.49	29.51

Table 1 **Comparation of Analytical and Simulation Results on Batch Queue with Time-out and Poisson arrival Process.**

4 Model Applications

4.1 Task Scheduling for Distributed Systems

Consider the sample task control flow graph shown in figure 3.1. The task is a example of a real-time application job on a distributed system. It consists of fifteen modules governed by a precedence relationship represented by the control flow graph. The task is processed on a three processor loosely coupled distributed system. We shall apply the model to two examples using the same control flow graph but different module assignment, module execution times and overheads. In both examples, we assume constant scheduling overhead and exponentially distributed execution times.

The module execution times, overhead, and module assignment used in our first example is shown in table 4.1. Total overhead is fixed at 20% of processing time for all modules. Fixed and variable overhead each take up 50% of the total overhead. This module assignment yields a fairly balanced load for the distributed system.

We apply the analytical model to this example to estimate the task response time under both BST and FCFS scheduling algorithms. The response time estimates are obtained for different maximum batch sizes and time-out constants of the module batch queues. The task is repeatedly invoked with Poisson arrival process. The validity of the analytical model is checked by comparing with results from PAWS simulations.

The comparison of the analytical and simulation results is shown in figure 4.1. Our analytical results agree with simulations to within 15% in most cases with the exception of very high invocation rates. Our simulation results agree closely with the behavior of BST algorithm in this example, despite the fact that the independent Poisson process assumption for module invocation is not appropriate for high invocation rates due to the fact that module invocations occur at different processors which introduces extra randomness, our

The two algorithms' utilization of bottleneck processor is compared in figure 4.2. Under this module assignment, the bottleneck processor is CPU 2. The figure shows the utilization of this processor under different invocation rate. For both algorithms, the resulting graphs are straight lines. The slope of these lines is the weighted mean processing load to the processor:

$$\sum_i \frac{\lambda_i}{\lambda}(\overline{X'_i} + \overline{D'_i})$$

where

$\overline{X'_i}$ is the mean module i execution time;

$\overline{D'_i}$ is the mean module i overhead;

λ_i is the mean module i invocation rate;

λ is the sum of invocation rate of modules assigned to the bottleneck processor.

Under the FCFS algorithm, this quantity is a constant and is independent of invocation rate. It depends on the ratio of module invocation rates which is unchanged for the control flow graph. With the BST algorithm, a straight line relationship indicates mean batch sizes remain unchanged under different invocation rates. This results from defining time-out constants as a constant factor of the reciprocal of the module invocation rate. For the low overhead in this example, this set of module maximum batch sizes and time-out constants do not allow the BST algorithm to yield substantial saving in utilization as shown in figure 4.2.

In the second example, the fixed overhead is increased to a significant portion of the processing time for each module. The module execution times and overhead are specified in table 4.2a. The modules are assigned in such a way that CPU 2 is again the bottleneck processor. In addition, the processing load handled by CPU 2 is significantly heavier than those handled by the other two CPUs. The module assignment is given in table 4.2b. Note that the four most frequently invoked modules are assigned to CPU 2. Hence, we want to batch invocations of these four modules to lower the utilization and significantly reduce waiting time at the bottleneck pro-

cessor. Furthermore, delay at the batch queue can be minimized by employing batching mechanism only to these more frequently invoked modules of the task. Different batch sizes and time-out constants are chosen.

The results are compared in figure 4.3. The bottleneck processor saturates at much lower invocation rate under FCFS algorithm than in the previous example. With BST algorithm, the processor can sustain a high task invocation rate. The larger the average batch sizes, the more overhead can be saved, and the more the utilization can be reduced. In another word, the larger the batch size, the higher is the task invocation rate that is needed to saturate the system. However, larger batch size results in more delay at low task invocation rates. In this example, time-out constants of module batch queue are again expressed as a factor of the reciprocal of module invocation rates. As in the previous example, this results in the linear increase of utilization with task invocation rate shown in figure 4.4.

Figures 5 and 6 show that utilization of the bottleneck processor is reduced as task invocation rate increases. The waiting time at batch queue decreases as the invocation rate is increased. BST outperforms FCFS after a critical invocation rateis reached. This critical invocation rate increases with larger batch sizes.

4.2 Scheduling for Disk I/O

The second example applies the BST to a disk I/O processor. Since disk I/O access time is often several order of magnitude larger than execution time, the BST algorithm can be applied to substantially reduce disk I/O overhead. The algorithm should yield lower response time for a larger range of task invocation rates than in the previous example.

The model for this example has a central processor servicing jobs that require disk I/O with little computation. In addition, there is a batch and a processor queue. Jobs arrive from a single source and depart from the system upon being serviced. The system parameters are given

as follows:

Fixed Disk I/O Access Overhead = 1

Variable Disk I/O Overhead = 0.01

Mean Job Execution Time = 0.001

The additional parameters that need to be specified are the M and T_o of the batch queue. The analytical model is applied to obtain the mean job response time. Comparisons are made between FCFS and BST algorithm with different M and T_o. The comparisons are shown in figures 4.5 and 4.6.

From figure 4.5, we can see that BST has two advantages over the FCFS algorithm. The first advantage is the response time improvement for the high invocation rate, although FCFS yields lower mean response time during low invocation rate. The second advantage is that the BST algorithm enables the system to sustain heavier load and reach saturation at a substantially higher invocation rate.

Notice in figure 4.6 that as both the maximum batch size and time-out constant increase, the response time at low invocation rates increase while at high invocation rates it decreases. At low invocation rates, larger maximum batch size and longer time-out would increase the waiting time at batch queue. Thus mean response time increases for BST with larger M and T_o. However, the delay at batch queue is reduced as the average batch size increases under high invocation rate. Both factors contribute to the response time improvement with the BST algorithm with larger M and T_o.

Modules	Mean Execution Time (Execution time is exponentially distributed)	Fixed Overhead	Variable Overhead
1,2,3,4,5	0.08	0.01	0.01
6,7,8,9,10	0.16	0.02	0.02
11,12,13,14,15	0.24	0.03	0.03

Table 2a **Module Execution Times and Overheads For First Example.**

CPU 1	CPU 2	CPU 3
M_1, M_5, M_7, M_9, M_{12}, M_{13}	M_2, M_4, M_8, M_{11}, M_{15}	M_3, M_6, M_{10}, M_{14}

Table 2b **Module Assignment For First Example.**

Modules	Mean Execution Time (Execution time is exponentially distributed)	Fixed Overhead	Variable Overhead
1,2,3,4,5	0.03	0.06	0.01
6,7,8,9,10	0.06	0.12	0.02
11,12,13,14,15	0.09	0.18	0.03

Table 3a **Module Execution Times and Overheads For Second Example.**

CPU 1	CPU 2	CPU 3
M_3, M_4, M_5, M_6, M_7, M_8, M_9	M_1, M_2, M_{14}, M_{15}	M_{10}, M_{11}, M_{12}, M_{13}

Table 3b **Module Assignment For Second Example.**

Figure 6 Comparison of Analytical and Simulation Results
(Distributed System Application Task Example of Table 1).

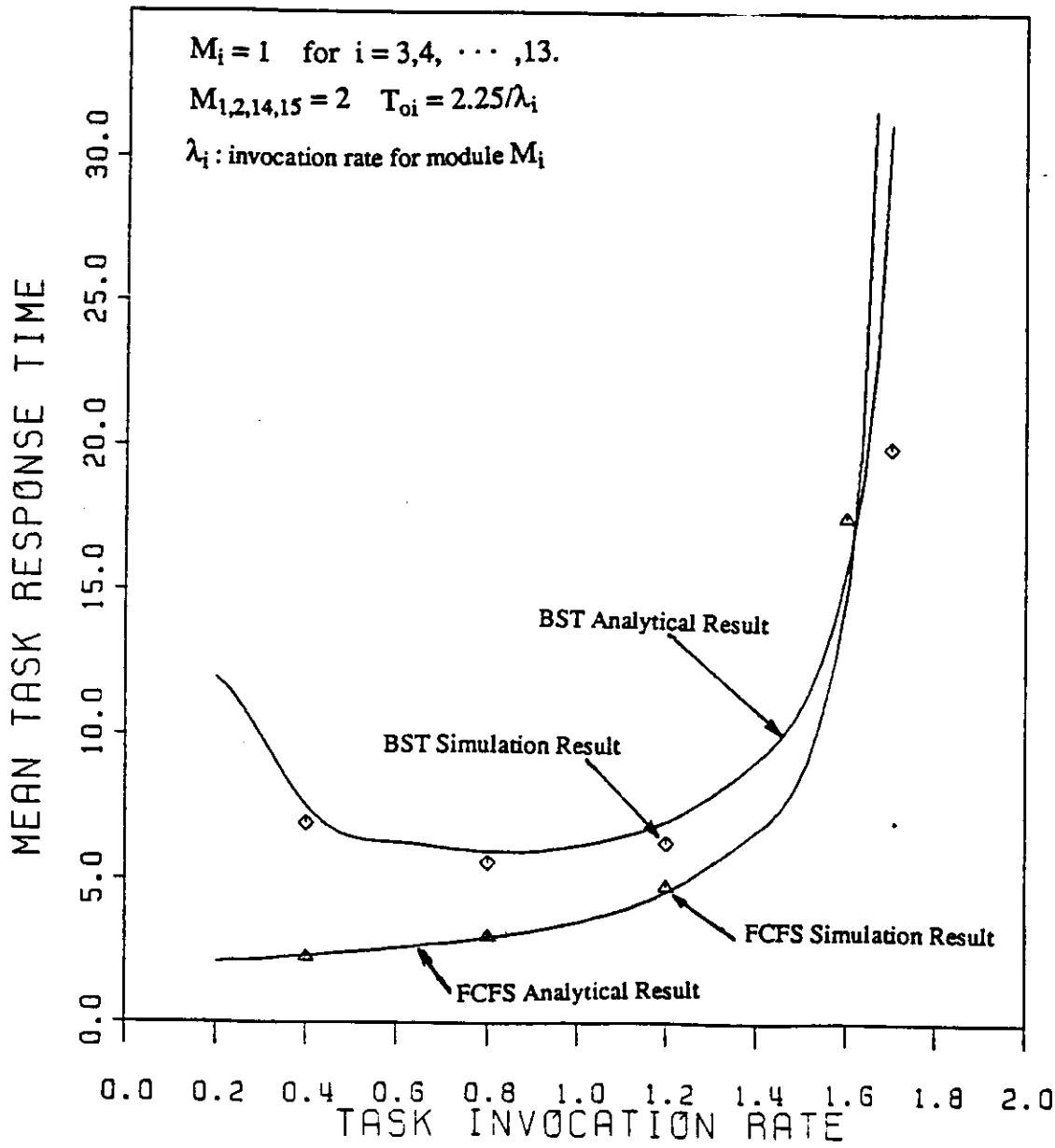


Figure 7 Utilization of Bottleneck Processor
(Distributed System Application Task Example of Table 1).

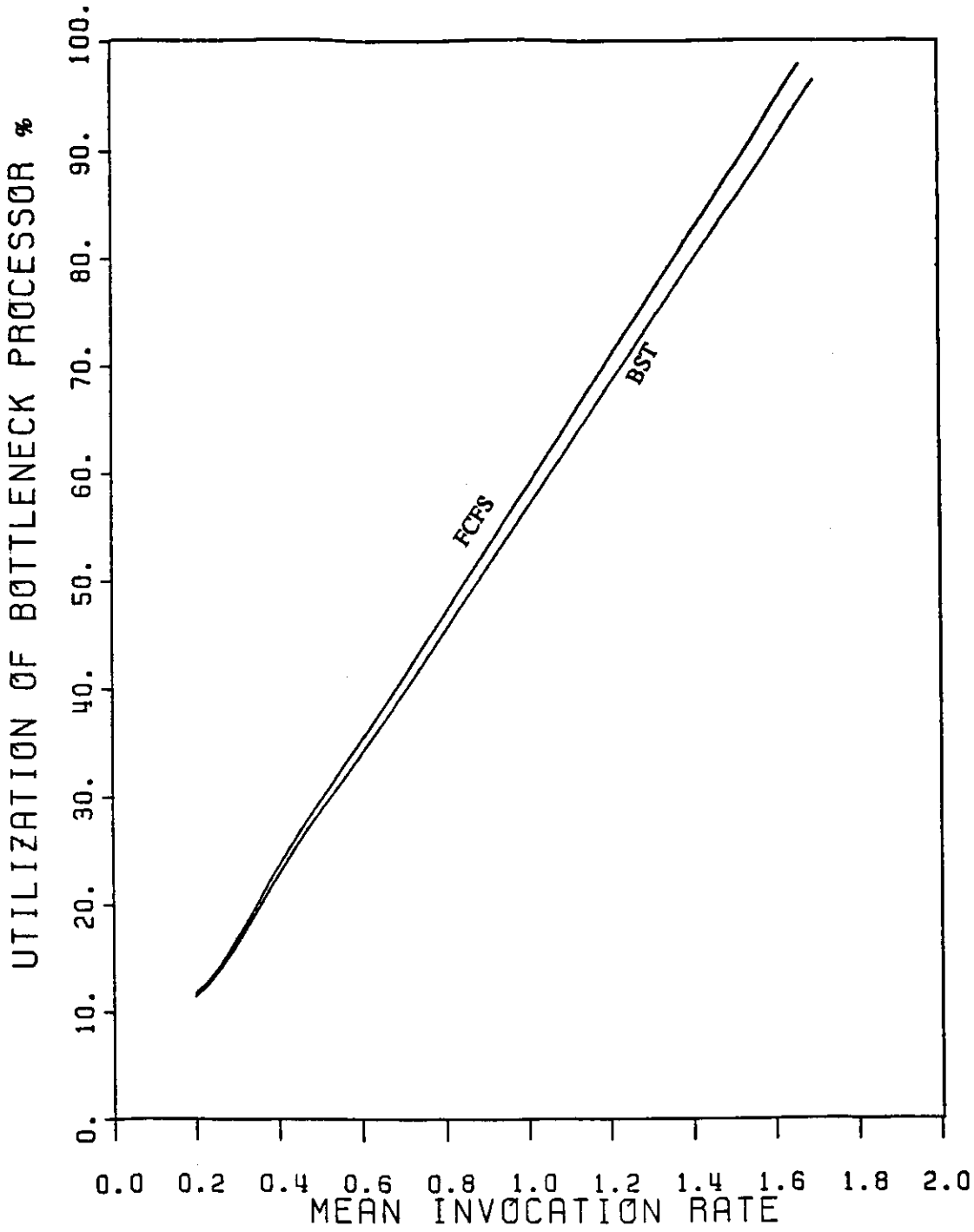


Figure 8 Comparison of Analytical Results
(Distributed System Application Task Example of Table 2).

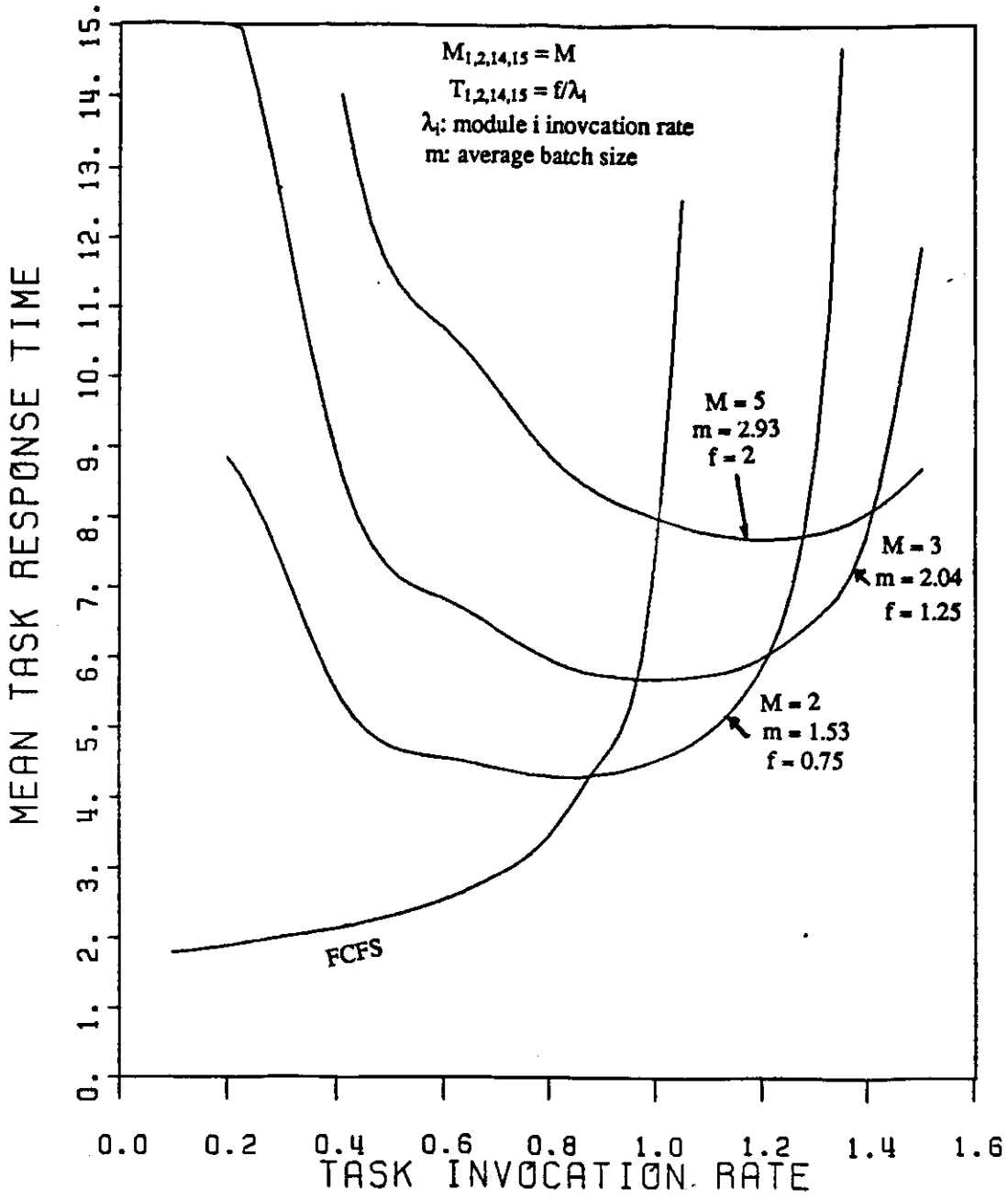


Figure 9 Utilization of Bottleneck Processor
(Distributed System Application Task Example of Table 2).

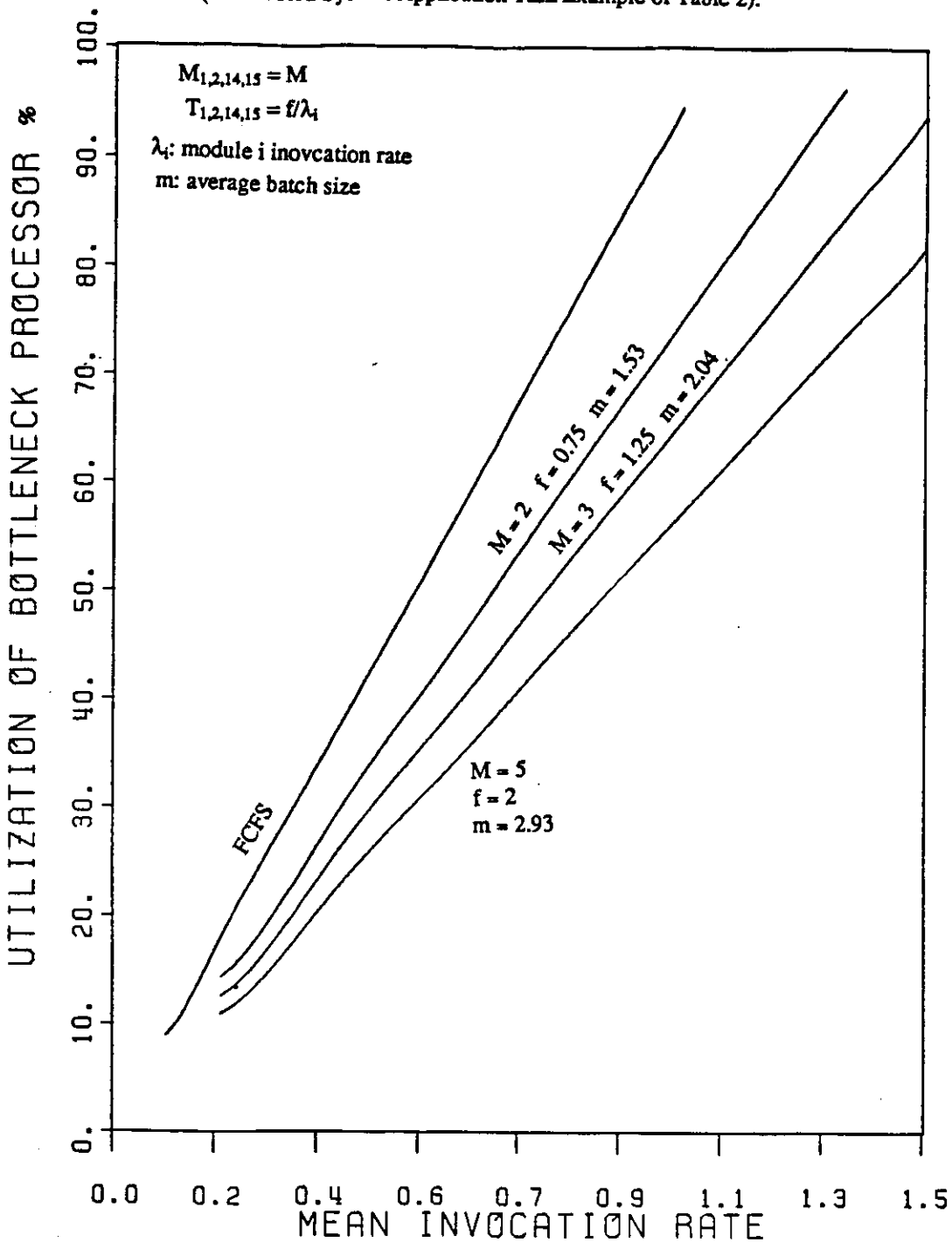


Figure 10 Comparison of FCFS and BST Algorithm for Disk I/O Example.

