Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

DISTRIBUTED DATA BASE MANAGEMENT FOR REAL-TIME BMD APPLICATIONS

Wesley W. Chu
M. T. Lan
K. K. Leung
R. C. Lee
M. A. Merzbacher

September 1987
CSD-870059

DISTRIBUTED DATA BASE MANAGEMENT FOR REAL-TIME BMD APPLICATIONS

FINAL REPORT FOR THE PERIOD

FROM: May 28, 1986

TO: Sept. 30, 1987

Contract No. DASG60-85-C-0059

Prepared For:

US Army Strategic Defense Command

Huntsville, Alabama 35807

Sept. 30, 1987

University of California, Los Angeles

Wesley W. Chu, Principal Investigator

Researchers: M.T. Lan, K.K. Leung, R.C. Lee,

M.A. Merzbacher, C.M. Sit

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 assignment and scheduling for distributed systems, testbed validation of design techniques, performance of concurrency control algorithm for distributed database systems, and query processing with domain semantic.

1.0 Module Assignment for Real-Time Distributed Processing Systems

Module assignment is a key issue that effects system performance in distributed systems. Response time is intimately related with module assignment. Therefore, we shall use response time as a performance measure in our research. We have investigated two related areas. The first area considers the module precedence effect on module assignment, and the second considers the replicated module assignment to provide load balance and improves response time.

1.1 Module Assignment and Precedence Relations for Distributed Real-Time Distributed Systems

It is well known that module assignment should consider module precedence relationships. However, most of the published task allocation work has not considered the precedence effect. This motivates us to study and understand the effects of precedence relationship (PR) among program modules on response time. A new loading function that includes the Intermodule Communication (IMC) and Accumulative Execution Time (AET) of each module is also proposed. Our study reveals that minimizing the most heavily loaded (bottleneck) processor is a good principle for module assignment. Further, the PR module effect can also be integrated into

the above assignment principle. When module PR is considered in the task assignment, it yields better performance than without considering the PR effect. Detailed results are summarized in Chapter 2.

1.2 Module Replication and Assignment for Real-Time Distributed Systems

An analytical 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 reveals that the task response times for a given module assignment (with replications) generated by the algorithm compare closely with that of the simulation and exhaustive search. A series of experiments is also performed to characterize the behavior of the algorithm.

2.0 A Batch Service Scheduling Algorithm with Time-Out for Real-Time Distributed Processing Systems

A new scheduling algorithm for reducing overhead and thus response time is proposed for distributed processing systems. The algorithm groups several module invocations into a

batch and processes them together to reduce certain scheduling overhead. A time-out clock is used to avoid excess delay in forming a batch. The clock is set when the first invocation arrives at the batch queue. The batch is formed when either the number of invocations reaches the prespecified maximum batch size or the time-out period ends. We denote this scheduling algorithm Batch Service with Time- out (BST). An analytical model is developed to estimate response time for this scheduling algorithm. The response time of a module using the BST algorithm depends on the invocation rate, scheduling overhead, execution time, maximum batch size, and time-out period. The assumptions used in the model are validated by simulations.

Comparing performance of a system using BST with that of using first-come- first-served (FCFS) scheduling algorithm, we note that the amount of improvement depends on the ratio of the fixed scheduling overhead to the incremental scheduling overhead. At heavy invocation rates, more batches will be formed when using the BST algorithm, therefore fixed scheduling overhead is reduced and more response time improvement can be achieved (See Chapter 3). As a result of reduction in overhead, the system using BST provides more capacity than that of using FCFS.

3.0 Testbed-Based Validation of Design Techniques

During the past three years, we have been jointly working with Unisys SDC at Huntsville to study and develop design methodology for tightly coupled distributed systems. Experimentation on the testbed provides us with insights on algorithm behavior. We have developed a fault tolerant locking (FTL) algorithm for the tightly coupled multiple processing system [1], designed the experiments, and studied its feasibility and performance. Experimental results reveal that the FTL is capable of detecting a processor failure during update and recovering data inconsistency among replicated copies. The overhead for performing the fault-tolerant locking

protocol depends on the lock frequency and its application. The parameters that may affect system performance are: time-out period, lock granularity (record or a group of records), and lock protocol (e.g., exclusive lock for write and shared for read, or reserve, upgrade, or exclusive lock).

We have also used the testbed for studying the performance of lock granularity (e.g. record, file) and the performance of reserve-upgrade locking protocol. Because of the read/write pattern of the radar tracking application, the results reveal that simple record locking provides better response time than file locking and reserve-upgrade locking. The detailed results are summarized in Chapter 4.

4.0 Performance of Concurrency Control Algorithms for Real-Time Distributed Database Systems

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.

To study the different concurrency control techniques, we introduced new performance measures such as accuracy and weak consistency for characterizing the performance for data consistency. We have experimentally studied three types of protocols via simulation: Primary Site Locking (PSL) [2], Exclusive Writer Protocol (EWP) [3], and Time Stamp with modified Rollback (TMR) [4] and compared their performance in terms of response time, communication overhead, query rate, update/query ratio, consistency, and accuracy. Detailed discussions are given in Chapter 5. Protocols that assure weak consistency yield better performance (response time) than those assuring strong consistency. The data consistency requirement for different

types of data is application dependent. Further, it also depends on how the data is used for decision making. It is possible that when data state is x, weak consistency is sufficient. While data state becomes y, strong consistency is required.

5.0 A Methodology of Knowledge Acquisition for Semantic Query Processing

Query processing is a key consideration in database management systems. Conventional approach uses a domain-independent approach for query processing design. Queries are transformed algebraically to determine the optimal access plan for retrieving the answer. *Semantic query optimization* uses a set of integrity constraints and reasoning to transform a given query into a different but more efficient query yet yields the same answer. It has been shown that this technique has great promise for improving system performance. However, knowledge acquisition problem needed to be solved before this technique can be of practical use.

Data modelling provides a useful tool for database design and usage. Conventional data models such as *hierarchical, network,* and *relational* models provide a record-based data structure for modelling database application. However, because of the lack of expressiveness of the conventional data models in modelling various database applications, semantic data models have been developed to provide a rich set of semantic constructs to describe various situations of the application. Database designer then uses his knowledge about the application to define the database schema. Most of the semantic data models focus only on providing structural specification and ignore the importance of knowledge for designing the database. However, this knowledge is very useful to semantic query processing and should be saved and used in query processing.

In this research, a semantic data model is developed that provides a knowledge specification capability associated with the semantic constructs for schema specification. This data model provides not only the necessary semantic expressive capability to model various

database applications, but also specifies domain knowledge which can be used to improve query processing performance. A knowledge acquisition tool is developed for systematically collecting useful domain knowledge for semantic query processing. A semantic database management system (SDBMS) is proposed that integrates semantic data modelling with semantic query processing. SDBMS provides facilities to systematically acquire semantic knowledge and use them to improve query processing. Detailed discussions are given in Chapter 6.

References

[1] Chu, Wesley W. 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] Stonebraker, Michael, "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.

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

[4] Jajodia, Sushil and Catherine A. Meadows, "Mutual Consistency in Decentralized Distributed Systems," *Proceedings Third International Conference on Data* Engineering, vol. 3, pp. 396-404, IEEE, 1987.

CHAPTER II

2.1 MODULE ASSIGNMENT AND PRECEDENCE RELATIONS FOR DISTRIBUTED REAL-TIME SYSTEMS

Task Allocation and Precedence Relations for Distributed Real-Time Systems

WESLEY W. CHU, FELLOW, IEEE, AND LANCE M-T. LAN, MEMBER, IEEE

Abstract—**In a distributed processing system with the application software partitioned into a set of program modules, allocation of those modules to the processors is an important problem. This paper presents a method for optimal module allocation that satisfies certain performance constraints. An objective function that includes the intermodule communication (IMC) and accumulative execution time (AET) of each module is proposed. It minimizes the bottleneck-processor utilization—a good principle for task allocation. Next, the effects of precedence relationship (PR) among program modules on response time are studied. Both simulation and analytical results reveal that the program-size ratio between two consecutive modules plays an important role in task response time. Finally, an algorithm based on PR, AET, and IMC and on the proposed objective function is presented. This algorithm generates better module assignments than those that do not consider the PR effects.**

Index Terms—**Distributed processing, intermodule communication (IMC), interprocessor communication (IPC), minimum bottleneck, module assignment, parallel processing, precedence relationship (PR), real-time systems, response time, task allocation algorithms.**

I. Introduction

ALTHOUGH computer speed has been increased by several orders of magnitude in recent decades, the demand for computing capacity increases at an even faster pace. The required processing power for many real-time applications cannot be achieved with a single processor. One approach to this problem is to use distributed data processing (DDP) that concurrently processes an application program on multiple processors. If properly designed and planned, DDP provides a more economical and reliable approach than that of centralized processing systems.

Task partitioning and task allocation are two major steps in the design of DDP systems. If these steps are not done properly, an increase in the number of processors in a system may actually result in a decrease of the total throughput [5]. Assuming the software for an application (a *task*) has been partitioned into a set of program *modules* (or subroutines), in this paper we study how to optimally allocate these modules to the set of processors in the DDP system.

Manuscript received September 10, 1985; revised August 11, 1986. This work was supported by the Ballistic Missile Defense Advanced Technology Center under Contracts DASG60-79-C-0087 and DASG60-83-C-0019.

W. W. Chu is with the Department of Computer Science, University of California, Los Angeles, CA 90024.

L. M-T. Lan was with the Department of Computer Science, University of California, Los Angeles, CA 90024. He is now with the Cellular Telecommunications Laboratory, AT&T Bell Laboratories, Whippany, NJ 07981.

IEEE Log Number 8713985.

First, we shall present two important parameters for task allocation: intermodule communication (IMC) and accumulative execution time (AET) of each module. The load of a processor consists of AET and IMC. We propose an objective function for task allocation that is based on minimizing the load on the most heavily loaded processor ("bottleneck"). The precedence relation (PR) among program modules, that specifies the execution sequence of the modules, is another parameter that affects module assignment. It is studied analytically and experimentally. A series of experiments are presented which reveal that the *program-size ratio* between two consecutive modules plays an important role in determining whether two modules should be colocated. An analytical model is developed that enables us to decide whether to assign consecutive modules to the same processor. Finally, a heuristic algorithm is developed that considers PR, IMC, and AET to search for the minimum-bottleneck assignment. Examples are given to illustrate the performance of the algorithm and also the improvement that may be obtained when considering PR in task allocation.

II. A New Objective Function for Task Allocation

In this section we shall first describe the two important parameters, AET and IMC, for task allocation. An objective function based on these parameters that minimizes system bottleneck is proposed. Then, we present the behavior of the proposed objective function.

A. IMC and AET

The AET for module M_j during time interval (t_h, t_{h+1}) is the total execution time incurred for this module during that time interval, i.e.,

$$T_j(t_h, t_{h+1}) = N_j(t_h, t_{h+1})\, y_j(t_h, t_{h+1})$$

where $N_j(t_h, t_{h+1})$ = number of times module M_j executes during (t_h, t_{h+1}), and $y_j(t_h, t_{h+1})$ = average execution time of M_j during (t_h, t_{h+1}). Both the y_j and the AET T_j can be expressed in units of machine language instructions (MLI). Although the execution time of a machine language instruction varies from one instruction to another, based on a given instruction mix we can use the *mean* instruction execution time. Our study reveals that both the number of module executions and the AET are almost independent of module assignments when the load offered to the system is fixed. For example, the AET's produced by five different assignments for a module in a space-defense application, the Distributed

0018-9340/87/0600-0667$01.00

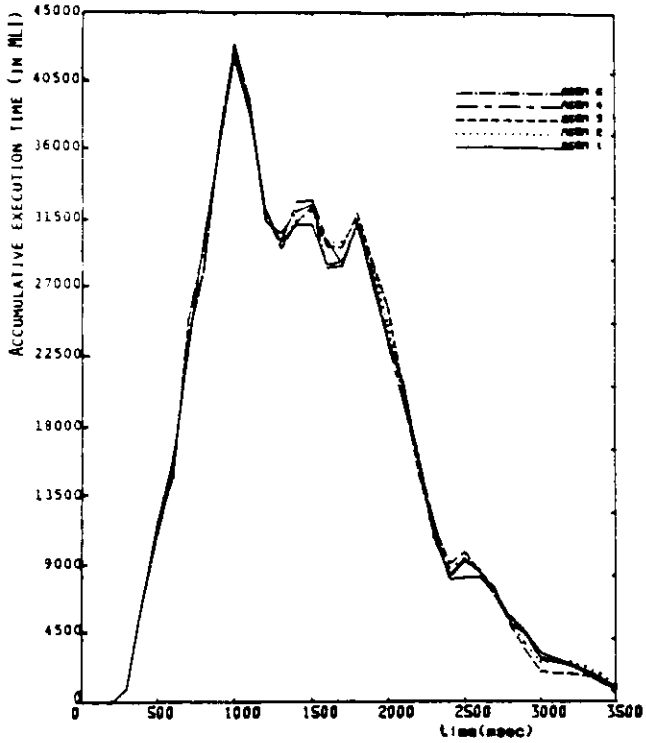


Fig. 1. Accumulative execution time of module M_8, $T_8(t, t + 100$ ms).

Processing Architecture Design (DPAD) system, are almost identical (Fig. 1).

IMC is the communication between program modules and file modules. When a module on a processor writes to or reads from a shared file on *another* processor, such IMC incurs IPC (interprocessor communication) and requires processing overhead. *Control IMC* is another type of IMC. As discussed in [8], it can be treated in the same way as the file-access IMC when we consider the *control files*, as opposed to the application files. The importance of IPC minimization has been recognized by many researchers [5], [14]. IPC can be reduced by assigning a pair of heavily communicating modules to the same processor. Like AET, the IMC can also be assumed to be independent of module assignments [16]. A method for estimating both IMC and AET has been reported in [8].

IPC occurs only when two communicating modules are assigned to different processors. If two modules reside on different processors and communicate through a *replicated* shared file, then the file is assumed to be replicated on each processor. When a module updates the file, it updates the copy on its local processor and sends the updates to the remote processor. This results in IPC, which requires processing load on both the sending and receiving processors. Even if the actual transfer of the update words is done in the background by some I/O processors, the sending processor still needs to spend time on message formatting and address initialization for the I/O processor. The receiving processor, on the other end, will spend time on extracting the message contents and notifying the destination module. Such IPC overhead is eliminated if the two modules are assigned to the same processor since both modules would share the same local file copy. Module assignments also effect IPC for other file structures such as partitioned files or single-copy files.

B. The Objective Function

Assuming each module is assigned to one and only one processor, then there are S^J different ways to assign J modules to S processors. This can be represented by an assignment tree. This tree has J levels, each representing a module. At each nonleaf node there are S downward branches, each representing the choice of a processor to host the particular module. Therefore, the tree has S^J leaves, each leaf corresponding to a possible assignment.

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

Existing approaches to task allocation can be divided into three categories: graph-theoretic [15], [20], [2], [3], integer 0-1 programming approach [4], [6], [18], and the heuristic approach [13], [10]. Many of these methods try to minimize a task's total cost which is defined as the sum, over all processors, of both the processing cost (i.e., AET) and the IPC cost of that task. This might be acceptable for a distributed system *shared* by multiple simultaneous nonreal-time applications (tasks), each having program modules running on some or all of the multiple processors. Such applications attempt to maximize the total throughput. For a distributed system with identical processors, this formulation is equivalent to the minimization of IPC since the total AET is fixed.

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

The processor with the heaviest loading in a distributed system is the one that causes the *bottleneck*. For instance, for a system with three processors, an assignment resulting in 58, 60, and 61 percent of processor utilizations might have a better response time than another assignment yielding 20, 40, and 90 percent utilizations, although the total processor utilization of the first assignment is higher than the second. This is mainly due to the fact that the second assignment has a *bottleneck processor* more heavily loaded than the first assignment, and queueing delay is a nonlinear function that rises rapidly with the level of bottleneck (processor load).

The *processor load* consists of the loads due to program module execution and IPC. Therefore, both AET and IPC play important roles in module assignment and influence task response time. AET is usually represented in machine lan-

guage instruction (MLI). The number of transferred IPC words can be converted into the MLI's spent by both the processor that sends the IPC and the processor that receives it.

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

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

$$= \text{AET}(r; X) + \text{IPC}(r; X) \tag{1}$$

where $X = [x_{jr}]$ is the assignment matrix in which $x_{jr} = 1$ or 0 indicates whether module M_j is assigned to processor r. The first term in the equation is the AET for all modules assigned to processor r. The second term is IPC overhead due to both the IPC originated from processor r to other processors, and incoming IPC destined to processor r from other processors. For a system whose file-update messages dominate the IPC traffic, we can ignore other types of IPC such as module-enablement messages and system-control messages. The total overhead due to outgoing IPC at processor r is

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

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

The AET, T_j, for a module M_j is represented as a single value in (1). Also, the IMC between a module and a file, V_{jk}, in (2) is represented as a single value. However, the measured T_j and V_{jk} vary from one time interval to another (e.g., see Fig. 1). Since we are concerned with system performance during the peak-load period, we shall use the *average* T_j and V_{jk} values during the peak-load period for the terms T_j and V_{jk} in (1) and (2) to compute our objective function.

Similar to (2), the total overhead at processor r for incoming IPC from all remote sites is

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

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

$$\text{Bottleneck}(X) = \max_{1 \le r \le S} \{L(r; X)\}. \tag{4}$$

We want to find the assignment that yields the *minimum bottleneck* [7] among all possible assignments in the assignment tree, i.e.,

$$\min_{X} \{\text{Bottleneck}(X)\}. \tag{5}$$

Substituting (1) and (4) into (5) yields

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

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

A good assignment can be obtained by reducing IPC while balancing processor loads among the set of processors. A minimum-bottleneck assignment generally has low IPC and fairly balanced processor loads because of the following.

1) If the given assignment resulted in a large volume of IPC, the sum of processor loads over all processors would be high, which would yield high bottleneck.

2) If the loads were not fairly balanced for an assignment, the bottleneck (highest load of all processors) would be high which would not yield a minimum-bottleneck assignment.

Our minimum-bottleneck approach, (6), is different from the commonly used measure of minimizing the *sum* of processor loads (e.g., [20]),

$$\min_{X} \left\{ \sum_{r=1}^{S} [\text{AET}(r; X) + \text{IPC}(r; X)] \right\}. \tag{7}$$

An assignment obtained from (7) can be quite unbalanced. In a homogeneous system all modules will be assigned to a single processor as discussed before. Our minimax principle [7] is also used in [21] which considers only the *single execution* of a task, instead of using the processor load. Since each external stimulus causes a task execution in our formulation, the processing load is based on *multiple* executions of a task.

C. Behavior of the Proposed Objective Function

To illustrate the characteristics and performance of the proposed objective function, we apply the objective function to the Distributed Processing Architecture Design (DPAD) system. The DPAD system was developed to manage the data processing and radar resources for a space-defense application [11], [12], [18], [19]. The control-and-data-flow graph (similar to Fig. 12) consists of 23 modules which are to be assigned to three processors.

The average AET (T_j) and IMC (V_{jk}) during the peak-load period (from 1.0 to 2.0 s of mission time) for all modules of the DPAD system are calculated. For example, $T_8 = 32\,055$ MLI is the average of ten measured AET values for M_8 within the period at each increment of 100 ms.

A program was developed to compute the proposed objective function for *every* assignment (corresponding to a leaf of the assignment tree), performing an exhaustive search for the minimum-bottleneck assignment. When an assignment yields a bottleneck value lower than the smallest bottleneck obtained so far, that assignment is recorded. The last ten recorded assignments, denoted as assignment 1–10, are shown in Table I. The 23 digits under the "assignment" column represent the

TABLE I
TOP TEN ASSIGNMENTS FROM EXHAUSTIVE SEARCH

	ASSIGNMENT	LOAD-1	LOAD-2	LOAD-3	BOTTLENECK	TOTAL LOAD
10th	11111 11121 00330 13323 123	75612	75546	70420	75612	221578
9th	11111 11121 00330 13322 123	75421	75546	70709	75546	221576
8th	11112 11121 00330 13222 221	75413	75353	71643	75413	220408
7th	11112 12131 00220 13211 132	75174	74275	73829	75174	223282
6th	11112 12131 00220 13231 232	75413	74564	73829	75013	223806
5th	11112 12333 00220 13211 112	74414	74275	74023	74414	222712
4th	11112 12331 00220 13211 312	74240	74275	74312	74312	223836
3rd	12123 23212 00330 21322 113	74308	73971	74275	74308	222456
2nd	12123 23212 00330 21322 213	74414	74038	74275	74275	222732
MINIM. BOTTLE-NECK	12213 13121 00330 11323 231	74004	73805	74275	74275	222084

NOTE: 1. LOAD-I IS EACH PROCESSOR'S LOAD PER 100 MSEC (IN UNIT OF MLT).
2. AN ASSIGNMENT WITH THE MINIMUM TOTAL LOAD IS NOT THE ASSIGNMENT WITH THE MINIMUM BOTTLENECK.

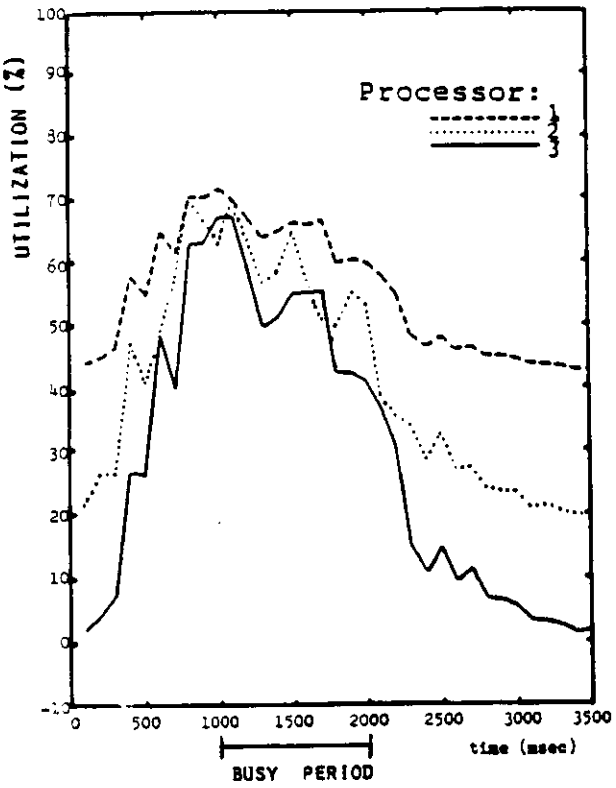


Fig. 2. Processor utilization for the best module assignment selected by exhaustive search.

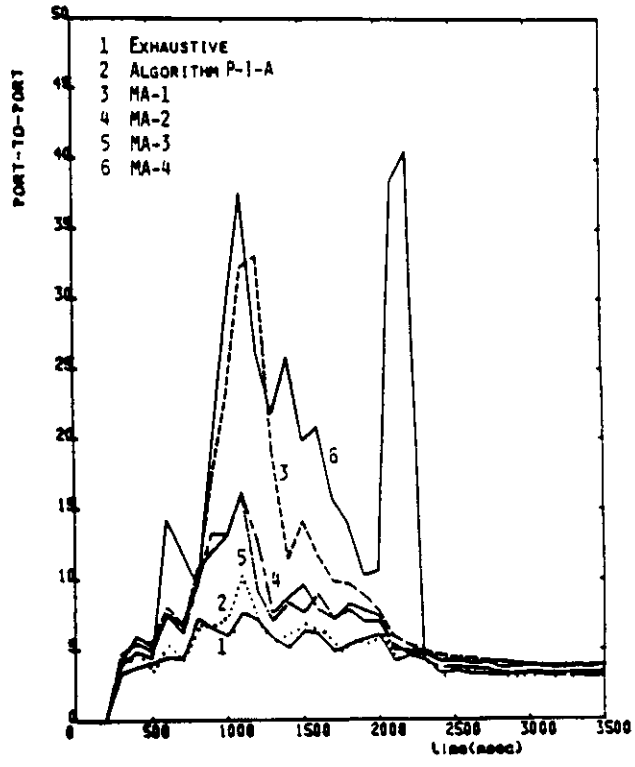


Fig. 3. Precision-Tracking Thread response times—compare the best assignment from exhaustive search, Algorithm P-1-A, and the four assignments from Ma *et al.* [18].

assignment of the 23 modules to the three processors (three of the 23 modules were not implemented in the DPAD system and are indicated here by a zero), columns 2, 3, and 4 provide the loading on the three processors, column 5 shows the bottleneck processor load (the largest one of the three processor loads), and the last column is the sum of the three processor loads. These ten assignments were simulated with the DPAD simulator, and their performance compared. Fig. 2 shows the CPU utilization for the minimum-bottleneck assignment 1. Note that the loads for the three processors are quite balanced during the peak-load period. The processor loads for assignments 2–10 are also fairly balanced. This verifies our conjecture that the minimum-bottleneck objective function provides *balanced* loads among processors. Fig. 3 shows the Precision-Tracking *port-to-port time* (response time for a task thread) for assignment 1 (curve 1). The assignments MA-1–MA-4, reported in [18] for the DPAD system, minimize the sum of AET and IPC and, thus, do not generate balanced-load assignments as discussed in Section II-B. As a result, their response times (curves 3–6) are higher than that generated by our objective function. (Curve 2 will be discussed later.)

III. Precedence Relationship and Module Assignment

The precedence relationship (PR) among program modules is another important factor that needs to be considered in task allocation. In this section we shall present several experiments to illustrate the effect of PR on response time. These experiments provide us with enough insight to formulate an

analytical model to quantitatively study the PR effect on task allocation. The quantitative PR effect will be used in module grouping in our module-assignment algorithm.

A. PR Experiments

In experiment 1, we compare three assignments of a task, consisting of nine modules, to three processors. The control-flow graph [Fig. 4(a)] shows the strong PR relationship among the modules. Assume that the task arrival is a Poisson process with rate λ. When a module completes its execution, it enables its succeeding module according to the control-flow graph. The enabled module is placed at the end of the *ready queue* of its residence processor in a first-come-first-served manner. Let the execution times for all modules be identical and equal to *one* time unit. To clearly observe the PR effect on response time we further assume there is no IMC between the modules and thus there is no IPC overhead among the processors. Three assignments [Fig. 4(b), (c) and (d)] were simulated using the PAWS simulator [1]. The results are presented in Fig. 5. Note that assignment 2 (pipelined) yields the best task response time. The vertical bars in the figure represent 90 percent confidence intervals for each simulation point. The response time varies substantially among these assignments in spite of the fact that all the three assignments have equal and balanced loads and there is no IPC overhead. This discrepancy is solely due to the PR effect among modules.

In experiment 2, the execution time of each module is *exponentially distributed* (instead of being a constant), with an *average* of one time unit. All other parameters remain unchanged from experiment 1. Experimental results reveal that the task response times for the three assignments are comparable (Fig. 5). Due to the memoryless property of the exponential distribution, the job queue at each processor can be approximated by an M/M/1 queue. Since the service-time distributions of all modules are identical and all modules are invoked for execution at identical arrival rate, the three load-balanced processors can be represented as three identical queueing systems. Thus, the wait-time is the same for all modules and all three assignments yield the same response times.

Experiment 1 reveals that precedence relationship *does* have an impact on task response time. Experiment 2 shows that the PR effect on response time is also influenced by module-execution-time distributions. In experiment 3, we shall study the effect of module size on response time. We assume that every module's execution time is exponentially distributed, but with a different mean value, as shown in Fig. 6. The simulation results for the three assignments reveal that assigning two consecutive modules to the same processor yields good response times *if the execution time of the second module is much larger than that of the first module* (Fig. 7). We shall denote this as PR Principle 1. For example, because y_2 is considerably greater than y_1, M_1 and M_2 should be assigned to the same processor. This principle was used in assignment 1 (Fig. 6) which yielded the best performance. Likewise, in assignment 1 module pairs (M_3, M_4) and (M_5, M_6) are allocated to processors 2 and 3, respectively.

If the second module is much smaller than the first one,

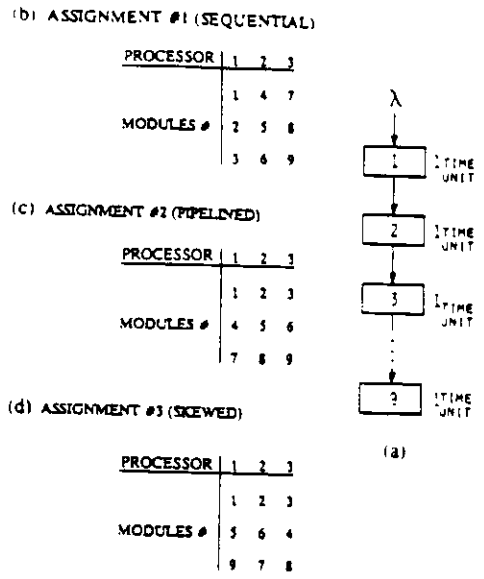


Fig. 4. Precedence-relationship experiment 1. (a) Task control-flow graph. (b) Sequential assignment. (c) Pipelined assignment. (d) Skewed assignment.

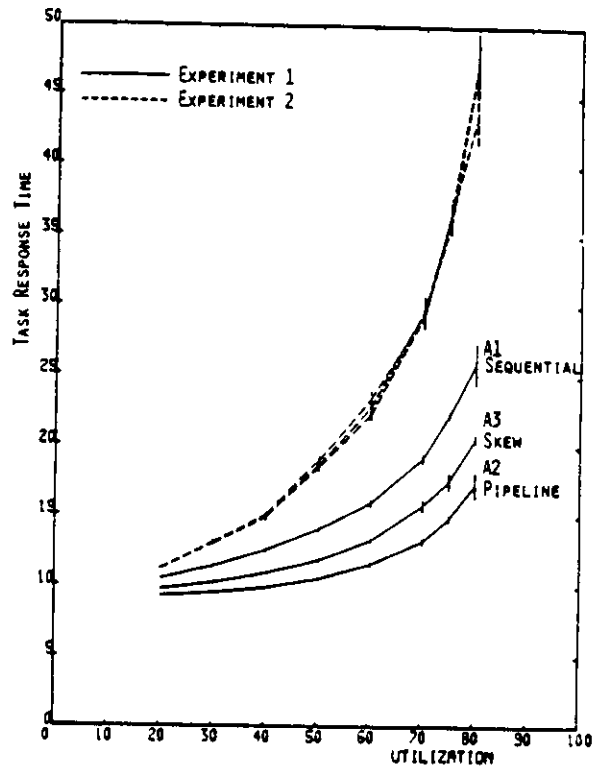


Fig. 5. Compare the response time of three module assignments. Experiment 1 uses deterministic execution time. Experiment 2 uses exponential execution time.

separating the two consecutive modules and assigning them to two different processors yields better response time. We shall denote this as PR Principle 2. Since y_3 is much less than y_2 in this example, M_2 and M_3 should be assigned to different processors. Assignment 1 satisfies the PR Principles for all pairs of consecutive modules. Therefore, it yields the best response time. Assignment 2 is the worst of the three assignments because it violates the PR Principles for all module pairs. Assignment 3 violates PR Principle 1 for some

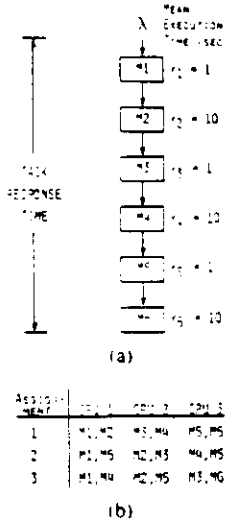


Fig. 6. (a) Task control-flow graph and (b) module assignments for PR experiment 3.

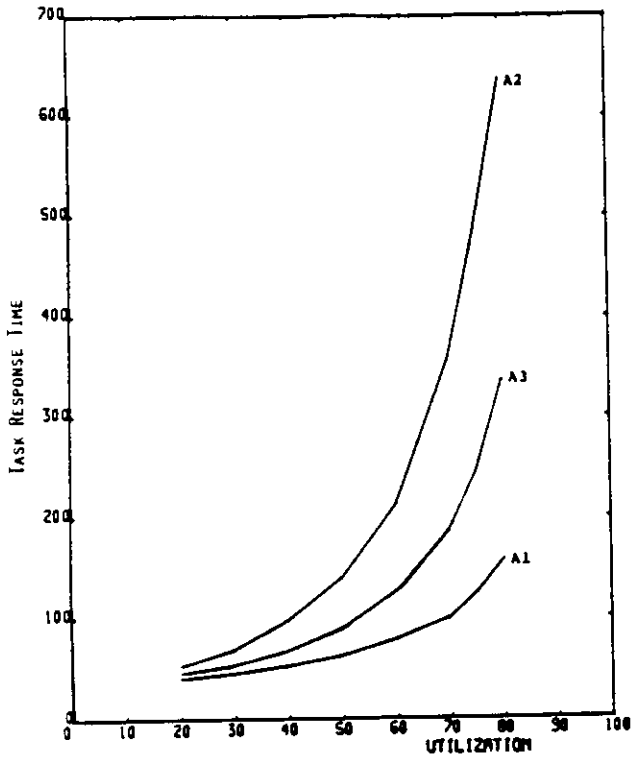


Fig. 7. Performance of three module assignments for PR experiment 3.

module pairs (e.g., separation of M_1 from M_2) and satisfies PR Principle 2 for some other pairs (e.g., separation of M_2 from M_3), therefore its performance lies between that of assignments 1 and 2. We repeated these experiments with deterministic execution times and obtained similar results.

Let us now discuss the reasons why good response time can be obtained from following the PR Principles. When a job arrival process is deterministic, the workload is *evenly spread* over time. The average queue length at every processor and thus, the average module wait time, should be smaller than that of a bursty arrival process. If two consecutive modules are assigned to the same processor and if the execution time of the second module is much larger than that of the first, the second one will act as a *regulating valve* which controls the task flow into the next processor. This makes the arrival process at the next processor *more deterministic* than the arrival process of the first module at the first processor. It is well known from queueing theory that for a given queueing system with a given arrival rate, the deterministic-arrival case yields less wait time than that of the bursty arrival case.

We can explain the results of experiment 3 based on the above reasoning. In assignment 1, M_2 at processor 1 has a large execution time, regulating the task flow into processor 2. Therefore, even though there are bursty arrivals for M_1, the invocation arrivals for M_3 at processor 2 are spread fairly evenly over time. As a result, the queue that contains invocations for M_3 and M_4 at processor 2 would be short and thus yield short wait times for M_3 and M_4. Likewise, M_4 acts as a regulating valve for the task flow into processor 3.

Assignment 2 yields poor response time. Since the size of M_1 is small, each group of bursty invocation arrivals for M_1 results in bursty arrivals for M_2 at processor 2 (i.e., there is no regulating valve between processors 1 and 2). As a result, there is a high probability of having many arrivals for M_2 waiting in the queue at processor 2. A newly arrived invocation for M_2 (called M_2 invocation) at processor 2 has a high probability of finding other previously arrived M_2 invocations in the queue. Execution of the first M_2 invocation in the queue generates an invocation for M_3 which is placed *at the end* of the queue at processor 2. There is a *long* wait time for the execution of this M_3 invocation due to the large number of M_2 invocations in front of it. This process is repeated with the execution of other M_2 invocations in the queue, thus contributing to the large response time. Furthermore, since M_3 (at processor 2) is small, the consecutive M_3 invocations finish their execution *rapidly* (although they each have a long wait time). This generates bursty invocations for M_4 at processor 3, again causing long wait time for those modules assigned at processor 3.

From these experiments, we note that module-size (service time) distribution and the *module-size ratios* of consecutive module pairs influence response time. In the following, we shall use analytical methods to derive quantitative guidelines for determining whether or not a consecutive module pair M_i and M_j, with given module-size distributions and average sizes y_i and y_j, should be colocated in the same processor.

B. PR Analysis

Consider the control-flow graph with two separate threads in Fig. 8(a) where all modules have deterministic execution times. Assume no IMC exists among modules. Let $y_1 = y_3$, $y_2 = y_4$. Thus, the module-size ratio $r = r_{1,2} = y_2/y_1 = r_{3,4} = y_4/y_3$. Furthermore, let the job arrival rates $\lambda_1 = \lambda_2 = \lambda$. Under the above condition, both assignments 1 and 2 [Fig. 8(b)] yield equal processor loads. However, they yield different response times. We wish to derive analytical results so that the module-size ratio r can be used as a parameter for determining whether M_1 and M_2 (also M_3 and M_4) should be colocated; that is, if r is greater than some threshold value, M_1

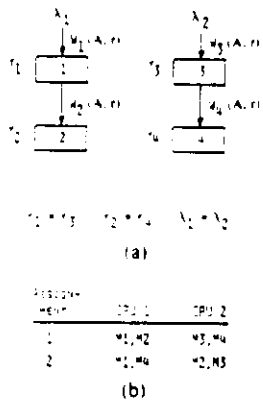


Fig. 8. PR analytical study. Two threads of consecutive modules for studying wait-time ratio between assignments 1 and 2 as a function of size ratio between the consecutive modules.

and M_2 should be assigned to the same processor; otherwise they should be separated.

The response time for the left control thread in Fig. 8(a) is $w_1(A, r) + y_1 + w_2(A, r) + y_2$ while the response time for the right thread is $w_3(A, r) + y_3 + w_4(A, r) + y_4$, where $w_i(A, r)$ is the queueing wait time experienced by module M_i. The queueing wait time is a function of both the assignment A and the module-size ratio r. Because of the symmetry of the two threads and the balanced loading on both processors, both threads have the same response time for assignment 1, denoted as A_1. The two threads also have the same response time for A_2. Therefore, it is sufficient to compare A_1 and A_2 using the response time of only one thread. The left thread is chosen for the following analysis.

The thread response times for A_1 and A_2 are $w_1(A_1, r) + y_1 + w_2(A_1, r) + y_2$ and $w_1(A_2, r) + y_1 + w_2(A_2, r) + y_2$, respectively. Since the values of y_1, y_2, y_3, and y_4 are fixed and independent of module assignment, they need not be considered. Thus, the *wait-time ratio* between assignments 1 and 2 is defined as

$$R = R(r) = \frac{w_1(A_1, r) + w_2(A_1, r)}{w_1(A_2, r) + w_2(A_2, r)}. \tag{8}$$

If $R < 1$, then assignment 1 yields better response time than assignment 2. The response-time improvement is due to better handling of the PR effect. Under such conditions, we should assign the pair of consecutive modules M_1 and M_2 to the same processor, and the other pair M_3 and M_4 to the alternate processor. If $R > 1$, then assignment 2 has better response time than assignment 1 and the consecutive modules should be assigned to different processors. Thus, R [see (8)] allows us to select the better module assignment.

Let us now discuss how to compute $w_i(A, r)$ and thus, R. For a given control-flow graph, module assignment, and module-size distributions, the wait-time w_i's for all modules can be estimated via the analytical model reported in [9]. The Appendix shows how to use the model to derive the numerator and denominator for (8). Therefore, we are able to determine the wait-time ratio R for various module-size ratios r. When all the modules have *constant* service times, we can compute R as a function of processor utilization ρ for executing M_1 and M_2 [Fig. 9(a) and (b)]; $\rho = \rho_1 + \rho_2$, $\rho_1 = \lambda_1 y_1$ and $\rho_2 = \lambda_1 y_2$. Note that R increases as r decreases from 100 to 0.4 [Fig. 9(a)]. As r further decreases from 0.4, R then reverses the trend and starts to decrease [Fig. 9(b)]. Note that $R = 1$, occuring at $r = 2.5$, is the threshold value that determines whether two consecutive modules should be colocated. R varies slightly with processor utilization. In the same manner, when the module execution times are exponentially distributed, we can derive the relationship between the wait-time ratio R and the module-size ratio r, as shown in Figs. 9(c) and (d). In this case, the threshold value $R = 1$ occurs at $r = 1$.

Note from Fig. 9 that when each module execution time is exponentially distributed, R is less sensitive to r, as compared to the case of deterministic module execution time. Results from the analytical model also confirm our observation in experiments 1 and 2 that response time is more sensitive to precedence relationship when module service times are deterministic than when they are exponentially distributed.

We have extended the above analysis to encompass the case where each control thread consists of *three* consecutive modules. M_1, M_2, M_3 are consecutive modules in one thread and M_4, M_5, M_6 in another. Let $y_1 = y_4$, $y_2 = y_5$, $y_3 = y_6$, and $\lambda_1 = \lambda_2$. Assignment 1 allocates all the consecutive modules to the same processors, i.e.,

Processor 1: M_1, M_2, M_3

Processor 2: M_4, M_5, M_6.

Assignment 2 allocates the consecutive modules to different processors, i.e.,

Processor 1: M_1, M_5, M_3

Processor 2: M_4, M_2, M_6.

Note that both assignments yield balanced loads. The wait-time ratio can be expressed similar to (8). The analytical results show that if we hold $y_1 (= y_4)$ fixed, then as module-size ratio $r_{2,3} = y_3/y_2 (= y_6/y_5)$ decreases, the wait-time ratio R between assignments 1 and 2 increases to a point and then reverses the trend and starts to decrease. This is similar to the case with two-module threads (Fig. 9). Similar relationships are observed for a control-flow graph consisting of four consecutive modules in a thread. All of these suggest that one way to handle the cases with more than two consecutive modules is to treat *each* pair of consecutive modules in a control-flow graph independently. Using the PR relation (Fig. 9) one can decide whether to allocate the two modules to the same processor. Our experience shows that assignments generated by such an approach yield good task response time.

IV. Module Assignment Algorithm

A. The Algorithm

Using exhaustive search to select an assignment from an entire assignment tree is prohibitively time consuming. Therefore, we shall propose a heuristic algorithm for module

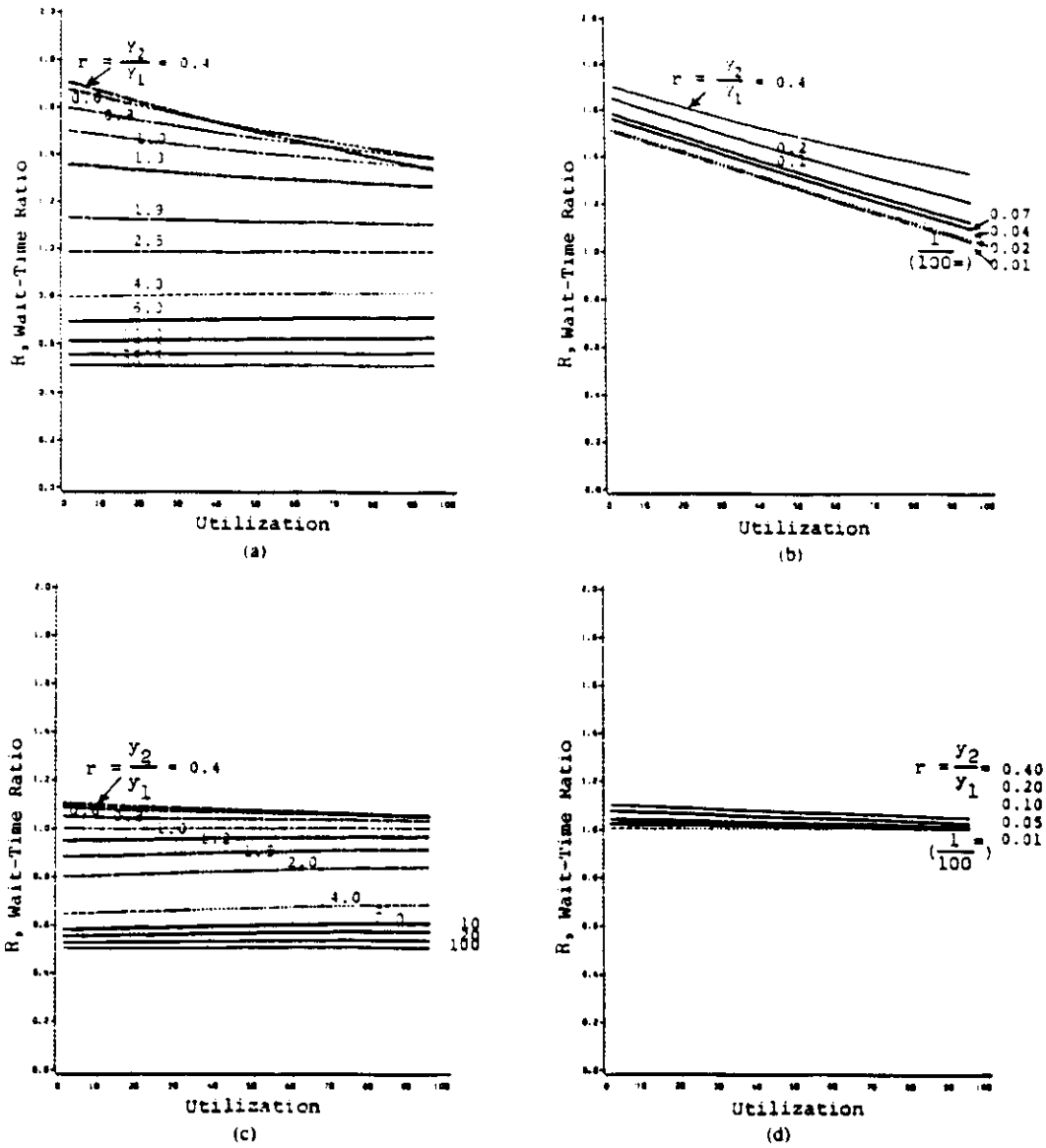


Fig. 9. Wait-time ratio between two assignments as a function of program-module-size ratio, for (a) and (b) deterministic execution times and (c) and (d) exponentially distributed execution times.

assignment (Fig. 10) that considers PR, IMC, and AET. We shall call it Algorithm P-I-A. This algorithm assumes that

1) there are J modules, $M_1, M_2, \cdots, M_J$, and S processors;

2) the AET (an average during the peak-load period) for each module M_j, T_j, ($j = 1, \cdots, J$) is given;

3) the IMC (an average during the peak-load period) between each module pair M_i and M_j, IMC_{ij}, ($i = 1, \cdots J; j = 1, \cdots J$) is given. Each IMC_{ij} can be derived from the V_{jk}'s [8].

The algorithm consists of two phases. Phase I reduces J modules to G groups ($G < J$) which corresponds to a much smaller assignment tree for Phase II. This grouping can be done with very little computation. Each group generated at the end of Phase I is a *set* of modules which will be assigned *as a single unit* to a processor. In Phase II these groups are assigned to the processors such that the bottleneck (in the most heavily utilized processor) is minimized.

The grouping of modules in Phase I is based on several factors. To reduce IPC, heavily communicating modules may be combined into groups. To do this, communicating module pairs are listed in descending order of the IMC volume (Step 1.1). Module pairs with large IMC are considered first.