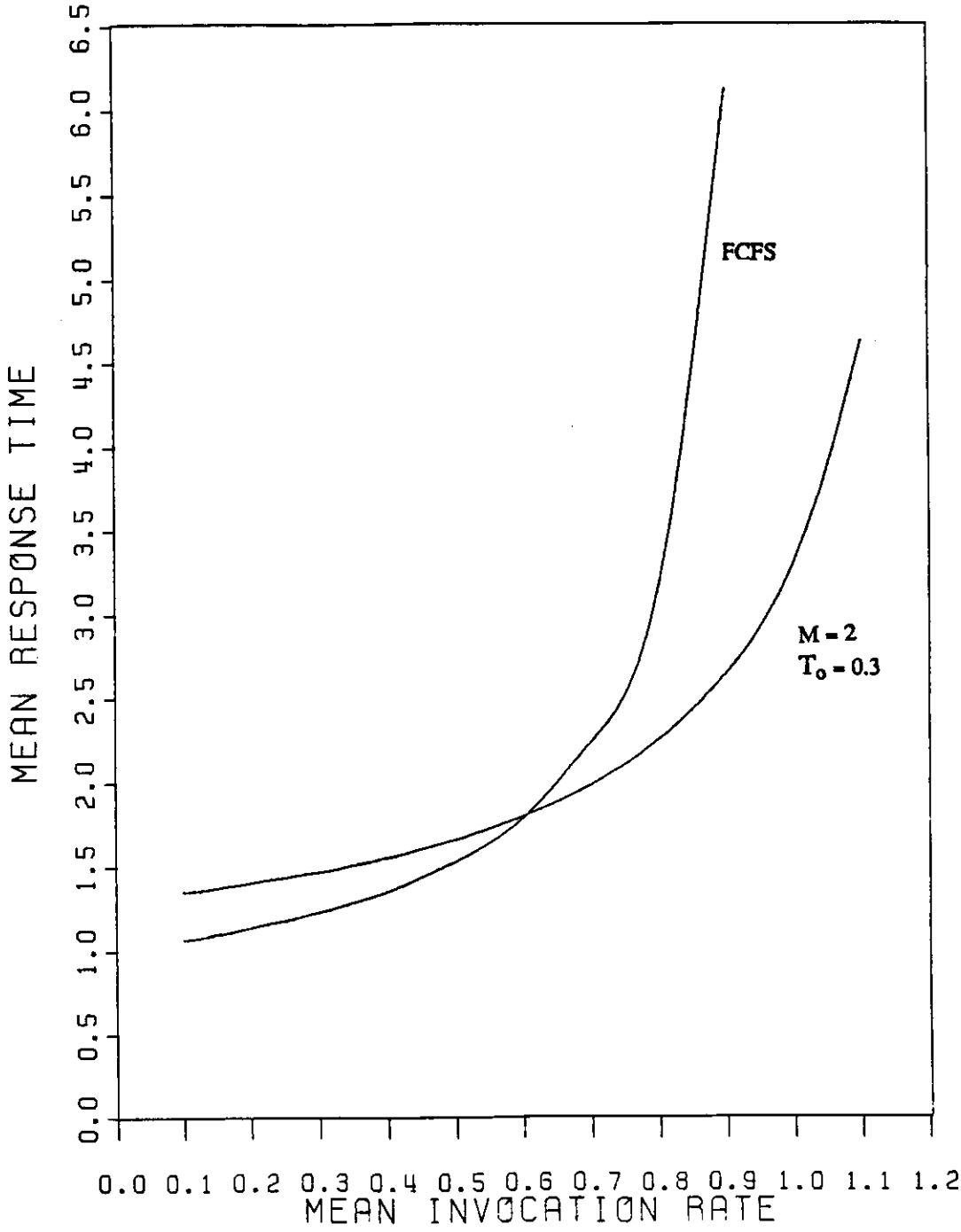
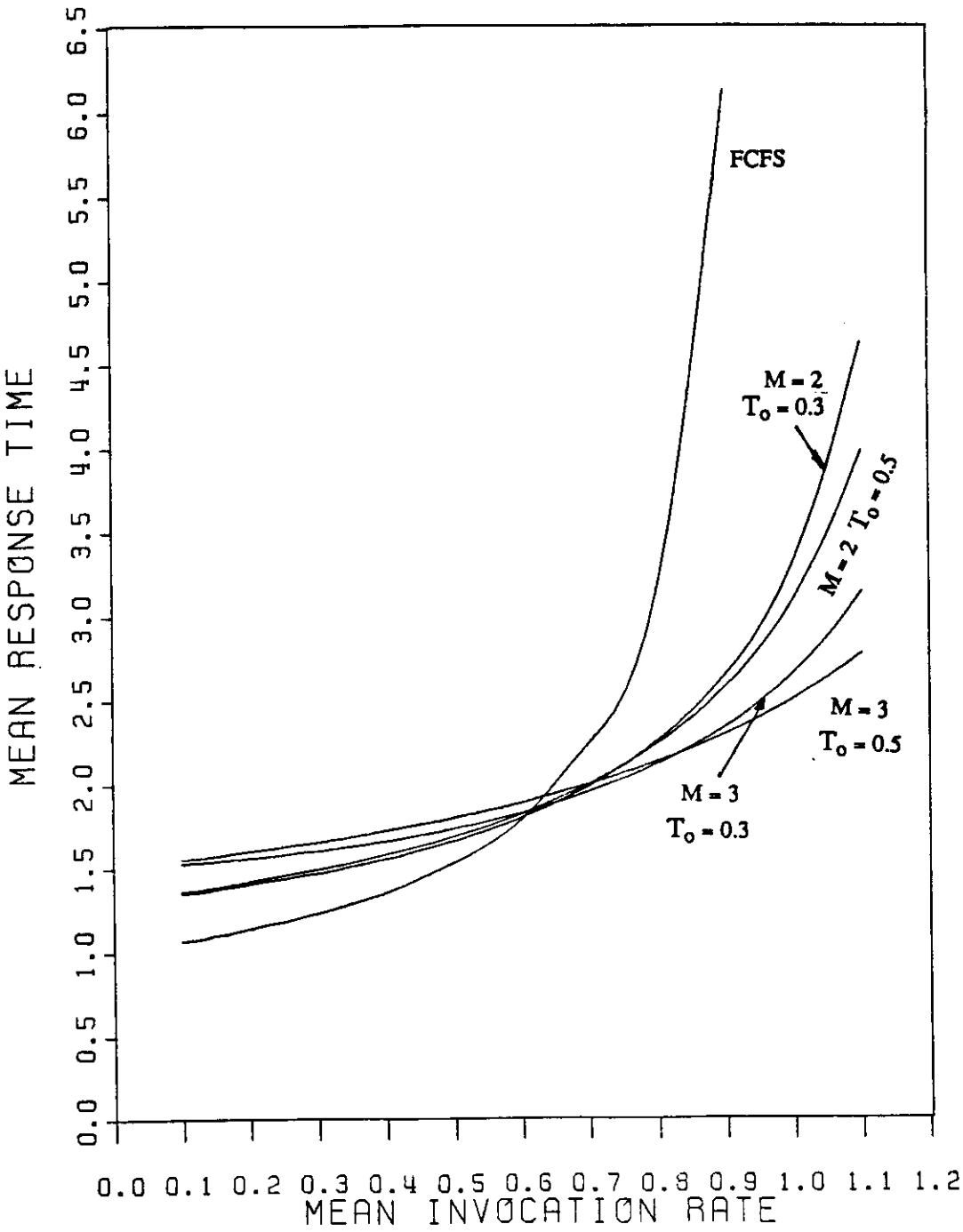


Figure 11 Comparison of BST Algorithm with Different M and T_0 for Disk I/O Example.



5 Conclusions And Future Work

A new scheduling algorithm that is based on Batch Service with Time-out (BST) is proposed. An analytical model for this algorithm which provides reasonably good estimates of task response time is developed. Comparing the performance of using BST with that of First Come First Served (FCFS). we note that the amount of improvement of BST over FCFS algorithm depends on the ratio of fixed overhead to incremental overhead and the level of system load. For example, with a high fixed to incremental overhead and a heavy system load, such as disk I/O access, BST performs significantly better than the FCFS algorithm. In addition to response time improvement, the BST algorithm enables the system to sustain a heavier load and tolerate a substantially higher task invocation rate before it reaches saturation. This scheduling algorithm can be applied in many application areas.

There are two areas where future work is needed. 1) For a given application task, optimal M and T_o need to be determined for module batch queues in order to attain minimum response time. This problem is complicated by the precedence relationships among modules and the large number of parameters that affect task response time. 2) In this study, module invocation process is assumed to be Poisson. Since batching mechanism is used, the module invocations are grouped and processed together. Thus, bulk Poisson process would be a more accurate module invocation representation. Carrying out the analysis with bulk Poisson process would yield more accurate performance prediction.

REFERENCES

[BASK75] F. Basket, K. M. Chandy, R. Muntz, F.G. Palacios, " Open, Closed, and Mixed Networks of Queues with Different Classes of Customers," *Journal of the ACM*, April 1975.

[BERR82] R. Berry, K. M. Chandy, J. Misra, and D. Neuse, *PAWS 2.0 Performance Analyst's Workbench System*, Dec. 1982.

[CHAU83] M. L. Chaudhry, J. G. C. Templeton, *A First Course in Bulk Queues*, John Wiley & Sons, 1983.

[CHU84] W. W. Chu, K. K. Leung, "Task Response Time Model & Its Applications For Real Time Distributed Processing Systems," *Proceedings of the Real Time Systems Symposium*, Dec. 1984.

[KLEI76] L. Kleinrock, *Queueing System Volume II: Computer Application*, Wiley, New York, 1976.

[LAVE83] S. S. Lavenberg (Editor), *Computer Performance Modeling Handbook*, New York: Academic Press, 1983.

CHAPTER IV

PERFORMANCE STUDY OF FAULT TOLERANT LOCKING

PERFORMANCE STUDY OF FAULT TOLERANT LOCKING

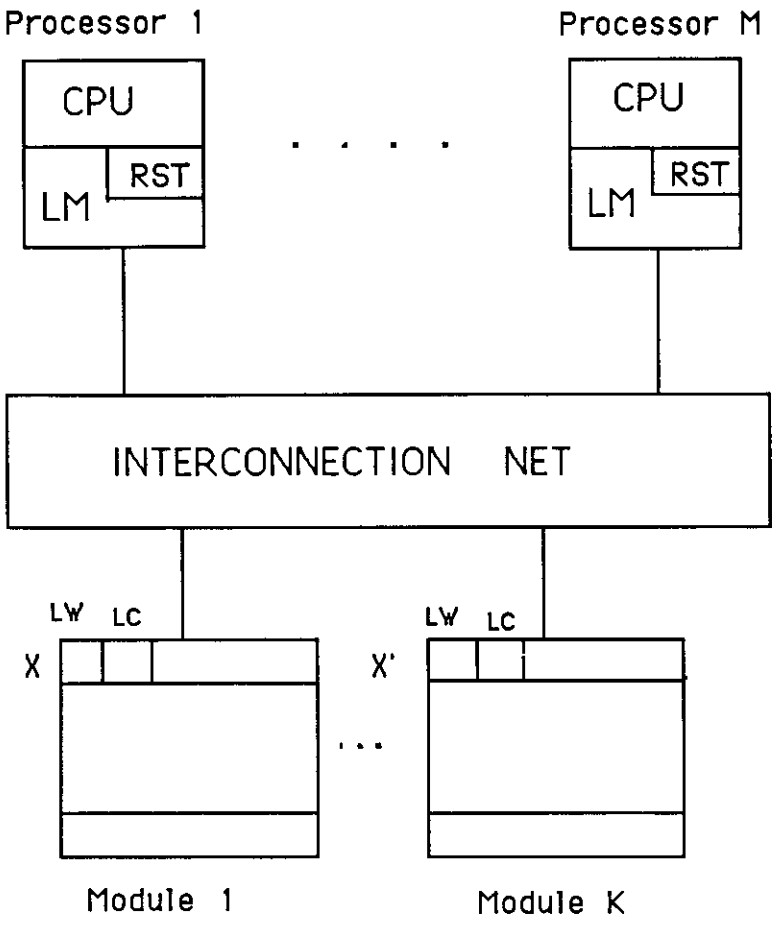
1. INTRODUCTION

In a tightly coupled distributed processing system, records are replicated and stored in shared memory modules to provide fault tolerance. Data updates should be applied to all copies to assure mutual consistency among them. If a failure occurs 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 failure, Fault Tolerant Locking (FTL) is proposed [CHU86]. In this paper, an analytic model is developed to study the effect of system parameters on FTL performance.

In this chapter we shall describe FTL operations, present a finite population model and a method for estimating the lock grant rate, lock grant time and average transaction response time for a given operating environment. Finally, numerical results are used to show the effect of system parameters on FTL performance.

2. FTL OPERATIONS

To implement FTL in a tightly coupled system, shared records are replicated in two shared memory modules as shown in Fig. 1. Thus, we obtain the primary copy and the secondary copy. Each processor maintains a record status table (RST) in its local memory to indicate the accessibility of the record copies. A lock word (LW) and lock count (LC) are appended to each record copy. LW indicates whether the copy is free (LW=0), locked (LW=1), being updated (LW=2) or failed (LW=3). The LC of a record copy informs us whether any other processors have locked this copy during two successive lock requests.



LM: Local Memory

X : Primary Copy

X' : Secondary Copy

RST: Record Status Table

Fig. 1 Tightly Coupled System with FTL

Before accessing a record copy, a processor first checks the RST to determine if the copy is accessible. If it is, the processor reads the LW and determines the status of the copy. If the copy is free, the processor will lock the copy by setting LW to 1 and then increasing the LC by 1. If this copy is "locked" or "update-initiated", the processor starts a fault detection procedure using a timeout method. Each processor has a timeout counter which indicates how many times the processor tries to lock the copy. In this procedure, the timeout counter is compared with timeout constant. The processor holding the copy is considered to have failed if a lock request is not granted before the timeout constant is exceeded. In addition, the processor checks the LC in this procedure. The purpose of doing so is to find out whether any other processor locks the copy during two succesive lock trials. If the LC is changed, the processor resets the timeout counter to prevent a false fault alarm. The processor then retries after a prespecified period (retry period). If the LW indicates the record is "failed", the processor records this information in its RST to avoid further access to this record.

After the record is locked, the processor copies the record from the primary module into its local memory and performs the required execution. To update the record, the processor sets the LW of both copies to "update-initiated" and perform the update. After completing the updates on both copies, the processor releases the lock on them and sets the LW to zero.

3. THE MODEL

3.1 Model Assumptions and Formulation

In our FTL model, we shall make the following assumptions:

1. There is no lock conflict on the secondary copy; that is, the lock request for the secondary copy is always successful at the first attempt. Since the secondary copy is released immediately after the primary copy is released, the possibility that the primary copy be released while the secondary is still locked is minimal.

2. All records are of identical size and have an equal likelihood of being referenced.

3. Memory requests are served in a FCFS order from memory request queues.

4. Although the actual aggregate lock requests for records are a non-Poisson process, for tractibility of analysis, we assume they are generated from a poisson process.

A finite population model is used to study the effect of retry period. The system consists of M processors and K memory modules. The primary and secondary copies of all records are evenly allocated on these memory modules. Assume that each processor has only one buffer where one FTL transaction request can be stored. An FTL transaction request is accepted by a processor only when the buffer is empty; that is, no transaction request is in the processor. When an FTL transaction request is accepted, the processor starts this transaction immediately. Thus, a processor can be in one of the five possible states, shown in Fig. 2. A processor is in state I if there is

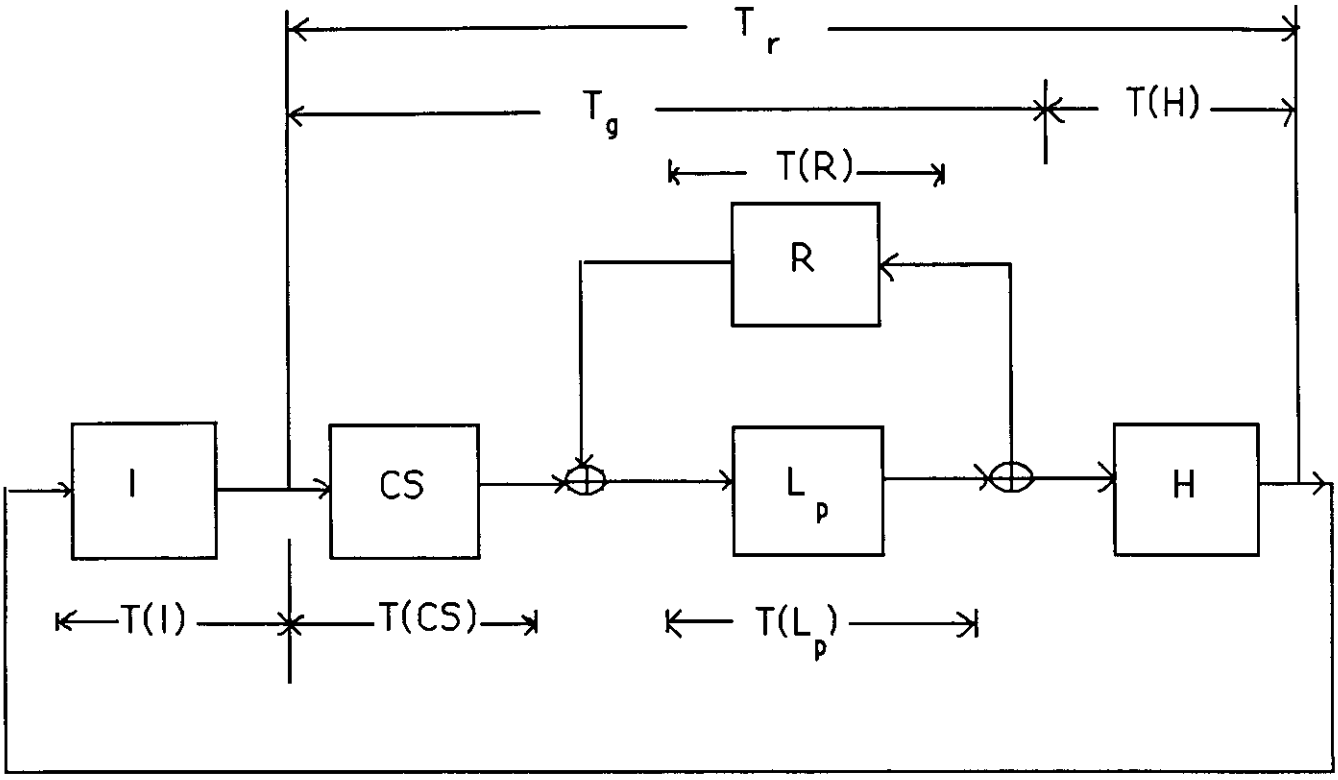


Fig. 2 State Diagram for One Processor

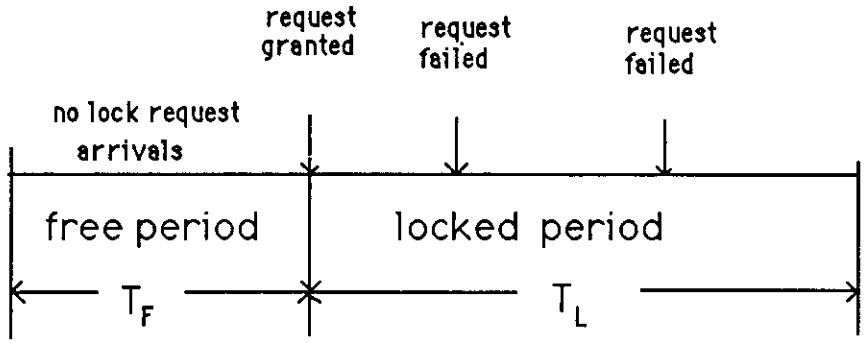


Fig.3 Free and Locked Period for One Primary Copy

no FTL transaction request. The time a processor remains in state I is defined as idle time. When one FTL transaction request arrives, the processor enters state CS. Here the processor checks its RST to determine if the primary copy is accessible. L_p is the state where processors issue requests to lock the primary copy. If a lock request is blocked, the processor will enter lock retry state, R. In state R, the processor performs the fault detection procedure and then waits for the completion of the retry period. Then, the processor reenters state L_p. Assuming the lock request is granted, the processor enters state H. In state H, the following operations are performed: lock the primary copy, issue lock request on the secondary copy, lock secondary copy, read the primary copy, perform local execution, update both copies and finally, release the locks on both copies.

Let us define the following system parameters:

N = total number of records in the system,

S = record size,

t_i = average idle time,

t_{rp} = average retry period,

t_{lr} = average time to issue a lock request without memory conflict,

t_l = average time to lock a record copy without memory conflict,

t_{rl} = average time to release a lock without memory conflict,

t_h = average time to read a copy, perform execution and update both copies without memory conflict,

t_r = average overhead to check the RST,

t_{fd} = average time to perform processor failure detection,

t = memory cycle time.

Note that in the following analysis, we assume that the number of records, N, is at

least greater than the number of memory modules, K.

3.2 Average Delay for Each State

Due to memory conflicts, it may take more than one memory cycle to finish a memory request. Let us define the access delay for shared memory system $\overline{T}_m$ as the time from the initiation of a shared memory request to its completion. The average delay due to memory conflict is equal to memory access delay minus memory cycle time; that is, $\overline{T}_m - t$. The delay increase at each state due to memory conflict depends on the number of shared memory accesses in that state. There are three types of instructions involving the shared memory access: read, update and modify data. Read and update data require only one memory access while modify data requires two memory accesses. One for reading the old value and another for writing the new value back to memory. We calculate the number of shared memory accesses for each operation to obtain the delay increase for each operation. The operations and their corresponding instructions involving shared memory accesses and the delay introduced by the operations that include memory conflicts are shown in Table 1 for each state.

The delay of state I is equal to the idle time. The delay of state CS is the overhead for determining the accessibility of the primary copy from the RST. Let T(x) be the delay for state x and $\overline{T}(x)$ be the average value of T(x). Thus, we have

$$\overline{T}(T) = t_i \tag{1}$$

and

$$\overline{T}(CS) = t_r \, . \tag{2}$$

The delay for states R, L_p and H can be obtained by adding the delay of all their

LW_p : LW of the primary copy
LW_s : LW of the secondary copy
LC_p : LC of the primary copy
LC_s : LC of the secondary copy

state	operation	instruction	delay
I	non-FTL operations	none	t_i
CS	determine the accessibility of the primary copy	none	t_r
L_p	Request for Locking the primary copy	check if LW_p is 0 ?	$t_{lr}+(\overline{T}_m-t)$
R	detection of processor failure	check if LC_p is changed ?	$t_{fd}+(\overline{T}_m-t)$
	wait for retry period	none	t_{rp}

Table 1. State Description and Delay for Each State

state	operation	instruction	delay
H	lock the primary copy	set LW_p to 1 and increment LC_p by 1	$t_l+3(\overline{T}_m-t)$
	determine the accessibility of the secondary copy	check RST	t_r
	request for locking the secondary copy	check if LW_s is 0 ?	$t_{lr}+(\overline{T}_m-t)$
	lock the secondary copy	set LW_s by 1 and increment LC_s to 1	$t_l+3(\overline{T}_m-t)$
	read the primary copy	perform read instruction on each byte	$t_h+(3S+2)(\overline{T}_m-t)$
	perform local execution	none	
	update the primary copy	set LW_p to 2 and perform update instruction on each byte	
	update the secondary copy	set LW_s to 2 and perform update instruction on each byte	
	lock release for the primary copy	reset LW_p to 0	$t_{rl}+(\overline{T}_m-t)$
	lock release for the secondary copy	reset LW_s to 0	$t_{rl}+(\overline{T}_m-t)$

Table 1. (Cont.)

operations in Table 1. We have,

$$\bar{T}(R)=t_{rp}+t_{fd}+\bar{T}_m-t \tag{3}$$

$$\bar{T}(L_p)=\bar{T}_m-t+t_{lr} \tag{4}$$

and

$$\bar{T}(H)=(3S+11)(\bar{T}_m-t)+t_d \tag{5}$$

where,

$$t_d=2t_l+2t_{rl}+t_r+t_{lr}+t_h\ .$$

Let us define the following parameters:

λ: the rate per unit time that lock requests for the primary copy are granted.

T_g: the time from the initiation of a lock request to the granting of the lock for the primary copy.

T_r: the time interval from the initiation of a lock request for a record to the release of the lock in the secondary copy of that record.

$\bar{T}_g$ and $\bar{T}_r$ are the average values for T_g and T_r. We shall use λ, $\bar{T}_r$, and $\bar{T}_g$ as the performance measures in our analysis.

3.3 Estimation of λ, $\bar{T}_r$ and $\bar{T}_g$

In our model (Fig. 2), the transaction response time refers to the time interval from the processors' entry to state CS until their departure from state H. The lock grant rate is referred to as the rate that processors enter state H so λ is the system throughput. Let the cycle time be the time interval between two succesive entries of a processor into state I. By applying Little's result [LIT61], we find that the total number of processors is equal to system throughput multiplied by the cycle time. Thus, we have

$$\lambda[\,t_i + \bar{T}_r\,] = M\,. \tag{6}$$

$\bar{T}_r$ can be expressed as a function of λ yielding:

$$\bar{T}_r = \frac{M}{\lambda} - t_i\,. \tag{7}$$

From Fig. 2, we have

$$\bar{T}_r = \bar{T}_g + \bar{T}(H)\,. \tag{8}$$

$\bar{T}_g$ can be expressed in terms of $\bar{T}_r$ from (8)

$$\bar{T}_g = \bar{T}_r - \bar{T}(H)\,. \tag{9}$$

To estimate λ, we consider the primary copy of each record as a resource which goes through free and locked periods depending on whether or not the copy is locked by a processor (Fig. 3). During one alternative free and locked period, only one lock request is granted. Let $\bar{T}_L$ and $\bar{T}_F$ be the average length of the locked and the free periods, respectively. By using the renewal theory [KLE75], we find that the lock grant rate per record is $\frac{1}{\bar{T}_F + \bar{T}_L}$. Since it is assumed that all records are equally likely to be referenced (assumption 2), we have

$$\lambda = \frac{N}{\bar{T}_F + \bar{T}_L}\,. \tag{10}$$

According to the model, the average locked period for the primary copy is equal to the average delay incurred at state H minus average delay to release the secondary copy. Thus,

$$\bar{T}_L = \bar{T}(H) - [\,t_{rl} + (\bar{T}_m - t)\,]\,. \tag{11}$$

Let λ_l be the aggregate lock request rate. Since we assume that all records have an equal likelihood of being referenced, the lock request rate per record is $\frac{\lambda_l}{N}$. Further, since aggregate lock requests are assumed to be generated from the poisson process (assumption 4) we have

$$\bar{T}_F = \frac{1}{\frac{\lambda_l}{N}} . \tag{12}$$

The aggregate lock request rate λ_l is the rate that processors enter state L_p. If we let $\bar{M}(x)$ be the average number of processors in state x and apply Little's result to the L_p state, we have

$$\lambda_l = \frac{\bar{M}(L_p)}{t_{lr} + \bar{T}_m - t} . \tag{13}$$

By substituting (13) into (12), we find:

$$\bar{T}_F = \frac{(t_{lr} + \bar{T}_m - t)N}{\bar{M}(L_p)} . \tag{14}$$

Substituting (11) and (14) into (10), we have

$$\lambda = \bar{M}(L_p) N \{N(t_{lr} + \bar{T}_m - t) + \bar{M}(L_p)[\bar{T}(H) - t_{rl} - \bar{T}_m + t]\}^{-1} . \tag{15}$$

Let us estimate $\bar{T}_m$. Let $\bar{n}_m$ be the number of memory requests in the queue of a memory module seen by an incoming memory request. Since memory requests are served in a FCFS order (assumption 3), we have

$$\bar{T}_m = (1 + \bar{n}_m) t . \tag{16}$$

Let $\bar{N}_m$ be the time average number of memory requests in the queue of one memory module. By applying the state approximation method in [BAR81], we obtain,

$$\bar{n}_m = \bar{N}_m (1 - \frac{1}{M}) . \tag{17}$$

Memory requests are initiated by processors in states H, L_p, and R. Since all records have an equal likelihood of being referenced and record copies are evenly distributed on all memory modules, the average number of memory requests in each memory module equals $\frac{1}{K}$ of the total average number of memory requests issued by processors. Therefore,

$$\bar{N}_m = \frac{1}{K} [\bar{N}_m(H) + \bar{N}_m(L_p) + \bar{N}_m(R)] \tag{18}$$

where,

$\bar{N}_m(x)$=average number of memory requests in memory module queues issued by processors in state x.