Next, the PR effects are considered. The decision of whether to group two consecutive modules should be based on the two possibly conflicting factors: IMC volume and the effect of PR (i.e., module-size ratios). For a module pair (M_i, M_j), we propose to use *IMC index* $\gamma_{\text{IMC}}(i, j) = \text{IMC}_{ij}/\overline{\text{AET}}$ and *PR index* $\gamma_{\text{PR}}(i, j) = 1 - R(r_{ij})$, as defined in the initialization in the algorithm, to evaluate these conflicting factors. The IMC index indicates the relative IMC size normalized by the average module size in terms of the execution time. The typical index value should be between 0.1 and 0.5. An IMC with an index value below 0.1 may be considered negligible. Grouping two modules with the small IMC saves little IPC. The wait-time ratio R in the PR-index

Initialization:

0 Compute average AET, $\overline{AET}$, and average processor load, $\overline{PL}$.

$$\overline{AET} \leftarrow \sum_{i=1}^{I} T_i \quad / I$$

$$\overline{PL} \leftarrow \sum_{i=1}^{I} T_i \quad / S$$

Compute the IMC index and the PR index:

$$\gamma_{IMC}(i,j) = \frac{IMC_{ij}}{\overline{AET}} \qquad i = 1, \ldots, I, \quad j = 1, \ldots, I$$

$$\gamma_{PR}(i,j) = 1 - R(r_{ij}) \qquad i = 1, \ldots, I, \quad j = 1, \ldots, I$$

Do $\alpha = \alpha_1$ to α_2, with increment $\Delta\alpha$
Do $\beta = \beta_1$ to β_2, with increment $\Delta\beta$

Phase I — Combine modules with large IMC into groups to reduce total system load (i.e., to reduce the sum of processor loads):

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

Let each program module form a distinct group (a set):
$G_i \leftarrow \{M_i\} \quad i = 1, \ldots, I$

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

1.3 If $\alpha \times \gamma_{IMC}(i,j) + \gamma_{PR}(i,j) \leq 0$
go to Step 1.2.

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

1.5 If $T_s + T_t > (\overline{PL} \times \beta)$
go to Step 1.2.

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

1.7 Go to Step 1.2.

Phase II — Assign module groups to processors:

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

2.2 Record the minimum-bottleneck assignment.

end;
end;

Fig. 10. The Algorithm P-I-A.

can be computed from (8). For deterministic and exponential module execution times, from Fig. 9, R ranges between 0 and 2. $R = 1$ is the threshold value for deciding whether to group two consecutive modules. The condition $R < 1$ corresponds to a positive PR index $\gamma_{PR}(i, j)$, which favors the grouping of modules M_i and M_j. Likewise, $R > 1$ indicates a negative PR index favoring separation of M_i and M_j. Since the IMC index has a range between 0.1 and 0.5 and the PR index has a range between -1 and 1, a *scaling factor* α is introduced to combine the two indexes (Step 1.3). The α value can range from 1 to 10 and thus is a variable in Algorithm P-I-A.

Another factor to be considered is the size of a new group. If the new group, resulting from combining two subgroups, becomes too large, it would be impossible to obtain a balanced-load assignment during Phase II. Therefore, the concept of *processor-load threshold* ($\overline{PL} \times \beta$) is introduced (Step 1.5), where $\overline{PL}$ is the average processor load and β is a scale constant. If the size of a candidate new group is greater than the threshold, the two subgroups should not be combined. Note that a too small β would retard proper beneficial module grouping while a too large β makes it impossible to balance processor loads during Phase II. Our experiences on DPAD and other systems reveal that a good range for β is between 0.6 and 1.2 times of the average processor load.

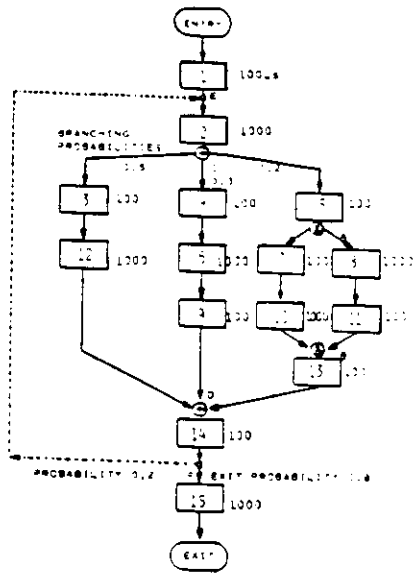


Fig. 11. Task control-flow graph for example 1.

For each pair of α and β values the algorithm generates a minimum-bottleneck assignment. We should select the assignment which corresponds to the *smallest* minimum-bottleneck among all sets of (α, β). If several assignments yield the same smallest minimum-bottleneck value, then we select the one with the smallest *total* processor load.

B. Examples

In this section, we shall use an example (denoted "example 1") to show that significant response-time improvement can be achieved when PR is considered in module assignment. Consider the control-flow graph in Fig. 11 where each program module has a deterministic execution time of either 100 or 1000 μs. Thus, the size ratios of most consecutive module pairs are either 0.1 or 10 (except for four pairs whose size ratios are 1.0). The job interarrival time is assumed to be exponentially distributed, with a rate of one hundred arrivals per *unit* interval. Each arrival makes an invocation to the entire control-flow graph. Some modules are executed more frequently than the others. Using the model presented in [8], the AET can be estimated for a specified time interval for each module. The estimated AET for each unit interval is shown in column 2 of Table II. Let us assume that the IMC sizes for all communicating module pairs are about equal, either 1400 or 1500 μs (see Table II and Fig. 12) which implies that IMC plays a lesser role than PR does. Given the PR, IMC, and AET, the module assignment generated by Algorithm P-I-A is shown in Table III along with the processor loads. In order to compare the PR effect, we generate a second assignment (also shown in Table III) which excludes the PR effect by replacing the Step 1.3 of Algorithm P-I-A with

1.3 If $\gamma_{IMC}(i, j) \leq 0.1$

go to Phase II.

TABLE II
AET T_i AND FILE-UPDATE IMC V_{ik} FOR EXAMPLE 1

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

* AET and Total IMC during a 100-arrival period

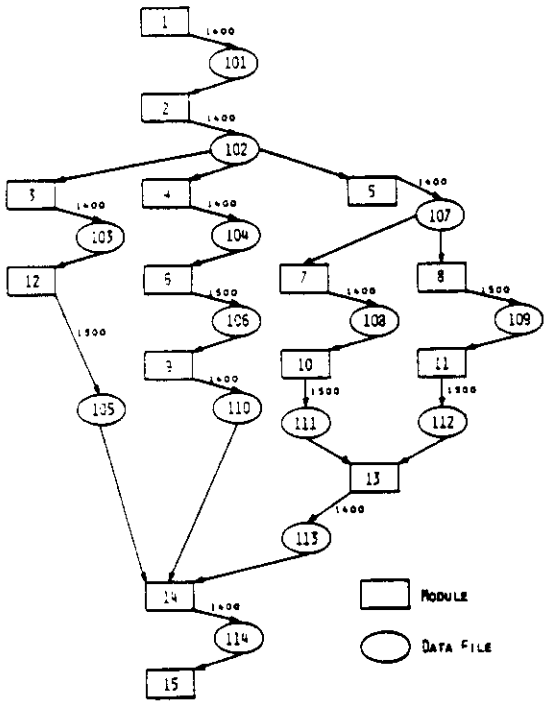


Fig. 12. Data-flow graph for example 1.

When $\gamma_{IMC}(i, j) \leq 0.1$, the IMC effect is negligible. We called this Algorithm I-A.

Note that in the assignment generated by Algorithm P-I-A, most module pairs are assigned (either colocated or separated) according to our PR Principles rather than the IMC sizes. For example, the module-size ratio $r_{4,6} = y_6/y_4 = 10$, thus M_4 and M_6 are colocated on processor 3. On the other hand, $r_{6,9} = 0.1$, thus M_6 is separated from M_9 although $IMC_{6,9}$ is larger than $IMC_{4,6}$.

These two assignments are simulated via the PAWS simulator. The average response time for each job arrival is measured from when the job arrives at the system until it finishes the execution of M_{15}. Fig. 13 portrays the response time for the two assignments. Note that the assignment generated by Algorithm P-I-A yields better response time than that generated by Algorithm I-A, with 10.8 percent improvement at processor utilization $\rho = 0.2$ and 25.7 percent improvement at $\rho = 0.8$ percent. Both assignments yield fairly balanced processor loads with similar bottleneck values. The difference in response time is due to the consideration of PR in module assignment.

We have applied both Algorithms P-I-A and I-A to the DPAD module assignment problem. The assignment generated from Algorithm P-I-A is the same as that generated from Algorithm I-A. This is due to the fact that there are very few consecutive modules in the DPAD system. Note that if a module is enabled by another through an OR branch with a low probability (say less than 0.5), the PR effect of such a module pair is greatly reduced. Therefore, they can logically be viewed as nonconsecutive modules because the second module is *not* always invoked for execution after the first module finishes its execution. Many module pairs in the DPAD belong to this type. The result also reveals that the performance of the best assignment obtained from Algorithm P-I-A is comparable with that of the exhaustive search (see Fig. 3). This demonstrates that the heuristic Algorithm P-I-A can generate an assignment which yields response time comparable to that of using the time-prohibitive exhaustive search method.

V. Summary

The three important parameters that influence task allocation are accumulative execution time (AET) of each module, intermodule communication (IMC), and precedence relationship (PR) among program modules. AET contributes to processor load and is independent of task allocation. IMC is the communication between program modules through shared files. When a module on a processor writes to or reads from a shared file on *another* processor, IMC becomes IPC (interprocessor communication) which requires extra processing and communication overhead. A task-allocation algorithm should minimize the IPC by assigning heavily communicating modules to the same processor.

An objective function for minimizing the bottleneck processor load (consisting of IMC and AET) has been proposed for task allocation. It is shown to generate load-balanced assignments with small IPC.

The third parameter for task allocation is the *precedence relationship* (PR). Due to PR, a program module cannot be enabled before its predecessor(s) finish executing. Both simulation and analytical study revealed that the module-size ratio of two consecutive modules affects task response time. Two principles were observed: 1) assigning two consecutive modules to a same processor yields good response times if the execution time of the second module is much larger than that of the first module; 2) if the second module is much smaller than the first one, the two consecutive modules should be separated and assigned to two different processors.

An analytical model was proposed to study the PR effect on response time which quantitatively determines whether two consecutive modules should be colocated in a processor. Our study reveals that this depends on the size ratio of the two consecutive modules, module-execution-time distribution, and processor load.

TABLE III
MODULE ASSIGNMENTS FOR EXAMPLE 1

	ASSIGNMENT I-A (W/O CONSIDERING PR)			ASSIGNMENT P-I-A (CONSIDERING PR)		
	CPU1	CPU2	CPU3	CPU1	CPU2	CPU3
MODULES	1 2 4	6 9 15	3 5 7 8 10 11 12 13 14	1 2 9	7 10 13 14 15	3 4 5 6 8 11 12
PROCESSOR LOADS	141550	144050	146850	143050	148300	147300
PERCENTAGE OF LOADS	32.73%	33.31%	33.96%	32.61%	33.81%	33.58%

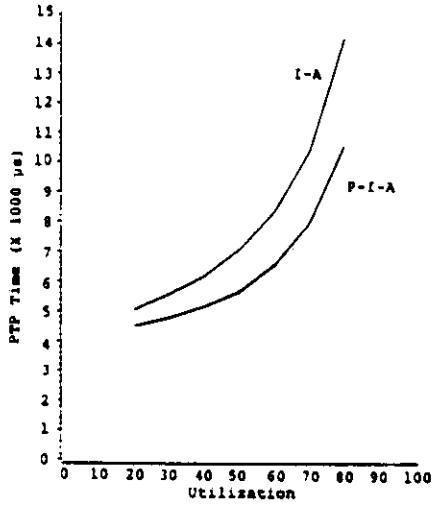


Fig. 13. Task-response-time comparison between assignments with and without PR consideration for example 1.

A heuristic algorithm that considers PR, IMC, and AET was developed for task allocation. In determining whether two consecutive modules should be colocated on the same processor, the effect of PR on response time may be in conflict with the effect of IMC. Therefore, the allocation algorithm jointly considers the effect of IMC and PR. Using the minimum bottleneck as an objective function, the algorithm was applied to two example systems. The results revealed that module assignments considering PR may yield better response time than assignments without PR consideration.

Further investigation is needed to generalize the algorithm to handle the assignment of replicated program modules. This could have a significant effect on task response time [22].

GLOSSARY

F_k—the kth file in the system
IMC—intermodule communication
IPC—interprocessor communication
J—number of program modules
K—number of files
L—processor loading
MLI—machine language instruction
M_j—the jth program module
$N_j(t_h, t_{h+1})$—number of times module M_j executes during (t_h, t_{h+1})
R—wait-time ratio of two assignments
S—number of processors (sites)
$T_j(t_h, t_{h+1})$—accumulative execution time (AET) for module M_j during (t_h, t_{h+1})
V_{jk}—IMC message volume sent from M_j to update the file F_k
$X = [x_{jr}]$—module assignment matrix in which x_{jr} (= 1 or 0) indicates whether module M_j is assigned to processor r
$r_{i,j}$—size ratio between modules M_j and M_i
$w_j(A, r)$—the queueing wait-time of module M_j for assignment A and module-size ratio, r
y_j—average execution time of module M_j per execution
δ_{ks}—indicating function to specify whether a copy of F_k resides at processor s

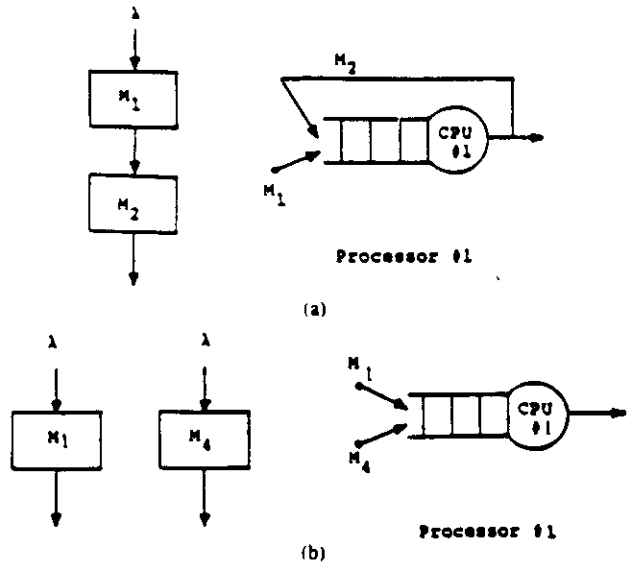


Fig. 14. Queuing model for computing module waiting time for (a) Assigment 1 and (b) Assignment 2.

λ—task arrival rate
ρ—processor utilization
ω—a normalizing constant for converting IMC to MLI's.

Appendix

I. Derivation of the Numerator for (8) (Assignment 1)

According to [9], the average wait-time $w_1(A_1, r)$ and $w_2(A_1, r)$ at Processor 1 for Assignment 1 can be obtained as follows. The mean wait-time for a given invocation of M_1 under FCFS scheduling policy is the average time to complete the executions of *both* the module invocation currently being served by the processor *and* all module invocations waiting in the job queue when the given M_1 invocation arrives. [See Fig. 14(a)]. Thus, we have

$$w_1(A_1, r) = w_{r1} + \overline{n_1}\,\overline{y_1} + \overline{n_2}\,\overline{y_2} \tag{A-1}$$

where

w_{r1} = mean residual module-execution time at processor 1 for assignment 1.

$$= \frac{1}{2}(\lambda \overline{y_1^2} + \lambda \overline{y_2^2})$$

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

c_i = coefficient of variation for the execution time of M_i

$\overline{n_i}$ = average number of M_i invocations waiting in the job queue.

To find the waiting time for M_2, we need to keep track of the queueing behavior starting from the arrival of the invocation for M_1. Let us consider a particular tagged invocation for M_1. After the completion of this tagged M_1 execution, its succeeding tagged invocation for M_2 is placed at the end of the job queue. The waiting time for this tagged M_2 invocation consists of three components. The first component is the total execution time of the new invocations for M_1 that arrive during the waiting and execution time of the tagged M_1 invocation. The second component is due to the executions of all the M_2 invocations which are enabled by the M_1 invocations that wait in the job queue when the tagged M_1 invocation arrives at processor 1. The last component is due to the execution of a M_2 invocation (with a probability of $\rho_1 = \lambda\overline{x_1}$). This M_2 invocation is enabled if module M_1 is in execution at the arrival of the tagged M_1 invocation. By adding these components, we have

$$w_2(A_1, r) = [w_1(A_1, r) + \overline{y_1})\lambda]\,\overline{y_1} + \overline{n_1}\,\overline{y_2} + \rho_1\,\overline{y_2}. \tag{A-2}$$

Since $\overline{n_i} = \lambda w_i$ (Little's result [23]) and $\rho_i = \lambda y_i$, (A-1) and (A-2) become

$$w_1(A_1, r) = w_{r1} + \rho_1 \cdot w_1(A_1, r) + \rho_2 \cdot w_2(A_1, r) \tag{A-3}$$

and

$$w_2(A_1, r) = (w_1(A_1, r) + \overline{y_1})\rho_1 + \rho_2 \cdot w_1(A_1, r) + \rho_1\,\overline{y_2}. \tag{A-4}$$

From (A-3) and (A-4), $w_1(A_1, r)$ and $w_2(A_1, r)$ can be solved as

$$w_1(A_1, r) = \frac{w_{r1} + \rho_1\rho_2(\overline{y_1} + \overline{y_2})}{1 - \rho_1 - \rho_2(\rho_1 + \rho_2)} \tag{A-5}$$

and

$$w_2(A_1, r) = \frac{w_{r1} + \rho_1\rho_2(\overline{y_1} + \overline{y_2})}{1 - \rho_1 - \rho_2(\rho_1 + \rho_2)}(\rho_1 + \rho_2) + \rho_1(\overline{y_1} + \overline{y_2}). \tag{A-6}$$

Therefore, the numerator of (8) is the sum of (A-5) and (A-6).

II. Derivation of the Denominator for (8) (Assignment 2)

With Assignment 2, Processor 1 can be treated as an M/G/1 queueing system with two types of "customers," M_1 and M_4. [See Fig. 14(b)]. The mean wait-time for these customers is given by

$$w_1(A_2, r) = w_4(A_2, r) = \frac{w_{r2}}{1 - \rho} = \frac{w_{r2}}{1 - \rho_1 - \rho_4} = \frac{w_{r2}}{1 - \lambda\overline{y_1} - \lambda\overline{y_4}} \tag{A-7}$$

where

ρ_i = processor utilization due to module i

w_{r2} = mean residual module execution time at processor 1 for assignment 2

$$= \frac{1}{2}(\lambda \overline{y_1^2} + \lambda \overline{y_4^2})$$

$$= \frac{1}{2}\lambda[(1 + c_1^2)\bar{y}_1^2 + (1 + c_4^2)\bar{y}_4^2].$$

Due to the symmetry in module threads (See Fig. 8), $w_2(A_2, r) = w_4(A_2, r)$. Thus, the denominator of (8) is equal to $2w_1(A_2, r) = 2w_{r2}/(1 - \lambda\overline{y_1} - \lambda\overline{y_4})$.

Acknowledgment

The formulation of the wait-time ratio as a function of module assignment and module-size ratio, was first proposed and studied by K. K. Leung at UCLA. The Appendix is

adapted from his dissertation [17]. The authors would also like to thank the referees for their comments which improved the organization of this paper.

REFERENCES

[1] R. Berry, K. M. Chandy, J. Misra, and D. Neuse, *PAWS 2.0—Performance Analyst's Workbench System: User's Manual*, Inform. Res. Ass., Austin, TX, Dec. 1982.

[2] S. H. Bokhari, "Dual processor scheduling with dynamic reassignment," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 341–349, July 1979.

[3] T. C. K. Chou and J. A. Abraham, "Load balancing in distributed systems," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 401–412, July 1982.

[4] W. W. Chu, "Optimal file allocation in a multiple computer system," *IEEE Trans. Comput.*, vol. C-18, pp. 885–889, Oct. 1969.

[5] W. W. Chu, D. Lee, and B. Iffla, "A distributed processing system for naval data communication networks," in *Proc. AFIPS Nat. Comput. Conf.*, vol. 47, 1978, pp. 783–793.

[6] W. W. Chu, L. J. Holloway, M. T. Lan, and K. Efe, "Task allocation in distributed data processing," *Computer*, vol. 13, pp. 57–69, Nov. 1980.

[7] W. W. Chu, J. Hellerstein, M. T. Lan, J. M. An, and K. K. Leung, "Database management algorithms for advanced BMD applications," Dep. Comput. Sci., Rep. UCLA-ENG-84-07 (CSD-840031), Univ. California, Los Angeles, Apr. 1984.

[8] W. W. Chu, M-T. Lan, and J. Hellerstein, "Estimation of intermodule communication (IMC) and its applications in distributed processing systems," *IEEE Trans. Comput.*, vol. C-33, pp. 691–699, Aug. 1984.

[9] W. W. Chu and K. K. Leung, "Task-response-time model and its applications for real-time distributed processing systems," in *Proc. 5th Real-Time Syst. Symp.*, Austin, TX, Dec. 1984, pp. 255–236.

[10] K. Efe, "Heuristic models of task assignment scheduling in distributed systems," *Computer*, vol. 15, pp. 50–56, June 1982.

[11] M. L. Green, E. Y. S. Lee, S. Majumdar, and D. C. Shannon, "A distributed real-time operating system," in *Proc. Symp. Distributed Data Acquisition, Comput. Contr.*, Dec. 1980, pp. 175–184.

[12] ——, *Phase III of Distributed Processing Architecture Design (DPAD) System—The DDP Underlay Simulator Experiment: Tactical Applications and d-RTOS Models*, TRW Defense Space Syst. Group, Special Rep. 35010-79-A005, May 15, 1980.

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

[14] K. B. Irani and K-W. Chen, "Minimization of interprocessor communication for parallel computation," *IEEE Trans. Comput.*, vol. C-31, pp. 1067–1075, Nov. 1982.

[15] C. J. Jenny, "Process partitioning in distributed systems," in *Proc. NTC 1977*, pp. 31:1-1–31:1-10.

[16] L. M-T. Lan, "Characterization of intermodule communications and heuristic task allocation for distributed real-time systems," Ph.D. dissertation, Rep. CSD-850012, Univ. California, Los Angeles, Mar. 1985.

[17] K. K. Leung, "Task response time and module assignment for real time distributed processing systems," Ph.D. dissertation, UCLA, Dec. 1985.

[18] P. Y. R. Ma, E. Y. S. Lee, and M. Tsuchiya, "A task allocation model for distributed computing systems," *IEEE Trans. Comput.*, vol. C-31, pp. 41–47, Jan. 1982.

[19] D. Palmer, "On the design of distributed data processing systems," in *Proc. COMPSAC 78*, Chicago, IL, invited paper.

[20] G. S. Rao, H. S. Stone, and T. C. Hu, "Assignment of tasks in a distributed processing system with limited memory," *IEEE Trans. Comput.*, vol. C-28, pp. 291–299, Apr. 1979.

[21] 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. Comput.*, vol. C-34, pp. 197–203, Mar. 1985.

[22] W. W. Chu and K. K. Leung, "Module replication and assignment for real-time distributed processing systems," *Proc. IEEE*, May 1987.

[23] J. D. C. Little, "A proof of the queueing formula $L = \lambda W$," *Oper. Res.*, vol. 9, pp. 383–387, 1961.

CHAPTER II

2.2 MODULE ASSIGNMENT FOR REAL-TIME DISTRIBUTED PROCESSING SYSTEMS

Module Replication and Assignment for Real-Time Distributed Processing Systems

WESLEY W. CHU, FELLOW, IEEE, AND KIN K. LEUNG, MEMBER, IEEE

Invited Paper

Response time is an important design criterion for real-time systems. A new analytic model is developed to estimate task response time. It considers such factors as interprocessor communication, module precedence relationship, module scheduling, interconnection network delay, and assignment of modules and files to computers. Since module assignment as well as its replication have great impact on task response time, a new algorithm is developed to iteratively search for module assignments and replications that reduce task response time. An objective function is introduced that is based on the sum of task response time and delay penalty for the violations of thread response time requirements. With this objective function, good module allocations and replications, which minimize task response time and yet satisfy the thread response time requirements, can be determined by the proposed algorithm.

To validate the algorithm, we compare the assignments generated by the algorithm for some sample distributed systems to the optimal module assignments obtained from exhaustive search. It shows that with a very small number of initial module assignments, our algorithm is able to generate the optimal or close-to-optimal assignments. The algorithm is also applied to a real-time distributed system for space defense applications where exhaustive search for the optimal assignment is not feasible. The generated module assignments (with replications) satisfy the specified thread response times, and compare closely with the simulation results. A series of experiments is also performed to characterize the behavior of the algorithm. In conclusion, the algorithm can serve as a valuable tool for assigning modules with replications for distributed systems.

I. 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 graceful performance degradation in case of failures. In this paper, we consider a class of real-time distributed processing systems (RTDPS) in which there is a single application task and message passing is used for communication between processors.[1]

The application task of a real-time system is often partitioned into a set of software modules (or simply, *modules*). The assignment of 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 [1]. 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 balanced.

Stone [2] and Rao *et al.* [3] use graph-theoretic algorithms which are tractable only for systems with two computers. Algorithms proposed in [4]–[7] 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 delay from multiple invocations which is a significant portion of the response time is ignored. 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 the processors according to the loading conditions. Each invocation for a replicated module is routed to one of its resident processors for execution. A special algorithm is required to perform the routing. One simple startegy is to route invocations for a replicated module to its resident processors in a round-robin fashion. Thus module replications may improve system load balancing, response time, and system reliability and availability.

[1] *Processor* and *computer* are interchangeably used in this paper.

Manuscript received September 1, 1985; revised January 7, 1987. This work was supported in part by the U.S. Army under Contract DASG60-79-C-0087, in part by Micro under Grant co-sponsored by Hughes Aircraft Co., and in part by the University of California under Contract 4-482516-19900.

W. W. Chu is with the Department of Computer Science, University of California, Los Angeles, CA 90024, USA.

K. K. Leung was with the Department of Computer Science, University of California, Los Angeles. He is now with AT&T Bell Laboratories, Holmdel, NJ 07733, USA.

IEEE Log Number 8714298.

To remedy the shortcomings in the existing module assignment algorithms, the module assignment proposed in this paper considers repeated task invocations, queueing effects, module precedence relationships, interconnection network delays, and module replications.

Task response time[2] in an RTDPS is the time from an invocation of the application task to the completion of its execution. Key factors (parameters) that affect task response time include interprocessor communications, processor loading, module precedence relationships, and interconnection network delay. A new task response time model [8] has been introduced which considers these key parameters. Often the application task consists of several sequences of modules which are referred to as *threads*. For some applications, real-time constraints may exist for certain threads (e.g., radar scheduler); the response times of individual threads rather than the entire task are of interest. In these cases, 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.

In this paper, we shall first describe the task response time model. Next, an objective function based on task response time and penalty for violating the thread response times are introduced. An algorithm is proposed to search for module assignment that minimizes the objective function. The algorithm considers both the module replication and assignment together. The effectiveness of the algorithm is evaluated by applying it to a set of sample distributed systems where optimal assignments can be determined by exhaustive search. Finally, using a real-time distributed system for space defense applications as an example, a series of experiments are performed to characterize the behavior of the algorithm.

II. Task Response Time Model

The application task in an 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 Fig. 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)* [9]. 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 distributed data files. IPC also incurs interconnection network delay. Therefore, IPC has a more significant impact on system performance than IMC within a computer.

[2]We use *task response time* to refer to its mean response time unless otherwise stated.

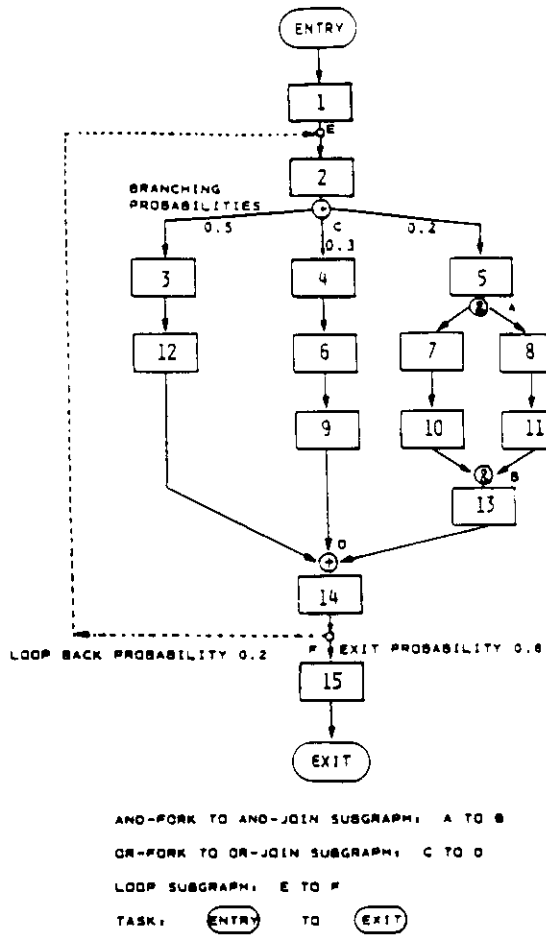


Fig. 1. A sample task control-flow graph.

Simulation techniques may be used to estimate the response time for the RTDPS, but such approaches are time-consuming and expensive. Queueing networks [10]–[12] 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 graph consists of these forks and joins, the routing scheme in the queueing network model becomes inadequate to represent the logical relationship 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 (queueing) 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* refers to the sum of a module's waiting time and its execution time. *Precedence waiting time* is the intermodule synchronization delay resulting from the precedence relationships among modules. Our 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.

A. 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 part of the module execution time. These IPCs are transmitted over the interconnection network, and eventually arrive at their destinations. On the destination computers, these input IPCs 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 computed. This computation depends on the distributed system under consideration, and will be illustrated by some examples given later in this paper.

Let module execution times be characterized by probability distribution functions. Then each computer can be modeled as a single-server queueing system with its resident modules (customers of different types) of specified service distributions. Based on the module assignment, the logical structures among modules, and task invocation rate, module invocation rates (customer arrival rates) on each computer can be determined. If several modules on the same computer are invoked simultaneously, this forms a bulk arrival.

In our model, we assume that 1) module invocation arrivals (single or bulk) are independent of each other, and 2) module invocation interarrival times are exponentially distributed. Under these assumptions, each computer in the RTDPS 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[3] for module executions.

Consider a computer that has h distinct types of module invocations (single or bulk invocations). Let the arrival rate for the ith type of module invocations be λ, and the Laplace Transform (LT) of the service requirement be $Y_i^*(s)$ for $i = 1, 2, \cdots, h$. One of these h types of module invocations (say the cth) represents all the input IPC on the computer. Then λ_c and $Y_c^*(s)$ become the arrival rate and LT of processing time for the input IPC, respectively. Suppose the ith type of module invocation consists of a set B_i of distinct module(s). Then

$$Y_i^*(s) = \prod_{j \in B_i} X_j^*(s)$$

where $X_j^*(s)$ is the LT of the execution time of module j. In case the ith invocation just consists of single module, B has one element.

Based on assumptions 1 and 2, this queueing system is an extension of the FCFS M/G/1 queue with total arrival rate

$$\lambda = \sum_{i=1}^{h} \lambda_i.$$

The LT of execution time for an arbitrary 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 (i.e., the time period from the invocation arrival to the start of its first module execution) are

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

and

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

where

$\overline{y_i^n}$ nth of moment service time for ith module invocation
ρ server utilization $= \sum_{i=1}^{h} \lambda_i \overline{y_i}$
$\overline{w}$ average module invocation waiting time.

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

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

For 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 $j_1, j_2, \cdots, j_{k-1}, j_k, j_{k+1}, \cdots$. 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 is the module invocation waiting time (independent of module invocations as FCFS is used) and $x(j_i)$ is the execution time for module j_i.

Note that w and the summation term in (4) correspond to the 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 (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 (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

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

i, and σ_{ij}^2 is given in (3). For the case of a single-module invocation, there will be only one module in the execution sequence.

B. 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 ((5) and (6)) onto the control-flow graph as arc weights (Fig. 2). The

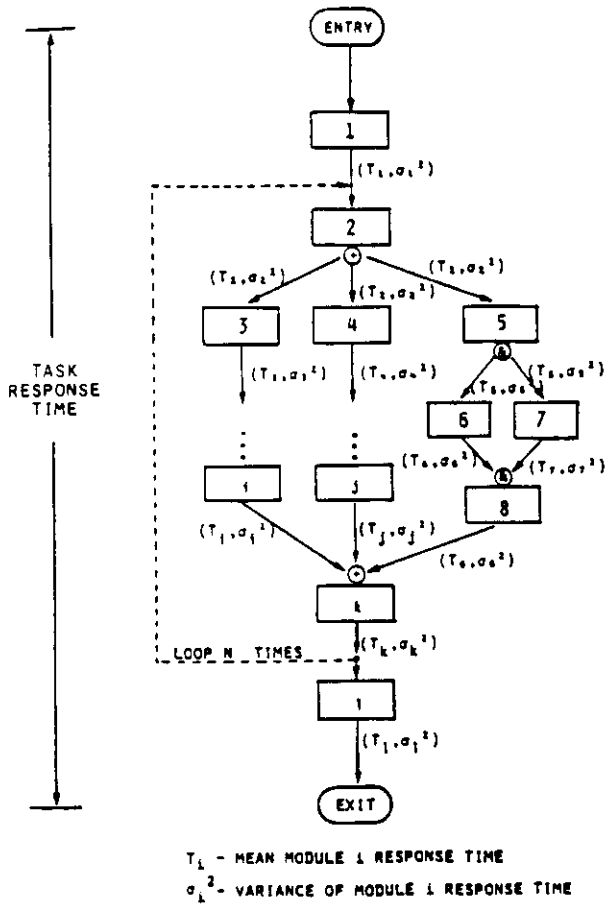


Fig. 2. Weighted control-flow graph for response time estimations.

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 variance of network delay[4] can be added to the weight of the arc from module i to j. Then the task response time can be estimated from this weighted control-flow graph model.

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 (Figs. 3–6). A task control-flow graph may contain a set of subgraphs which are a combination of these 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.

1) Sequential Thread Subgraph: The sequential thread subgraph (Fig. 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

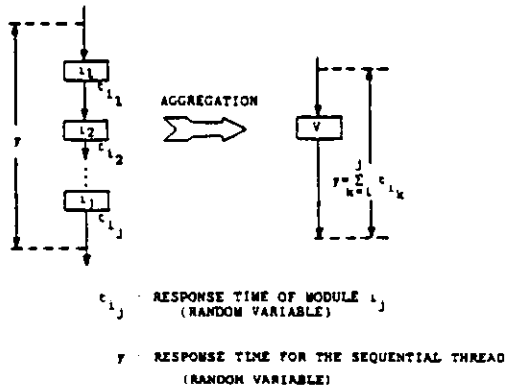


Fig. 3. Sequential thread.

thread. Treating the module response times (represented by the arc weights) as random variables, the total response time of the sequence thread is the sum of the arc weights of all modules in the thread.

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 Fig. 4. This

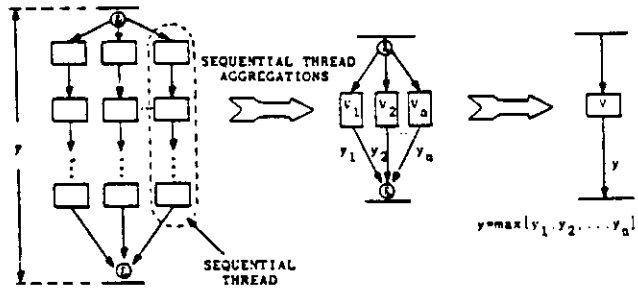


Fig. 4. And-fork to and-join subgraph.

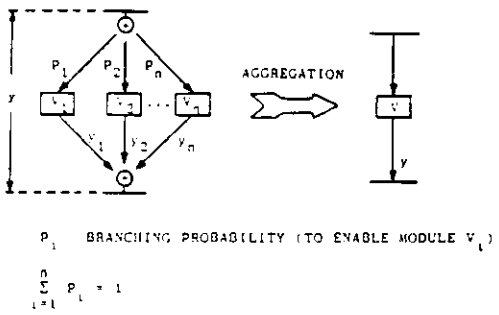
subgraph may correspond to the case in which the modules assigned to different computers require concurrent processing. Since sequential threads can be reduced to a single node as above, the and-fork to and-join subgraph can be aggregated into n nodes, V_i, with response time y_i for $i = 1, 2, \cdots, n$ where n is the number of threads from the and-fork (Fig. 4). Because of the and-join function, the response time for the subgraph is the maximum of y_i's.

Computing the response time for this subgraph requires the probability distribution functions for y_i's, which is rather complicated. In this study, we shall emphasize the *mean*

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

task response time, which can usually be determined by the first two moments of module response times. Therefore, these moments are derived from the module response time model, and are mapped onto the task control-flow graph as arc weights. According to the coefficients of variation of y_i's, they can be approximated by either Erlangian or hyper-exponential distribution functions [13]. 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.

3) Or-Fork to Or-Join Subgraph: This type of subgraph consists of an or-fork and an or-join as depicted in Fig. 5.



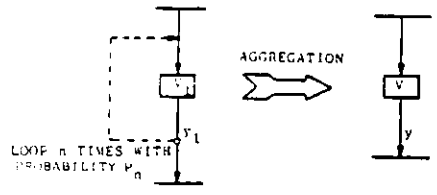
P_i BRANCHING PROBABILITY (TO ENABLE MODULE V_i)

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

Fig. 5. Or-fork to or-join subgraph.

At the or-fork, the module enables one of its succeeding modules. At the or-join, the module can be enabled by any one of its preceding modules. This type of subgraph facilities the system to process one of several threads based on certain selection criteria. The branching probability of each thread can be measured or estimated. The response time for the subgraph is the sum of the thread response times weighted by their invocation probabilities.

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 cyclic single-node graph, as shown in Fig. 6. The arc weight is the



y_i RESPONSE TIME FOR A SINGLE LOOP

y RESPONSE TIME FOR THE LOOP SUBGRAPH

Fig. 6. Loop subgraph.

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.

5) Integration of Periodic Modules with Task Control-Flow Graph: In the preceding discussion, we are mainly emphasizing the task control-flow graph with precedence and logical relationships. For certain applications, some of the modules may not be involved in any logical or precedence relationships with other modules, and are invoked by the system periodically rather than asynchronously by other modules. Although these periodic modules have no precedence relationships with other modules, these modules still need to communicate with each other via message passing and/or sharing common data files, as shown in Fig. 7. Thus periodic module invocations may facilitate the system to perform certain application functions in a timely fashion. Task response times in such cases depend on the logical structure of the task control-flow graph and its interrelationship with the periodic modules.

The task response time for these systems can be defined as the weighted sum of the aggregated response time for the task control-flow graph and response times for the periodic modules. The corresponding weighting factors are the task and periodic module invocation rates normalized by the total module invocation rate (see Fig. 7). When the task control-flow graph consists of sequential threads and/or or-fork to or-join subgraphs, the task response time can be reduced to the weighted sum of response times of all modules. An example of integration of such a system is presented in Section V-A.

C. Validation of the Analytical Task Response Time Model

Simulation experiments [8] have been conducted to validate model assumptions used by the task response time model. Since adjacent modules may be allocated at different processors, and since computers invoke modules asynchronously, the independent module invocations can be used as a good approximation and generate fairly accurate module response time estimations. In case the independent module invocation assumption generates results that are not accurate enough, a more complex model that considers dependent invocation arrivals may be used to estimate the modules response times [8]. More validation of the response time module with Poisson and non-Poisson module invocations will be presented in Section V-C as part of the overall performance assessment of the module assignment algorithm.

III. Module Assignment Algorithm with Module Replications

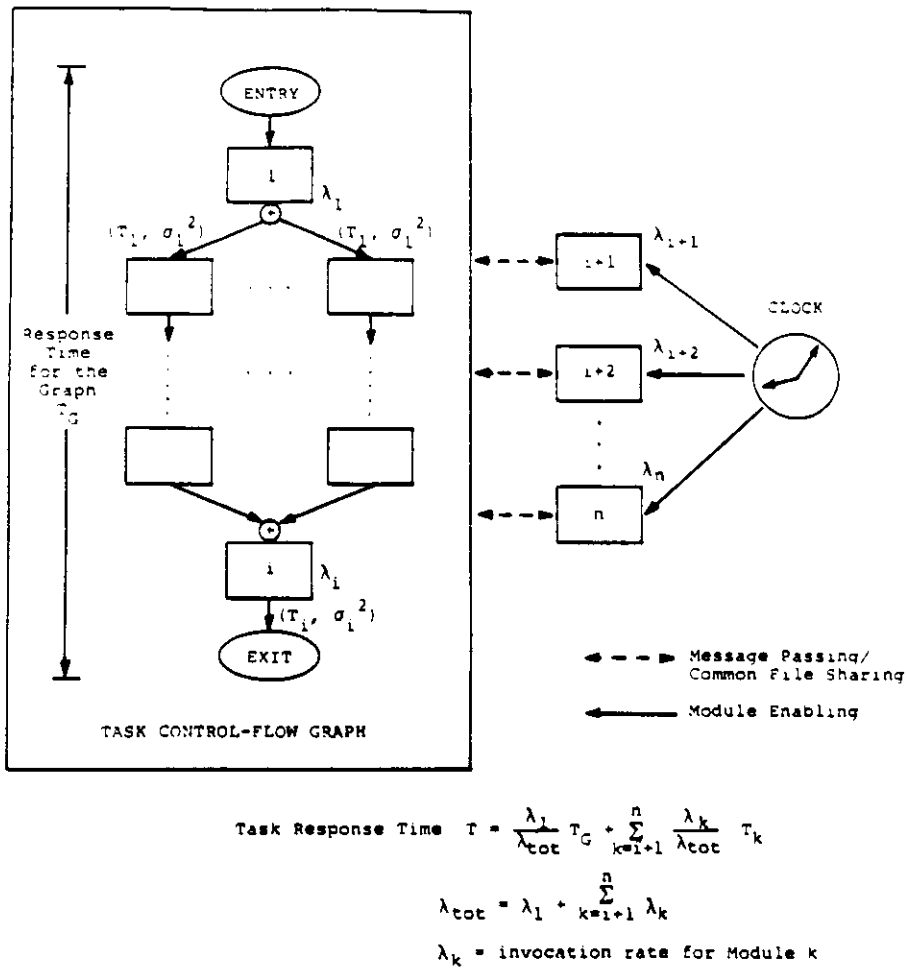
The replication and assignment of modules to computers in a distributed system is referred to as the *replicated module assignment problem* (RMAP). The RMAP minimizes the task response time by: 1) determining the optimal module multiplicities (i.e., number of copies for each module), and 2) allocating those module copies to computers, such that system performance objectives are satisfied. Since both module multiplicities and assignment of module copies to computers affect system performance, the problems are considered jointly. For simplicity, we use *module assignment* in the following text to refer to the replication and assignment of modules to computers.

A. Assumptions

Let us make the following assumptions for the RTDPS:

1) All of the computers in the systems do not have memory space constraints.

2) Data files are stored in main memory at a processor where its resident modules need to read and/or update the files.



$$\text{Task Response Time } T = \frac{\lambda_1}{\lambda_{tot}} T_G + \sum_{k=1+1}^{n} \frac{\lambda_k}{\lambda_{tot}} T_k$$

$$\lambda_{tot} = \lambda_1 + \sum_{k=1+1}^{n} \lambda_k$$

λ_k = invocation rate for Module k

Fig. 7. Integration of periodic module invocations with task control-flow graph.

3) If a module is replicated on several processors, the invocation rates for the module copies are equally divided among the processors.

4) All processors in the system are identical. Thus the execution time for each module is the same at any processor.

5) The network delay is independent of module assignment. Although different module assignments may generate different IPC traffic volume in the network, we assume that the network has sufficient bandwidth such that the delays do not depend on module assignment.

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

B. A New Objective Function

To search for optimal module assignment, we need to establish an objective function. Since the thread response time requirements are usually specified by users, the RMAP has two objectives: 1) to minimize task response time, and 2) to satisfy response time specifications for the threads. We shall combine these two objectives into a single objective function as follows:

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

where

- $T(A)$ task response time for module assignment A
- $T_{dp}(A)$ a positive-valued delay penalty function for module assignment A
- a a positive scaling constant to weigh the impact of violating thread response time requirements with respect to the task response time.

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

Let us define the following system parameters that will be used to compute the task response time:

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_{net}^2	variance of network delay,
λ	task invocation rate.

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

$\bar{t}_c(i, j)$	average processing time for the IMC from module i to j
$\sigma_c^2(i, j)$	variance of processing 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.[5] 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

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

C. Delay Penalty Function

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

$$d_i(A) = \begin{cases} t_i(A) - R_i, & \text{if } t_i(A) > R_i \\ 0, & \textit{otherwise} \end{cases}$$

where $i \in [1, k]$. We can express the thread response time overruns of 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 (i.e., $t_i(A) > R_i$), then $d_i(A)$ represents the discrepancy between that thread's actual response time and its requirement. Let S_i be the set of all (n_i) modules in thread i, and $\bar{w}_i$ be the *average allowable module waiting time* for each module in thread i, then

$$\bar{w}_i = \frac{R_i - \sum_{j \in S_i} \bar{x}(j)}{n_i} \tag{8}$$

where $\bar{x}(j)$ is the mean execution time for module j. We can express the average allowable module waiting times for all the threads as a vector $W_R = [\bar{w}_1, \bar{w}_2, \cdots, \bar{w}_k]$. For a given R_i, the numerator in (8) is the maximum sum of average waiting times for all modules of thread i. Since thread i consists of n_i modules, $\bar{w}_i$, as defined by (8), represents the average allowable waiting time for each module in thread i. Clearly, the smaller the value of $\bar{w}_i$, the faster the response time is required by thread i.

To provide an efficient search for good module assignments, we define the delay penalty function T_{dp} as a function of $D(A)$ and W_R so that it will have the following desirable properties:

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

Property 2: For a given module assignment, if two threads have the same thread response time overrun, then the thread with the stricter response time requirement (i.e., with a smaller $\bar{w}_i$) contributes a larger component of the total delay penalty.

Property 1 is self-explanatory. Property 2 is desirable in that the search algorithm is guided to reduce T_{obj} by satisfying those threads with stricter response time requirements before those with less stringent requirements. Based on these properties of the delay penalty function, the objective function, T_{obj} in (7), can guide us to search for module assignments that reduce response time overrun for all threads. Therefore, we define the delay penalty function according to these properties as follows:

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

where

$$\ell_i = \frac{\max\{\bar{w}_1, \bar{w}_2, \cdots, \bar{w}_k\}}{\bar{w}_i}, \quad \text{for all } i \in [1, k].$$

Note that (9) satisfies Property 1. Based on the definition of ℓ_i, we know that if $\bar{w}_i > \bar{w}_j$, then $\ell_i < \ell_j$. Thus Property 2 holds. By the definition of $d_i(A)$, if all thread response time requirements are satisfied, then $T_{dp}(D(A), W_R)$ is equal to zero. Substituting (9) into (7), we obtain the objective function for the

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

RMAP

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

For a given module assignment in a distributed system, each computer processes a set of assigned modules. The module invocation rates at each computer can be determined from the task invocation rate and the algorithm for serving replicated modules. Based on the task response time model, the task response time and thread response times can be computed. From the thread response times and their specified requirements, the delay penalty can then be calculated.

D. Search Algorithm for the RMAP

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

To minimize

$$\begin{aligned} T_{obj}(A) &= T(A) + aT_{dp}(D(A), W_R) \\ &= F(G, A, X, \sigma^2(x), T_c, \sigma^2(c), D_{net}, \sigma^2_{net}, \lambda, m, n) \\ &\quad + a \sum_{i=1}^{k} \ell_i d_i(A) \end{aligned} \quad (11)$$

with constraints

$$1 \le \sum_{i=1}^{n} A_{ij} \le n, \qquad \text{for all } j \in [1, m].$$

The constraint inequalities[6] indicate that each module must be allocated to at least one processor or may be replicated to every processor in the system.