Substituting (17) and (18) into (16), we have

$$\bar{T}_m = t + \frac{1}{K} (\bar{N}_m(H) + \bar{N}_m(R) + \bar{N}_m(L_p)) (1 - \frac{1}{M}) t . \tag{19}$$

Substituting (19) into (5), we obtain

$$\bar{T}(H) = \frac{(3S+11)}{K} [(\bar{N}_m(H) + \bar{N}_m(L_p) + \bar{N}_m(R)(1 - \frac{1}{M}) t] + t_d . \tag{20}$$

λ_l and λ can be obtained by substituting (19) into (13) and (19) and (20) into (15).

$$\lambda_l = \bar{M}(L_p)[t_{lr} + \frac{1}{K}(N_m(H) + \bar{N}_m(R) + \bar{N}_m(L_p))(1 - \frac{1}{M}) t]^{-1} \tag{21}$$

and

$$\lambda = \bar{M}(L_p) N \{ N t_{lr} + \bar{M}(L_p) t_d - \bar{M}(L_p) t_{rl} + \frac{[(3S+10)\bar{M}(L_p) + N]}{K}$$

$$(\bar{N}_m(H)+\bar{N}_m(L_p)+\bar{N}_m(R))(1-\frac{1}{M})t\}^{-1}. \tag{22}$$

Substituting (22) into (7), we have

$$\bar{T}_r=\frac{M}{\bar{M}(L_p)N}\{N t_{lr}+\bar{M}(L_p)t_d-\bar{M}(L_p)t_{rl}+\frac{[(3S+10)\bar{M}(L_p)+N]}{K}$$

$$(\bar{N}_m(H)+\bar{N}_m(L_p)+\bar{N}_m(R))(1-\frac{1}{M})t\}-t_i. \tag{23}$$

The average lock grant time $\bar{T}_g$ can be obtained by substituting (20) and (23) into (9).

$$\bar{T}_g=\frac{M}{\bar{M}(L_p)N}\{N t_{lr}+\bar{M}(L_p)t_d-\bar{M}(L_p)t_{rl}+\frac{[(3S+10)\bar{M}(L_p)+N]}{K}$$

$$(\bar{N}_m(H)+\bar{N}_m(L_p)+\bar{N}_m(R))(1-\frac{1}{M})t\}-t_i$$

$$-(3S+11)\frac{1}{K}[(\bar{N}_m(H)+\bar{N}_m(L_p)+\bar{N}_m(R))(1-\frac{1}{M})t]-t_d \tag{24}$$

For our analysis, we solve $\bar{N}_m(x)$, and $\bar{M}(x)$ via a set of nonlinear equations of these variables. We shall now set up this set of nonlinear equations.

3.4 Nonlinear Equations for $\bar{N}_m(x)$ and $\bar{M}(x)$

Since the rate at which processors enter states H, T and CS equals the lock grant rate, λ, and the rate at which processors enter state R equals the lock contention rate, λ_l - λ, we can obtain the following equations by applying Little's result to each state.

$$\lambda \bar{T}(H) = \bar{M}(H) \tag{25}$$

$$\lambda \bar{T}(T) = \bar{M}(T) \tag{26}$$

$$(\lambda_1 - \lambda)\bar{T}(R) = \bar{M}(R) \tag{27}$$

$$\lambda_1 \bar{T}(L_p) = \bar{M}(L_p) \tag{28}$$

$$\lambda \bar{T}(CS) = \bar{M}(CS) \tag{29}$$

From Table 1, processors entering state H will issue (3S+11) memory requests. The rate at which processors enter state H is λ. Therefore, the memory request rate issued by processors in state H is $\lambda(3S+11)$. Applying Little's result to memory requests in memory queues issued by processors in state H, we can obtain the following equation:

$$\lambda (3S + 11)\bar{T}_m = \bar{N}_m(H) \tag{30}$$

Considering memory requests issued by processors in states R and L_p and applying Little's result we find,

$$(\lambda_1 - \lambda)\bar{T}_m = \bar{N}_m(R) \tag{31}$$

$$\lambda_1 \bar{T}_m = \bar{N}_m(L_p) \tag{32}$$

Since the total number of processors in the system is fixed, the sum of processors in all states is equal to M. Hence,

$$\bar{M}(H) + \bar{M}(T) + \bar{M}(CS) + \bar{M}(L_p) + \bar{M}(R) = M \tag{33}$$

Substituting (1)-(5) into (25)-(29) and (19)-(22) into (25)-(32), we can obtain non-linear equations of $\bar{M}(x)$ and $\bar{N}_m(x)$.

$$\bar{M}(L_p)\, N \{ N t_{lr} + \bar{M}(L_p)\, t_d - \bar{M}(L_p)\, t_{rl} + \frac{[(3S+10)\bar{M}(L_p) + N]}{K}$$

$$(\bar{N}_m(H)+\bar{N}_m(L_p)+\bar{N}_m(R))(1-\frac{1}{M})t\}^{-1}$$

$$\{\frac{(3S+11)}{K}[(\bar{N}_m(H)+\bar{N}_m(L_p)+\bar{N}_m(R))(1-\frac{1}{M})t]+t_d\}-\bar{M}(H)=0 \quad (34)$$

$$\bar{M}(L_p)N\{N t_{lr}+\bar{M}(L_p)t_d-\bar{M}(L_p)t_{rl}+\frac{[(3S+10)\bar{M}(L_p)+N]}{K}$$

$$(\bar{N}_m(H)+\bar{N}_m(L_p)+\bar{N}_m(R))(1-\frac{1}{M})t\}^{-1}t_i-\bar{M}(I)=0 \quad (35)$$

$$\{\bar{M}(L_p)[t_{lr}+\frac{1}{K}(N_m(H)+\bar{N}_m(R)+\bar{N}_m(L_p))(1-\frac{1}{M})t]^{-1}$$

$$-\bar{M}(L_p)N\{N t_{lr}+\bar{M}(L_p)t_d-\bar{M}(L_p)t_{rl}+\frac{[(3S+10)\bar{M}(L_p)+N]}{K}$$

$$(\bar{(\bar{N}}_m(H)+\bar{N}_m(L_p)+\bar{N}_m(R))(1-\frac{1}{M})t\}^{-1}\}$$

$$[t_{rp}+t_{fd}+\frac{1}{K}(\bar{N}_m(H)+\bar{N}_m(R)+\bar{N}_m(L_p))(1-\frac{1}{M})N t]-\bar{M}(R)=0 \quad (36)$$

$$\bar{M}(L_p)[t_{lr}+\frac{1}{K}(N_m(H)+\bar{N}_m(R)+\bar{N}_m(L_p))(1-\frac{1}{M})t]^{-1}$$

$$[t_{lr}+\frac{1}{K}(\bar{M}(H)+\bar{M}(R)+\bar{M}(L_p))(1-\frac{1}{M})N t]-\bar{M}(L_p)=0 \quad (37)$$

$$\bar{M}(L_p)N\{N t_{lr}+\bar{M}(L_p)t_d-\bar{M}(L_p)t_{rl}+\frac{[(3S+10)\bar{M}(L_p)+N]}{K}$$

$$(\bar{N}_m(H)+\bar{N}_m(L_p)+\bar{N}_m(R))(1-\frac{1}{M})t\}^{-1}t_r-\bar{M}(CS)=0 \quad (38)$$

$$\overline{M}(L_p)\,N\,\{\,N\,t_{lr}+\overline{M}(L_p)\,t_d-\overline{M}(L_p)\,t_{rl}+\frac{[\,(3S+10)\,\overline{M}(L_p)+N\,]}{K}$$

$$(\,\overline{N}_m(H)+\overline{N}_m(L_p)+\overline{N}_m(R)\,)\,(1-\frac{1}{M})\,t\,\}^{-1}$$

$$(3S+11)[\,t+\frac{1}{K}(\,\overline{N}_m(H)+\overline{N}_m(R)+\overline{N}_m(L_p)\,)\,(1-\frac{1}{M})\,t\,]-\overline{N}_m(H)=0 \qquad (39)$$

$$\{\overline{M}(L_p)[t_{lr}+\frac{1}{K}(N_m(H)+\overline{N}_m(R)+\overline{N}_m(L_p))(1-\frac{1}{M})t]^{-1}$$

$$-\overline{M}(L_p)\,N\,\{\,N\,t_{lr}+\overline{M}(L_p)\,t_d-\overline{M}(L_p)\,t_{rl}+\frac{[\,(3S+10)\,\overline{M}(L_p)+N\,]}{K}$$

$$(\,\overline{N}_m(H)+\overline{N}_m(L_p)+\overline{N}_m(R)\,)\,(1-\frac{1}{M})\,t\,\}^{-1}\}$$

$$[\,t+\frac{1}{K}(\overline{N}_m(H)+\overline{N}_m(R)+\overline{N}_m(L_p))(1-\frac{1}{M})\,t]-\overline{N}_m(R)=0 \qquad (40)$$

$$\overline{M}(L_p)[t_{lr}+\frac{1}{K}(N_m(H)+\overline{N}_m(R)+\overline{N}_m(L_p)\,)(1-\frac{1}{M})\,t]^{-1}$$

$$[\,t+\frac{1}{K}(\overline{N}_m(H)+\overline{N}_m(R)+\overline{N}_m(L_p))\,(\,1-\frac{1}{M}\,)\,t\,]-\overline{N}_m(L_p)=0 \qquad (41)$$

Since the system reaches its equilibrium state, the above set of nonlinear equations provides exactly one solution for $\overline{M}(H)$, $\overline{M}(T)$, $\overline{M}(CS)$, $\overline{M}(R)$, $\overline{N}_m(H)$, $\overline{N}_m(L_p)$, and $\overline{N}_m(R)$.

4. DISCUSSION OF RESULTS

In this section, we use the results generated from the model to study the behavior of the FTL. The overhead values used in our study are

$t_r = 30.0$

$t_{fd} = 15.0$

$t_l = 2.0$

$t_{lr} = 2.0$

$t_{rl} = 2.0$.

All time units are represented in memory cycle time.

To reduce memory conflict, the number of memory modules in the system, K, is equal to the number of processors or equal to the number of records, whichever is less; that is, K=min[M, N].

Lock grant rate is the amount of lock requests that the system can support for a given operating environment. We study the lock grant rate, the transaction response time and the lock grant time as a function of lock retry period for selected M and N. Figure 4 presents the lock grant rate of selected M for N = 100. We note that the lock grant rate is higher for larger values of M. Further, the lock grant rate is more sensitive with retry period as M increases. This is because increasing retry period reduces lock request rate. The reduction increases as the number of processors goes up. Increasing retry period also reduces the lock retry rate thus reducing the lock grant rate. Due to the large number of records available in the system and the assumption that all the records have an equal likelihood of being referenced, there is little lock conflict thus, yielding low lock grant time (Figure 6). The lock grant rate for N = 10 is shown in Figure 5. We note that lock grant rate is not very sensitive to M. This is mainly due

to the small number of records available. Besides, since the number of records is small, the records become the bottleneck. A high lock conflict rate causes the saturation of the system. Therefore, the lock grant time increases as the number of processors increases (Figure 7). The curves for M = 48 and M = 32 in Figure 5 show some degradation with small retry periods. This is mainly due to memory conflicts. High lock conflict results in a large number of retries. Thus, a small retry period causes a great deal of memory conflict. With small retry periods (M=48 and M=32), the lock grant time decreases as retry period increases (Figure 7). In this case, increasing retry period reduces the memory conflict and lock conflict. Therefore, lock grant time decreases as retry period increases with small retry periods.

Transaction response time is the sum of the lock grant time, the lock holding time for the primary copy (including memory conflict delay) and the time used to release the secondary copy. Figures 8 and 9 present the transaction response time as a function of retry period for N=10 and N=100 respectively. Note that the curves are similar to those in Figures 6 and 7. The difference between transaction response time and lock grant time decreases as retry period increases due to the associated reduction of memory conflict.

Figures 10 - 15 present the effect of the idle time on lock grant rate, lock grant time and transaction response time for t_i = 0, 1000, 2000, and M = 16 and 48. We note that increasing the idle time reduces the lock request rate which in turn reduces both memory and lock conflicts. The effect of the idle time on lock grant rate is shown in Figures 10 and 11. Besides, as the idle time increases, lock grant time and response time decreases. Since there are more lock conflicts with fewer records, the amount of increase in lock grant rate is less sensitive to increasing idle time. Further, the reduction in lock grant time and lock response time are more sensitive with variation in idle

time for the fewer records.

5. CONCLUSION AND FUTURE RESEARCH

In this chapter, an analytic model for FTL has been proposed. This model is used to study the behavior of FTL. Our study reveals that the number of processors, number of records, idle time, and lock retry period are key parameters that influence FTL performance. The amount of memory and lock conflicts depend on idle time, the number of processors and the number of shared records. Increasing the number of processors, reducing the number of memory modules and/or shared records, or reducing the idle time results in an increase in memory and lock conflicts.

To obatain low lock grant time, the retry period should be as small as possible. However, reducing the retry period increases memory conflicts and lock conflicts. Thus, there is an optimal retry period for for FTL with a given operating environment.

The FTL protocol has been implemented on the Crossbar Multicomputer Testbed at SDC Huntsville, AL. The testbed contains six processors connected to twelve 32K byte memories by a crossbar switch network. Records are replicated on shared memory modules. FTL protocol is used by processes for accessing records in shared memory. The measurement results from these experiments will be compared with results from our analysis to validate the assumptions and approximations used in our model. More specifically, to compute the lock grant rate and the average transaction response, we need the following parameter values from the testbed:

t_i: average time interval from the completion of one FTL transaction to the beginning of the next FTL transaction,

t_{rp}: average retry period,

t_{lr}: average time to issue a lock request without memory conflict,

t_l: average time to lock one copy without memory conflict,

t_{rl}: average time to release a lock for one copy,

t_h: average time to read record, perform local execution and update record without memory conflict,

t_r: average time to check the RST,

t_{fd}: average time to perform processor failure detection.

We plan to perform this validation jointly with SDC during the coming year.

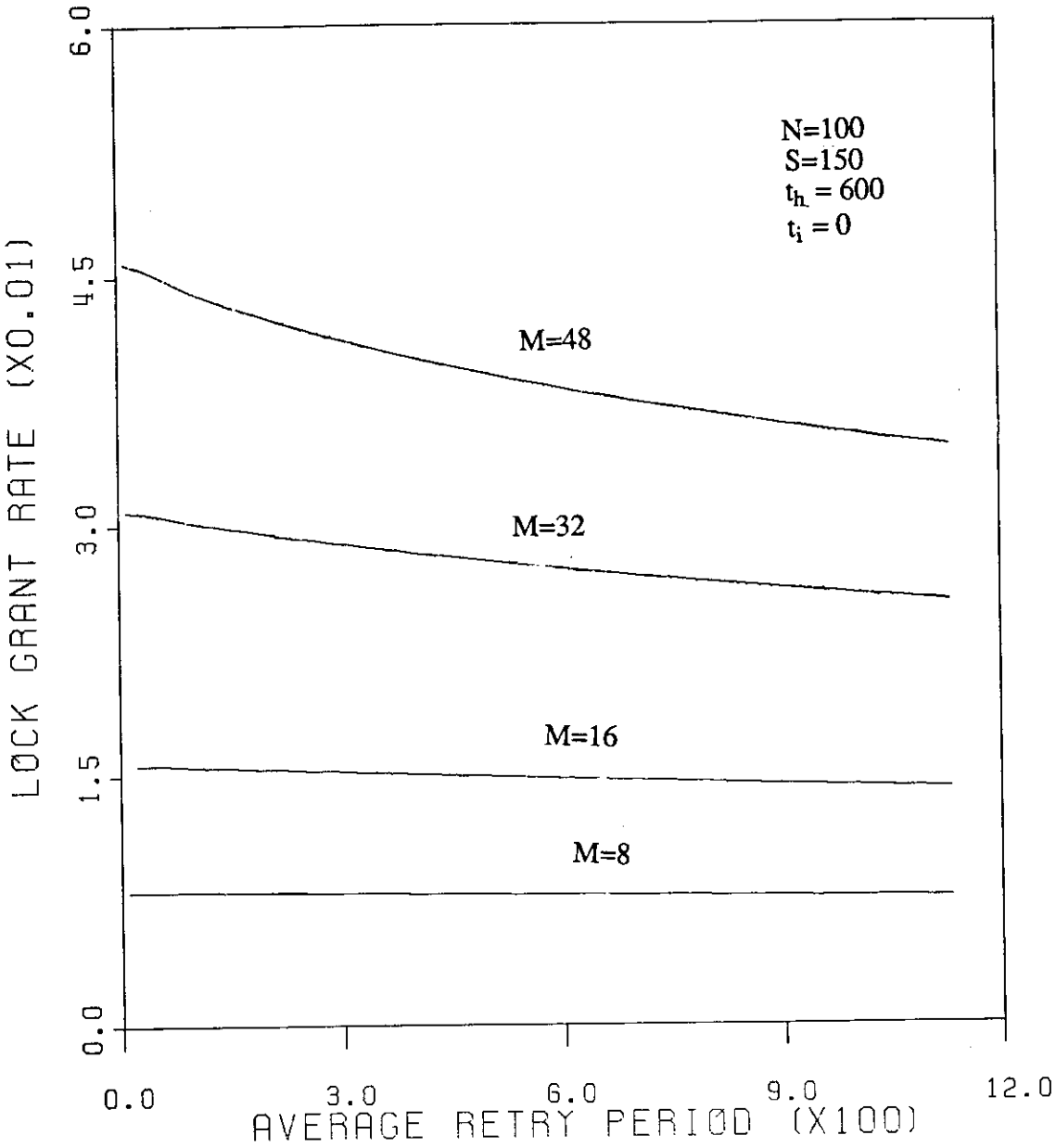


Figure 4 Lock Grant Rate as a Function of Retry Period for Different M (N=100 case)

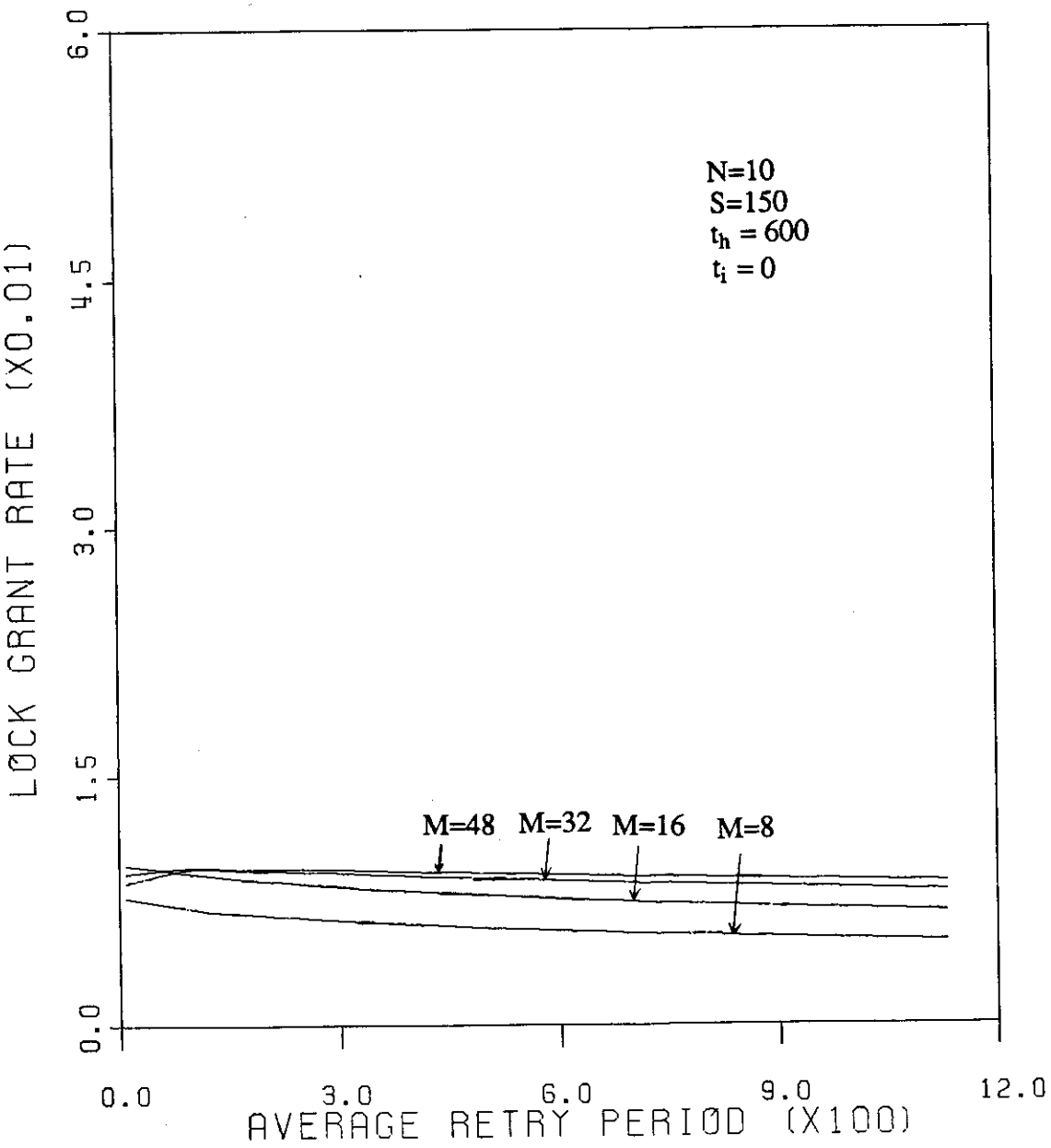


Figure 5 Lock Grant Rate as a Function of Retry Period for Different M (N=10 case)

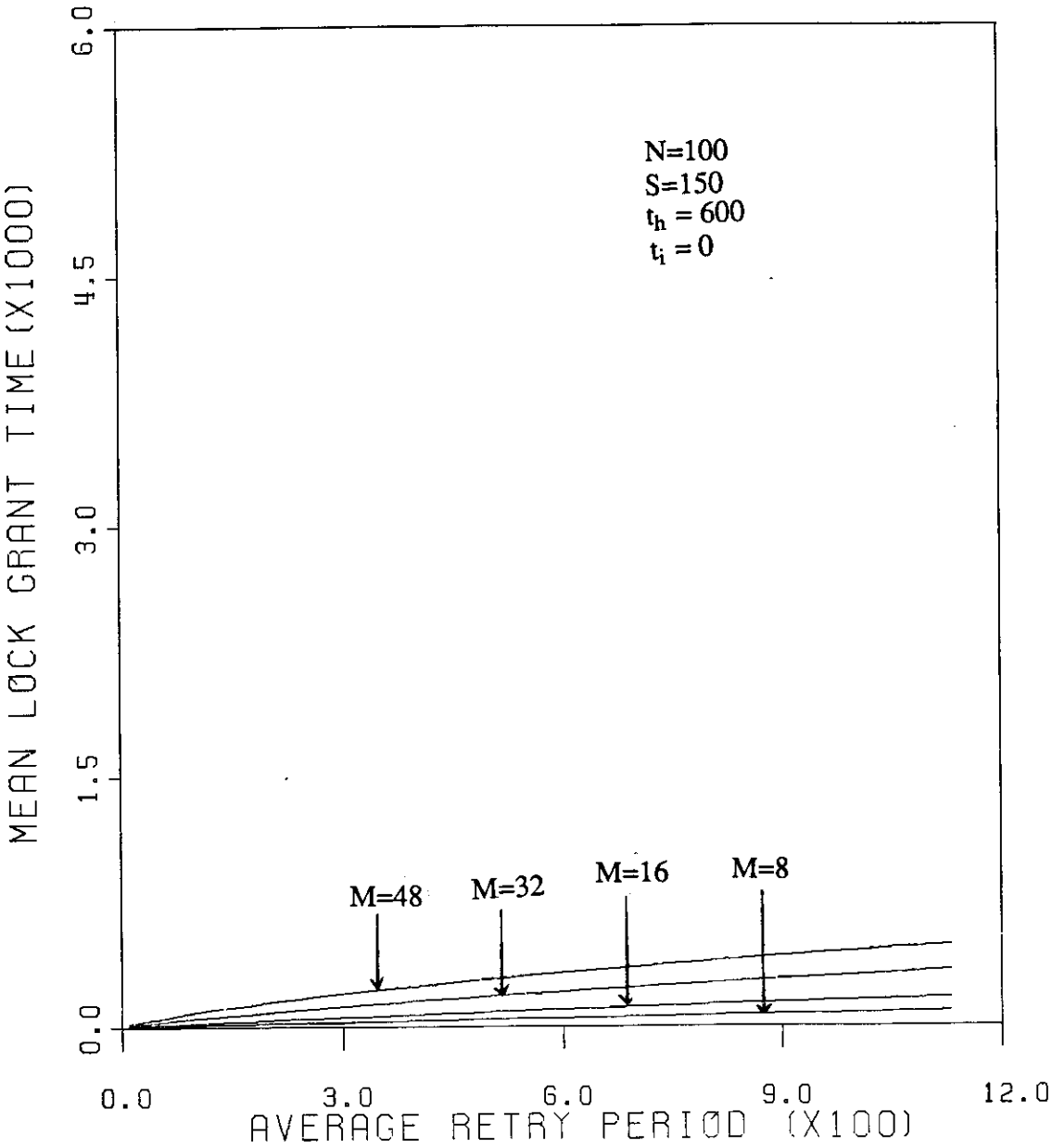


Figure 6 Lock Grant Time as a Function of Retry Period for Selected M (N=100 case)