The application task for the RTDPS requires repeated task invocations, and consists of various logical and precedence relations among modules. Therefore, the module assignment problem is more complicated than the multiprocessor scheduling problems [14] 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 [15]. However, due to the complexity and the characteristics of the RMAP, we propose an algorithm that searches for local optimal solutions and then selects the final solution from this set of local optimals.

The RMAP algorithm consists of three major components:

1) *Relocating Module from Longest Wait Processor to Shortest Wait Processor*

For a given module assignment, let the processor in the system that has the longest average module waiting (queueing) time, and the one with the shortest average module waiting time[7] 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 Replicating Module on SWP*

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. If certain threads violate their response time requirements, the candidates for further replications on the SWP are the modules in those threads. If all thread response time specifications are satisfied, those modules currently residing on the LWP are the candidates for further replication onto the SWP. We replicate the candidate module one at a time on the SWP. After such replication, the processor loading will be altered and certain modules may require to be relocated from the new LWP to the SWP to improve T_{obj}. We repeat such replication process, and finalize the replication of a module on the SWP that yields the minimum T_{obj}. Note that T_{obj} may not always be improved from such module replications on the SWP because it may increase IPC and/or violate thread response time requirements.

3) *Deleting Module 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 1) deleting modules may reduce IPC in the system and 2) deleting some replicated modules of those 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
Relocating Module from LWP to SWP

1) Determine initial module multiplicities (see Section III-E 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_0 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, $G, X, \sigma^2(x), T_c, \sigma^2(c), D_{net}, \sigma^2_{net}, \lambda, m$, and n, compute the assignment-dependent parameters for assignment A_0 (including IPC arrival rate and processing time for each processor).
   - 3.2 Compute the process or utilization on each computer for assignment A_0. If any computer(s) is saturated (i.e., its utilization $\ge$ 100 percent), stop or go to Step 1 for next iteration; otherwise continue.
   - 3.3 Invoke the task response time model:
     - 3.3.1 Compute $T_{obj}(A_0)$ for assignment A_0 and
     - 3.3.2 Identify the computers with the longest and shortest average model waiting times. (Denote them as $LWP(A_0)$ and $SWP(A_0)$, respectively.)
   - 3.4 Let S_L be the set of modules residing on $LWP(A_0)$ but not residing on $SWP(A_0)$. For each module $j \in S_L$, perform

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

[7]When different types of modules have different average waiting times for the given scheduling discipline (e.g., head-of-line priority) on a processor, the average module waiting time on the processor is defined to be the sum of all average module waiting times weighted by their respective invocation rates normalized by the total module invocation rate.

3.4.1 Temporarily relocate module j from $LWP(A_0)$ to $SWP(A_0)$ 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 $T_{obj}(A_j) \leftarrow \infty$; 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 $T_{obj}(A_j) \leq T_{obj}(A_0)$ for any $j \in S_L$ tested in Step 3.4, then perform

3.5.1 Set $A_0 \leftarrow A_j$, $T_{obj}(A_0) \leftarrow T_{obj}(A_j)$, $LWP(A_0) \leftarrow LWP(A_j)$ and $SWP(A_0) \leftarrow SWP(A_j)$ 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.)

Replicating Modules on SWP

4) Compute thread response time overrun $d_i(A_0)$ for all threads i where $i \in [1, k]$ and identify $LWP(A_0)$ and $SWP(A_0)$ for assignment A_0.

5) If there exists $d_i(A_0) > 0$ for any $i \in [1, k]$, then let S_R be the set of modules of all threads where $d_i(A_0) > 0$ for all $i \in [1, k]$. (Some thread response time requirements violated);
Otherwise, let S_R be the set of modules residing on $LWP(A_0)$. (All thread response time requirements satisfied.)

6) For each module $j \in S_R$ not residing on $SWP(A_0)$, perform:

6.1 Temporarily replicate module j onto $SWP(A_0)$ 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_{j0} (as Step 3 does).

7) If there exists $T_{obj}(A_{j0}) \leq T_{obj}(A_0)$ for any $j \in S_R$ from Step 6, then

7.1 Set $A_0 \leftarrow A_{j0}$, $T_{obj}(A_0) \leftarrow T_{obj}(A_{j0})$, $LWP(A_0) \leftarrow LWP(A_{j0})$, and $SWP(A_0) \leftarrow SWP(A_{j0})$ where $T_{obj}(A_{j0}) = \min_{i \in S_R} \{T_{obj}(A_{i0})\}$:
(To finalize a single module replication on SWP.)

7.2 Go to Step 4.

Deleting Modules from LWP

8) Otherwise, let S_L be the set of modules residing on $LWP(A_0)$. For each module $j \in S_L$ which has more than one copy, perform

8.1 Temporarily delete module j from $LWP(A_0)$ and form a new assignment A_j;

8.2 Perform Step 6.2 to obtain the local optimal assignment A_{j0}.

9) If there exists $T_{obj}(A_{j0}) \leq T_{obj}(A_0)$ for any $j \in S_L$ from Step 8, then

9.1 Set $A_0 \leftarrow A_{j0}$, $T_{obj}(A_0) \leftarrow T_{obj}(A_{j0})$, $LWP(A_0) \leftarrow LWP(A_{j0})$, and $SWP(A_0) \leftarrow SWP(A_{j0})$ where $T_{obj}(A_{j0}) = \min_{i \in S_L} \{T_{obj}(A_{i0})\}$.
(To finalize a single module deletion from LWP.)

9.2 Go to Step 4.

10) Otherwise, stop or go to Step 1 for next iteration. (Reach the final local optimal assignment A_0.)

E. Initial Module Multiplicities

During the execution of the algorithm, module multiplicities are changed due to the module replications and deletions resulting from the search for better assignments. The algorithm is re-iterated with a number of randomly selected assignments[8] to provide a good module assignment for a given distributed system. The final suboptimal 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 carefully 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 processing requirement of each module copy, which is equal to the invocation rate times the mean module execution time, should be less than processor 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) Based on the invocation rate of each module and the mean execution time, we can compute its processor utilization, ρ_i, for $i = 1, 2, \cdots, m$, where m is the total number of modules in the task.

2) Compute the mean processor utilization due to a module

$$\bar{\rho} = \sum_{i=1}^{m} \rho_i / m.$$

3) Compute the initial multiplicity α_i for module i for $i = 1, 2, \cdots, m$:

3.1 $\alpha_i = \left\lceil \frac{\rho_i}{\bar{\rho}} \right\rceil$

3.2 if $(\rho_i/\alpha_i) > 1$, then let α_i be the smallest integer such that $(\rho_i/\alpha_i) < 1$.

Note that α_i should be less than the total number of processors in the system. The initial module multiplicities determined by the above procedure may not initially provide satisfactory thread response times. However, it provides a starting point for the RMAP to search and revise the module multiplicities for the module assignment that minimizes T_{obj}.

IV. Algorithm Validation

In order to validate the RMAP algorithm, we apply it to two simple distributed systems and compare the best gen-

[8]A similar technique was used in [16] for the traveling salesman problem.

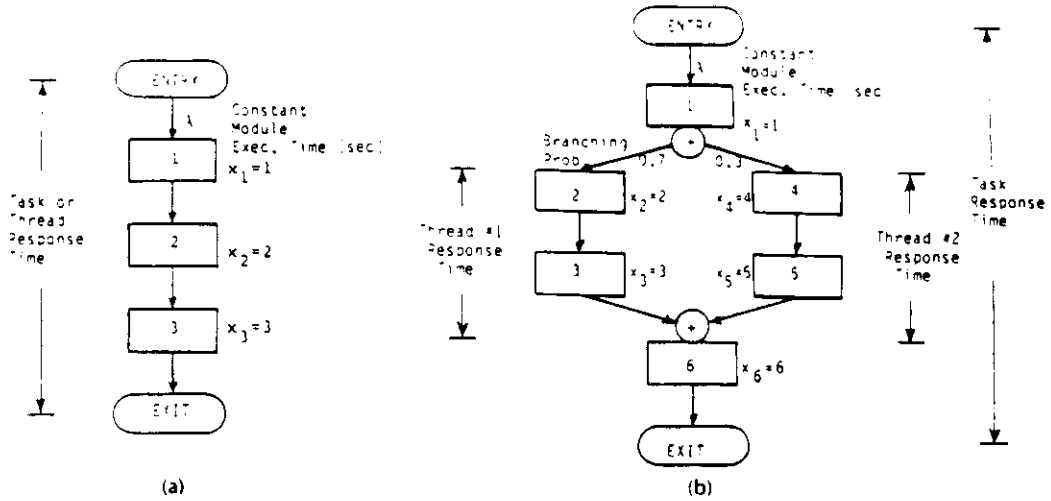


Fig. 8. Task control-flow graphs for the sample distributed systems. (a) Task *A*. (b) Task *B*.

erated assignments with the optimal assignment determined from exhaustive search. The distributed system consits of three identical processors. Two application tasks, task *A* and task *B*, are studied. Task *A* (Fig. 8(a)) represents a sequential thread of three modules, while task *B* (Fig. 8(b)) contains an or-fork to or-join subgraph with a total of six modules. All modules are assumed to have deterministic processing times. Modules are processed on an FCFS basis. Since task *A* has a single thread (*thread #1*), the thread response time is equal to the task response time. Task *B* has two threads in the or-fork to or-join subgraph: *thread #1* corresponds to M_2 and M_3 and *thread #2* consists of M_4 and M_5.

We assume certain modules in the system cannot be replicated and therefore only exist as single copies. We let M_2 in task *A* and M_1 and M_5 in task *B* be single-copy modules. Other modules can be arbitrarily replicated. Table 1 shows eight cases which represent different task invocation rates, with and without IPC processing for tasks *A* and *B*. For simplicity, the IMC among a pair of modules on the same processor is assumed to be zero. If a pair of adjacent modules (i.e., modules linked by an arc in the graphs) reside on distinct processors, then the execution times for both modules are increased by 1 s to account for the IPC processing. For the replicated module cases, although two adjacent modules are colocated on the same processor, they may generate IPC if the preceding module invokes its succeeding module at a remote processor (for load balancing) instead of on the local processor. Interconnection network delay is assumed to be negligible. The optimal module assignments, obtained by exhaustive search over all possible solutions, are presented for comparison with the best assignment obtained from the proposed algorithm.

To apply the RMAP algorithm, we set $a = 10$ (selection of a will be discussed in detail in the next section). The corresponding thread response time requirements for all cases are shown in Table 1. The procedure given in Section III-E was used to determine the initial module multiplicities. The RMAP algorithm was then reiterated with 5 random initial module assignments for cases #1 through #4 and 25 for cases #5 to #8. The best module assignments (in terms of T_{obj}) generated by the algorithm and their corresponding response times are given in Table 2. We note that: 1) in most of the cases, the algorithm generates the optimal solutions, 2) the response times of the suboptimal solutions generated by the algorithm deviate only a few percent from those of the optimal assignments, and 3) the solutions generated by the algorithm satisfy the thread response time requirements. Because of the relatively small size of solution space for task *A*, the optimal module assignments were generated during the first iteration. While for task *B*, cases #5 to #8, the best module assignments were generated during the first 20 iterations. The solution space of task *B* is in the order of $7^4 \times 3^2 = 21\,603$ possible assignments (as four modules can be replicated on one to three processors, while the other two can reside only on one processor). In order to understand

Table 1 Global Optimal Module Assignments by Exhaustive Search

Case No.	Task Invocation Rate, λ	Control-Flow Graph	IPC Exist?	Thread Response Time Requirements (s)					Thread Response Time (s)		Task Response Time (s)
				Thread #1	Thread #2	CPU #1	CPU #2	CPU #3	Thread #1	Thread #2	
1	0.1	*A*	No	8		1,2	1,3	1,3	6.87		6.87
2	0.35	*A*	No	15		1,3	2	1,3	14.17		14.17
3	0.1	*A*	Yes	15		1,2,3	3	3	10.02		10.02
4	0.2	*A*	Yes	25		1,2	3	3	19.33		19.33
5	0.1	*B*	No	9	15	1,2,3	4,6	5,6	6.84	13.28	18.83
6	0.135	*B*	No	11	19	1,2,3	4,6	5,6	9.77	15.52	23.72
7	0.1	*B*	Yes	13	16	1,4,5	2,3,6	2,3,6	11.53	13.53	24.65
8	0.125	*B*	Yes	19	21	1,2,4,5	2,3,6	2,3,6	14.78	17.09	31.15

Table 2 Best Module Assignments Generated by the RMAP Algorithm

Case No.	Thread Response Time Requirements (s)					Thread Response Time (s)		Task Response Time (s)	Percent Off From Optimal Task Response Time
	Thread #1	Thread #2	CPU #1	CPU #2	CPU #3	Thread #1	Thread #2		
1	8		1,2	1,3	1,3	6.87		6.87	0
2	15		1,3	2	1,3	14.17		14.17	0
3	15		1,2,3	3	3	10.02		10.02	0
4	25		1,2	3	3	19.33		19.33	0
5	9	15	1,2,3	4,6	5,6	6.84	13.28	18.83	0
6	11	19	1,2,3,4	3,6	5,6	9.77	15.52	24.39	2.8
7	13	16	1,2,4,6	3,6	2,3,6	11.57	14.52	24.92	1.1
8	19	21	1,2,4,5	2,3,6	2,3,6	14.78	17.09	31.15	0

the response time distribution of the solution space, we collected the response time statistics of all possible assignments for cases #5 to #8 from exhaustive search. We noted that only about 0.09 to 1.84 percent of the possible assignments can yield response times that lie within the range of 5 percent from the optimal task response time. In spite of the few assignments that yield good response times and the very large solution space for each case, the algorithm needed only to search about 400 to 850 assignments for generating a good module assignment with a few minutes computation time on the VAX 11/780. Similar results were obtained when using a different delay penalty constant (e.g., $a = 1000$) and different thread response time requirements.

V. Algorithm Application: The Sentry System

Next, we shall apply the RMAP algorithm to a real-time distributed system, the Sentry System [17], 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 replication and allocation generated by the proposed algorithm meet the specified thread response time.

A. 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. Its control-flow graph is given in Fig. 9. Three of these modules, M_{10}, M_{11}, and M_{12}, are periodically invoked 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 selects a thread in accordance with the type of the return signal received. The names of various threads are given in Fig. 9. 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. To integrate modules M_{10} and M_{12} which do not belong to any thread with the or-fork task control-flow graph, we define the task response time for the system as the weighted sum of the average module/thread response times.

$$T = \frac{\lambda_{OS}}{\lambda_{tot}}(T_1 + T_2) + \frac{\lambda_{OV}}{\lambda_{tot}}(T_1 + T_3 + T_4) + \frac{\lambda_{TI}}{\lambda_{tot}}(T_1 + T_5)$$
$$+ \frac{\lambda_{OT}}{\lambda_{tot}}(T_1 + T_7) + \frac{\lambda_{OD}}{\lambda_{tot}}(T_1 + T_8 + T_9)$$
$$+ \frac{\lambda_6}{\lambda_{tot}} T_6 + \frac{\lambda_{10}}{\lambda_{tot}} T_{10} + \frac{\lambda_{12}}{\lambda_{tot}} T_{12}$$
$$+ \sum_{\substack{i=1 \\ i \neq 11}}^{12} \frac{\lambda_i}{\lambda_{tot}} \sum_{\substack{j=1 \\ j \neq 11}}^{12} p_{ij}\, \delta_{ij}\, D_{net}(i, j)$$
$$= \sum_{\substack{i=1 \\ i \neq 11}}^{12} \frac{\lambda_i}{\lambda_{tot}} T_i + \sum_{\substack{i=1 \\ i \neq 11}}^{12} \frac{\lambda_i}{\lambda_{tot}} \sum_{\substack{j=1 \\ j \neq 11}}^{12} p_{ij}\, \delta_{ij}\, D_{net}(i, j)$$

where

λ_{xx} invocation rate for thread xx, $xx \in \{OS, OV, TI, OT, OD\}$,

λ_i invocation rate for M_i,

$\lambda_{tot} = \sum_{\substack{i=1 \\ i \neq 11}}^{12} \lambda_i$ total invocation rate for all modules,

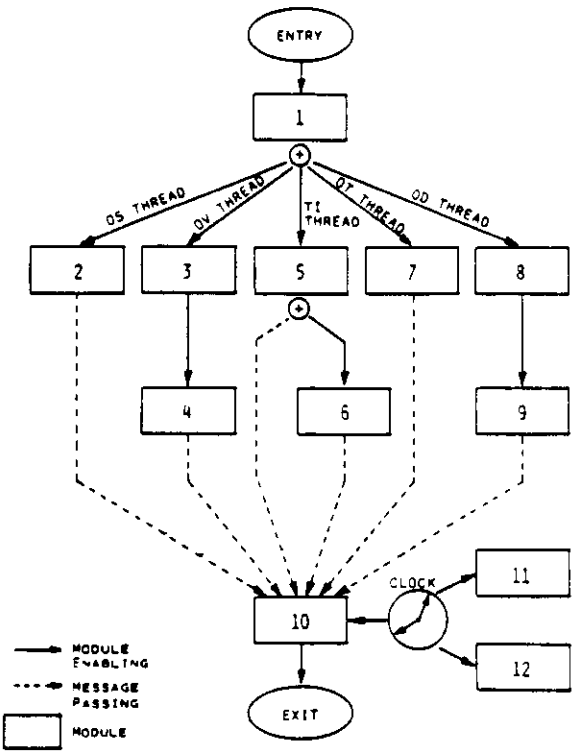


Fig. 9. Task control-flow graph for the Sentry System.

Table 3 Module Execution, File Access Times, and Invocation Rates for the Sentry System

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

Note: All times are in microseconds.

T_i mean response time for M_i (averaged over the response times for all copies if M_i has replicated copies),

p_{ij} probability that M_i enables M_j,

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

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

The data flow of shared file access is presented in Fig. 10. 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 both reads and updates the file.

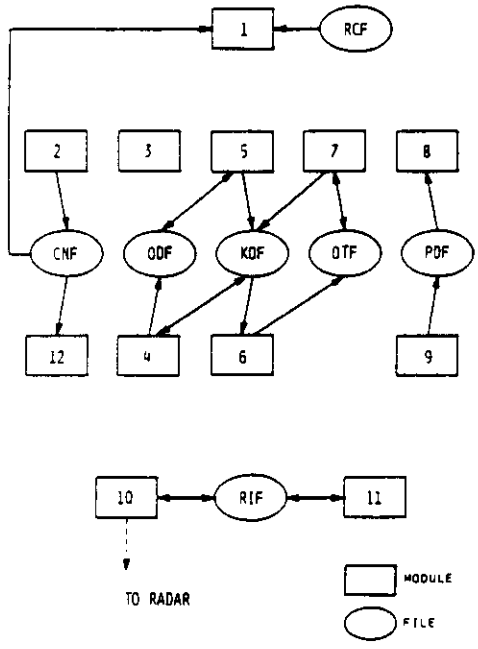


Fig. 10. Shared data files in the Sentry System.

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 3. Modules communicate with other modules by sharing common (in-memory) 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 in Table 4. If two communicating modules are located on distinct processors, the IMC becomes IPC which requires processing on both the transmitting and the receiving processors. The processing time for the IPC is called *IPC time* (IPC_{ij}). In the Sentry System, the IPC times on transmitting and receiving processors are different. $IPC_{ij} = 80\ \mu s$ 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 ms.

B. Characteristics of the RMAP Algorithm

To study the characteristics of the RMAP algorithm, we experimented with it under different environments such as varied thread response time requirements, initial module multiplicities, and delay penalty scaling constant. Four selected sets of thread response time requirements, R_A, R_B, R_C, and R_D, were used, as shown in Table 5. Among these requirements, R_A is the least stringent, R_D is the strictest, 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, were used in the experiments. α_A was generated in accordance with the procedure in Section III-E; M_1 has two copies, M_5 has five copies, and all other modules have a single copy. In α_B, since the processor utilization for M_5 is 95 percent, M_5 is initially duplicated into two copies to avoid possible processor saturation; while all other modules consist of a single copy. Three delay penalty scaling constants, 1, 10, and 1000, were used in the experiments.

Eleven experiments were performed. Each of them used

Table 4 IMC Times for Various Module Pairs

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

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

Table 5 Selected Sets of Thread Response Time Requirements for the Sentry System

Thread \ Thread Response Time Requirements (ms)	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	3.0	3.2	3.15	3.1
OT	4.5	4.0	3.95	3.9
OD	5.5	5.0	4.95	4.9

a different combination of thread response time requirements, initial module multiplicities, and delay penalty scaling constants. Experiments #1 through #9 used α_A as initial module multiplicities, while Experiments #10 and #11 used α_B. The scaling constant for Experiment #1 was 1. Experiments #2 to #5 and #10 used 10, while Experiments #6 to #9 and #11 used 1000 as the scaling constant. In each experiment, the RMAP algorithm was iterated with a prespecified number (500 or 1000) of random initial module assignments. For each initial assignment, the algorithm generated a suboptimal assignment. Only certain local optimal assignments could satisfy the thread response time requirements. The final module assignment was selected from this set of local optimals that yielded 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 6. The corresponding module assignments are shown in Table 7.

Since R_D is the most stringent thread response time requirement, no module assignment was generated to meet the thread response time requirements (Experiments #5 and #9), even using 1000 randomly selected initial module assignments. The number of module assignments searched

Table 6 The Thread Response Times of the Module Assignments Generated by the RMAP Algorithm

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 (ms)	*OS*	1.741	1.394	1.699	1.653	1.678	1.441	1.586	1.616	1.678	1.642	1.767
	OV	6.051	5.894	6.052	6.136	6.153	5.657	6.236	6.309	6.131	6.044	6.261
	TI	3.204	3.186	3.173	3.101	3.169	3.247	3.171	3.128	3.160	3.167	3.189
	OT	3.710	3.639	3.838	3.667	3.743	3.657	3.578	3.767	3.738	3.687	3.776
	OD	4.977	5.057	4.954	4.797	4.848	4.556	4.772	4.798	4.844	4.990	4.995
T_{obj}		1.138	1.058	1.148	1.119	1.837	1.062	1.106	1.151	61.59	1.131	1.151
Local Optimals Meet Requirements		15.8%	99.8%	15.9%	0.44%	0%	99.8%	14.5%	0.33%	0%	13.2%	14.3%

*Algorithm was iterated with 1000 initial random module assignments, while others with 500 initial random module assignments.

Table 7 Module Assignments Generated for the 11 Experiments

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
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

and CPU time used to obtain the final module assignment varied from one experiment to another. They ranged from 19 100 to 45 000 module assignments with various module multiplicities. The required CPU times ranged from 1.48 to 3.48 h on the VAX 11/780 machine. We also observed 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[9] for the cases with stricter thread response time requirements. The modules in a thread with a strict response time requirement should be allocated to lightly loaded processors in order to avoid violating the stringent thread response time specification. Modules with less stringent response time requirements may be allocated to more heavily loaded processors. This restricts the freedom of the search algorithm to perform 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) Delay penalty scaling constant:

For threads that do not require strict response time requirements, the scaling constant does not have much effect on module assignment. However, for stringent thread response time requirements, selection of the scaling constant is critical. Experiment #1 used a very small scaling constant ($a = 1$). Note that the *TI* thread in the experiment violated its response time requirements, yet the algorithm was not able to detect this violation. This is because the scaling constant a is so small that the final suboptimal assignment yielded 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 assignments which have one or more threads violating their response time specifications. For example, T (task response time) for the Sentry System is about 1 ms 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 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 had the same thread response time requirements R_B. Experiments #3 and #7 used α_A as initial multiplicities whereas Experiments #10 and #11 used α_B. 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

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

insensitive to the initial module multiplicities provided that no single module copy for the initial multiplicities would saturate a processor.