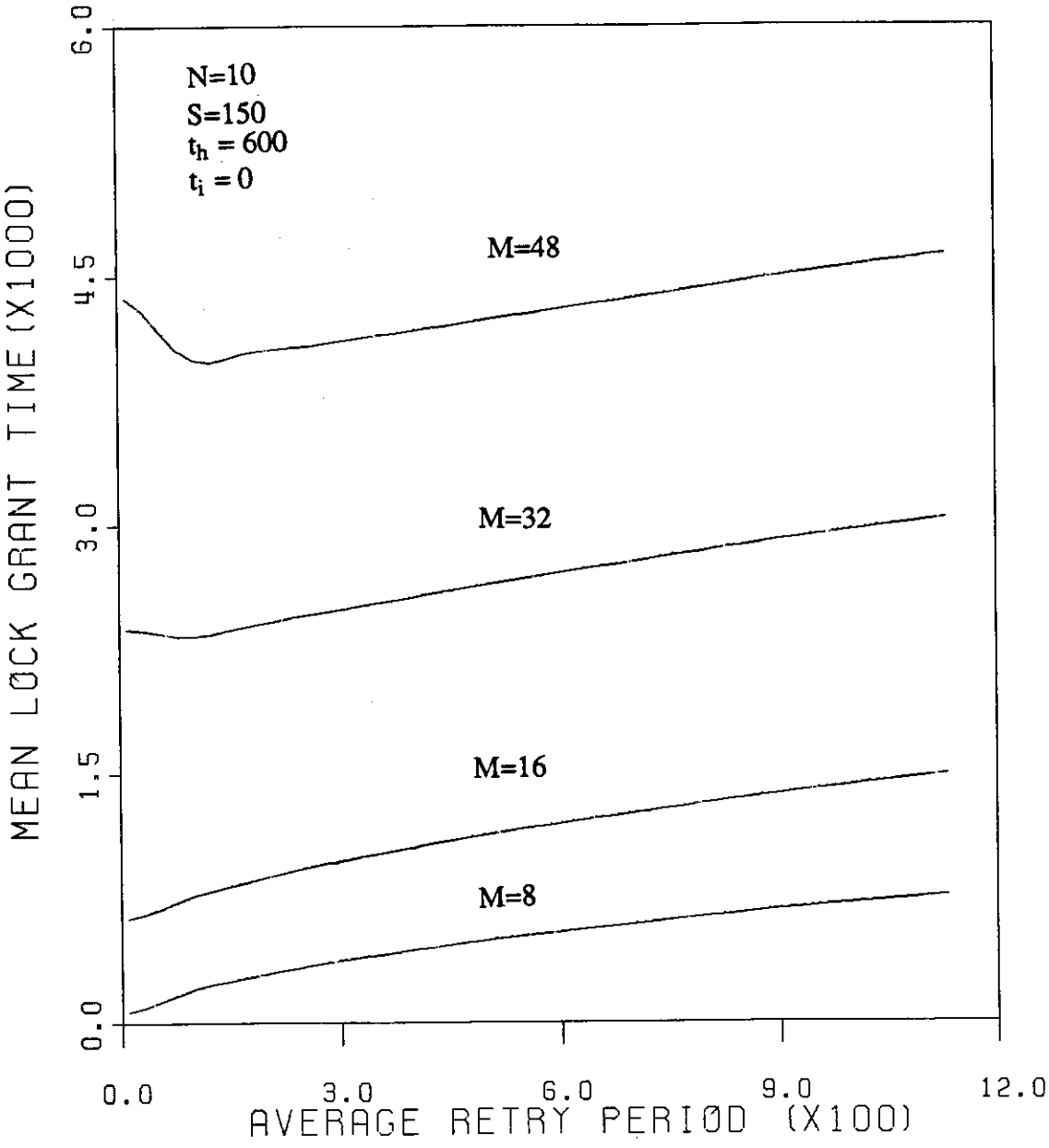


Figure 7 Lock Grant Time as a Function of Retry Period for Selected M (N=10 case)

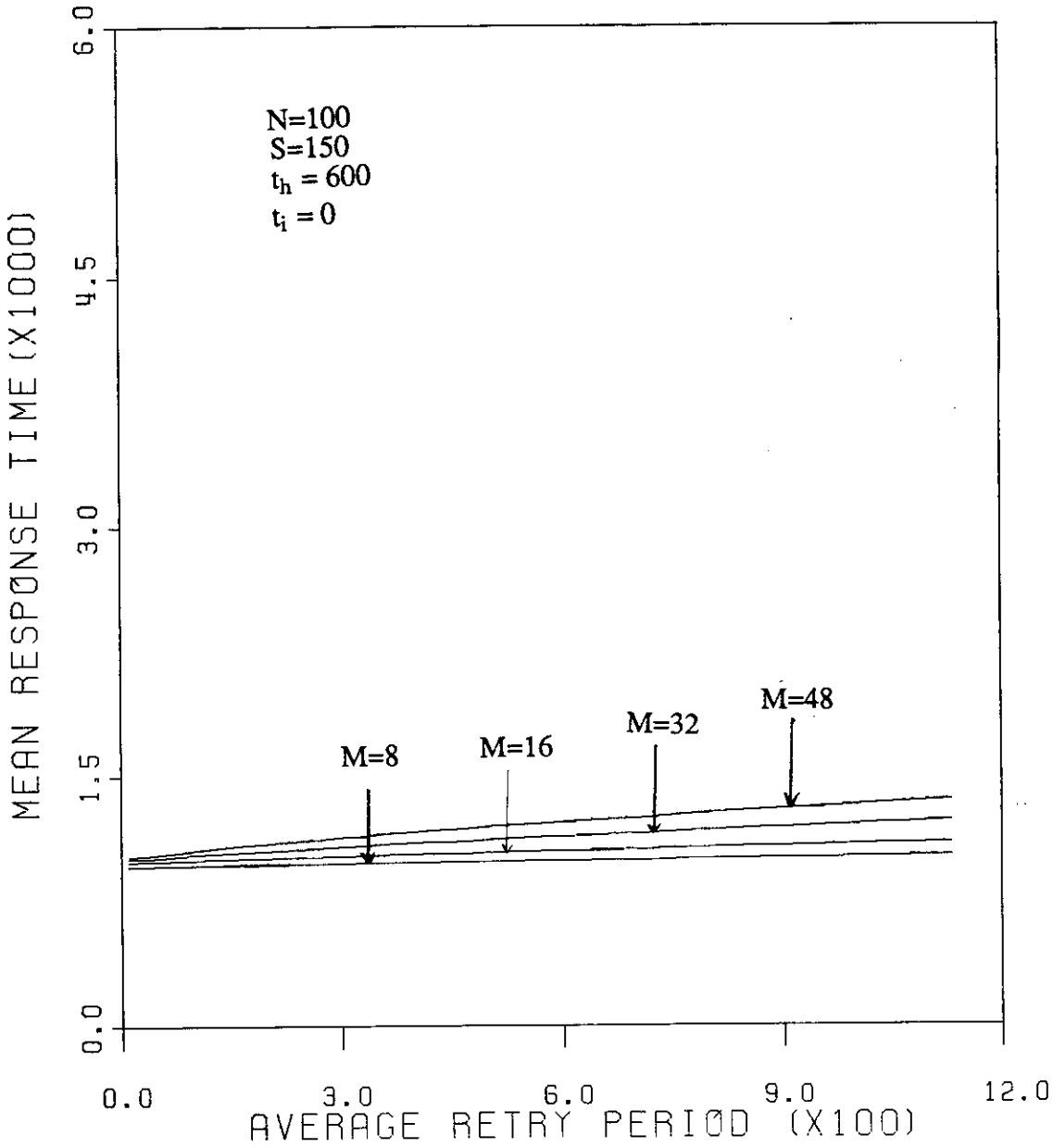


Figure 8 Lock Response Time as a Function of Retry Period for Selected M (N=100 case)

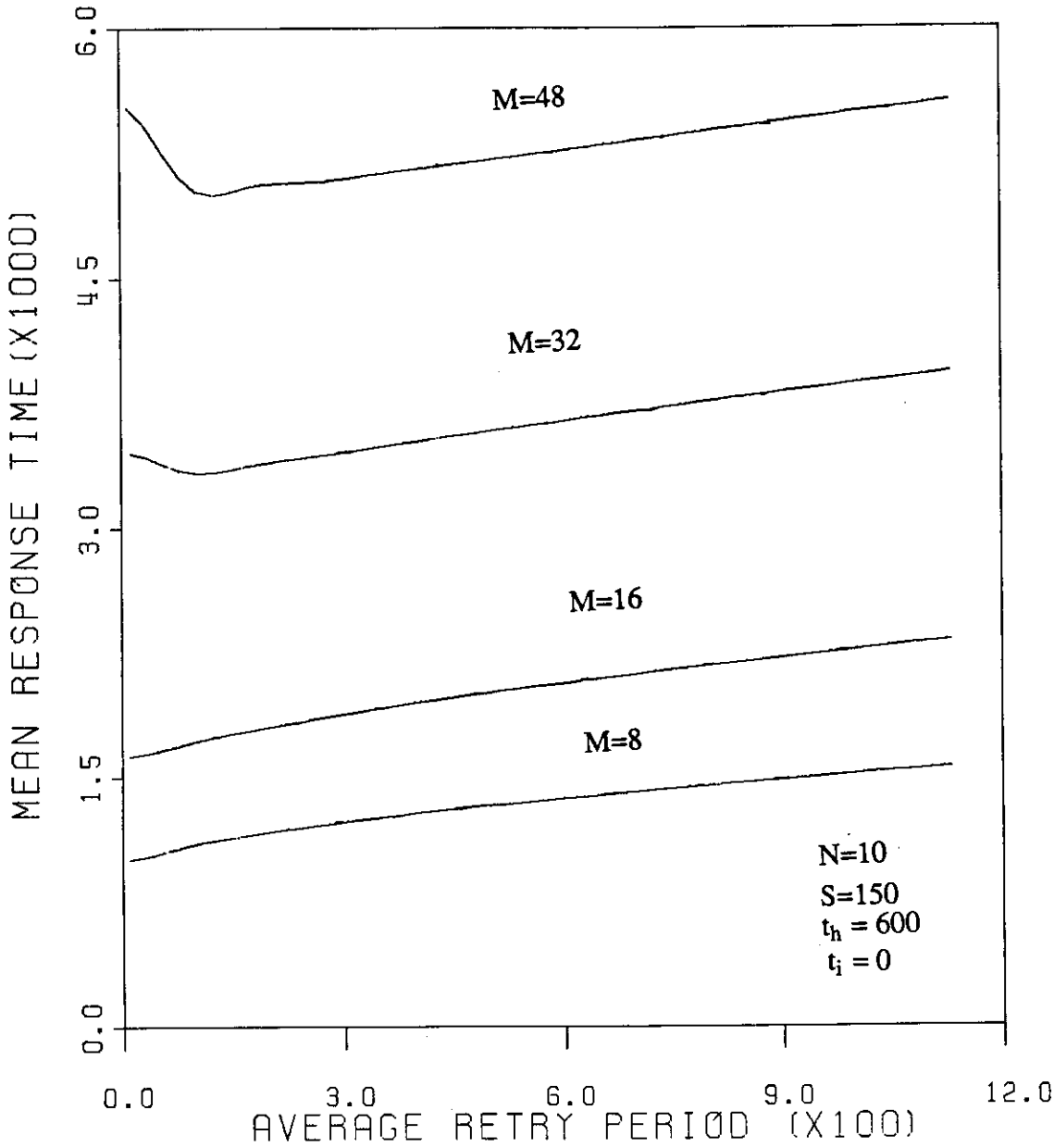


Figure 9 Lock Response Time as a Function of Retry Period for Selected M (N=10 case)

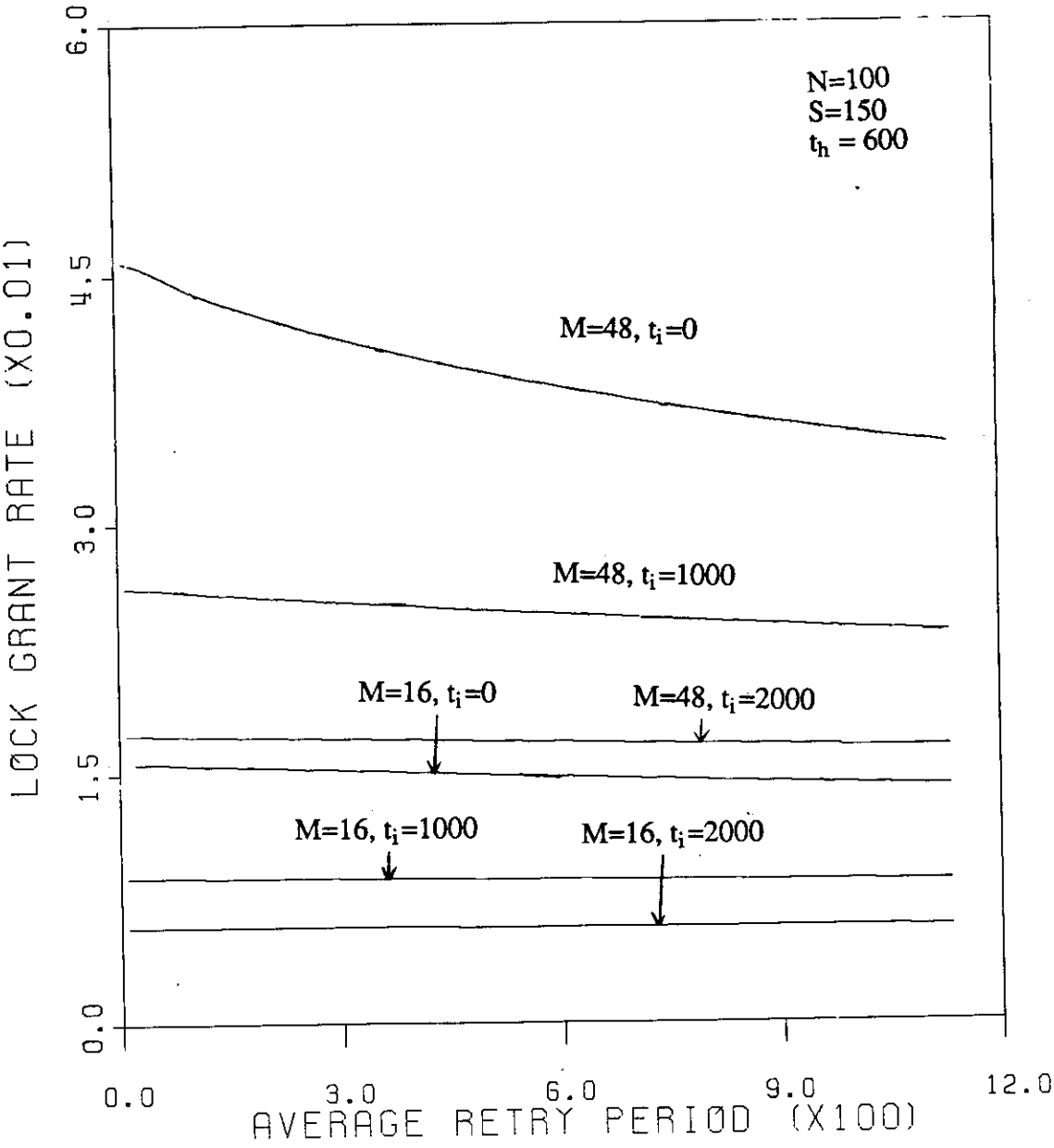


Figure 10 Lock Grant Rate as a Function of Retry Period

for Selected Pairs of M and t_i (N=100 case)

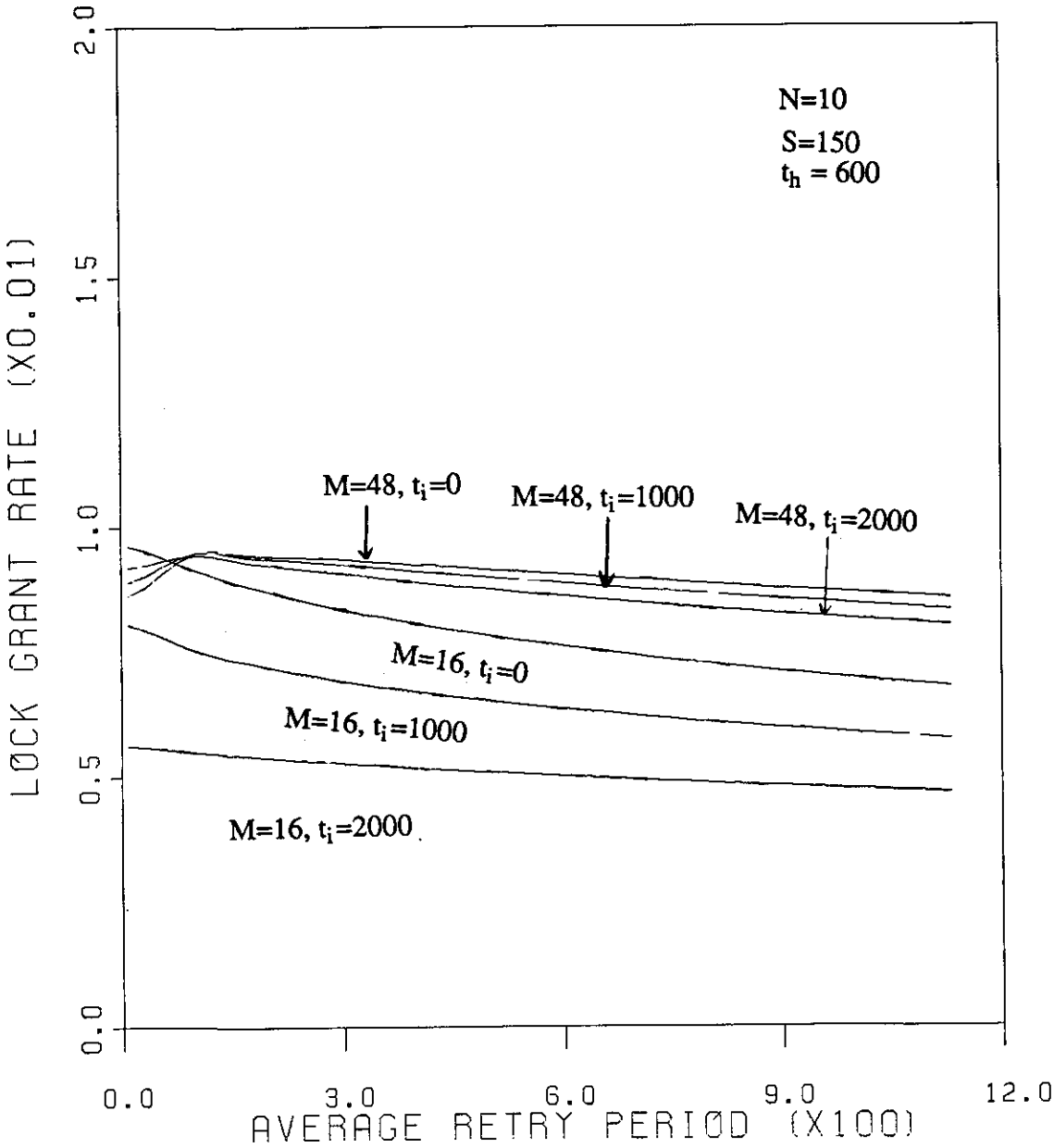


Figure 11 Lock Grant Rate as a Function of Retry Period

for Selected Pairs of M and t_i (N=10 case)

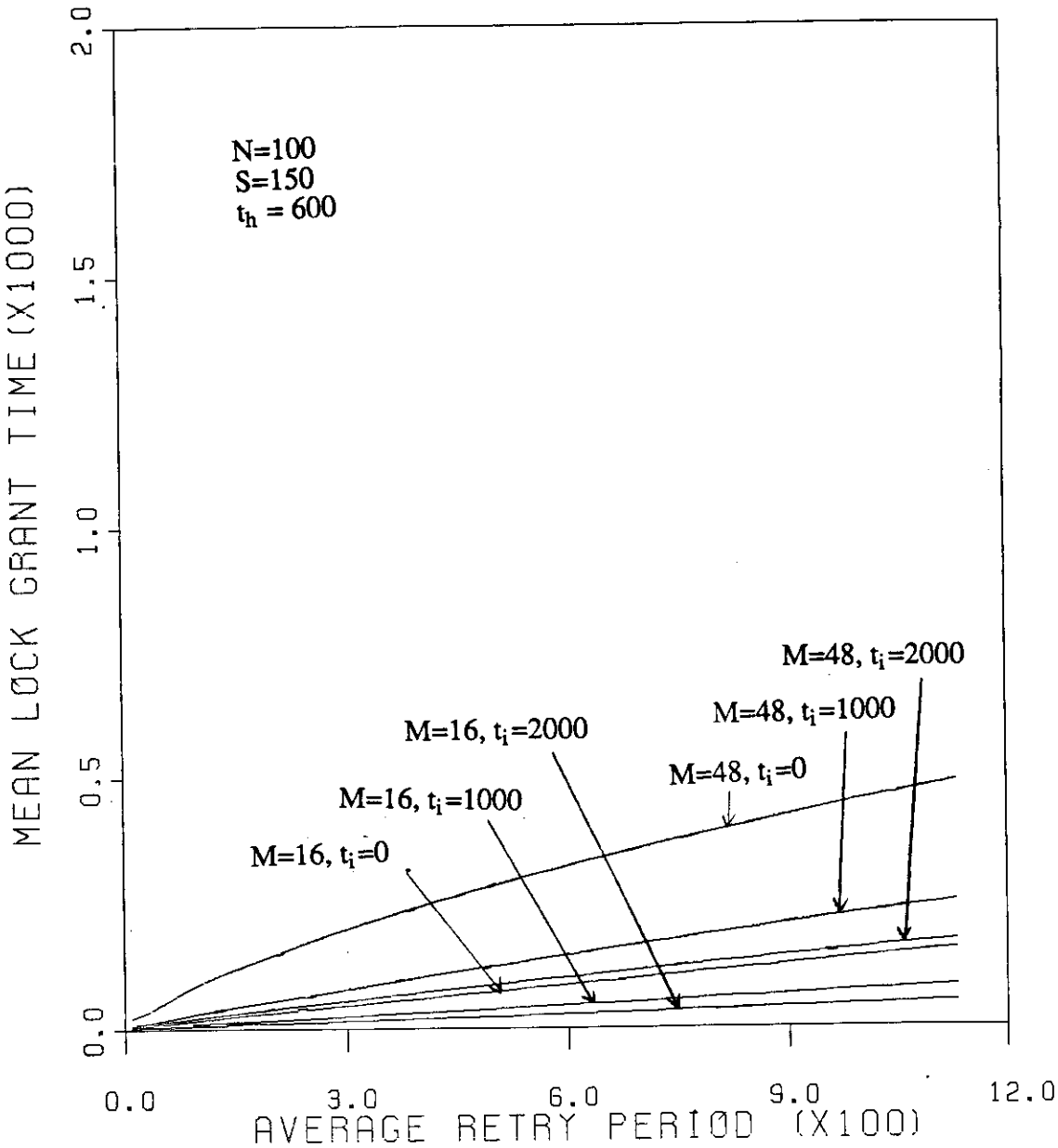


Figure 12 Lock Grant Time as a Function of Retry Period
for Selected Pairs of M and t_i (N=100 case)

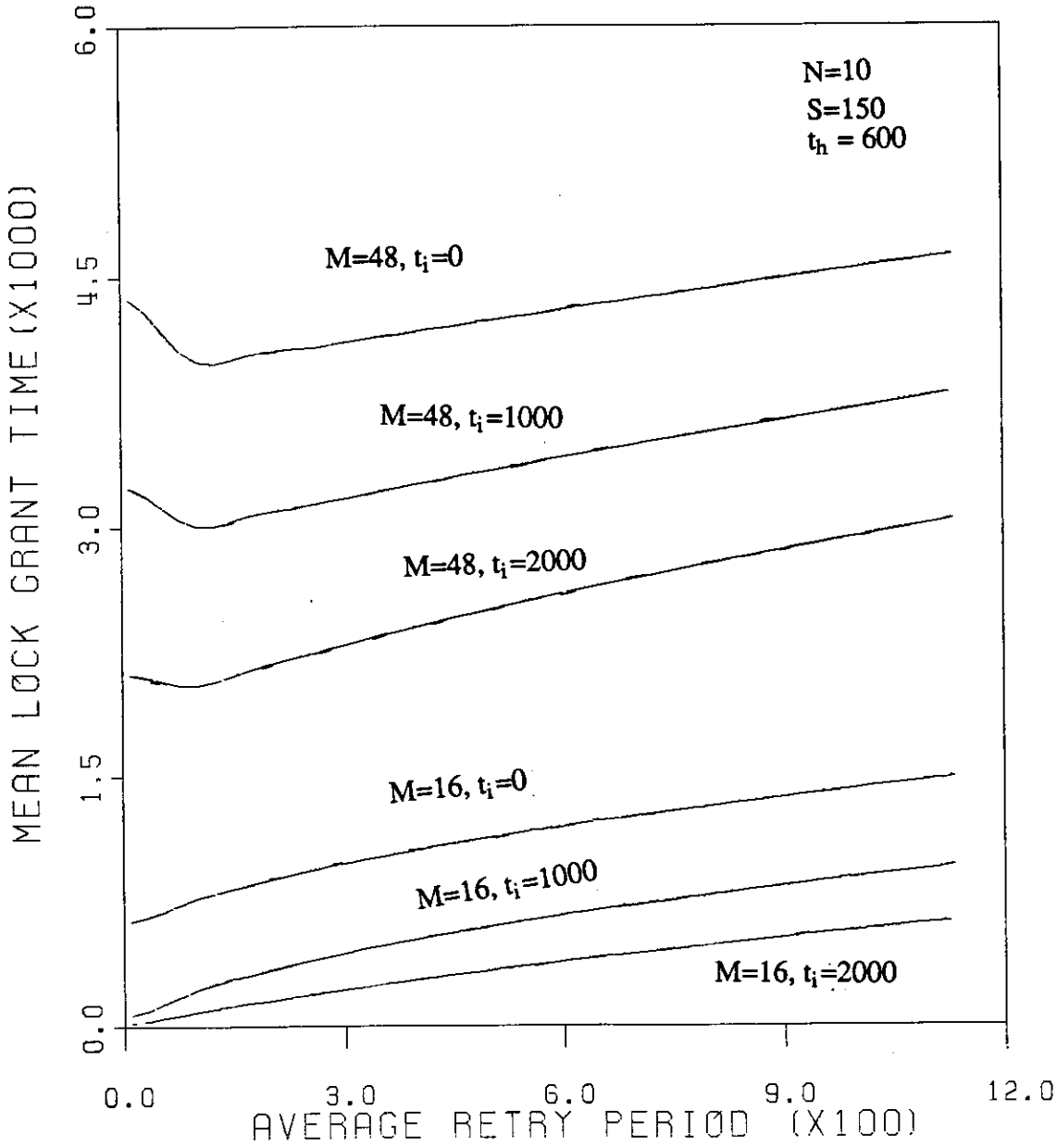


Figure 13 Lock Grant Time as a Function of Retry Period
for Selected Pairs of M and t_i (N=10 case)