C. Validation of the RMAP Results via Simulations

In this section, we shall assess the performance of the module assignments generated by the RMAP algorithm and also validate the accuracy of the task response time model. The thread response times for the assignments from Experiments #2, #3, and #4 were simulated and compared with the thread response time requirements and the response time predictions from the analytical model. We note that the interarrival times of radar return signals (i.e., task/thread invocations) for the Sentry System are not exponentially distributed. Further, the invocation rates for *OT* and *OD* threads are highly variable with time (Fig. 11). Thus the invocation arrivals are non-Poisson and time-variant. To evaluate the RMAP algorithm and the accuracy of the analytical model (which assumes Poisson arrivals), both Poisson and non-Poisson (using actual arrival statistics) thread invocation arrivals were used in the simulation. From Table 8, we note that the simulated response times for both Poisson and non-Poisson radar signal return cases compare closely to the analytical predictions. The response times for the Poisson arrivals match closer with the analytical predictions than those of the non-Poisson cases, especially for *OT* and *OD* threads.

When the task invocation arrival processes differ significantly from a Poisson process, the module assignment generated by the RMAP algorithm may produce a high deviation from the response time predictions (see the *OD* thread in Experiment #2 in Table 8). 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 be able to generate module assignments that satisfy the required thread response times.

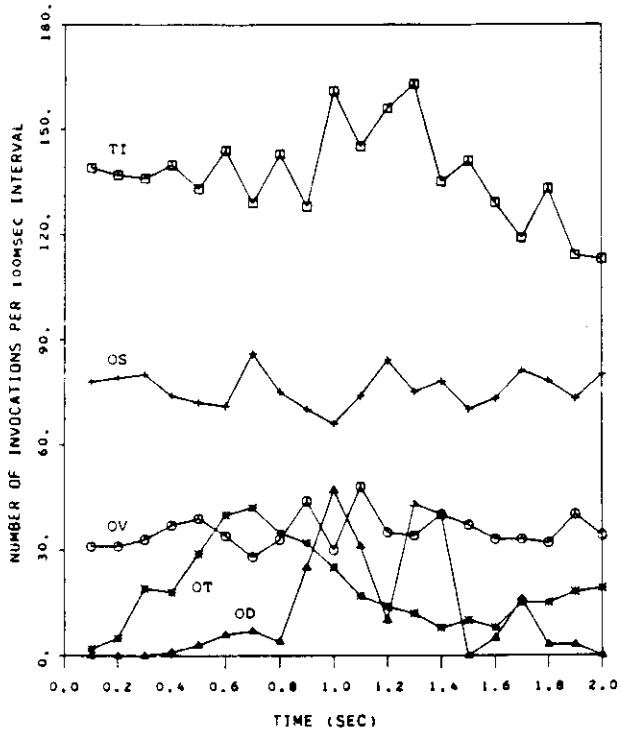


Fig. 11. Thread invocation rates for the Sentry System.

VI. Summary

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 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 which uses the sum of task response time and delay penalty as the objective function, to perform module assignment and replication for distributed systems.

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 algorithm reaches a local optimal assignment where the objective function cannot be improved further. The algorithm is repeated with a prespecified number of random initial assignments. The final module assignment

Table 8 Comparison of Response Time from Analytical Predictions with that of Simulation

Threads \ Response Time	Experiment #2				Experiment #3				Experiment #4			
			Simulations				Simulations				Simulations	
	R_A	Anal. Pred.	P	N-P	R_B	Anal. Pred.	P	N-P	R_C	Anal. Pred.	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

Anal. Pred.: analytical predictions.
P: Poisson radar signal returns.
N-P: non-Poisson radar signal returns.

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 set of sample distributed systems and a real-time distributed system for space defense applications. For the sample distributed systems, exhaustive search was performed to obtain the optimal assignments. With a small number of initial module assignments, the algorithm has been able to generate the optimal solutions for most cases; while in a few other cases, the solution assignments generated by the algorithm are practically identical to the response time performance of the optimal solutions.

Because of exponential growth in computation requirements, exhaustive search for optimal assignments for large-size systems is not feasible. Therefore, the proposed algorithm was used to generate the module assignments for a real-time distributed system for space defense applications. The assignment yields satisfactory task response time while meeting the set of thread response time specifications. A series of experiments was performed to characterize the behavior of the algorithm. The experiments indicate that the final module assignment is rather insensitive to initial module multiplicities. Further, the algorithm is quite robust over a wide range of the delay penalty scaling constant. 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 can be used in many instances for approximating non-Poisson arrival cases. Few cases of deviation are noted where the inputs 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 could be otherwise prohibitive. Therefore, the proposed algorithm is a valuable tool for module assignment with replications for distributed processing systems.

Acknowledgment

The authors wish to thank J. Huang and D. Townsend of Titan Systems, Inc., Los Angeles, CA, for performing the simulation for the Sentry System; J. Hellerstein of IBM Thomas Watson Research Center, Yorktown Heights, NY, and L. Lan of AT&T Bell Laboratories for their constructive comments on the earlier version of this paper.

References

[1] W. W. Chu, L. J. Holloway, M. T. Lan, and K. Efe, "Task allocation in distributed data processing systems," *IEEE Computer*, vol. 13, no. 11, pp. 57–69, Nov. 1980.

[2] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Trans. Software Eng.*, vol. SE-3, no. 1, pp. 85–93, Jan. 1977.

[3] G. S. Rao, H. S. Stone, and T. C. Hu, "Assignment of tasks in a distributed processing system with limited memory," *IEEE Trans. Comput.*, vol. C-28, no. 4, pp. 291–299, Apr. 1979.

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

[5] K. Efe, "Heuristic models of task assignment scheduling in distributed systems," *IEEE Computer*, vol. 15, no. 6, pp. 50–56, June 1982.

[6] T. C. K. Chou and J. A. Abraham, "Load balancing in distributed systems," *IEEE Trans. Software Eng.*, vol. SE-8, no. 4, pp. 401–412, July 1982.

[7] 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. Comput.*, vol. C-34, no. 3, pp. 197–203, Mar. 1985.

[8] W. W. Chu and K. K. Leung, "Task response time model and its applications for real-time distributed processing systems," in *Proc. 5th Real-Time Symp.* (Austin, TX, Dec. 1984), pp. 225–236.

[9] W. W. Chu, M. T. Lan, and J. Hellerstein, "Estimation of intermodule communication (IMC) and its applications in distributed processing systems," *IEEE Trans. Comput.*, vol. C-33, no. 8, pp. 691–699, Aug. 1984.

[10] F. Basket, K. M. Candy, R. Muntz, and F. G. Palacios, "Open, closed, and mixed networks of queues with different classes of customers," *J. ACM*, vol. 22, no. 2, pp. 248–260, Apr. 1975.

[11] P. Heidelberger and K. S. Trivedi, "Queueing network models for parallel processing with asynchronous tasks," *IEEE Trans. Comput.*, vol. C-31, no. 11, pp. 1099–1109, Nov. 1982.

[12] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[13] C. H. Sauer and K. M. Chandy, *Computer System Performance Modeling*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[14] M. R. Garey and D. S. Johnson, *Computers and Intractability*. San Francisco, CA: W. H. Freeman, 1979.

[15] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Englewood Cliffs, NJ: Prentice-Hall, 1982.

[16] S. Lin, "Computer solutions of the Traveling Salesman problem," *Bell Syst. Tech. J.*, vol. 44, pp. 2245–2269, Dec. 1965.

[17] Titan Systems, Inc., "Distributed data base management (DDBM) analysis," Semi-Annual Rep., Contract No. DASC60-83-C-0080, CDRL No. A004, July 19, 1985.

$P[\hat{i}(M-l_1)<t]$. Further, it can be expressed as

$$P[\hat{i}(M-l_1)<t] = 1-P[\hat{i}(M-l_1)\geq t] = 1-\sum_{j=0}^{M-l_1-1} P[A(t)=j] \quad (16)$$

where

$$P[A(t)=j] = \begin{cases} e^{-\lambda_B t} & j=0 \\ \sum_{i=1}^{j} \binom{j-1}{j-1} \theta^i (1-\theta)^{j-i} \frac{\lambda_B t^i}{i!} e^{-\lambda_B t} & j>0 \end{cases} \quad (17)$$

is the probability of exactly j invocations arriving in a period t [3].

By substituting (17) into (16) and differentiating with respect to t, we have

$$P[t<\hat{i}(M-k)\leq t+dt] = \left\{ \sum_{j=1}^{(M-k)-1} \sum_{i=1}^{j} \binom{j-1}{j-i} \theta^{j-i}(1-\theta)^{j-(j-i)} e^{-\lambda_B t} \left[\frac{\lambda_B(\lambda_B t)^{i-1}}{(i-1)!} - \frac{\lambda_B(\lambda_B t)^{i-1}}{(i-1)!} \right] + \lambda_B e^{-\lambda_B t} \right\} dt \quad (18)$$

Case II : the tagged invocation belongs to the g^{th} group, $G=g$, for $g = 2,3,\ldots,M$

The tagged invocation waiting time depends on the number of invocations in the batch queue at the instant of its arrival. Thus, we have

$$E[W_B \mid G=g] = \sum_{m=g}^{M-1} E[W_B \mid G=g, L=m] P[L=m \mid G=g] \quad \text{for } g = 2, 3, \ldots, M-1. \quad (19)$$

where

$L = \sum_{j=1}^{g} L_j = m$ = total number of invocations in the g groups,

$P[L=m \mid G=g]$ = probability that the total number of invocations in the g groups is m given that the tagged invocation belongs to the g^{th} group,

$E[W_B \mid G=g, L=m]$ = expected waiting time at the batch queue given that there are m invocations in the g groups and the tagged invocation belongs to the g^{th} group.

Since the batch is made up of at least g groups of invocations, the total number of invocations in the first $g-1$ groups, L', must be less than M; that is, $L' = \sum_{j=1}^{g-1} L_j = m' < M$. Thus

$$P[L=m \mid G=g] = \sum_{m'=g-1}^{M-1} P[L=m \mid L'=m', G=g] P[L'=m' \mid G=g] \quad (20)$$

Using (6) and normalizing over the probabilities of all the possible values of m', it can be shown that

$$P[L'=m' \mid G=g] = \frac{\binom{m'-1}{g-2} \theta^{g-1}(1-\theta)^{m'-(g-1)}}{\sum_{j=g-1}^{M-1} \binom{j-1}{g-2} \theta^{g-1}(1-\theta)^{j-(g-1)}} \quad (21)$$

The conditioned probability of the total number of invocations in the g groups in (20) is

$$P[L=m \mid L'=m', G=g] = P[L_g=(m-m')] = \theta(1-\theta)^{m-m'-1} \quad (22)$$

where $P[L_g=m-m']$ is the probability of having $m-m'$ invocations in the g^{th} group.

Substituting (21) and (22) into (20), we obtain the probability of the total number of invocations at the batch queue at the arrival instant of the tagged invocation. This allows us to determine that how many additional invocations are needed to form a batch with maximum size before the time-out period expires. To determine the amount of time remaining before the time-out period ends, we also need to know the arrival time of the last invocation group. The remaining time is the maximum waiting period for the g^{th} invocation group. Hence, we condition $E[W_B \mid G=g, L=m]$ on the arrival instant of the g^{th} invocation group, t_g.

$$E[W_B \mid G=g, L=m] = \int_{t=0}^{T_o} E[W_B \mid G=g, L=m, t<t_g\leq t+dt] P[t<t_g\leq t+dt \mid L=m, G=g] \quad (23)$$

where

$P[t<t_g\leq t+dt \mid L=m, G=g]$ = probability of the tagged invocation arriving at t given that it is in group g and $L=m$.

Since the arrival instant of the g^{th} invocation group is not affected by the total number of invocations in the n invocation groups, we have

$$P[t<t_g\leq t+dt \mid L=m, G=g] = P[t<t_g\leq t+dt \mid G=g] \quad (24)$$

Since the first invocation group always arrives at the instant $t_1=0$, $P[t<t_g\leq t+dt]$ can be interpreted as the instant of $(g-1)^{th}$ arrival from a Poisson source. Therefore, t_g is an Erlangian distribution [4]. Thus,

$$P[t<t_g\leq t+dt \mid G=g] = \frac{\frac{\lambda_B(\lambda_B t)^{g-2}}{(g-2)!} e^{-\lambda_B t} dt}{\int_{t=0}^{T_o} \frac{\lambda_B(\lambda_B t)^{g-2}}{(g-2)!} e^{-\lambda_B t} dt} \quad (25)$$

The denominator of (25) is the normalization constant which is the sum of the probabilities for $0<t_g<T_o$.

Given that m invocations have arrived at the batch queue at the time instant t, then the tagged invocation waiting time is either the time for the arrivals of M-m invocations (1st term of (26)), or T_o-t if less than M-m invocations arrive before time-out period ends (2nd term of (26)). Thus, in the same manner as in case I, the conditional waiting time at the batch queue in terms of $\hat{i}(M-m)$ is

$$E[W_B \mid G=g, L=m, t<t_g\leq t+dt] = \int_{t'=0}^{T_o-t} t' P[t'<\hat{i}(M-m)\leq t'+dt'] + (T_o-t) \int_{t'=(T_o-t)}^{\infty} P[t'<\hat{i}(M-m)\leq t'+dt'] \quad (26)$$

By substituting (25) and (26) into (23), and (20) and (23) into (19), we can obtain the mean waiting time of the tagged invocation given that it is not among the first invocation group. By further substituting the results from cases I (13) and II (19) into (4), the mean waiting time at the batch queue can be expressed as a function of the invocation group arrival rate, λ_B; the average number of invocations in a group, $1/\theta$; the maximum batch size, M; and the time-out period, T_o.

3.2 Waiting Time At Processor Queue, W_P

Module invocations are grouped into batches and sent to the processor queue to wait for scheduling and execution as shown in Figure 2. Since invocations of different modules require different execution time and scheduling overhead, the processor can be modeled as a single server queuing system with multiple types of customers [5]. In addition, we assume that the batched module invocations arriving to the processor queue are batch Poisson arrivals. To characterize the arrival processes for each of the k modules assigned to the processor,

we need to specify the average batch arrival rate, λ_i, and the probability distribution of the number of invocations in a batch (hereafter will be called batch size), $P[Y_i=y_i]$ for $y = 1, 2, .., M_i$. The detailed derivation is shown in the Appendix A. The average batch arrival rate can be expressed in terms of the probability distribution and the invocation arrival rate to the batch queue.

$$\lambda_i = \begin{cases} \dfrac{\lambda_i'}{E[Y_i]} & \text{for external invocations} \\ \dfrac{\lambda_i''}{E[Y_i]} & \text{for internal invocations} \end{cases}$$

where $E[Y_i = y_i]$ is the expected batch size for module i. The mean waiting time of an invocation at the processor can be divided into the mean waiting time for the batch, W_g, and the mean waiting time of the invocation after the processor starts processing the batch, W_e, that is,

$$E[W_p] = E[W_g]+E[W_e] \tag{27}$$

By considering a batch of invocations as a single customer to the processor, the mean batch waiting time at the processor queue can be obtained by using the Pollaczek-Khinchin formula for M/G/1 queuing systems [6].

$$E[W_g] = \frac{\lambda_p \overline{X_p^2}}{2(1-\lambda_p \overline{X_p})} \tag{28}$$

where

$\lambda_p = \sum_{i=1}^{k} \lambda_i$ = total batch arrival rate to the processor,

$\overline{X_p}$, $\overline{X_p^2}$ = the average and the second moment of the batch processing time respectively.

$\overline{X_p}$ and $\overline{X_p^2}$ can be obtained from the Laplace Transform of the batch processing time distribution, $X_p^*(s)$, which is

$$X_p^*(s) = \sum_{i=1}^{k} \frac{\lambda_i}{\lambda_p} \sum_{y_i=1}^{M_i} X_b^*(s \mid Y_i=y_i) P[Y_i=y_i] \tag{29}$$

where

$X_b^*(s \mid Y_i=y_i)$ = the Laplace Transform of the processing time distribution for a batch with y_i invocations,

λ_i/λ_p = probability that a batch contains module i invocations.

When a batch of invocations arrive at the processor queue, a given invocation in the batch has to wait for the completion of processing of all the preceding invocations. The waiting time of the invocation, $E[W_s]$, is part of the overall waiting time (27) at the processor queue. For a batch with y_i invocations for module i, we have

$$E[W_s \mid Y_i=y_i] = \frac{1}{y_i} \sum_{j=1}^{y_i-1} F_i+j(V_i+X_i) = (y_i-1)\left[\frac{(V_i+X_i)}{2} + \frac{F_i}{y_i}\right]$$

By unconditioning $E[W_s \mid Y_i=y_i]$, we have,

$$E[W_s] = \sum_{i=1}^{k} \frac{\lambda_i E[Y_i]}{\lambda_e} \sum_{y_i=1}^{M_i} E[W_s \mid Y_i=y_i] P[Y_i=y_i] \tag{30}$$

where

$\lambda_i E[Y_i]/\lambda_e$ = probability that an invocation is for module i,

$\lambda_e = \sum_{i=1}^{k} \lambda_i E[Y_i]$ = total invocation rate to the processor.

Substituting (28) and (31) into (27), we obtain the mean waiting time of an invocation at the processor queue.

3.3 Module Scheduling Time, D

The batch scheduling overhead for module i depends on the fixed overhead, F_i; the variable overhead, V_i; and the number of invocations in the batch, Y_i. The mean scheduling time is the weighted sum of the scheduling time for all the possible batch sizes.

$$E[D_i] = \sum_{y_i=1}^{M} E[D_i \mid Y_i=y_i] P[Y_i=y_i] \tag{31}$$

where $E[D_i \mid Y_i=y_i]$ can be obtained from (1).

The module response times (3) can be obtained by summing (4), (27), (31) and the module execution times. The thread response times can be computed by aggregrating the response time of all the modules in the thread [2].

4. Examples

4.1 Module Scheduling for Distributed Systems

Let us apply the BST to the application task (Figure 1) on a two-processor loosely coupled distributed system. The task can be decomposed into two threads. Thread 1 consists of modules 1, 2, 4, 8 and thread 2 consists of modules 1, 3, 5, 6, 7, 8. Table 1 shows the module assignment and the BST parameter values used in this example. The parameters include the mean execution times, fixed and variable overheads, the maximum batch sizes and the time-out periods. The task is repeatedly invoked according to a Poisson process with an average task invocation rate λ_T. From the task control flow graph, we note that 90% of the task invocations flow through thread 1. Thus the modules in thread 1 are invoked more often than other modules. To reduce scheduling overhead, we use BST for the modules in thread 1. The time-out periods are set in accordance with the maximum batch sizes and the average module invocation rates. For module 1, the time-out period starts when a module invocation arrives and finds that the batch queue is empty. The average time for the next invocation arrivals is equal to the reciprocal of the module invocation rate. Therefore, the time-out period is set equal to the interarrival time of module 1 which is equal to the interarrival time of task invocations. By assigning equal maximum batch sizes to all the modules in thread 1, invocations arriving at the batch queues for modules 2, 4, and 8 will be in groups. As a result, a smaller time-out period is used for these modules.

The optimal module assignment to the processors using the FCFS scheduling algorithm is shown in Table 1. This assignment yields the minimum task response time. We use this module assignment to compare the thread response time between the BST and the FCFS algorithms. The thread response times of FCFS are computed using the techniques in [2]. To compute the thread response times of BST, we first compute the module response times from our analytical model. We then aggregate the modules in a given threads to obtain its thread response time. The response time of threads 1 and 2 for the BST and the FCFS scheduling algorithms are shown in Figures 4 & 5. Note that for FCFS, the thread response time increases exponentially as the invocation rate increases. At low invocation rates, BST yields higher response times than those of FCFS. This is mainly due to the delay in forming batches of module invocations. As traffic increases, the delay in grouping invocations into batches decreases, which in turn decreases the thread response time. Therefore, at high invocation rates, BST yields lower response times than those of FCFS. Since the BST algorithm reduces fixed scheduling overhead, systems using BST can support a larger module invocation rates than those using FCFS. We have also simulated the example via

the PAWS simulation [7] with Poisson task invocation arrivals. We note that our analytical results compared closely with the simulations which provides a validation of the assumptions used in our model.

	EXEC. TIME	SCHEDULING OVERHEADS		BATCH QUEUE PARAMETERS		
MODULE	X	F	V	M	T_o	MODULE ASSIGNMENT
1	1.5	1.2	0.3	2	$1/\lambda_T$	CPU2
2	1.5	1.2	0.3	2	2.0	CPU2
3	0.5	0.3	0.2	1	N/A	CPU2
4	1.5	1.2	0.3	2	2.0	CPU1
5	0.5	0.3	0.2	1	N/A	CPU2
6	0.5	0.3	0.2	1	N/A	CPU1
7	0.5	0.3	0.2	1	N/A	CPU1
8	1.5	1.2	0.3	2	2.0	CPU1

Table 1. Parameters for the Module Scheduling Policy Example

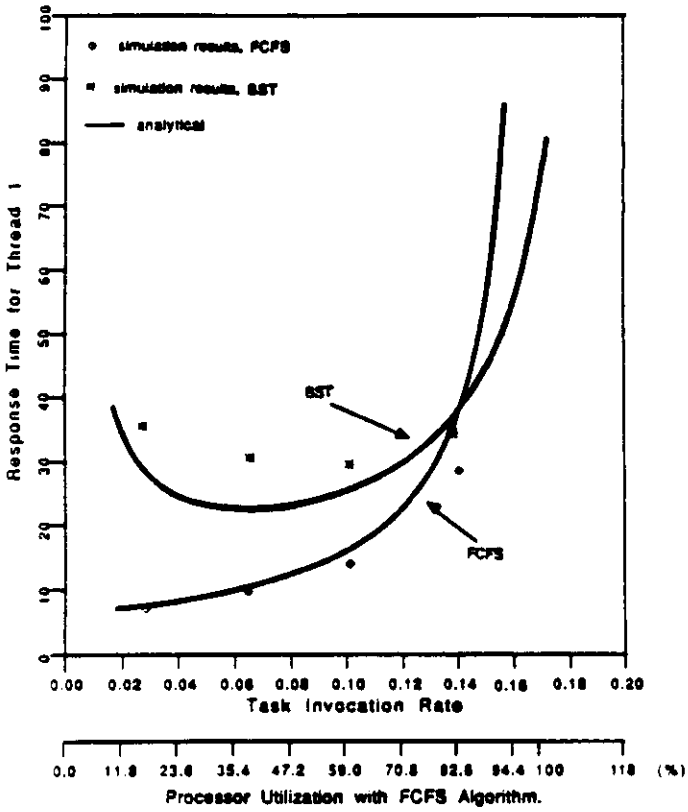


Figure 4. Average Response Time for Thread 1 of the Module Scheduling Example .