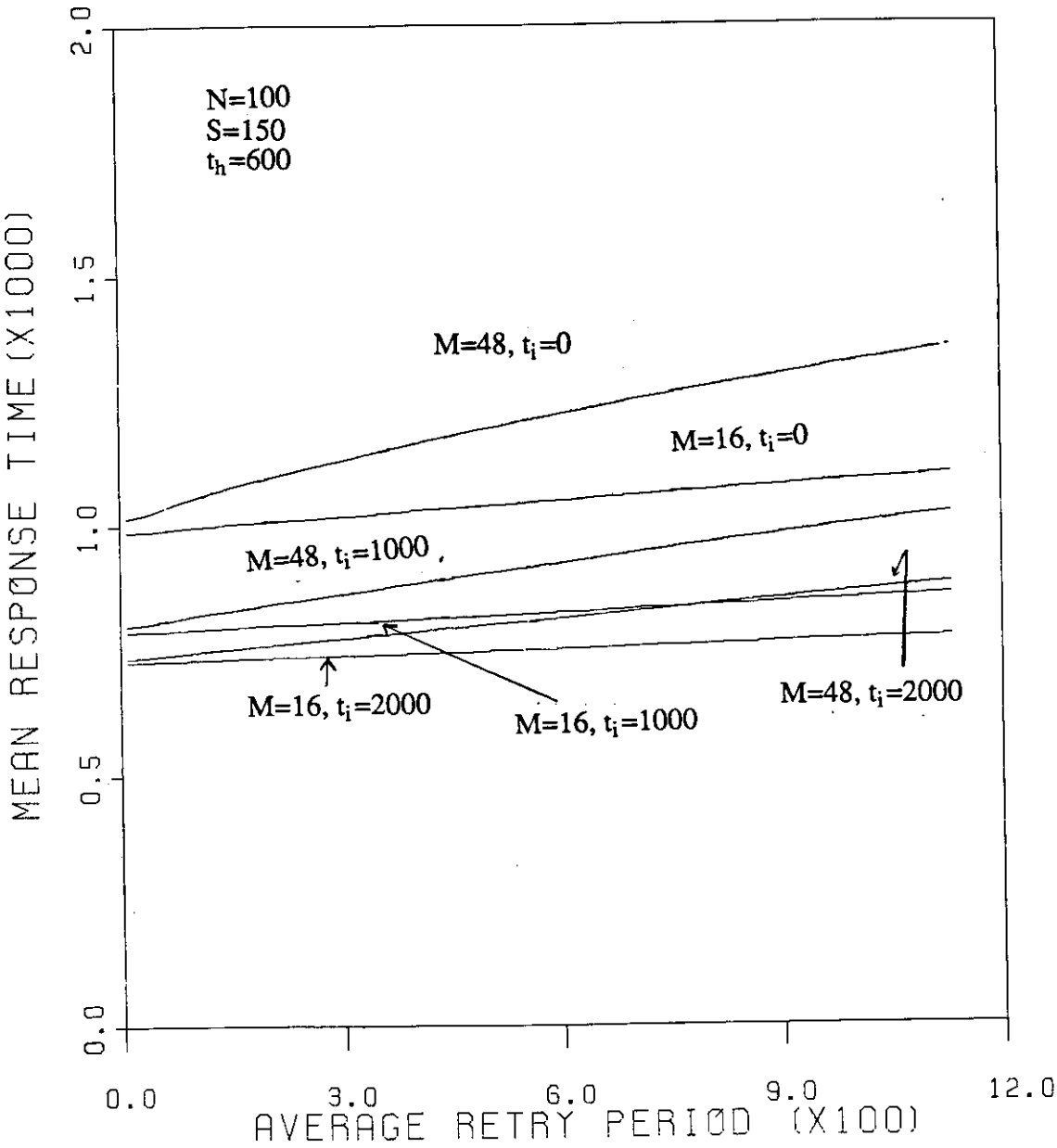


Figure 14 Lock Response Time as a Function of Retry Period for Selected Pairs of M and t_i (N=100)

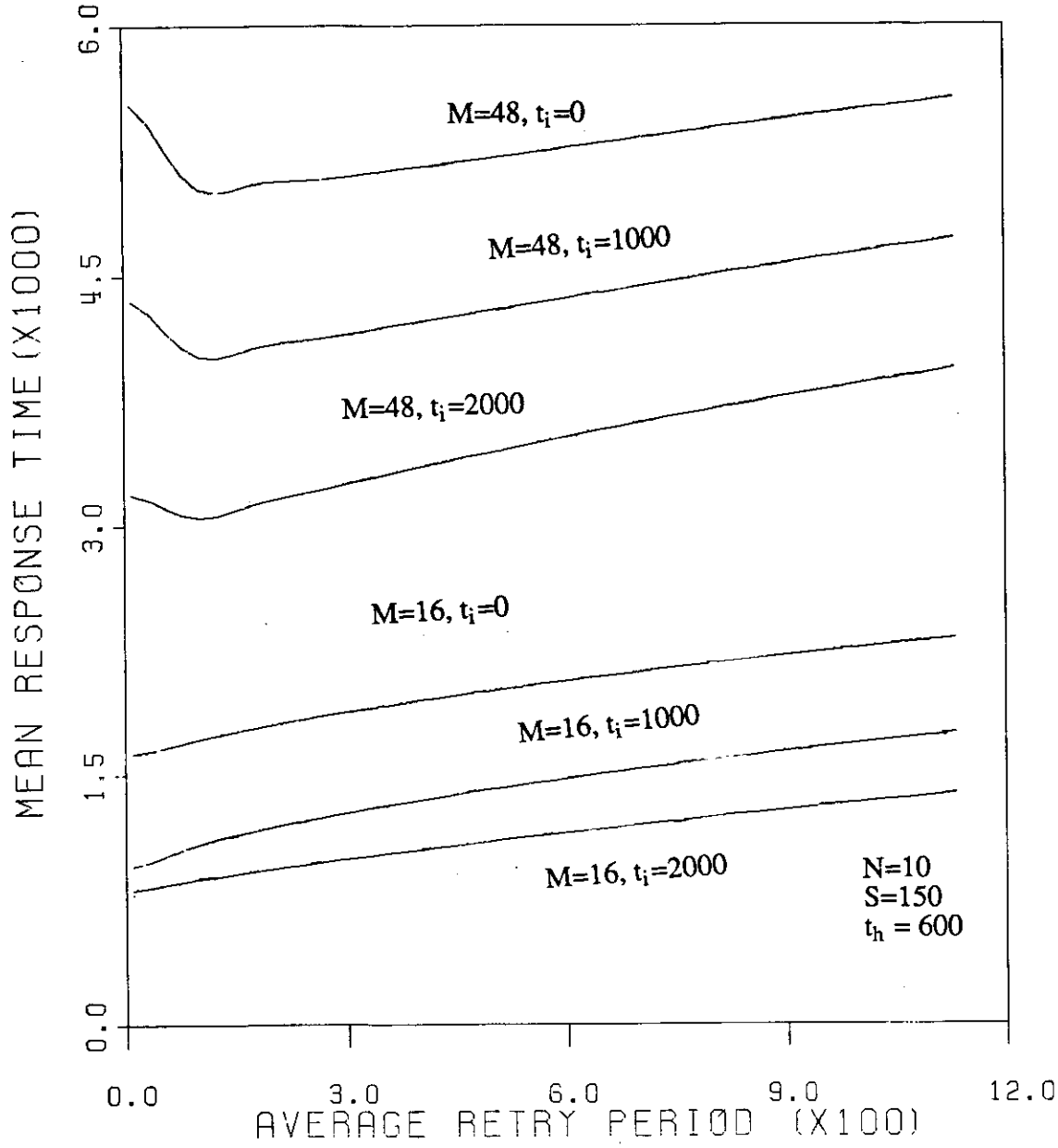


Figure 15 Lock Response Time as a Function of Retry Period for Selected Pairs of M and t_i (N=10)

6. REFERENCES

[BAR81] Y. Bard, "A Simple Approach to System Modeling" Performance Evaluation pp 225-248 1981

[CHU86] Wesley W. Chu and Jung M. An, "Fault Tolerant Locking (FTL) For Tightly Coupled System" Proceeding of Fifth Symposium on Reliability in Distributed Software and Database System, pp 49-55 Jan. 1986

[KLE75] L.Kleinrock, Queueing System, Vol. 1 : Theory, pp 221 Wiley-Interscience, 1975

[LIT61] J.D.C. Little, "A Proof of The Queueing Formula $L = \lambda W$," Operations Research, pp 383-387 1961

CHAPTER V

RESILIENT REAL-TIME DISTRIBUTED DATABASE MANAGEMENT IN THE SDI ENVIRONMENT

RESILIENT REAL-TIME DISTRIBUTED DATABASE MANAGEMENT IN THE SDI ENVIRONMENT

1. INTRODUCTION

The Strategic Defense Initiative (SDI) environment poses several data management problems. We must maintain consistency amongst several files which are distributed over a network. Different sites in the network may contain different files. Our software must operate in real-time. Although there are several protocols to maintain file consistency in a distributed database, most of them are not real-time. In addition, our applications must continue operation despite failures of one or more sites. This chapter presents the resilient Exclusive Writer Protocol (EWP) [Chu 85] and Primary Site Lock Protocol (PSL) [Stonebraker 79], two proposed real-time protocols for distributed data management. [An 85]

2. ENVIRONMENT

Figure 1 shows a set of six sites covering different sections of the sky. Some regions are covered by more than one processor. In the corresponding logical environment (Figure 2), links are drawn between those sites which share common areas. Thus, for site 1 to send a message to site 5, the message must pass from 1 to 2, from 2 to 3 , and from 3 to 5. Even if site 3 crashes, we can still send the message via site 4. However, if site 2 crashes, then site 1 will be cut off from the rest of the network and the message will be lost or stalled. This situation is called *partitioning*.

Any configuration of sites into a network is a composition of several *simple* configurations. Figure 3 shows several different simple configurations. Initially, we can analyze these simple configurations before tackling more difficult compositions. Notice the composite network in Figure 3 is composed of a three node strongly connected section and a tree, attached

by a ring. Alternatively, we could consider the strongly connected section to be a ring of size 3.

3. INITIAL ASSUMPTIONS

To make some initial calculations, we will first restrict the problem. Then, once we have considered a simplified version, we can relax our restrictions. We assume that partitioning of the network will not occur. After studying and solving the simplified problems, we will introduce partitioning and propose solutions to the problems which arise.

We also assume that a message sent by a site will eventually be received by a destination site as long as the destination is alive and that sites will either operate properly or crash. We will never have a site which is sending information which is undetectably incorrect. These are hardware tolerance assumptions.

4. PROTOCOL DESCRIPTIONS

For all protocols, all sites are numerically ordered. The lowest numbered site is the *Primary Site* or *Exclusive Writer*. Updates are sent out based on this ordering (lowest site first). Different files can have different numberings, so that each file can have a different primary site. To maintain a failure check, each site must send *I_AM_UP* messages. These messages will be sent to all sites, or just neighbors (or some compromise) depending on communication and time restrictions. If a site stops sending I_AM_UP messages it is assumed that site has failed (crashed). When the primary site fails, the next lowest site takes over as the new primary site.

Figure 4 shows the instructions for the primary site under EWP. This site monitors incoming update-requests and sends the update out to all other sites which have a copy of the file affected. In addition, the primary site must maintain a network directory to determine which sites receive a copy of the update. This directory is adjusted as sites fail and recover.

Other sites process normally until a file must be updated. When that happens, they send an update-request to the primary site. The request should include a *sequence number* specifying the last update on this file at this site. The primary site can use this number to determine which sites have fallen behind in updates and which updates must be discarded.

If the site is the lowest numbered "normal" site in the network, then it has the additional task of monitoring the primary site in case of failure. If the primary site fails, then this site takes over and must broadcast the most recent update to all other sites, in case the primary site failed half way through a transmission. As a result, duplicate messages may be sent. These duplicates can be discarded based on the sequence number.

Primary Site Locking is similar to EWP. The main difference is that the updating sites request a lock from the primary site. Once the lock is granted, the updating site sends out all of the updates. If a lock is not readily available, then the requesting site must either be denied the lock or wait for the lock to be granted. In either case there is a synchronization delay and an extra set of messages must be passed to grant or reject the lock request.

To determine which update to accept under EWP or which lock to grant under PSL, sequence numbers are kept for each file. A sequence number starts at one and increments after each update to the file. The sequence number is included in all update requests when they are sent to the EW. The EW can then determine if the requested update was based on the most recent version of the file or not. Perhaps the site has slipped off the network briefly and then rejoined, missing an update in the interim. The EW can detect this and can resend the update from the log backup.

The main advantage of these protocols is that they use very little inter-site communication. We expect communication to be our major cost in this application. We sacrifice some consistency control when we reduce the number of messages; we can still have conflicting update

messages. However, this application will generate frequent writes to the database as objects are tracked. Thus, even though we may have two sites updating the position of an object at the same time, the updates will be almost the same. Soon, another update will occur which will overwrite the previous updates with new information anyway.

If we want further data consistency, we can implement checking and voting algorithms on top of EWP or PSL. These algorithms are not discussed here.

5. OVERHEAD ESTIMATIONS

Let us first consider the overhead of I_AM_UP messages. Assuming files are replicated on all sites, and the network is strongly connected, we have several important parameters.

- m = number of sites
- n_c = number of I_AM_UP messages
- l_c = size of I_AM_UP messages
- n_u = number of UPDATE messages
- l_u = size of UPDATE messages
- n_{rc} = number of REQUEST CONFLICT messages
- l_{rc} = size of REQUEST CONFLICT messages
- n_r = number of RECOVERY messages
- l_r = size of RECOVERY messages

Based on these parameters, we find the following:

$$n = n_c + n_u + n_{rc} + n_r$$

$$n_c = \begin{cases} m & \textit{if messages are sent to one neighboor} \\ m^2 & \textit{if messages are sent to all sites} \end{cases}$$

$$\textit{Communication Cost} = n_c \times l_c + n_u \times l_u + n_{rc} \times l_{rc} + n_r \times l_r$$

Assuming:

- $l_c = 1$
- ignore recovery ($n_r = 0$)
- $l_u = l_{rc}$
- $n_{rc} = f(n_u)$ since more updates mean more conflicts

Total Traffic in the two cases for I_AM_UP is approximately:

$$m + (n_u \times m + m_{rc}) \times l_u$$

$$m^2 + (n_u \times m + m_{rc}) \times l_u$$

This analysis means that if too many I_AM_UP messages are sent, the network will become swamped with messages, while if too few messages are sent, we won't recognize sites which have failed fast enough. Figure 6 contains a qualitative analysis of our expected results. The top three graphs show expected communication costs depending on how many sites receive each I_AM_UP message. Notice that if all other sites receive the message, the quadratic nature will cause communication costs to be prohibitive for a large number of sites.

The middle graph of Figure 6 presents a family of curves showing the expected recovery time depending on the reliability of sites and the frequency of I_AM_UP messages. If reliability is high, then I_AM_UP messages need not be sent frequently. However, if reliability is low, we must send frequent I_AM_UP messages or risk waiting longer for a node which has failed. If the I_AM_UP frequency is too high, our network begins to get flooded with messages.

The bottom graph is a composite of the results from the top of the page and the middle graph. On the left side, extra messages are sent to failed nodes; on the right, the network is clogged. In between these undesirable areas, there is an optimal area which we will determine using simulation techiniques.

6. CURRENT WORK

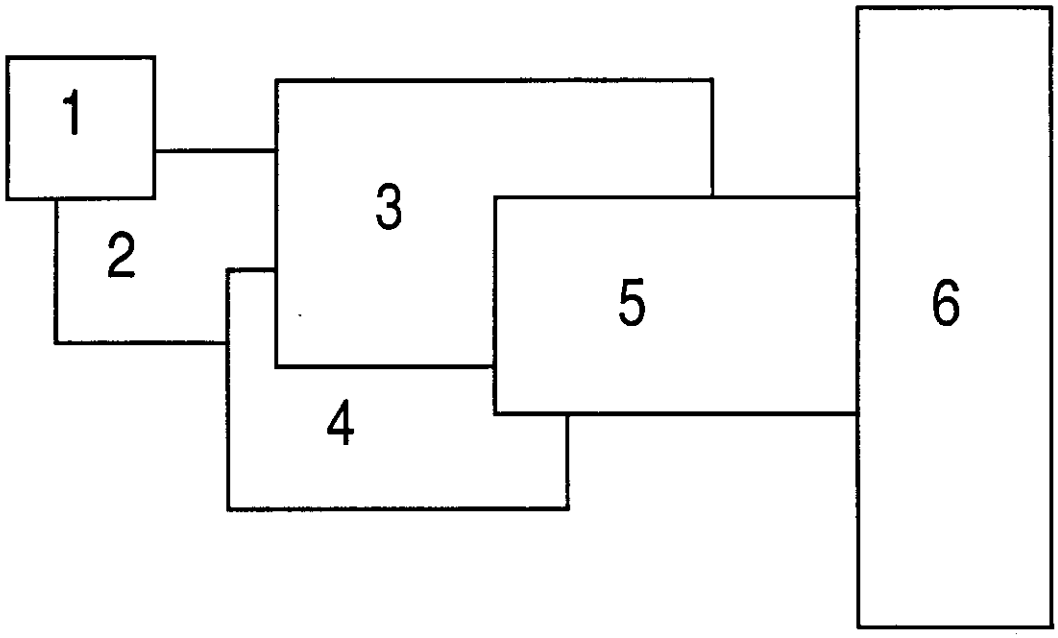
Currently, we are building a simulation for a distributed real-time system. The simulation will be able to measure communication and computation costs along with the amount of time spent waiting for other processes. The results of the simulation will rely on such key parameters as number of sites, frequency of I_AM_UP, operation costs, frequency of update messages, message size, and the reliability of individual sites in the network.

The simulation consists of several data structures. The first is a list of commands for each site along with the associated costs. Specificially, there is an 'EW' list and a 'Non-EW' list of instructions. An ordered list of sites sorted by local time provides the next process to select. That site then 'executes' one step in the instruction list and is inserted into the site list with an updated time. If a message is sent, then it is put into a queue at the correct site.

A flow-chart of the simulation is shown in Figure 5. The data is initialized, and all important parameters are read in from a data file. Then, one by one, processes are 'popped' from the ordered site list. Thus, the site with the smallest local time is found. The psuedo-instruction for this site is found, along with its cost. The instruction is completed; messages are sent and files are 'written' as appropriate. Then, the new site time calculated, and the site is re-inserted into the list in its new proper place. The simulation repeats this until one of the processes has a local time which is greater than the specified end time.

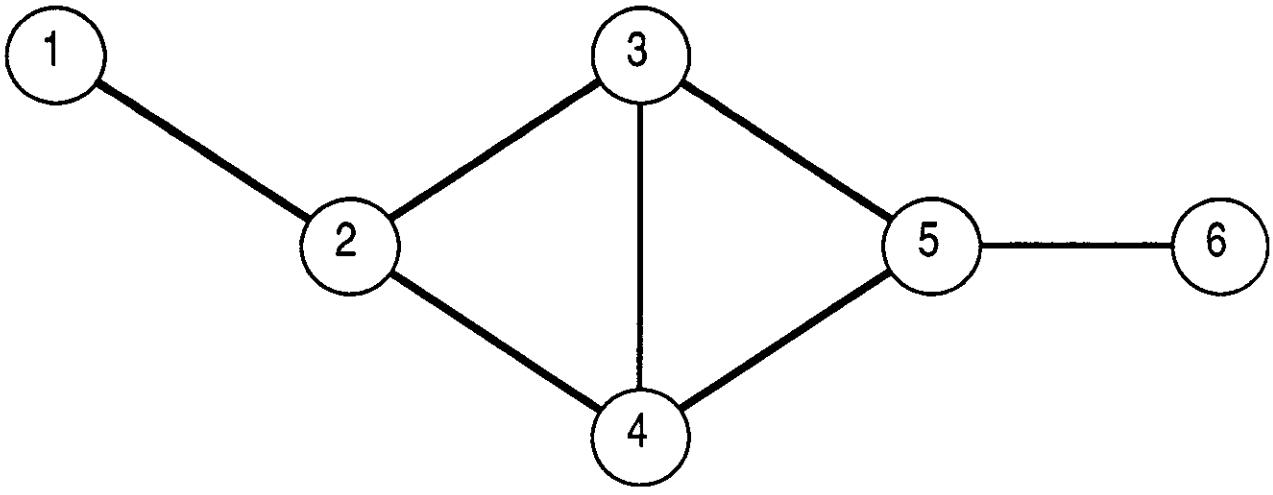
We plan to use the simulation to verify our analytical results and charactarize the behavior of the resilient protocol. We can then use the simulation to determine the frequency of I_AM_UP messages, and the loading limitations of a given system. We can also use the simulation to inject faults into the system and determine the resilience with respect to amount of recovery time and number of simultaneous faults.

Using the results of this simulation, we can to determine the relationships between different system parameters, providing us with insights on the behavior and feasibility of the proposed protocols for operating in the SDI BM/C^3 environment. We can then compare simulation results with the expected results of Figure 6.



Physical Environment

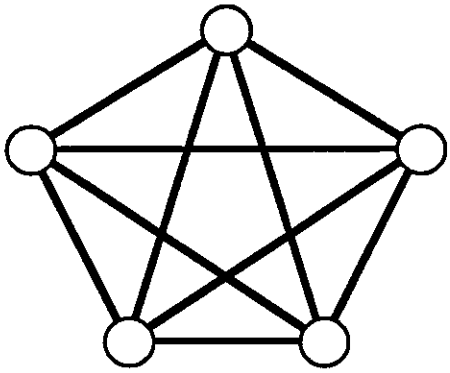
Figure 1



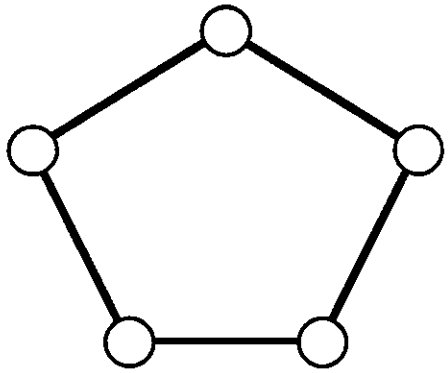
Logical Environment

Figure 2

Figure 3
Possible Physical Configurations

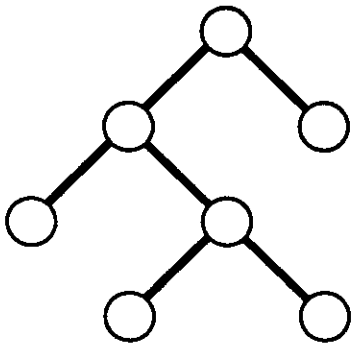


Strongly Connected
all nodes directly connected

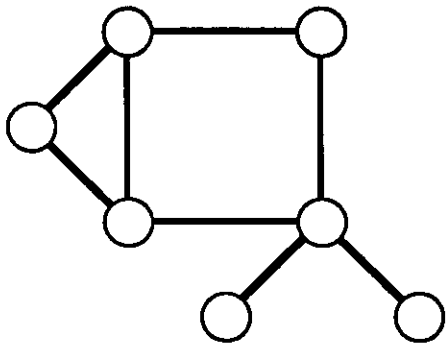


Ring
2 paths to each node

Line
minimal connection



Tree
minimal connection



Composite

Figure 4
Instructions for Primary Site

1. When an Update Request arrives:
   Check Sequence Number to determine acceptence
   If the Update Request is accepted:
      "Stamp" Update with Sequence Number
      Send Update to all nodes with a copy of the file
      Add Update to Log
      Increment Sequence Number
   If the Update Request is rejected:
      Send "reject" to originating site

2. Maintain a current network directory
   I_AM_UP messages determine status at each site
   I_AM_UP contains current Sequence Number

3. If a site fails
   Mark site as "down" in directory
   Stop sending messages to that site

4. If a site returns after a failure
   Send "back" message if there aren't too many
   Resume sending messages to that site