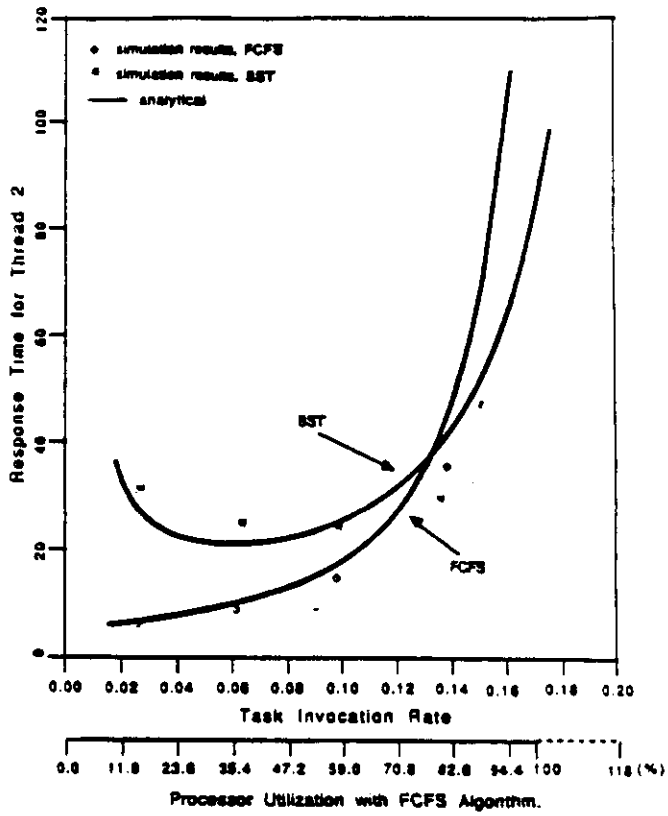


Figure 5. Average Response Time for Thread 2 of the Module Scheduling Example

4.2 Scheduling for Disk I/O

In a disk I/O access, the time to move the disk arm to the data track on the disk is the fixed overhead. The time to read the track data depends on the data size and is the variable overhead. The time required to forward the data block to the CPU is the execution time. Assuming the data to be retrieved in the batch are stored on the same or neighboring data tracks, then the fixed overhead can be shared. We shall show that using BST algorithm can substantially reduce disk I/O overhead. Let us use the following normalized system parameters :

Fixed Disk I/O Access Overhead = 1
Variable Disk I/O Overhead = 0.01
Mean Execution Time = 0.001

Figure 6 depicts the response time for the FCFS and the BST algorithms with selected values of M and T_o. We note that BST yields lower response time than FCFS for high disk access rate. Due to the time-out period delay in BST, FCFS yields lower response time than BST in the low invocation region. Because the BST algorithm substantially reduces fixed scheduling overhead, the I/O processor using BST can support a larger number of disk access than that of FCFS.

Next, we study the effect of batch size and time-out period on the response time. We note that at low disk I/O rate, larger maximum batch size and longer time-out period increase the waiting time at batch queue and thus the response time. At high disk I/O rate, the delay at the batch queue is reduced as the batch size increases. Therefore, as the maximum batch size and time-out constant increase, the response time increases at the low disk access rate region and decreases at the high disk access rate region.

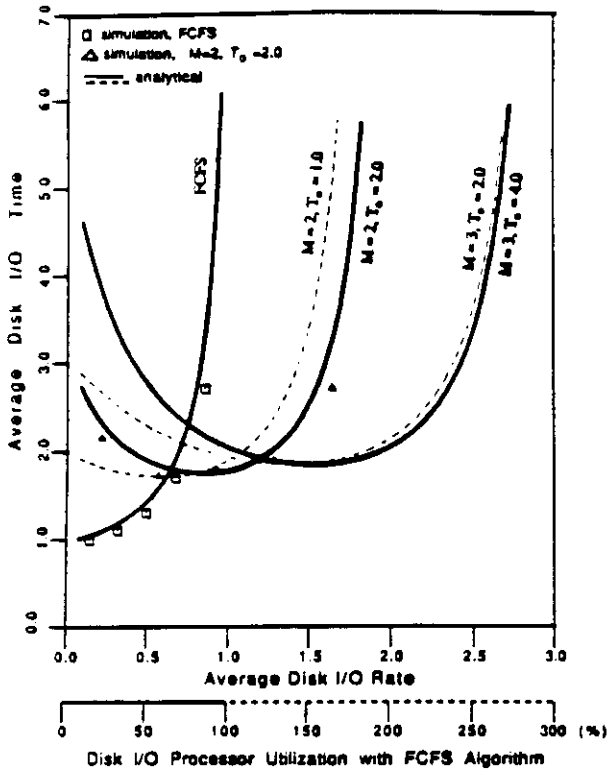


Figure 6. Average Disk I/O Time for FCFS and BST Algorithms with selected values of M and T_o

5. Conclusions

A new scheduling algorithm, Batch Service with Time-out (BST), is proposed for distributed processing systems for executing repeatedly invoked task(s). The analytical model developed provides module response time estimates for this algorithm and thus thread response time can be predicted. Comparing performance of a system using BST with that of using first-come first-served (FCFS) scheduling algorithm, we note that the amount of improvement depends on the ratio of the fixed scheduling overhead to the incremental scheduling overhead. At heavy invocation rates, more batches will be formed when using the BST algorithm, therefore fixed scheduling overhead is reduced and more response time improvement can be achieved. As a result of reduction in overhead, the system using BST provides more capacity than that of using FCFS. To optimize the performance of BST, it is desirable to use a larger M and T_o at heavy invocation rates and a smaller M and T_o at light invocation rates.

Appendix A - Probability Distribution of Module Batch Size,[1] P[Y=y]

When an invocation group arrives to a batch queue, the invocations in the group may be larger than the maximum batch size. As a result, some of the invocations in the group will be left behind at the batch queue and become the first invocation group of the next batch. For mathematical tractability in estimating the invocation waiting time at the batch queue, we assume that there are no *left-over invocations*. Although the resulting batch queue waiting times compare closely with simulation results, the left-over invocations have significant impact on the batch size. Therefore, we relax the above assumption in estimating the probabilities of the batch size distribution.

Let the number of left-over invocations from a given batch queue be α, and let the probability that the batch size equals y be P[Y=y].

$$P[Y=y] = \sum_{z=0}^{M-1} P[Y=y \mid \alpha=z]P[\alpha=z] \qquad (A.1)$$

where

$P[Y=y \mid \alpha=z]$ = probability that the batch size is y given that there are z left-over invocations from the previous batch,

$P[\alpha=z]$ = probability that there are z left-over invocations.

Since the number of invocations left behind from a batch depends on the number of invocations left behind by the previous batch, we have a recursive relationship as follows.

$$P[\alpha_r=u] = \sum_{v=0}^{M-1} P[\alpha_r=u \mid \alpha_{r-1}=v]P[\alpha_{r-1}=v] \qquad \text{for } u,v = 0,\ldots,M-1 \qquad (A.2)$$

Assuming the limiting probabilities, $\lim_{r\to\infty} P[\alpha_r=u]$ and $\lim_{r\to\infty} P[\alpha_{r-1}=v]$ exist and are equal to $P[\alpha=u]$ and $P[\alpha=v]$ respectively, then (A.2) becomes:

$$P[\alpha=u] = \sum_{v=0}^{M-1} P[\alpha_r=u \mid \alpha_{r-1}=v]P[\alpha=v] \qquad \text{for } u,v = 0,\ldots,M-1 \quad (A.3)$$

Conditioning on the number of groups in the r^{th} batch, N_r, we have

$$P[\alpha_r=u \mid \alpha_{r-1}=v] = \sum_{n_r=1}^{M-v} P[\alpha_r=u \mid \alpha_{r-1}=v, N_r=n_r]P[N_r=n_r \mid \alpha_{r-1}=v] \qquad \text{for } u,v = 0,\ldots,M-1. \quad (A.4)$$

where $P[\alpha_r=u \mid \alpha_{r-1}=v, N_r=n_r]$ and $P[N_r=n_r \mid \alpha_{r-1}=v]$ can be expressed in terms of λ_B, M and T_o. Thus $P[\alpha=u]$ for $u = 0,\ldots,M-1$ can be obtained by first substituting (A.4) into (A.3), then solving $\sum_{u=0}^{M-1} P[\alpha=u] = 1$ and the set of linear equations in (A.3).

In the same manner as in (A.4), the probabilities $P[Y=y \mid \alpha=z]$ for $z = 0,\ldots,M-1$ in (A.1) can be obtained by conditioning on the number of invocation groups in the batch. The resulting conditioned probabilities can be expressed in λ_B, M and T_o. The probability distribution of batch size, P[Y=y] can be estimated by substituting (A.4) and (A.3) into (A.1).

[1] Since the analysis is the same for all modules, we drop the subscirpt i in this Appendix.

Acknowledgement

The authors wish to think Dr. Kin K. Leung of AT&T Bell Laboratories for his critical reading of a draft of this paper.

References

[1] W. W. Chu, K. K. Leung, "Module Replication and Assignment for Real-Time Processing Systems," *Special Issue of the IEEE Proceedings on Distributed Database Systems*, May 1987, pp. 547-562.

[2] 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, pp. 225-236.

[3] W. W. Chu, "Buffer Behavior for Mixed Input Traffic and Single Constant Output Rate," *IEEE Transactions on Communications*, April 1972, pp. 230-235.

[4] L. Kleinrock, *Queueing Systems Volume I: Theory*, Wiley, New York, 1975.

[5] 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, pp. 248-260.

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

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

CHAPTER IV

TESTBED-BASED VALIDATION OF DESIGN TECHNIQUES FOR RELIABLE DISTRIBUTED REAL-TIME SYSTEMS

Testbed-Based Validation of Design Techniques for Reliable Distributed Real-Time Systems

WESLEY W. CHU, FELLOW, IEEE, K. H. KIM, SENIOR MEMBER, IEEE, AND WILLIAM C. McDONALD, MEMBER, IEEE

Invited Paper

Two tightly coupled multi-computer testbeds, one providing efficient inter-node communications tailored to the application, and the other providing more flexible full connectivity among processors and memories are used to support validation of the design techniques for distributed real-time systems. The testbeds are valuable tools for evaluating, analyzing, and studying the behavior of many algorithms for distributed systems. We have used the testbeds in studying distributed recovery block scheme for handling hardware and software faults. A testbed has also been used to analyze database locking techniques and a fault-tolerant locking protocol for recovery from faults that occur during updating of replicated copies of files in tightly coupled distributed systems. Testbeds can be configured to represent the operating environments and input scenarios more accurately than software simulation. Therefore, testbed-based evaluation provides more accurate results than simulation and yields greater insight into the characteristics and limitations of proposed concepts. This is an important advantage in the complex field of distributed real-time system design evaluation and validation. Therefore, testbed-based experimentation is an effective approach to validate system concepts and design techniques for distributed systems for real-time applications.

I. INTRODUCTION

The complexity and sophistication of the real-time data processing problems encountered in computer-based weapons systems severely tax all aspects of advanced data processing technology due to requirements in reliability, availability, cost, performance, and growth. Furthermore, data processing solutions are required for a wide range of system concepts and operational environments. Conventional techniques such as pipelining and cache memory have significantly improved computer performance. Advances in circuit technology have increased processor capacity. However, current and projected needs still represent significant challenges.

In real-time systems, such as those for ballistic missile defense, the data processing problem is dominated by the necessity of meeting response times while achieving total system throughput requirements. Response times may be as low as a few milliseconds with throughput requirements ranging up to hundreds of millions of instructions per second. Frequently, the data processing system must also remain dormant for months, activate on minutes' notice, and operate unattended with ultra-reliability for periods ranging from a few hours to several days.

Distributed computing promises to satisfy the requirements of these systems by utilizing moderately priced contemporary hardware in networks. To achieve this promise, research and development are being pursued in all aspects of real-time distributed computing technology. Many techniques have been proposed for achieving reliability through redundancy, detecting and recovering from errors, distributing and managing shared data, communicating reliably between processes, allocating and scheduling resources in the presence of failures and overloads, and achieving high throughput through architectural innovation. These techniques must be proven experimentally before they can be used in a real-time system. Individually and collectively they impose overheads that create problems in satisfying real-time requirements. As a result, solutions to the problems of real-time distributed computing must be proven in a realistic environment for the specific real-time application. This approach, although initially more expensive, is in the long run cost-effective because of the availability of low-cost microcomputers that can be used to develop flexible multi-microcomputer systems.

In this paper, two tightly coupled multi-computer testbeds (Section II) are presented, and their use in evaluating fault-tolerant software and database management techniques for real-time systems is discussed. The fault-tol-

Manuscript received December 30, 1985; revised January 8, 1987. This work was supported by the U.S. Army under Contracts DASG60-85-C-0059, DASG60-85-C-0061, DASG60-84-C-0115, and DASG60-82-C-0019.

W. W. Chu is with the Computer Science Department, University of California at Los Angeles, Los Angeles, CA 90024, USA.

K. H. Kim is with the Computer Engineering Program at the Department of Electrical Engineering, University of California, Irvine, CA 92717, USA.

W. C. McDonald is with UNISYS Corp., Huntsville, AL 35805, USA.

IEEE Log Number 8714297.

erant software and database management experiments were performed for radar-tracking application tasks. Testbed experience has proven the feasibility of the distributed recovery block (DRB) scheme and database locking techniques. Further, experimental results provide insight into the behavior of these algorithms and allow selection of the appropriate system parameters for optimizing the system performance. Performance of the DRB scheme in terms of overhead and response time is presented in Section III. Experiment results on database management, lock granularity, and fault-tolerant locking (FTL) showing performance in terms of overhead, loc ontention rate, and response time are presented in S on IV. Section V discusses how the DRB and FTL schemes can be incorporated together into a system that supports reliable real-time operation.

II. Testbeds for Real-Time Distributed Systems

This section describes two flexible distributed system testbeds that have been established to support the development, analysis, test, evaluation, and validation of research in distributed computing for real-time applications.

A. Tightly Coupled Network (TCN)

The establishment of tightly coupled network (TCN) [1] facilities at the University of South Florida (relocated to the University of California, Irvine, 1987) was motivated by the desire to address in detail applications with extremely stringent time constraints. Therefore, efficient inter-node communication was an important design requirement. A Z8001-based single-board microcomputer, henceforth called the OEM-Z8000, was used as the primary building block of the TCN hardware facilities. The amount of on-board RAM available on an OEM-Z8000 currently ranges from 32 to 120 kbytes. The interconnection approach adopted is based on the use of a two-port buffer memory as a medium for connecting a pair of OEM-Z8000s. The access time of the two-port buffer memory developed in house is the same as that of the on-board memory of the OEM-Z8000. A sufficient number of these buffer memory modules were constructed to configure a variety of network topologies involving six microcomputer nodes.

Fig. 1 shows an example of the TCN configurations. The example configuration consists of six nodes linked with ten two-port buffer memories. Each node in the network is housed on a small backplane card and comprises an OEM-Z8000 microcomputer, local memory extension boards, and two-port buffer memory boards. The OEM-Z8000 is equipped with an interval timer and two serial I/O ports. The timer generates 1200 interrupts per second. It is used to construct a software implemented real-time clock in the TCN.

The decision to use the two-port buffer memory as a connection medium was based on the following objectives: 1) to match the network structure closely with the computation structure of a chosen real-time application, and 2) to minimize the set of components shared among processing nodes. It was thought that the fewer shared components the network contained, the easier it would be to prevent undesirable interference among processing nodes; it would then be easier to assess and contain the damages caused by faults.

To support efficient development of distributed application software to run on the TCN, a number of software tools were established. A major effort was devoted to the establishment of a virtual machine (VM) called the Extended Concurrent Pascal Machine (ECPM) on each node. ECPM is a combination of a software nucleus and node hardware. It supports concurrent programs consisting of asynchronous processes communicating through monitors [2]. ECPM also contains a VM code interpreter. A compiler translates a concurrent program written in the Extended Concurrent Pascal language into a VM code. The VM code is executed by ECPM after being loaded on the OEM-Z8000.

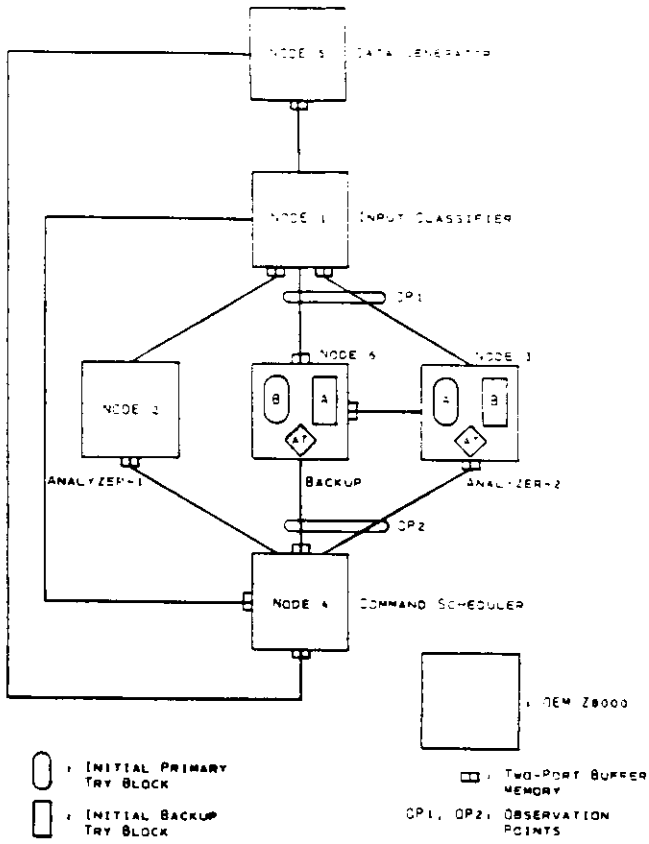


Fig. 1. Network configuration used at USF for experimentation with DRB.

The distributed operating system containing the ECPM and running on the TCN hardware was established. It has an extensible structure for incorporating various fault handling capabilities as well as deadline-driven scheduling strategies [3]. A real-time application program running on the TCN was also developed. (The abstract structure is depicted in Fig. 1.) The real-time application program, the distributed operating system, and the TCN hardware together formed the core of the testbed supporting experimentation with various fault tolerance schemes.

Other software tools established include a variety of concurrent programming language processors, recovery block (RB) translators, inter-high-level-language translators, graphic displays of real-time application status, and performance measurement programs.

B. The Crossbar Multi-Microcomputer System (CMS)

The CMS [4]–[8] was developed to support evaluation and analysis of distributed software technologies for real-time

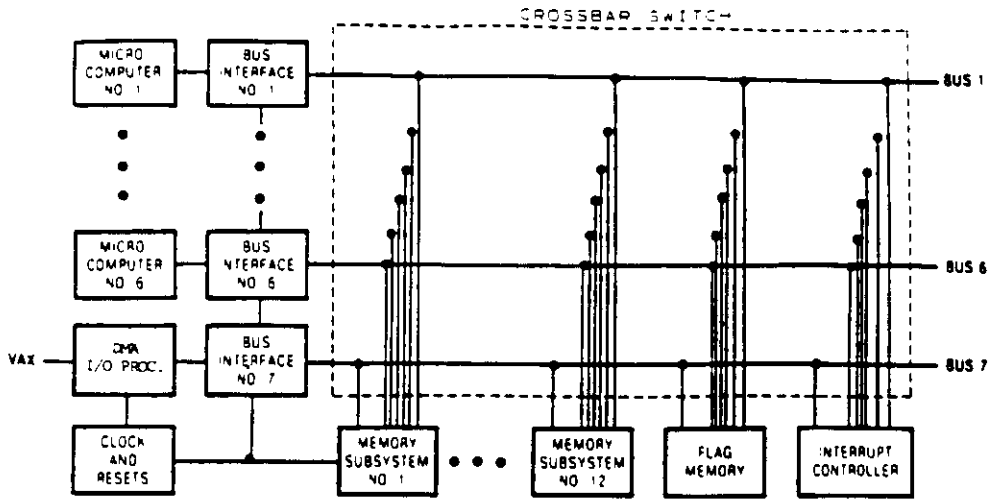


Fig. 2. Crossbar multi-microcomputer system.

systems. The system is designed for flexibility to support a range of fundamental architectures and for maximum interconnectivity. Both shared-data and message-based constructs can be investigated. It is composed of six computers (OEM-Z8000) interconnected to 12 shared memory modules through a crossbar switch as illustrated in Fig. 2. The fully parallel crossbar switch transfers the normal Z8001 memory access signals (address, data, control) directly to the shared memory. Such an interconnection avoids bus conflict. Thus the performance obtained from the testbed is independent of bus contention. Each shared memory module is assigned to a separate 64-kbyte memory space segment in the Z8001 address space, and shared memory access is by normal Z8001 memory read/write operations. Memory-mapped access to special functions, such as a real-time clock, interprocessor interrupt facilities, and synchronization flag memory, is provided in the same manner. The real-time clock provides a common 31-bit, 500-kHz time source accessible by all processors without contention for performance evaluation. Interprocessor interrupt facilities enable any processor to interrupt any other processor and provide the basic facilities needed to implement message based communication. The flag memory provides 16 384 1-bit flags that can be used for synchronizing access to shared memory. The system is hosted by a VAX 11/780 that provides the user interface, supports experiment development and analysis, and controls experiment execution. The VAX 11/780 also provides mass storage for the experiment files.

The crossbar system is designed using low-power Schottky TTL logic components. The entire system is driven from a single 6-MHz clock to simplify design and test. Each microcomputer has 120 kbytes of local RAM plus ROM located in segments zero and one, and can access any location in shared memory at any time. Each processor is able to access individual shared memory modules without contention on the crossbar switch. Each shared memory module includes an arbitration unit that implements a first-come first-served policy for resolution of conflicting memory requests. A fixed priority scheme resolves simultaneous requests. Instructions may be executed from either shared or local memory.

Each microcomputer and the VAX 11/780 have associated interrupt registers. A processor is interrupted when another processor writes data to the interrupt register of the former. The interrupted processor can then read the data. Contention logic to resolve simultaneous interrupt requests is implemented in microsequencer firmware. Any processor can interrupt itself or read the contents of any interrupt register. Each processor also has an interrupting interval timer that can be used for future event scheduling or for setting watchdog timers.

Flag memory provides synchronization mechanisms for access to shared data. Each read request to a flag implements a true test and set operation. The value of the flag is returned to the processor and the flag is set or cleared based on the least significant bit of the flag address.

The VAX 11/780 has access to the crossbar switch through a general-purpose Unibus interface that supports five types of transfers: shared memory access, interrupt request, read real-time clock, flag memory operations, and microprocessor reset. An I/O processor supports message transfers, file operations, and full access to the facilities of the testbed. RS-232 links between VAX and the microcomputers supports diagnostics, debugging, and data collection.

The operating system support for experimentation on this distributed system testbed is supplied by three major components—the master operating system, MOS; the application operating system, AOS; and the kernel operating system, KOS [9]. MOS running on the VAX provides global testbed control, performing activities that require a global view of the system. It supports interactive access to the multi-microcomputer testbeds through the VAX, and provides services for developing, configuring, loading, and executing experiments. AOS provides execution-time resource control, performing activities that require knowledge of an experimenter's specific needs. It is application-specific and controls the experimenter's virtual architecture and application processes. KOS provides low-level direct hardware control, performing activities requiring knowledge of the processor's local state. Services include task loading, scheduling, and control; local and shared memory management; synchronization lock management; Interrupt and message handling; and event scheduling. KOS, which resides in each microcomputer and includes low-level debug aids, coordinates all local activities and

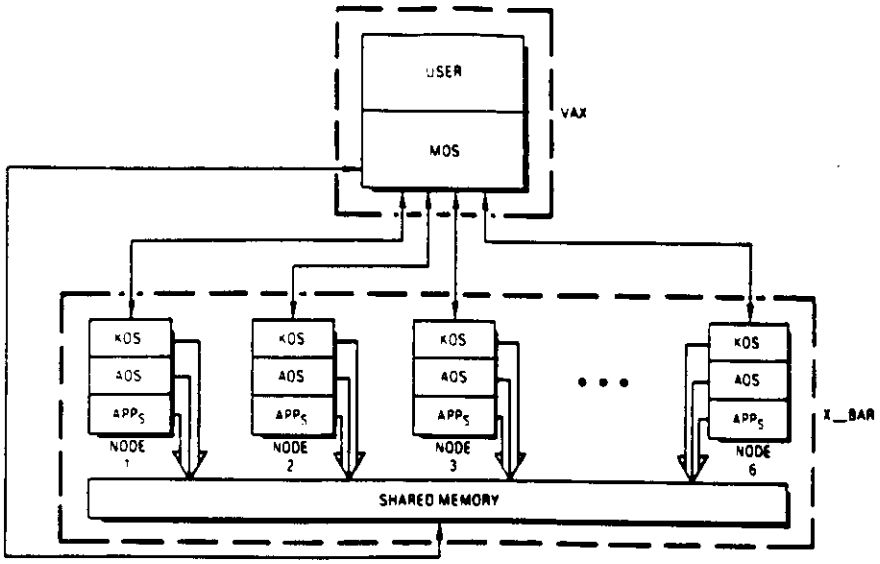


Fig. 3. Crossbar multi-microprocessor system software configuration.

provides services to AOS and MOS. Fig. 3 illustrates the relationship of these OS components and the application program on the CMS.