Figure 5
Flowchart of Simulation

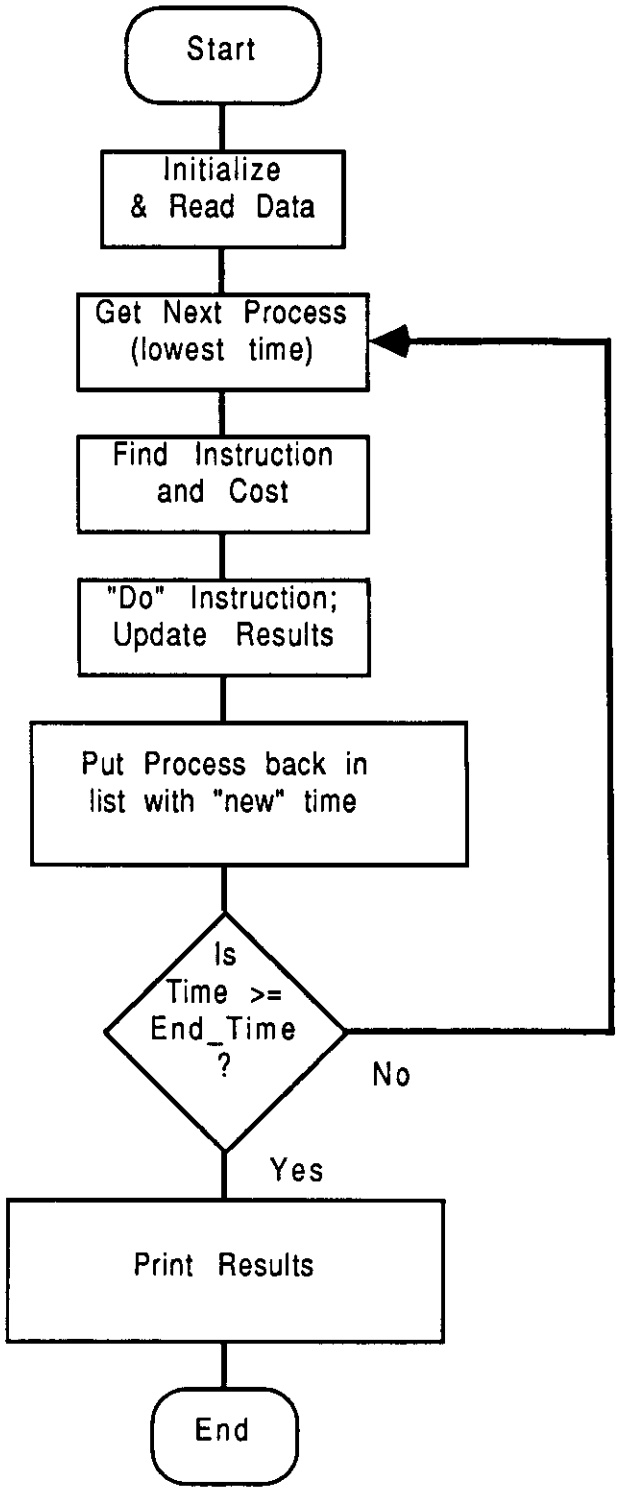
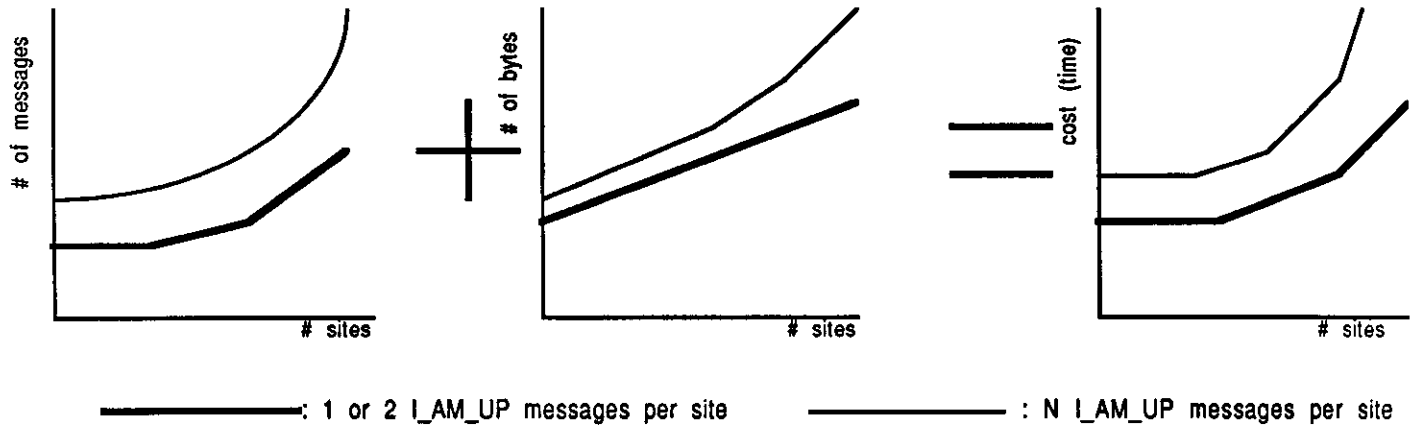
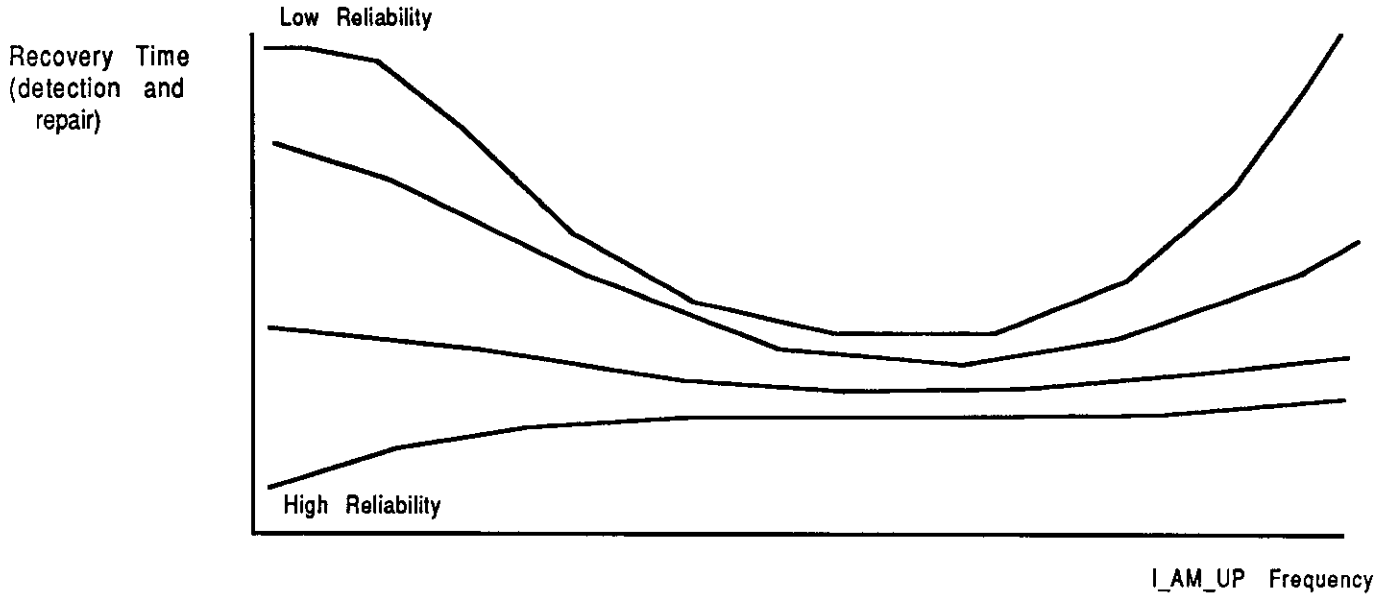
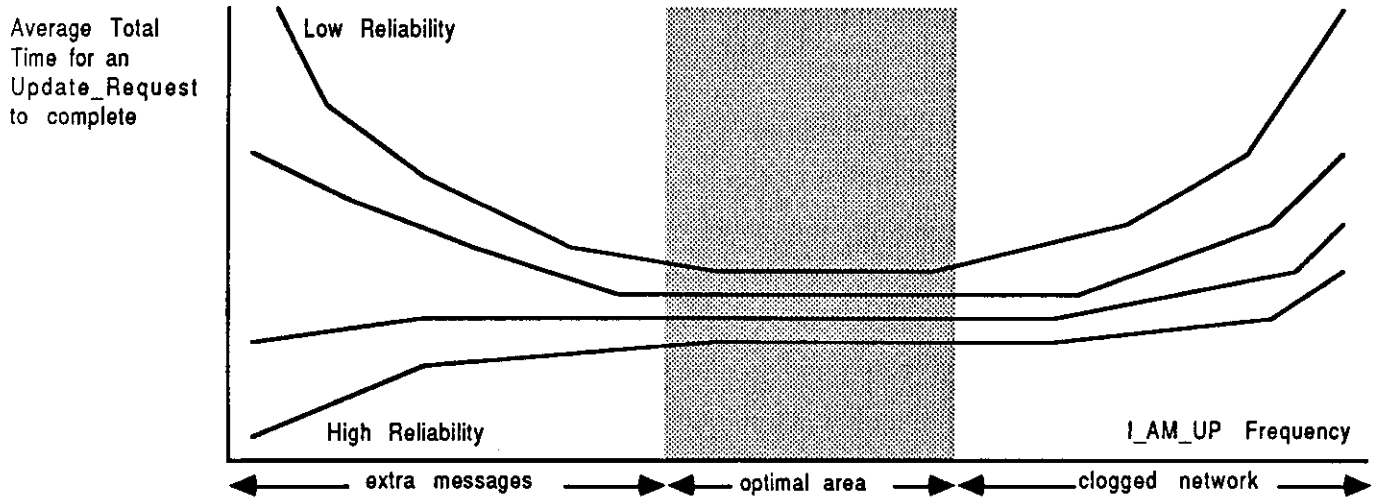


Figure 6

Proposed Qualitative Analysis of Update Completion Time







References

[An 85] Jung Min An and Wesley W. Chu, "A Resilient Commit Protocol for Real Time Systems," *Proceedings of the 1985 Real Time Systems Symposium*, San Diego, California, December 1985.

[Chu 85] Wesley W. Chu and Joseph Hellerstein, "The Exclusive Writer Approach to Updating Replicated Files in Distributed Processing Systems," *EEE Transactions on Computers,* pp. 489-500, June 1985.

[Sto 79] Michael Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," *IEEE Transactions on Software Engineering,* Vol. SE-5, No. 3, pp. 188-194, May 1979.

CHAPTER VI

OPTIMAL QUERY PROCESSING WITH SEMANTIC REASONING

OPTIMAL QUERY PROCESSING WITH SEMANTIC REASONING

1 INTRODUCTION

With the advent of computer network technology, the reliable, efficient and economical transfer of data among computers and terminals becomes feasible. Advances in this field provide us with a technological foundation to implement distributed computing and distributed database on a computer network. The need of geographically separate databases to process common information files has motivated their connection into a distributed database. Interconnecting the remote databases together provides real-time retrieval, update, and distribution of large quantities of information in addition to the sharing of data files. Distributed database systems exhibit tolerance to site failure and fault tolerance due to multiple copies of data files.

When organizing and planning a distributed database system, many problems need to be addressed. These include file allocation policy, directory design and distribution, deadlock avoidance, integrity and consistency in updating multiple copies of data files, optimal query processing policy, reliability and recovery as well as privacy and security issues. In this chapter, we shall consider the problem of query optimization. We shall investigate the applicability of the semantic knowledge of the application domains to query processing in an effort to reduce query response time. This technique is often referred as Semantic Query Optimization [HAMM80, KING81, XU83, CHAK84, JARK84a].

Conventional approaches to the query optimization problem use domain-independent techniques to transfer the original query to a set of operation sequences. Optimization usually involves the determination of the sequence of operations and sites for performing this set of operations such that the operating cost (communication cost and processing cost) for processing

a given query is minimized [CHU82].

In contrast to conventional query optimization, semantic query processing uses knowledge of application semantics to transform the original query into a semantically equivalent one that is cheaper to process. The result is not an alternative processing strategy as in conventional query processing, but rather a different query that is cheaper to process while still yielding the same result as the original query because of the semantics of the application.

Since the domain knowledge varies from application to application and knowledge may change as the state of the database changes, we propose a rule-based architecture to implement the Semantic Query Processor (SQP). With this approach, new knowledge can be easily added and out-of-date knowledge easily removed from the knowledge base at any point of time. The SQP consists of two components: the inference engine and the knowledge base. The inference engine integrates heuristics which are specified as sets of rules to guide the selection of knowledge for semantic query transformation and these heuristics seldom change among different domains. Knowledge in the knowledge base is expressed declaratively as rules and facts to make the knowledge base modular and easy to change. Because knowledge and heuristics are specified declaratively, both can be modified, added, and deleted without affecting the operation of the system. This makes the system highly modular and extendable. Since the heuristics guiding the query transformation are implemented independent of a specific application domain, the same set of optimization heuristics can be used for many application by interchanging the knowledge base describing different application domains. All of these factors make a rule-based approach ideal for implementing SQP.

In the sections that follow, we present an overview of the semantic query optimization, a general architecture for the SQP system, examples of how the knowledge and transformation techniques are specified and work in SQP, and finally, the development of SQP and future research plans.

2 SEMANTIC QUERY OPTIMIZATION

The goal of semantic query optimization is to transform the original query into a semantically equivalent but more efficient form with the use of database application domain knowledge. We provide an example that illustrates the necessity and effectiveness of semantic query optimization.

2.1 AN EXAMPLE OF SEMANTIC QUERY OPTIMIZATION

In the following discussion, we will take examples from a SHIP database management system that monitors the movements of about 10,000 ships. Each ship visits about 20 ports per year. A monthly data tape listing the ports visited by each ship is sent to the DBMS. Such a database consists of two entities:

SHIP = (Ship_Id, Ship_Type, Draft, Registry)

PORT = (Port_Id, Port_Type, Depth, Country)

and one relationship

VISIT = (Ship_Id, Date, Port_Id, Reason).

Ship_Id is the key for the relation SHIP. Port_ID is the key for the relation PORT. Ship_id and Date together provide the key for the relation VISIT. The Entity-Relationship conceptual model of this database is shown in Figure 1. The SHIP and PORT relations contain some relatively permanent characteristics such as a ship's draft or a port's channel depth, etc. The VISIT relation contains the list of ports visited by each ship in addition to other information. Suppose that the database system uses a hierarchical implementation to reflect the way the data is received. In this case, each ship has a pointer indicating a list of ports it has visited (Figure 2).

The database supports diverse queries from various groups of users. Some classes of queries can be answered rapidly. For example, consider the query:

Q1: "*List the ports visited by ship X during the year 1980.*"

Pointers direct access to selected items in order to quicken the response time for these queries (Figure 3). However, this database structure is not well suited to process queries such as the following:

Q2: "*Determine the ship name and date of all visits to port A during the year 1980.*"

To answer this query, we need to access each of 10,000 ship records and check each of the 20 visits to find the ships that have visited port A. This is equal to 200,000 record retrievals (Figure 4).

Suppose the database administrator knows certain physical characteristics of port A that limit the types of ships that can visit it. For example, port A has a shallow channel depth of 20 feet. The database administrator can transform the query to exploit this knowledge and avoid inspecting every visit record. Instead of asking for all visits to port A, the user can ask for all visits to port A by ships whose draft is less than 20 feet. Only when the ship's draft is less than 20 feet, will the port records be scanned to answer the query. Adding the extra constraint replaces a scan of 10,000 ships plus 200,000 visits with a scan of 10,000 ships plus a number of visits which is far less than 200,000. If only five percent of the ships have drafts of less than 20 feet, then only about 10,000 visits need to be checked, which results in a significant reduction in retrieval cost. We call this type of improvement knowledge-based query optimization or semantic query optimization.

2.2 SEMANTIC EQUIVALENCE OF QUERIES

Two queries are considered to be semantically equivalent if they result in the same answer in any database instance that conforms to the semantic integrity constraints [HAMM75]. Semantic equivalence is not the same as logical equivalence. Two queries are logically equivalent if one can be transformed into the other by the application of standard logical equivalences such as De Morgan's Laws. Logically equivalent queries are obviously semantically equivalent, but semantically equivalent queries need not be logically equivalent. That is, two semantically equivalent queries might yield different answers when posed to the database in a state where some semantic integrity constraint is violated.

For example, suppose there is a semantic integrity constraint to the effect that a ship can only visit a port with a channel depth greater than the ship's draft. If the database conforms to this condition, then the query:

Q2: "*List the ship name and date of all visits to port A during 1980.*"

is semantically equivalent to the query:

Q2': "*List all visits to port A by ships whose draft is less than 20 during 1980,*"

assuming port A has a channel depth of 20 feet. The answers will be the same because the enforcement of the semantic integrity constraint guarantees that there is no item in the database that contradicts the aforementioned condition. However, if the constraint is violated, then these two queries will produce different results.

Database semantic integrity is enforced by the independence of semantic euqivalence and database state. Integrity checking ensures that every allowable state of the database is a valid instance of the application. No database state can be reached with a violation of the semantic integrity. There is a violation of the semantic integrity if the database contains values that cannot

be attained in the application. The semantic integrity constraint for the aforementioned example can be stated as:

IC1: "For each pair of (ship, port) in the relation VISIT, ship's draft must be less than or equal to port's depth."

2.3 SEMANTIC QUERY TRANSFORMATION

The general approach to the semantic query transformation is similar to the technique for integrity checking which is implemented by adding constraints to the query expressions. The principle of this approach is as follows: Let c_1, c_2, ..., c_n be a set of domain knowledge represented as integrity constraints satisfied by a database state. By a sequence of logical transformations, the original query Q is translated into Q' subject to c_1, c_2, ..., c_n such that Q' yields lower processing cost than Q. The semantic query optimization problem is to determine the set of c_1, c_2, ..., c_n that yields the minimum query processing cost; that is,

$$C(Q') = \min_{\substack{c_i \\ i=1,\ldots,n}} C(Q \Lambda c_1 \Lambda \cdots \Lambda c_n)$$

To illustrate this concept, suppose that we have a piece of knowledge of the form $P(x) \rightarrow C(x)$, where the variable x ranges over some relation R and C(x) is some constraint on x. This knowledge states that if the property of P(x) is satisfied, so is the property of C(x). Let's suppose query expression Q contains the term P(z) where the variable z ranges over the same relation as the variable x. By using the technique of integrity checking by query modification proposed by Stonebraker[STON75], we can transform Q into a semantically equivalent query expression Q" which contains the conjunction:

$$P(z) \Lambda (P(z) \rightarrow C(z)).$$

That is,

$$Q''(z) = Q(z) \wedge (P(z) \rightarrow C(z)).$$

However, by the logical equivalence

$$(A \wedge (A \rightarrow B)) \equiv (A \wedge B)$$

we can replace this conjunction by the simpler conjunction,

$$P(z) \wedge C(z)$$

that yields

$$Q''(z) = Q(z) \wedge C(z).$$

The effect is to use the original query constraint P(z) to infer the new constraint C(z) by means of query modification. By repeatedly applying this technique, we will derive an expression from which no further constraints can be inferred. Let Q' be this final expression. The technique used to infer new constraints is similar to the technique of mechanical theorem proving in AI research. A thorough discussion of the transformation technique can be found in [CHAN73].

3 SEMANTIC QUERY PROCESSOR ARCHITECTURE

As we mentioned earlier, the SQP system consists of two parts: the knowledge base and the inference engine. The knowledge base maintains the domain-dependent knowledge. The inference engine contains the domain-independent heuristics that guide the query transformation. All information that is specific to the application domain should be factored out of the transformation heuristics and maintained in the knowledge base.

Thus, we have proposed a rule-based architecture with a knowledge base and an inference engine to implement the Semantic Query Processor (SQP) (Figure 5). To make the knowledge base modular and easy to change, knowledge about the application domain is ex-

pressed declaratively as rules and facts in the knowledge base. Facts specify the characteristics of the underlying database application, such as:

F1: "There are about 5% ships having draft less than 20 feet,"

from our previous example in Section 2. This piece of information is specified in Prolog form and stored in the knowledge base as:

draft_statistics(10, 20, 0.05).
draft_statistics(20, 40, 0.25).
draft_statistics(40, 60, 0.35).
draft_statistics(60, 80, 0.40).

assuming that the drafts of ships range over 10 to 80 feet. This fact specifies the statistics of the ships' draft. The first value gives the lower bound, the second one the upper bound, and the last value the statistics.

Rules specify the relationships between attributes and relations, which are sometimes referred as integrity constraints. The format of the rules is given as $P(x) \rightarrow Q(x)$ which states that the property of P(x) implies the property of Q(x). In another words, the property of Q(x) can be satisfied if the property of P(x) is satisfied. For example, in our previous example, the rule:

R1: "Ship A can visit port X only if the draft of ship A is less than the channel depth of port X."

can be specified in Prolog form as:

visit(X, A) **:-**
    ship(X, _, Draft, _),
    port(A, _, Depth, _),

Draft < Depth.

In Prolog format, the string starting with a capital letter is a variable, so *X*, *A*, *Draft* and *Depth* are all variables and can be instantiated to any valid values. This piece of information is actually one of the integrity constraints stating the relationship among **visit**, **ship**, and **port** relations, and is useful for query transformation in the semantic query processing, as shown in our previous example.

In SQP, the transformation heuristics are specified as a set of **if-then** rules. The **if** part specifies the condition under which this rule can be used. The **then** part specifies the actions that should accompany this rule. The format is as follows:

if *condition*
then *action.*

The condition is a predicate which is evaluated to be *true* or *false* depending on the state of the database and the current query that is being processed. If the condition is satisfied, this rule will be fired; that is, the action part of this rule will be executed. Because the action may satisfy the conditions of some other rules, they may also be fired in response to the firing of this rule. This process will continue until no further rules can be fired.

The success of semantic query processing depends on the development of efficient heuristics for selecting the best of many possible transformations (adding a set of integrity constraints to the query). Omitting a constraint may result in a lost optimization, while including an irrelevant one may increase execution time. We are currently developing the heuristics for selecting the optimal set of constraints to optimize a query.

The conventional query processor (CQP) takes the given query as its input and generates an output plan (a sequence of retrieval operations in the physical database). The SQP functions

as a preprocessor that translates users' queries into semantically equivalent forms by adding additional constraints onto the original query expressions. The knowledge used to infer the additional constraints is represented in the same format as the integrity constraints in the data dictionary. The conventional query optimizer then takes the output from the SQP and selects the query processing policy that yields optimal performance.

The advantage of this approach to implementing the SQP is that new knowledge and heuristics for query transformation can be relatively easily added to a working system. Since the specifications of the domain knowledge and the transformation guiding heuristics are declarative, both can be modified, added, and deleted without affecting the operation of the system. This makes the system highly modular and extendable. Since the heuristics guiding the query transformation are implemented independent of a specific application domain, the same set of optimization heuristics can be used for many applications. All of these factors make a rule-based approach ideal for implementing SQP.

4 A SHIP DATABASE EXAMPLE

In this section, we shall use the SHIP database as an example to illustrate how SQP works in transforming the original query to a semantically equivalent yet efficient form. During this discussion, we shall assume the original queries are specified in Prolog form. This restriction is not necessary. We are using it for illustration only. For query languages like SQL or QUEL, we can design a front-end translator and back-end translator that convert the original query language to Prolog form, used in SQP, and back again.

Let us consider the SHIP database from Section 2 again. For convenience, we rewrite these three relations in Prolog form as follows:

ship(ship_id, ship_type, draft, registry).

port(port_id, port_type, depth, country).

visit(ship_id, date, port_id, reason).

Each clause defines the relation name and the associated attributes in each relation. The definiton of the database schema is also stored as part of the knowledge base. To simplify our discussion, we will assume the database systems used in this section are all relational.

For a query like:

Q2: "*List the ship name, its type, and date of all visit to port Francisco during 1980,*"

the corresponding query in QUEL is given as:

```
range of s is SHIP
range of v is VISIT
retrieve (v.ship_id, v.ship_type, v.date) where
        s.ship_id = v.ship_id and
        v.port_id = "francisco".
```

One way to specify this query in Prolog is given below:

```
answer( Ship, Type, Date ) :-
        ship( Ship, Type, _, _),
        visit( Ship, Date, francisco, _).
```

The conventional query processor would perform an equal-join on **ship** and **visit** relations and project the columns *Ship*, *Type* and *Date*. Nothing else can be done to improve the query except consider the size of the two relations and perform the equal-join in a more efficient way.

Suppose the following information has been stored in the knowledge base which can be used to augment the original query to yield a more efficient form.

F1: draft_statistics(10, 20, 0.05).

F2: draft_statistics(20, 40, 0.45).

F3: draft_statistics(40, 60, 0.40).

F4: draft_statistics(60, 80, 0.10).

R1: visit(Ship, _, Port, _) :-

ship(Ship, _, Draft, _),

port(Port, _, Depth, _),

Draft < Depth.

In this section, we shall assume that the knowledge base is existent already. (The discussion of development and maintenance of the knowledge base will be presented in the next section).

Let us consider the query **Q2** again. This query contains two relations: **ship** and **visit**. To evaluate this query, it inevitably has to perform the equal-join on these two relations. However, we can improve this query by restricting the size of the relations for performing the equal-join. The only way to restrict the size of the relation is to introduce restriction on some attributes of some relation before performing the equal-join. Thus, we have the following heuristic that can be used to transform any given query to a more efficient form:

H1: *Introducing selection*

if R is a relation referred in Q **and**

A is an attribute in R **and**

A is restricted by some value v (A θ v) **and**

# of tuples satisfying (A θ v) is small

then add constraint (R.A θ v) to Q.

That is, for a relation R in a given query expression Q, if there exists an attribute A in R which is

restricted by some value v, the original query can be improved by adding the new constraint "R.A θ v" which is more efficient to evaluate yet semantically equivalent. The resulting query is as follows:

$$\mathbf{Q'} = \mathbf{Q} \wedge (\text{R.A } \theta\ v).$$

The condition of "# of tuples satisfying (A θ v) is small" is included because the addition of more constraints does not guarantee that query improvement. For example, if the number of tuples satisfying the new constraint is approximately equal to the total number of tuples in the relation, adding further constraints will accomplish nothing but increasing the computation time to check the redundant constraint. This example illustrates the importance of heuristics in choosing the right constraints for query transformation.

Now consider the query **Q2** that contains the relations **ship** and **visit**. By searching the knowledge base, we find that rule **R1** might be a promising candidate for query transformation. Suppose rule **R1** is applicable. We instantiate the variable of Port to the value "francisco". Further suppose that the depth of the port Francisco is 20, known either by inference or by physically storage of this information in the knowledge base. This value is then used to restrict the possible draft values of ship tuples. By checking with the knowledge base, we find that the number of tuples satisfying this restriction is only 5% of the total number of tuples in the **ship** relation. This meets the last condition of **H1** that "# of tuples satisfying (A θ v) is small". Thus, we add the constraint "Draft < 20" to **Q2** and yield the following form:

Q2' = answer(Ship, Type, Date) :=

ship(Ship, Type, Draft, _),

visit(Ship, francisco, Date, _),

Draft < 20.

This transformed query **Q2'** will then be fed into the conventional query processor for processing. The general approach to evaluating this query is to perform the selection on the **ship** relation with the constraint "**ship**.Draft < 20" first, which will reduce the size of the **ship** relation before doing the equal-join down to at most 5%.

Now suppose the following information has been added into the knowledge base either by the database administors or the domain experts:

F5: index(**ship**, ship_type).

F6: repair_facility(angeles, submarine).

R2: visit(Ship, _, Port, repair) **:-**

ship(Ship, Type, _, _),

port(Port, Type, _, _).

F5 specifies that *ship_type* is an index of the relation **ship**. **F6** tells that port Angeles only has the repair facility for submarines. **R2** specifies that a ship can only be repaired at ports with repair facilities for that type of ships.

If we are now interested in answering the query Q3: "*List the ship and the date that the ship has visited port Angeles for repairs.*" The Prolog expression of this query is:

Q3: answer(Ship, Type, Date) **:-**

ship(Ship, Type, _, _),

visit(Ship, Date, angeles, repair).

Notice that both **Q2** and **Q3** have the same query expression, except that the constant value qualifying the tuples of the relation **visit** has been changed from *francisco* to *angeles*, and "repair" is added to give the reason for visit.

Suppose accessing files by index is always faster than scanning the files. We can improve the query by adding a constraint on a clustered indexed attribute of a relation in the query if one can be found. Thus, we have the following:

H2: *Introducing index*

if R is a relation referred in Q **and**
A is an indexed attribute in R **and**
A is restricted by some value v (A θ v) **and**
then add constraint (R.A θ v) to Q.

In the case of **Q3**, either **H1** or **H2** can be used to improve the query. **H1** (Introducing selection) can be applied to transfer **Q3** to **Q3'** in the same manner as given before. However, since we know port Angeles can only repair submarines, and ship's *Type* is a clustered index of the relation **ship**, by applying **H2**, we get an even better query, **Q3"**. Since accessing files by index is usually faster than scanning the files, **Q3"** will save more block accesses than **Q3'** can do.

Q3" = **answer**(Ship, Type, Date) :=
ship(Ship, submarine, Draft, _),
visit(Ship, Date, angeles, repair).

The relation **visit** and the constant "repair" in **Q3** indicate that Rule **R2** might be beneficial. Suppose this rule is selected, then *Port* in **R2** will be instantiated to "angeles", which will then trigger **F6** to be used. Since Rule **R2** forces that the ship's type must be the same as the facility type of the port it visited, the ship's type will then be instantiated to "submarine". Since **F5** states that ship's type is a clustered index of the relation **ship**, all the conditions of the heuristic **H2** are satisfied and **H2** is then applied to transform **Q3** to **Q3"**.

When several transformation techniques can be used for query transformation, the one which yields the lowest operating cost (computation and communication cost) or the best response time will be selected. It may also be the case that several transformation heuristics will be used in sequence to transform the given queries. In either case, the operating cost or the response time will be used as a performance measure for selecting the heuristics to be used to improve the queries.

In this section, we have shown how domain knowledge and transformation heuristics can be used to transform original queries into equivalent but more efficient queries. In the next section, we shall present methods for obtaining useful domain knowledge and for developing transformation heuristics.

5 DEVELOPMENT OF SQP

At UCLA, we have a SHIP database which is a relational database and a SDI Battle Management database which is in a flat file data format. We will carry out this research in two phases. In the first phase, we plan to develop a methodology for knowledge acquisition and representation. Next, we shall implement a Semantic Query Processor, with this knowledge represented in Prolog, on top of the SHIP database and evaluate the performance improvement derived from this knowledge.

5.1 Knowledge Acquisition and Representation

There are two categories of knowledge that are useful for semantic query optimization: namely domain-related and system-related. The domain-related knowledge involves the database application domain. It includes such information as the sizes of attributes, the sizes of the relations, the distribution of the attribute values, and the inter- and intra-relationship among attributes and relations. All the knowledge mentioned in previous sections falls in this category. For example, facts **F1** to **F4**, specifying the distribution of the draft values of ships, and rule **R1**

specifying the relationship among **ship** and **visit** relations are all domain-related knowledge.

This type of knowledge comes from several sources. It can be specified by the user, extracted from semantics of the database schema definition, or induced from the current instance of the database. For example, the statistics of the draft values of ships can be either given by the database administrator or calculated with the use of the finite differencing technique[PAIG81] during updating the ship relation. Every time there is an update, either adding/deleting tuples into/from the relation or changing content of a tuple in the relation, the statistics can be incrementally changed without recomputing from the entire database instance. Rule **R1** is extracted from semantics about the SHIP database which assures the integrity of the database.

The knowledge can also be induced from the current content of the database. Rule induction techniques, such as ID3 and ACLS [BUND85], have been widely used in AI research to model learning activity by inducing rules from a set of examples. Consider the following database instance:

ship:	ship_id	ship_type	draft
	001	fishing_boat	20
	002	oil_tanker	100
	003	fishing_boat	15
	004	oil_tanker	200

If we know there exists a relationship between the type of a ship and its draft, then by using the rule induction techniques, we can induce the following relationship from the current database instance:

if 10 ≤ Draft ≤ 20 **then**

ship_type **is** "fishing_boat".

if 100 ≤ Draft ≤ 200 **then**

ship_type **is** "oil_tanker".

The reader should notice the difference between this rule and the integrity constraint stating the relationship between the ship's type and its draft. The integrity constraints state the semantics of a database application domain and are seldom changed during the life time of the database. On the other hand, what is specified by this rule is correct only at this database state and may no longer be correct when the state changes.

With the use of the rule-based approach to implement the Semantic Query Processor, rules in the knowledge base may be modified, added or deleted as the state of the database changes. Finite differencing techniques can also be used here to incrementally update the knowledge base when changes to the content of the database occur with very little overhead added to the database update processing.

Knowledge about the system may also play an important role in query optimization. For example, in a distributed database environment, files may be replicated on several sites. The query processor may select the sites with the lowest average load to evaluate the subqueries. Knowledge about the average load of each machine as well as the speed of the access route can be very useful in reducing the query time in a dynamic environment. The system-related knowledge involves the physical structure of the database. Information such as the file organization, distribution of the files, access method, available indexes, average load of each machine, the level of communication link congestion, and the cost of each operation is all part of system-related knowledge. This knowledge can be useful for improving query processing, and can be easily added to or removed from the knowledge base dynamically.

5.2 Current Status

We are currently developing a prototype SQP system. This prototype system will be implemented in Prolog[CLOC81] and all the knowledge (facts, rules, and heuristics) will be expressed in Prolog form. The input and output queries of the SQP system will also be expressed in Prolog language. In the first stage, we will mainly focus on developing the methodology for knowledge acquisition, and heuristics for query transformation as well as studying the issue of the knowledge representation. Rule induction techniques will be used to automatically induce useful knowledge from the database. The facts, rules, and transformation heuristics will be integrated as a Prolog program working as a pre-processor on top of the conventional query processor. We plan to develop the heuristics to guide the selection of promising constraints for semantic query transformation. The methodology for incrementally modifying the knowledge base will also be developed. The performance improvement based on the number of block accesses saved and the overhead added to the transaction processing will be measured and analyzed to evaluate the cost/performance of this approach.

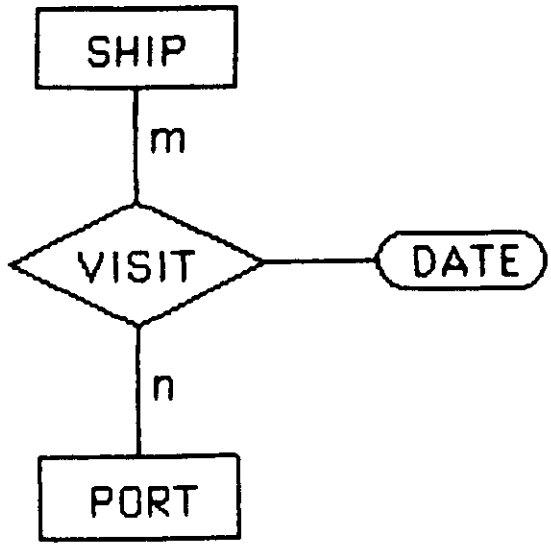


Figure 1. E-R Conceptual Model for SHIP Database

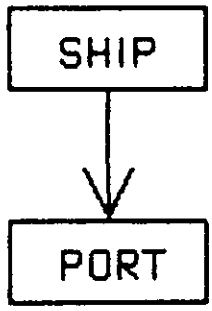


Figure 2. A Hierarchical Implementation of SHIP Database

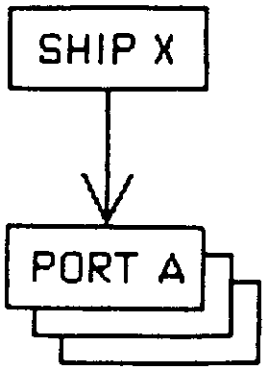


Figure 3. Query Evaluation of Common Queries

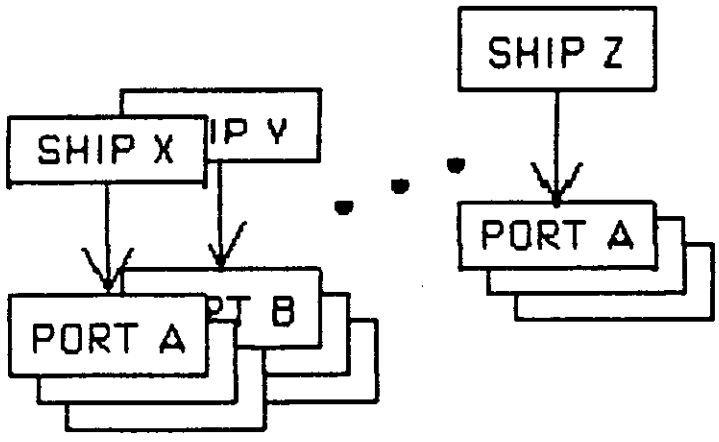


Figure 4. Unanticipated Query Results in Scanning Entire Database

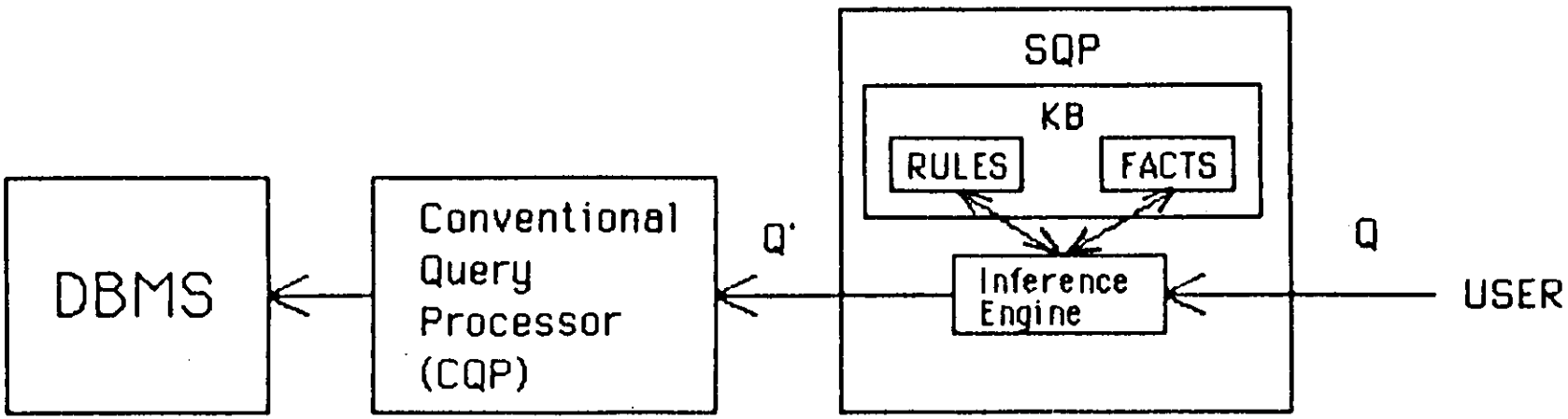


Figure 5. Architecture for Semantic Query Processor

6 REFERENCES

[BUND85] Bundy, A. et al, "An Analytical Comparison of Some Rule-Learning Programs," *Artificial Intelligence* 27 (1985) 137-181.

[CHAK84] Chakravarthy, U. S., Fishman, D., and Minker, J., "Semantic query optimization in Expert Systems and Database Systems," In *proceedings of the First International Workshop on Expert Database Systems* (Kiawah Island, Oct. 24-27), 1984, 326-341.

[CHAN73] Chang, C. L., and Lee, R., *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York and London, 1973.

[CHU 82] Chu, W. W., and Hurley, P., "Optimal Query Processing for Distributed Database Systems," *IEEE Trans. Comput.* C-31, 9, 1982, 835-850.

[CLOC81] Clocksin, W. F., and Mellish, C. S., *Programming in Prolog*, Springer-Verlag Berlin Heidelberg New Yrok Tokyo, 1981.

[HAMM75] Hammer, M., and Mcleod, D., "Semantic integrity in a relational data base system," In *Proceedings of the First International Conference on Very Large Data Bases*, IEEE, New York, pp. 25-47, 1975.

[HAMM80] Hammer, M. and Zdonik, S. B., Jr., "Knowledge-based query processing," In *Proceedings of the 6th International Conference on Very Large Data Bases* (Montreal, Oct. 1-3). IEEE, New York, pp. 137-147, 1980.

[JARK84] Jarkie, M., "External Semantic Query Simplification: A Grahp-Theoretic Approach and Its Implementation in Prolog," In *Proceedings of the First International Workshop on Expert Database Systems* (Kiawah Island, Oct. 24-27), 1984, 467-482.

[JARK84] Jarke, M. and Koch, J., "Query Optimization in Database Systems," *ACM Computing Surveys*, Vol.16, No.2, June 1984, 111-152.

[KING81] King, J. J., "QUIST: A system for semantic query optimization in relational databases," In *proceedings of the 7th International Conference on Very Large Data Bases* (Cannes, Sept. 9-11). IEEE, New York, pp. 510-517.

[MALL86] Malley, C. V. and Zdonik, S. B., "A Knowledge-Based Approach to Query Optimization," in *Proceedings of the First International Conference on Expert Database Systems* (Charleston, April 1-4), 1986, 243-257.

[PAIG81] Paige, R., "Applications of finite differencing to database integrity control and query/transaction optimization," *Advances in database theory, Vol. 2*, ed. H. Gallaire, J. Minker and J. M. Nicolas, Plenum Press, New York, 1981.

[STON75] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification," *Proc. of the 1975 SIGMOD Conference, ACM SIGMOD*, San Jose, June 1975.

[STON84] Stonebraker, M., "Adding Semantic Knowledge to a Relational Database System," in *Conceptual Modeling*, ed. M. Brodie, J. Mylopoulos, and J. Schmidt, Sprintger-

Verlag, 1984.

[XU83] Xu, G. D., "Search control in semantic query optimization," Tech. Rep. #83-09, Computer and Information Science Dept., University of Massachusetts, Amherst, Massachusetts, 1983.

CHAPTER VII

MULTI-AGENT PLANNING

MULTI-AGENT PLANNING

1. Introduction

A typical multi-agent problem solving system consists of a collection of agents, each with various skills such as sensing, communication, planning and acting. The group as a whole has a set of assigned tasks or a global goal. There are three main steps for problem solving in a multi-agent environment. The first step is to decompose the global goal into subgoals and assign them to appropriate agents. Each agent then designs its own plan to accomplish its assigned subgoals. Further negotiation is often required among these agents to accomplish the global goal. The final step is to carry out the global plan.

Contract Net protocol [DAV 81] provides a way to decompose the global goal into subgoals and assign them to appropriate agents. If these subgoals are assumed to be independent, then the agents can generate an individual plan for each subgoal as shown (Fig. 1):

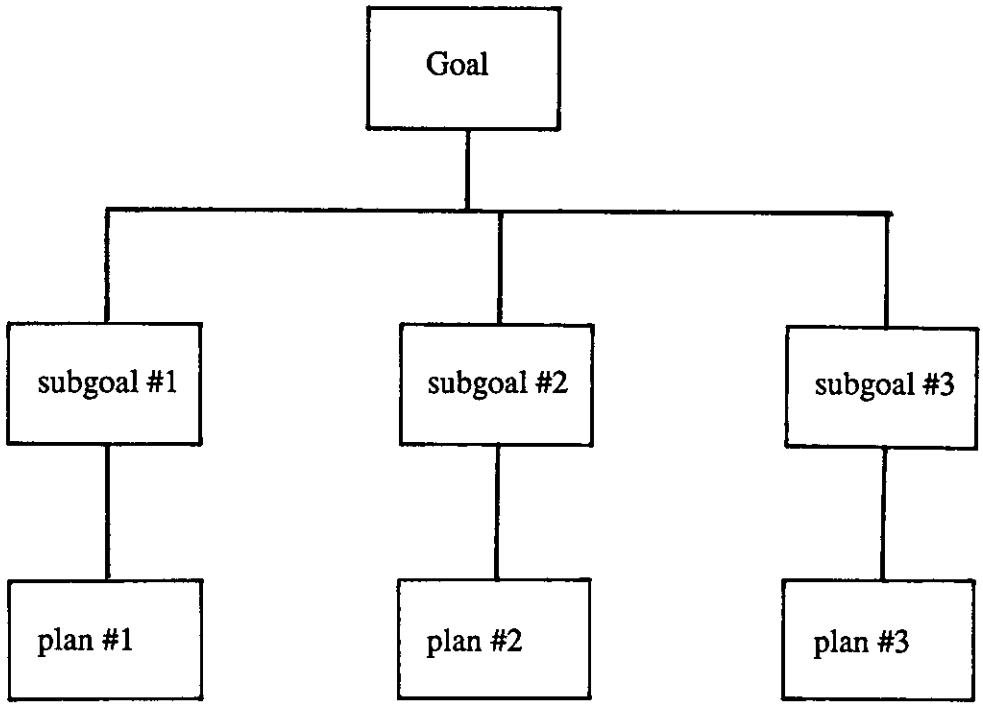


Fig. 1 Individual plan for subgoals

However, subgoals are usually not independent. The dependency of the subgoals among agents prevents each agent from devising its own plan. We shall describe the interaction problem in the following section and propose a two phase algorithm to iteratively generate a plan for multi-agent planning.

2. Interaction in the Multi-Agent Environment

Interactions usually arise because of resource sharing in the multi-agent environment. There are basically two types of interaction: cooperative and competitive. These will be discussed in the following sections.

2.1. Cooperation Between Agents

In order to illustrate this interaction problem, let us consider the following example using the "block world." Consider there are four blocks, red, blue, yellow and green, arranged as shown in Fig 2. The goal here is to move the green blocks to floor F4, the yellow to F5, the blue to F6 and the red to floor F There are two agents to do the jobs. Agent A is responsible for the green and yellow blocks while agent B is responsible for the red and blue blocks.

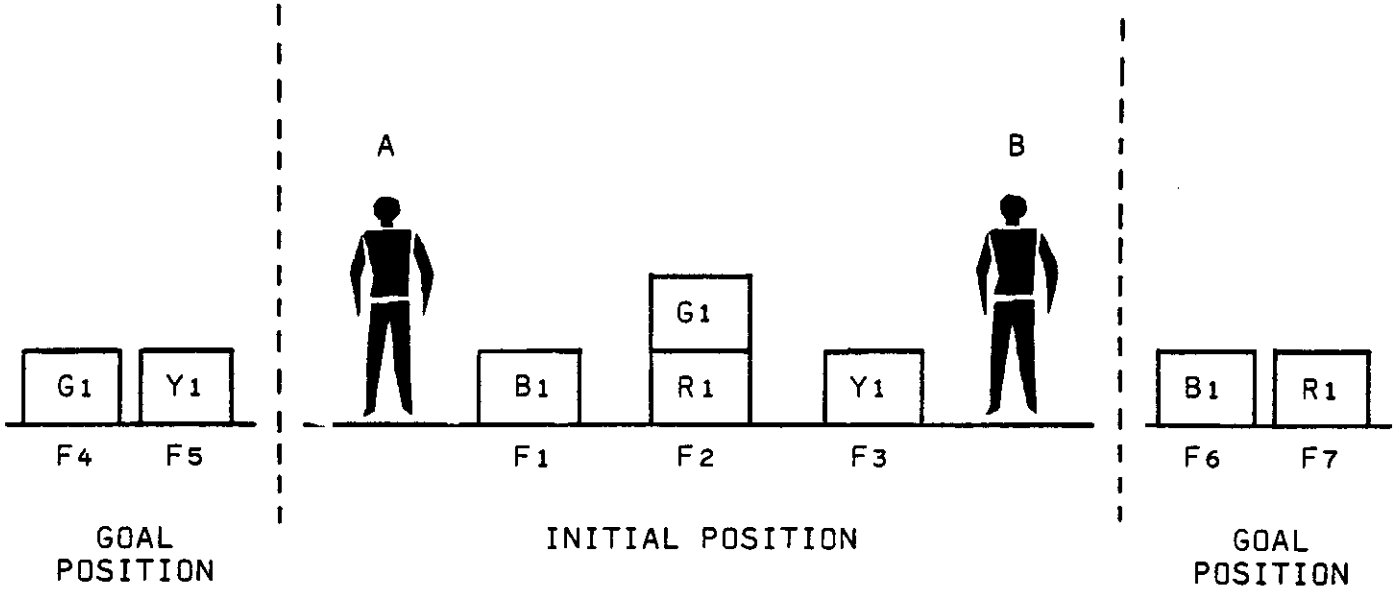


Fig 2 A block world example of cooperation

There are two possible plans to accomplish this goal:

PLAN I

Agent A	Agent B
move_on(G1, F4)	move_on(B1, F6)
move_on(Y1, F5)	wait_clear(G1)
	move_on(R1, F7)

PLAN II

Agent A	Agent B
move_on(Y1, F5)	move_on(B1, F6)

move_on(G1, F4) move_on(R1, F7)

Plan II is better than plan I because agent B finishes its work in 2 steps rather than 3. The main point of this example is to illustrate how one agent can help the other agent without paying extra cost (in our example, agent A can finish its job in two steps in either plan). In cases where an agent has to pay in order to help the other agent, the objective is to develop a plan that minimizes the overall cost.

2.2. Competition Between Agents

The last example shows that it is possible for the agents to interact cooperatively to benefit the overall goal. A situation with competitive interaction through resources will be illustrated in the following example.

This time we have two red, two green and one black blocks as shown in Fig 3. Agent A is responsible for moving the two red blocks to F4, and agent B for moving the two green blocks to F5. A block must reside directly or indirectly on one of the five floors. Notice that the blocks are numbered for easy reference; the agents, however, only see the difference between red and green

blocks.

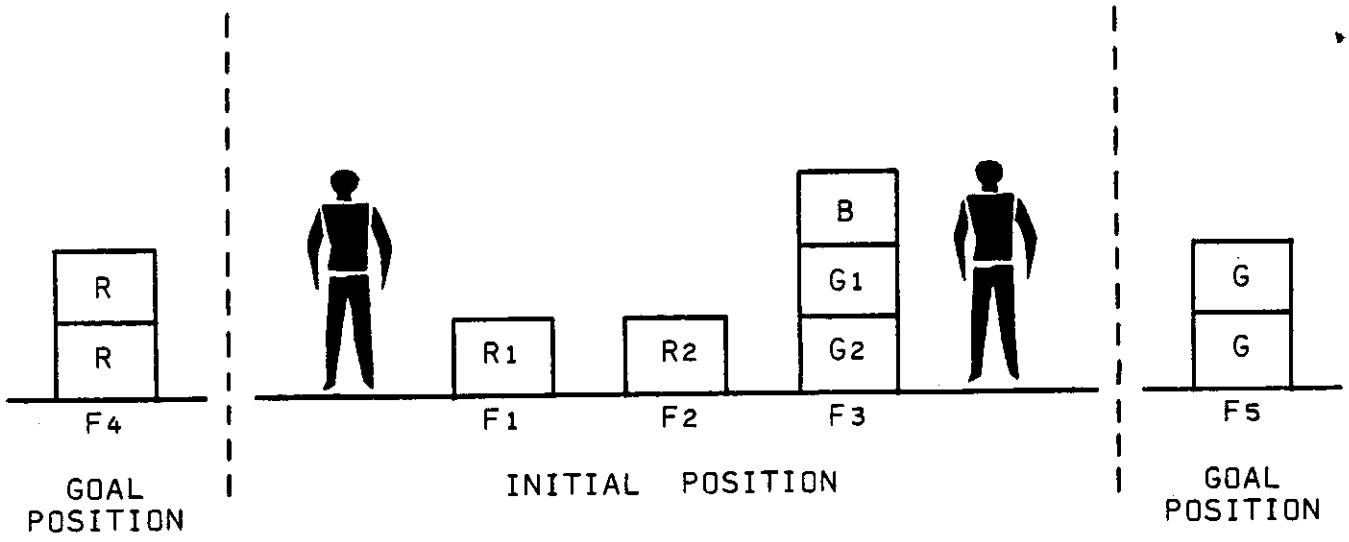


Fig 3 A block world example of competition

A possible plan (not an optimal one) is:

Agent A	Agent B
move_on(R1, F4)	move_on(B, R2)
wait_clear(G1)	move_on(G1, F5)
move_on(B, G2)	wait_clear(R2)
move_on(R2, R1)	move(B, F2)
	move(G2, G1)

Notice is that the black block is moved back and forth by agents A and B. As a result, the plans of both agents are slowed down. When the plan of one agent gets in the way of another agent, we say a competition or conflict has developed.

3. A Two Phase Approach for Multi-Agent Planning

A major task for multi-agent problem solving is to integrate the plans of all agents in a way that minimizes competition and maximizes cooperation. Difficulties arise when an agent has to devise its own plan while at the same time incorporating the plans of the other agents into

its own plan. The problem becomes even more complex when the inter-dependency between two agents, A and B, is so strong that A cannot proceed without knowing B's plan first and vice versa. Thus, we reach a deadlock situation. In order to alleviate these difficulties, we use a two phase algorithm similar to that used by Georgeff [GER 83] to generate plans for the multi-agent environment. We assume that the global goal of the system has been decomposed into subgoals and assigned to the appropriate agents. In the first phase, each agent generates its own individual plan to accomplish its own subgoal without considering subgoals of others. In the second phase, the agents compare and modify their plans with each other to minimize competition and maximize cooperation. We repeat the second phase until the global goal requirements are satisfied. The detailed algorithm is as follows:

PHASE I

Each agent does its own individual planning to accomplish its own goal as though no other agents exist (i.e., plans will be produced independently by each agent in parallel).

PHASE II

1. Request the individual plan of another agent, X, who has a dependent subgoal
2. Compare the two plans:
   a. Modify its own plan to eliminate instances of competition between the plans
   b. Further modify its own plan to incorporate all possible cooperation
3. Send the revised plan to X
4. Wait for X to send in its latest plan
5. If X makes changes in its plan, then repeat steps 2-5, otherwise, done with X
6. Repeat steps 1-5 for all the other agents who have dependent subgoals

This approach has the advantage that it is likely to achieve a high degree of parallel actions. The two phase algorithm starts by doing individual planning for every agent in parallel. Only when it is impossible to have parallel action due to resource contention or other conflicts are the parallel actions serialized. This parallelism is highly desirable in the multi-agent world.

4. Some Fundamental Problems in the Two Phase Approach

One problem that arises in this two phase approach, as in most systems with parallel activities, is the problem of inconsistency. In the first phase of the algorithm, every agent does its planning independently with no interference from other agents. However, in reality this is not the case. The interference of agents on each other will create an inconsistent view of the world when individual plans are merged in phase II. A condition or state assumed by an agent in accordance with the initial state of the world during phase I planning may no longer be true due to the actions of the other agents. For instance, a block "B" may be clear in the initial state of a given world. If an agent "X" does not do anything to block "B", it will assume that block "B" maintains the same position it did initially. Should agent "X" have to move block "B" later, it will plan to do so without having to clear block "B" first. However, due to the actions of the other agents, block B may not be clear any more by the time agent "X" wants to move it. In this case, extra steps must be added to the plan of agent "X" to accommodate the change. This problem is due to the parallel planning of multiple agents and we call it the *consistent world problem.*

Another problem associated with the multi-agent environment is the *resource availability problem.* In the single-agent world, we can safely assume that all resources specified in the system are available to the agent. However, in the multi-agent world, part of the resource may be currently used or operated on by one agent and thus not available to other agents at that time. This kind of resource contention where two agents both try to operate on a resource concurrently is easy to detect. Nevertheless, there is the less obvious case of resource contention where a resource not currently operated on by an agent is still needed by that agent and hence is not

available to other agents. For example, agent "X" is building a three block tower with block A at the top, block B in the middle and block C at the bottom. Let's say X is now in the final step of stacking A onto B. At this point in time, block B and C, which are not currently operated on by X, are still needed by X in order to build a three block tower and hence are not available to other agents. The system must somehow recognize these two types of resource contention and represent them efficiently. Again, since phase I planning is basically single agent planning, these resource availability problems must be addressed during phase II planning.

5. Implementation For the Two Phase Approach

The biggest potential problem in the two phase algorithm is with the process of "merging" individual plans to resolve interaction problems during the second phase. If not handled properly, the merging process might very well redo all the planning that has already taken place. If this is true, then there is no point in the agents going through the first phase of independent planning in parallel. Therefore, in order for the two phase algorithm to work efficiently, the phase I planner must provide enough information to the phase II planner for it to modify the initial plans. Otherwise, effort will be replicated by the phase II planner to figure out the objectives of the phase I plans. Information needed by the phase II planner includes temporal relationships, causal relationships and simultaneity. We shall go through the details in the following sections.

5.1. Knowledge and Data Representation

Some knowledge representation scheme must be used to represent the initial state of the world (the initial state), the goals which need to be accomplished (the goal state), and the plans generated to accomplish the goals. The initial state and the goal state are usually represented by a set of state predicates. This is the approach used by our algorithm.

There are two basic approaches for representing a plan. The state-based approach, represents a plan by a sequence of set of state predicates. This approach indicates the efforts of the actions carried out directly and indicates the actions to be carried out indirectly via the differences between consequent sets of state predicates. The second approach, the behavior or action approach, represents a plan by a sequence of actions. It indicates the actions to be carried out directly and the effects of the actions carried out indirectly by either updating the state-based description of the initial state of the world or by a distributed description of the change using add and delete lists like those in the procedural net [SAC 77]. In both approaches, the scheme will represent the actions, the effects of the actions and the temporal relationship of those actions. These are the minimum requirements of a representation scheme.

In our research, we shall adopt the behavior based approach. More specifically, the procedural net is used to represent the plan generated. The procedural net contains several levels of plan representation; each level more detailed than the previous one. A procedural net consists of a partially ordered sequence of nodes that represent goals at some level of abstraction. Each node in the procedural net is attached to its more detailed expansion in the next level; for example, the node representing the abstract goal *make coffee* may be expanded to handful of more detailed goals, such as *grind coffee, boil water, put the coffee in a filter, pour the water through it.* The statement of the problem goal is the top level node, representing a plan at a very high level. The top level node will be expanded repeatedly until a detailed plan is obtained. A procedural net is headed by the *planhead* node. Parallel plans will be represented by means of the *andsplit, andjoin, orsplit and orjoin* nodes.

The original procedural net scheme does not have a mechanism to represent the duration of the operation. In order to make the procedural net more general and more suitable for our purpose, a "time" field is added to every operator node to designate the time needed to accomplish that operation.

5.2. The Planning Algorithm

The backtracking approach, similar to that used in WARPLAN [WAR 74], is one way to plan. However, backtracking approaches are inadequate for our two phase algorithm because they overconstrain a plan. A planner is said to overconstrain a plan if it commits itself arbitrarily to orderings. For example, let M1, M2 and M3 represent three moves in a certain domain. (For example, in the block world domain, M1 can be "clear block A", M2 can be "clear block B", M3 can be "move A on B".) Let us use the notation "==>" to represent the temporal relationship between two actions. A backtracking planner may produce a plan like M1 ==> M2 ==> M3. (M1 followed by M2, followed by M3.) However, in some cases, the order in which M1 and M2 are executed may not matter, that is, M2 ==> M1 ==> M3 may work just as well. However, this information is not provided by the backtracking planner. If we use the backtracking approach in phase I of our algorithm, these overconstrained plans will present difficulties for the phase II planner when modification is necessary. For example, in the aforementioned case, suppose the plan M1 ==> M2 ==> M3 is given by Agent "X" as a feasible solution. However, another agent, "Y", may find that the alternative plan M2 ==> M1 ==> M3 will fit better with its own plan. However, since the plan is overconstrained, Agent "Y" will not know if the alternative plan is acceptable for Agent "X". Therefore, agent "Y" must go through the reasoning process to determine whether the alternative plan is acceptable for agent "X". Notice that even agent "X" does not know if the alternative plan is acceptable without further reasoning due to the termination of most backtracking algorithms after the first feasible solution is found.

Another planning approach is the least commitment approach. Planners using this approach are said to underconstrain a developing plan by not committing to any orderings except to avoid an interaction. For our research, we have implemented the least commitment approach. The planning algorithm is very similar to that used in NOAH [SAC 77]. Here, the basic idea is to repeatedly operate on the lowest level of the procedural net. Initially, a node is constructed for

the goal the planner is given as its task. A set of domain dependent procedures is used to expand this node. Once all the nodes in the current level have been expanded to produce a new level, the critics check for interactions before another level of expansion is tried. We shall elaborate on the ideas of node expansion and critics in the following sections.

5.2.1. Node Expansion

A rule-based system is used to develop the individual initial plan in phase I. A similar approach is used in STRIP [FIK 71] and WARPLAN [WAR 74]. For every effect the system wants to achieve, there are actions and preconditions associated with it. Using the "block world" as an example, the effect/action/precondition trio are as follows:

EFFECT	ACTION	PRECONDITION
on(U, floor)	move(U, V, floor)	clear(U) & on(U, V) & V /= floor
on(U, W)	move(U, V, W)	clear(U) & clear(W) & on(U, V) & U /= W
clear(U)	move(U, V, W)	clear(U) & clear(W) & on(U, V) & U /= W

5.2.2. Critics

Different critics are used in phase I and phase II of the two phase approach. For phase I, there are three commonly used critics which are very similar to those used in the NOAH system. The first one is the resolve precondition conflict critic. This critic examines conjunctive goals that are to be achieved in parallel. If an action taken to achieve one goal removes a precondition of an action in the other, the critic attempts to order the actions so that neither violates a precondition of the other. The second critic, the eliminate redundant operation critic, fixes the problem

where the same operation gets specified twice when it need only be done once. The third critic is the use existing operation critic. Formal objects, essentially placeholders, are used whenever there is not a clear choice of what value to give a variable. This critic will substitute a value when a clear choice becomes possible at a lower level of planning.

The Phase II planner employs a different set of critics. While the phase I critics deal only with individual plan, the phase II critics solve interplan interaction problems. Thus, phase II critics are more complicated and more involved than their phase I counterparts. We shall explain these critics in detail. We shall use the term subplans to represent the current individual plans of the agents.

(1) Resolve interplan precondition conflict critic

This critic addresses the consistent world problem mentioned in the previous section. It examines the operators or goals in different parallel subplans. If an action taken to achieve one goal in a subplan removes a precondition of an action in another subplan, the critic attempts to order the actions so that neither violates a precondition of the other. If the conflicts cannot be resolved by re-ordering or serializing the actions, then an extra action sequence may need to be inserted into the plan. In the worst case, substitution of an alternative plan may be needed in order to resolve this conflict.

(2) Resolve competitive resource sharing critic

This critic checks for operators that are using a common resource. An example of such a resource would be a block in the block world domain, or a runway in the air traffic control domain. If two plans are using a common resource in a competitive or conflicting way, this critic will try to resequence the actions or even change the plan in a way similar to that of the above critic to eliminate the competition.

(3) Resolve cooperative resource sharing critic

Sometimes two parallel subplans will perform the same operation on the same object. This critic will reorder and alter the subplans so that only one agent performs the operation while the other agent benefits from it. This critic will recognize operations with formal objects or placeholders as potential resource sharing opportunities.

(4) Resolve resource deadlock critic

Deadlock occurs when two parallel subplans require resources which are currently held by each other. For example, two vehicles (agents) moving toward each other in a single lane road. When a deadlock situation is detected, this critic will force one of the agents to give up its resource temporarily. This will usually mean inserting extra steps into the subplan of that agent.

(5) Resolve irreplaceable resource critic

In most cases, we assume that the shared resources are imperishable. However, sometimes a resource is consumable and irreplaceable. If two subplans require the same irreplaceable resource, this critic will determine which subplan will get to use it and what alternative arrangement will be made for the other subplan. This is basically a domain dependent critic.

(6) Mutually beneficial planning critic

When a single agent interchanges the content of two registers, an intermediate step is required. The equivalent problem in the block world of swapping two blocks requires the placement of one block in a temporary location while the second block is moved into place. However, if the job is handled by two agents that cooperate well, this intermediate step is not necessary. The job can actually be accomplished by just one move from each agent. The job of this critic is to recognize this potential for cooperation and simplify the plans accordingly.

6. A Block World Example

We shall illustrate the generation of cooperative plans by the two phase algorithm using the example in Fig 2. In phase I, the agents work independently in parallel to come up with two individual plans as shown in Fig 4a and Fig 4b. Then we enter the second phase of the algorithm. At this point, agent A sends its plan to agent B. Agent B then checks the subplans to see if there is any interaction between them by applying the various phase II critics. In this example, the resolve cooperative resource sharing critic discovers that both agents try to move the block g1. Further investigation reveals that the action "puton(g1,f4)" is parallel to another action in A's plan. Therefore, either action can be executed first. Agent B concludes that if it imposes the constraint that "puton(g1,f4)" of agent A be executed first, then it can eliminate its own step to move g1. Agent A's plan becomes that shown in Fig 5a. Since "puton(g1, Somewhere)" is the precondition of the action "puton(r1, f7)" in agent B's plan, agent B has to make sure that "puton(r1, f7)" waits until the action "puton(g1,f4)" is executed by agent A. This is accomplished by rearranging agent B's plan as shown in Fig 6.

Final plan of agent A:

1. start
2. put g1 on f4
3. put y1 on f5
4. end

Final plan of agent B:

1. start
2. put b1 on f6
3. put r1 on f7
4. end

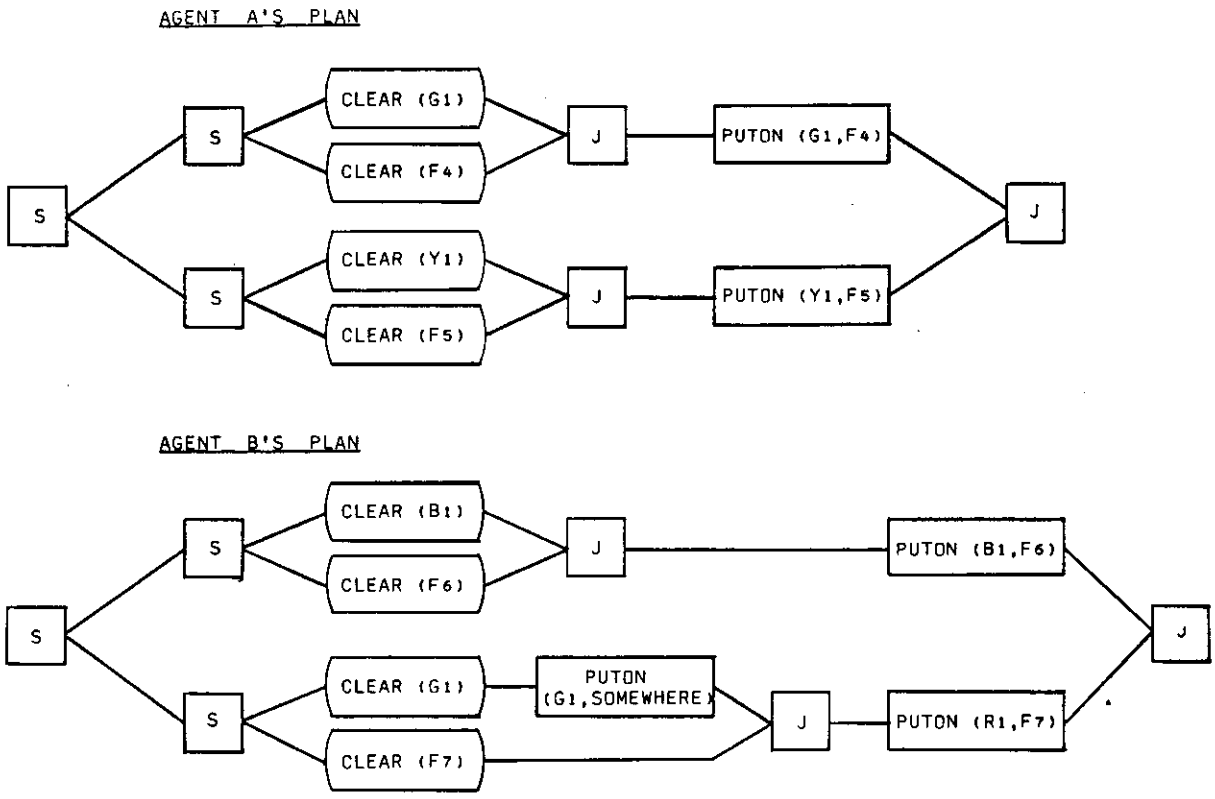


Fig 4 Subplans of A and B after phase I

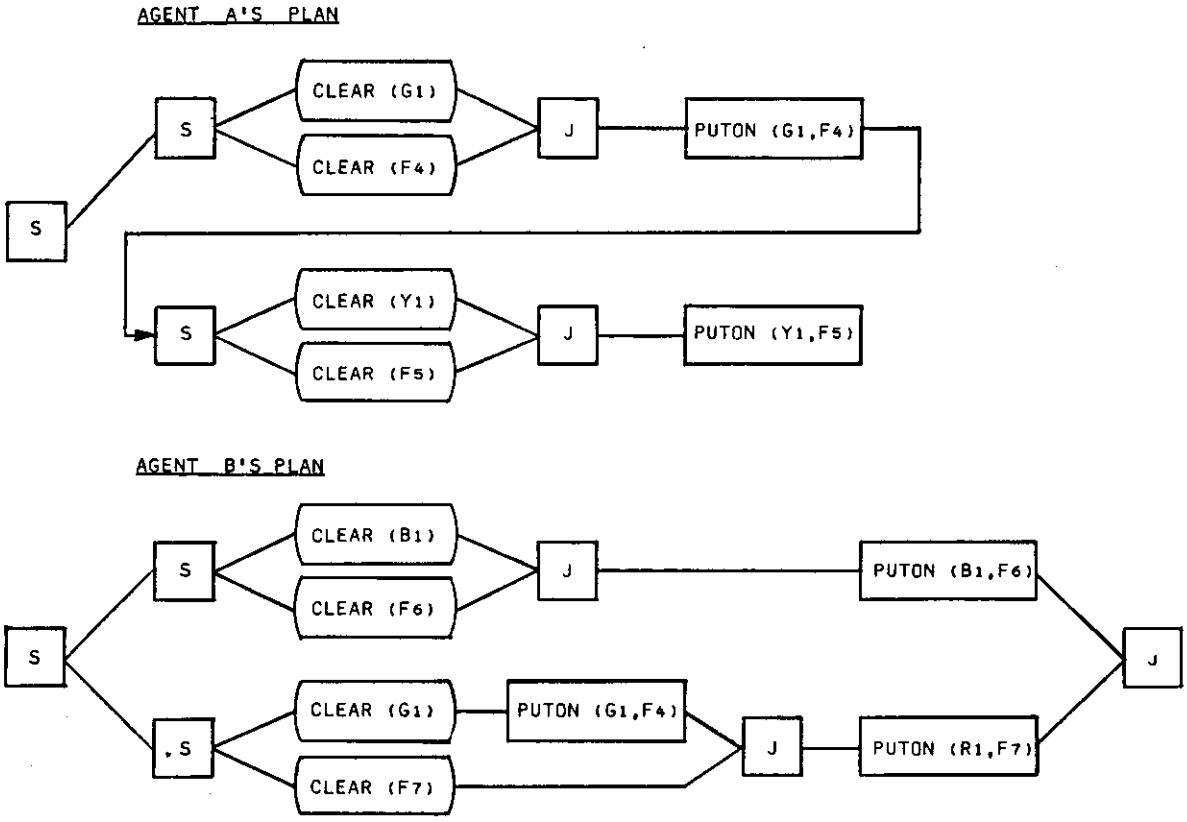


Fig 5 Subplans of A and B after the first criticism in phase II

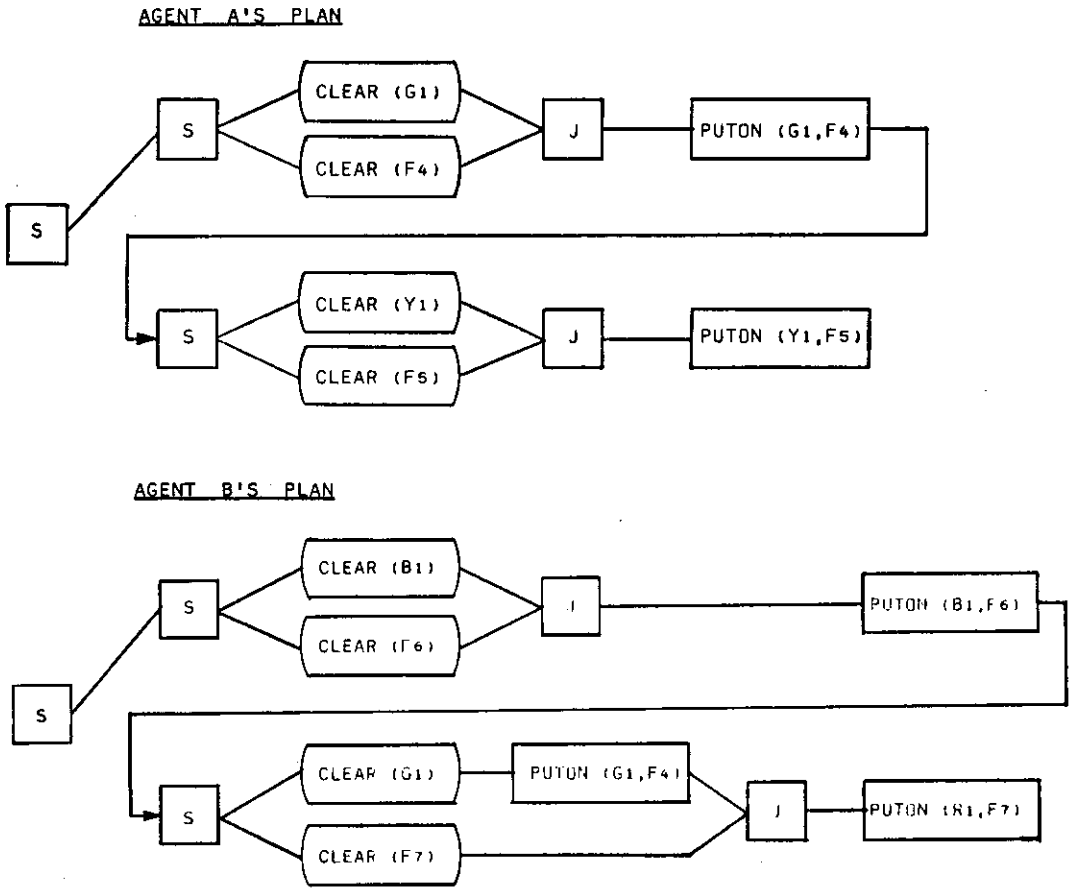


Fig 6 Final subplans of A and B after all phase II criticism

7. Applications of Multi-Agent Planning

Multi-agent planning is a general tool that has innumerable application possibilities in distributed systems. One possible application is in the CAD/CAM area. Due to the large amount of computing involved in a PC board layout software package, a main frame computer must be used to run the software; otherwise the response time will be very slow. With the advent of VLSI technology, one feasible way to meet the high computing power requirement is to use multiple microprocessors to do the job. In this case, each processor will deal with a particular part of the PC board layout. Obviously, the interconnection between one part of the PC board and another will create many interaction problems which must be solved by the system. Multi-agent planning provides the means to accomplish this.

Another application area is the Strategic Defense Initiative (SDI) Battle Management System. The SDI Battle Management System consists of a set of defense units (platforms), each spatially separated. Interaction problems may arise due to the overlapping of coverage areas of each of these defense units. Based on the input scenario, resource constraints (e.g., the current launch loading of a defense unit), amount of resource available (e.g., interceptor missile), and hit probability of an interception, multi-agent planning can provide the overall optimal strategic plan for battle management.

REFERENCE

[BRO 83] Brooks,R.S., "Experiments in Distributed Problem Solving with Iterative Refinement," Ph.D. Thesis, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, February 1983.

[CAM 83] Cammarata,S., McArthur,D., and Steeb,R., "Strategies of cooperation in distributed problem solving," N-2031-ARPA, December 1983.

[COR 83a] Corkill,D.D., "A framework for organization self-design in distributed problem solving networks," Ph.D. Thesis, Department of Computer and Information Science, university of Massachusetts, Amherst, Massachusetts, February 1983.

[COR 83b] Corkill,D.D. and Lessor,V.R., "The use of meta-level control for coordination in a distributed problem solving network," Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Washington,DC.

[DAV 81] Davis,R. and Smith,R.G., "Negotiation as a metaphor for distributed problem solving," AI Memo 624, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts.

[ERM 80] Erman,L.D., Hayes-Roth,F., Lessor,V.R. and Reddy,D.R., "The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty," Comput. Surveys 12 (1980) 213-253.

[FIK 71] Fikes,R.E. and Nilsson,N.J., "STRIPS: A new approach to the application of theorem proving to problem solving," Artificial Intelligence 2:189-208.

[GER 83] Georgeff,M., "Communication and interaction in multi-agent planning," AAAI-83, Washington, D.C. (1983) 125-129.

[LAN 85] Lansky,A.L., "Behavioral specification and planning for multiagent domains," Technical Note 360, SRI International, Artificial intelligence Center, Menlo Park, California, 1985.

[LES 80] Lessor,V.R. and Erman,L.D., "Distributed interpretation: A model and experiment," IEEE Transactions on Computers, C-29(12):1144-1163.

[LES 81] Lessor,V.R. and Corkill,D.D., "Functionally accurate cooperative distributed systems," IEEE Trans. Comput. 29 (1980) 1144-1163.

[LES 83] Lessor,V.R. and Corkill,D.D., "The distributed vehicle monitoring testbed," AI Magazine, Fall 1983.

[NIL 80b] Nilsson,N.J., "Two heads are better than one," SIGART Newsletter(73):43.

[SAC 77] Sacerdoti,E.D., "A Structure for Plans and Behavior," 1977, Elsevier North-Holland, Inc.

[SMI 78] Smith,R.G., "A framework for distributed problem solving," VMI Research Press, 1981; also: Stanford Memo STAN-CS-78-7000, Stanford University Stanford, CA, 1978.

[STE 84] Steeb,S., McArthur.D., Cammarata,S., Narain,S., and Giarla,W., "Distributed problem solving for air fleet control: Framework and implementations," N-2139-ARPA, April 1984.

[WAR 74] Warren,D., "WARPLAN: A system for generating plans," Memo 76, Computational Logic Dept., School of Artificial Intelligence, University of Edinburgh, 197

DISTRIBUTION LIST

1. Director
US Army Strategic Defense Command
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 Command
BMDSC-AOLIB
P.O. Box 1500
Huntsville, AL 35807-3801

4. Defense Technical Information Center
Cameron Station
Alexandria, VA 22134

5. General Research Corporation
ATTN: Dave Palmer
P.O. Box 6770
Santa Barbara, CA 91305

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

7. University of California, Berkeley
Dept. of Electrical Engineering & Computer Sciences
ATTN: C. V. Ramamoorthy
Berkeley, CA 94720

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

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

10. General Research Corporation
ATTN: Henry Minshew
307 Wynn Drive
Huntsville, AL 35805

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

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

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

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

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