An abstract simulator of a radar-tracking application, which is a distributed real-time program, has been developed that forms the basis for the experimental application. The simulation includes complete task control structures and data structures for the application. Synthetic workloads and functional performance models are provided for each application task to accurately model algorithm loading and performance. Experiments are executed in scaled real-time according to the ratio of the target processor performance to the testbed computer performance. In this way, timing of the sequence of events is maintained and results can be scaled to the target system.

Other software tools that reside on the host VAX 11/780 include a Pascal-based concurrent programming language facility for real-time applications and system programming, a Prolog Requirements Builder for specifying experiment requirements in Prolog form, a Design Builder for structuring the distributed system design, a configuration manager for maintaining the experiment software configuration, and performance measurement programs for data gathering, reduction, and analysis [9], [10].

III. Distributed Recovery Block Scheme and its Experimental Evaluations

A. Basic Principles of the Distributed Recovery Block (DRB) Scheme

The DRB (distributed recovery block) scheme is a technique for unified treatment of both hardware and software faults with minimal execution overhead. It provides efficient forward recovery in contrast to more time-consuming backward recovery such as rollback-and-retry. It is based on a combination of both the distributed processing and the recovery block structuring concepts. Recovery block (RB) [11], [12] is a language construct supporting the incorporation of program redundancy into a fault-tolerant program in a concise and easily readable form. The syntax of RB is as follows:

ensure *T*
by *B*1
else by *B*2
. . .
else by *Bn*
else error.

In the above description, *T* denotes the *acceptance test (AT)*, *B*1 the *primary try block*, and *Bk*, $2 \leq k \leq n$, the *alternate try blocks*. All the try blocks are designed to produce the same or similar computational results. The acceptance test is a logical expression representing the criterion for determining the acceptability of the execution results of the try blocks. A *try* (i.e., execution of a try block) is thus always followed by an acceptance test. In a sense, RB is an enclosure of some recoverable activities of a process.

The DRB scheme exploits concurrent execution of try blocks to facilitate fast forward recovery. The scheme also utilizes both hardware and software components efficiently to maximize the lifetime of computer systems. In [13], three different DRB schemes were explored and the conclusion was that a scheme called *C* was the most promising. Therefore, in this paper, only scheme *C* is discussed.

The real-time distributed computer systems considered in our study are assumed to have the following characteristics:

1) A computer system consists of multiple computing stations, each executing one and only one RB.

2) The result produced from a computing station may become an input to another computing station or to the application environment.

3) A computing station may consist of one or more computing nodes. Multiple computing nodes within a computing station can be used either in a load-sharing or in a redundant processing mode.

For simplicity, only two try blocks in an RB, the primary and the backup, are used to illustrate the DRB scheme. The specification of the maximum execution time allowed for each try block is an integral part of the DRB scheme. The

allowable execution time specifications of try blocks are used to set watchdog timers with the objective of ensuring timely completion of tries. A try not completed within the time limit due to hardware faults or excessive looping is treated as a failure. Therefore, the acceptance test can be viewed as a combination of both logic and time acceptance tests.

The DRB scheme realized with two nodes is depicted in Fig. 4. Both primary and backup nodes contain the same

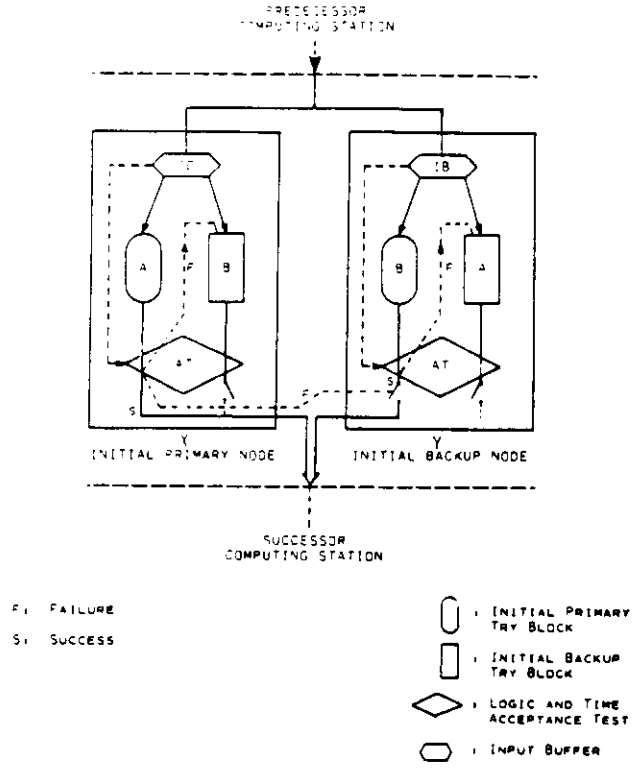


Fig. 4. Basic DRB configuration.

acceptance test, consisting of logic and time tests, and the same set of try blocks, *A* and *B*. However, the roles of the two try blocks are assigned differently in the two nodes. Primary node *X* uses try block *A* as the primary try block initially, whereas backup node *Y* uses try block *B* as the initial primary. Therefore, until a fault is detected, both nodes receive the same input data, process the data by use of two different try blocks (i.e., block *A* on node *X* and block *B* on node *Y*), and check the results by use of the acceptance test. Both nodes perform all these tasks concurrently. The time acceptance test is used to ensure timely behavior of both nodes.

In a fault-free situation, both nodes will pass the acceptance test with the results computed with their primary try blocks. In such a case, the primary node notifies the backup of its success in the acceptance test. Thereafter, only the primary node sends its output to the successor computing station. However, if the primary node fails and the backup node passes its test, the backup node assumes the role of the primary, i.e., the nodes exchange their roles. To be more specific, the primary node attempts to inform the backup node upon its failure in passing the acceptance test. The backup node will take over the role of the primary as soon as it receives notice. If the primary node is completely lost, the backup node will recognize the failure of the primary upon expiration of the preset time limit. It will then become the new primary. Since these interactions between two nodes are done asynchronously, the scheme does not suffer from synchronization overhead. On the other hand, if the backup node fails first, the primary node need not be disturbed. In both cases, the failed node attempts to serve as a backup node; it attempts to roll back and retry with its alternative try block to bring its application computation state or local database up-to-date. This attempt does not disturb the primary node.

Under the DRB scheme, the recovery time is minimal because the maximum concurrency is exploited in the redundant try block execution. The scheme uses forward recovery and does not use any special-purpose software components tailored to handling particular types of faults. Fast forward recovery is achieved regardless of whether faults occur in the hardware or software components.

In the case where a computing station uses *n* nodes for load sharing (i.e., using multiprocessing to obtain a higher throughput than a single processor), a straightforward application of the DRB scheme would be to group *n* computing nodes into *n*/2 primary-backup node-pairs. This should be used when the system application requires the fastest possible recovery from failures. Such an arrangement reduces the throughput potential to one half of what is achievable with an irredundant operation of nodes. In the applications where rollback-and-retry is acceptable, the arrangement described above is regarded as wasteful. A more cost-effective approach is to connect the nodes loosely through queues containing data sets and allow each node to dynamically select its next task among the primary try block, the alternate try block, and the acceptance test. Fig. 5 presents such an approach. A data set is defined as a set of data that communicates between computing stations and activates an execution of a processing algorithm. In this scheme, each node may select its next job from any of the four queues, Input Data Queue (IDQ), Try Result Queue (TRQ), Arrival Time Record Queue (ATRQ), and Retry Data Queue (RDQ). We shall now discuss the operations performed with the data set picked up from each of these queues.

1) Input Data Queue (IDQ): A node that selected a data set from this queue executes the primary try block (*A*) with the data set and deposits the result (together with the original input data) into the Try Results Queue.

2) Try Results Queue (TRQ): A node that selected this try result data set first removes a record in the Arrival Time Record Queue (ATRQ) which corresponds to the selected result data set. The node then executes the logic acceptance test. If the result is acceptable, an irrevocable update of the computing station database is carried out with the result. The result is then moved into the Validated Results Queue to be picked up by the successor computing station. If not acceptable, the data set is moved to the Retry Data Queue.

3) Arrival Time Record Queue (ATRQ): Each data set in this queue is an arrival time record containing both the arrival time and the maximum expected processing time of the corresponding data set in the Input Data Queue. This information is used in determining if a timing fault has occurred. In other words, a node which picked an arrival time record checks if the record is too old. If so, the node

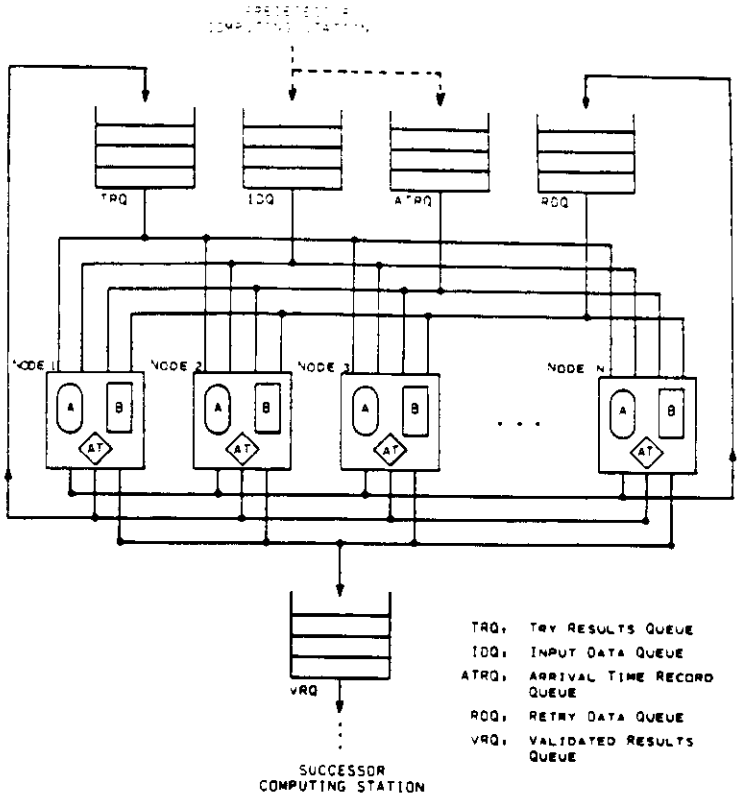


Fig. 5. DRB with load balancing.

moves the corresponding data set from the Input Data Queue to the Retry Data Queue.

4) Retry Data Queue (RDQ): A node that picked a data set from this queue executes the alternate try block *B* and deposits the result into the Try Results Queue.

As an illustration of the operation of this load-sharing multi-node computing station, assume that a data set, say *D*, has just been produced by the predecessor computing station. The data set *D* is entered into IDQ and at the same time its arrival time record is entered into ATRQ. Now assume that node 3 has been idle and looking for work. When node 3 checks IDQ, it discovers data set *D* and picks it up. Node 3 then executes the primary try block *A* with data set *D* and deposits the processed result *D*1 into TRQ. Suppose that node 2 is idle at this time and it soon discovers the result data set *D*1 in TRQ. Node 2 then executes the acceptance test with *D*1 and the original input data set *D*. Suppose that the result was a failure. Node 2 then moves *D* and the corresponding arrival time record into RDQ. Another idle node, say node 1, will soon discover *D* in RDQ, executes the alternate try block *B* with *D*, and deposits the result *D*2 into TRQ. Node 3 is idle at this time and it soon discovers *D*2 in TRQ. Node 3 thus executes the acceptance test with *D*2 and *D*. Now the result is a success and node 3 updates the computing station database with *D*2. Node 3 also removes *D* (and the corresponding arrival time record) from RDQ and moves *D*2 into Validated Results Queue (VRQ) for pickup by the successor computing station.

One drawback of this scheme is that an "insane" node can disrupt a significant portion of the network, thereby causing a significant performance degradation. For example, it may either get stuck to a queue or repeatedly pick up new data sets and produce unacceptable results. Or it may continue to pick up data sets from the Try Results Queue and reject them even if they are good. However, it may be possible to implement the scheme in such a way that the probability of an "insane" node causing significant disturbance becomes very small. This is a subject worthy of further study. There are also other ways of combining the DRB and load balancing schemes.

B. Experimental Evaluation with the TCN

To test the execution efficiency of the DRB scheme, two experimental implementations and measurements have been done, one on TCN and the other on the CMS.

We shall first describe the experimentation performed on TCN. The network configuration used is shown in Fig. 1. The DRB scheme was incorporated into node 3 (Analyzer-2) and node 6. The application program is written in Extended Concurrent Pascal [1]. Node 5 (Data generator) simulates a real-time device which generates stimulus data to the rest of the network and accepts the response (command). The remaining five data processing nodes execute *input* (i.e., stimulus data) *classification process*, various *analysis processes* constituting the intelligence of the solution algorithm, and a *control command scheduler process* that delivers the network's response to the real-time device. The stimulus data from node 5 are first handled by the input classification process which distributes inputs to the rest of the network. The command scheduler honors requests from various analysis processes to schedule commands for the real-time device.

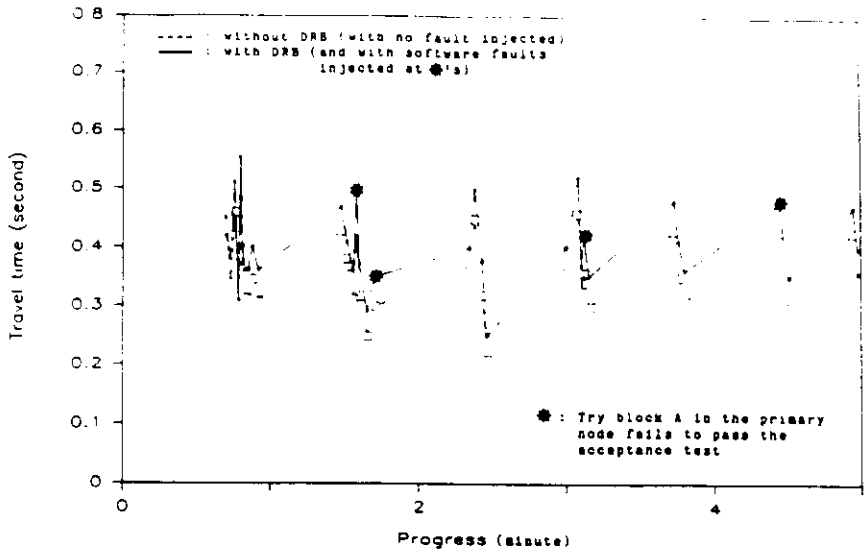


Fig. 6. Data travel time measured.

In the actual implementation of the DRB scheme, specific choices had to be made for various parameters. The most important parameter was the node reconfiguration strategy (i.e., reassignment of the roles to nodes and try blocks upon detection of failure). After extensive analysis, the following strategy was adopted: *the current primary node always uses A as the primary try block and B as the backup try block whereas the current backup node uses try blocks in the reverse order.* Therefore, once the primary node fails to produce an acceptable result, the roles of the primary and backup nodes are reversed, as well as the roles of the primary and backup try blocks in both nodes. If the backup node fails to produce an acceptable result, it merely retries with try block *A* and then returns to try block *B* for subsequent use as its primary. This strategy is attractive for two reasons. First, two nodes always execute different try blocks. Secondly, the current primary node always uses *A* as the primary try block; try block *A* is generally designed to produce the same or better quality output than try block *B*. If try block *A* has a residual design error that repeatedly manifests itself, it is possible to have a frequent exchange of (primary and backup) roles between the two nodes. However, this probability is very small if the DRB scheme is properly used.

The travel time of data set passing through a computing station was measured to determine the execution overhead caused by the introduction of the DRB scheme into the network. As a part of facilitating this measurement, "observation points" were established in the network. When a data set arrives at the designated observation point in the network, the node stamps the real time and saves a copy of the time-stamped data in its local memory. When enough measured data are obtained, the time-stamped data are transferred to another computer system for data analysis. The observation points are usually established at the points where the nodes are ready to send messages to the successor nodes and also at the points where the nodes have received messages.

In this experiment, two observation points were set up in the network. Fig. 1 shows these points established in the network. Observation point 1 (OP1) is set up where the primary and backup nodes have taken the data set from the queue buffer connected to the predecessor nodes. Observation point (OP2) is set up where both nodes are ready to put the data set into the queue buffer connected to the successor nodes.

During experimentation, faults were injected to examine their impacts on system performance. The types of faults studied include: 1) total node failure (simulated by node reset); 2) transient hardware faults (e.g., transient faults of main memory); and 3) software faults such as infinite looping, arithmetic overflow, etc. The DRB incorporated into nodes 3 and 6 in Fig. 1 was written in Extended Concurrent Pascal and executed on an OEM-Z8000 microcomputer with a clock rate of 4 MHz.

The DRB overhead consists of interprocess communication among nodes and the execution of the acceptance test. Fig. 6 shows such overhead for incorporating the DRB scheme into the network. The solid curve represents the delay between OP1 and OP2 in the case of using the DRB whereas the broken curve represents the delay in the case without the DRB. The gap between the solid and broken curves is the execution time increase due to the incorporation of the DRB. The average execution time increase is approximately 30 ms. Moreover, the solid curve in Fig. 6 also shows instances (marked by stars) where arithmetic overflows occur in the primary node and the fast recovery capabilities of the DRB scheme are exercised. We noted that the fault occurrences and subsequent recovery actions did not cause any visible degradation of the system performance. In the absence of fault, the execution time increase is caused mainly by the execution of the acceptance test and the communication of the acceptance test success to the backup node.

Considering the inefficient implementation language (Extended Concurrent Pascal), and the slow processor (4-MHz Z8001) used, the amount of execution time increase shown in Fig. 6 is at least 20 times higher than that expected in the systems built with current off-the-shelf hardware and software tools. For example, use of a processor running at

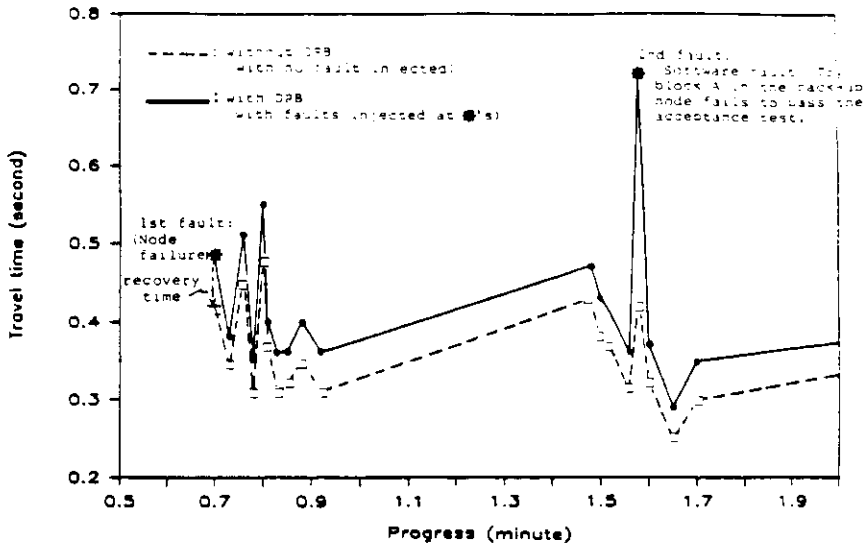


Fig. 7. Data travel time measured.

20 MHz will result in speedup by a factor of 5. Use of a more efficient language (an Assembly language in the extreme case) will result in additional speedup by a factor of 4.

Fig. 7 shows the case where the primary node is reset, resulting in the permanent loss of the node. Later an arithmetic overflow occurs in the remaining node. The recovery from the first fault (the loss of the primary node) took about 60 ms. This recovery time is largely a function of the timeout period used in the DRB. When the second fault (the arithmetic overflow) occurred in the remaining node after the permanent loss of the first node, the node had no choice but to roll back and retry with try block *B*. Therefore, the

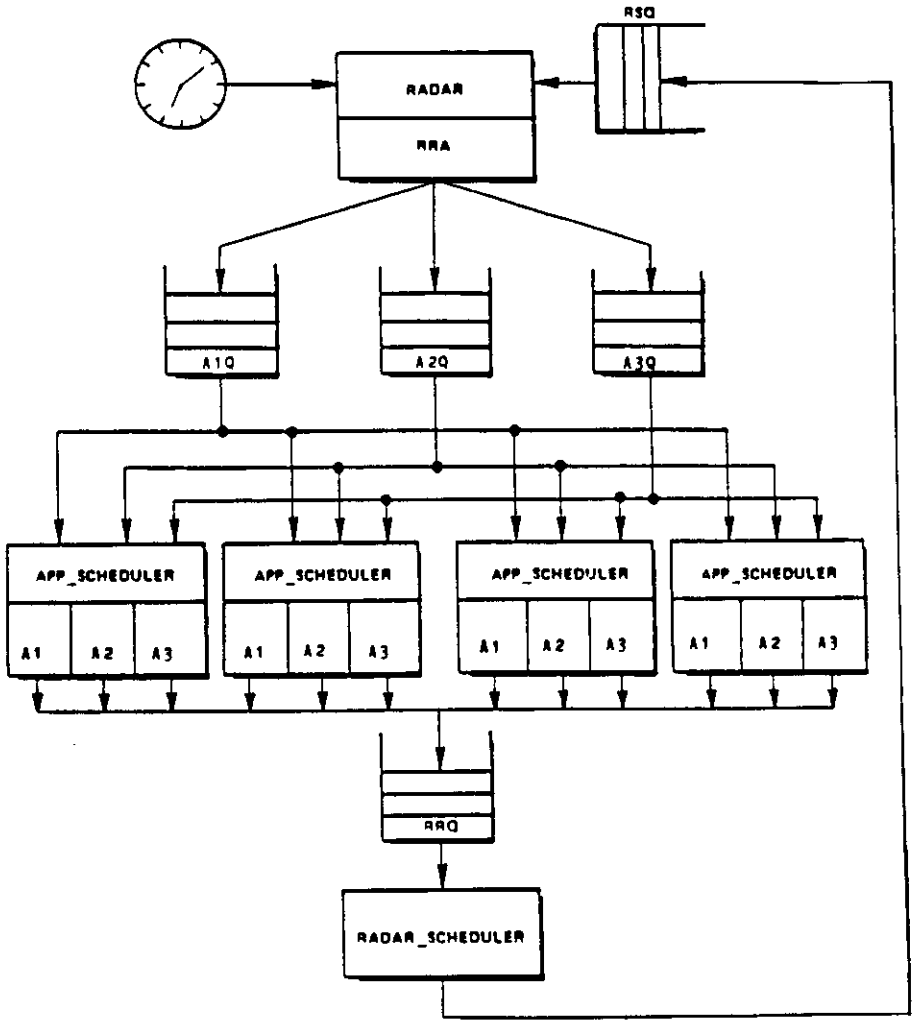


Fig. 8. Baseline network configuration for radar control.

recovery time was very high, i.e., about 290 ms, as shown in the figure. Again, the recovery time can be easily reduced by a factor of 20 by implementing real application systems with current off-the-shelf tools.

C. Experimental Evaluation with the CMS

The primary objectives of the experiment with the CMS [14] were: 1) to establish the initial feasibility of real-time recovery from hardware failures, failures of an algorithm to produce reasonable results, and failures of task completions; 2) to measure the impact of the recovery mechanisms on processing resource utilization and response times, and 3) to demonstrate that the DRB approach can be applied to real-time application processes and distributed operating systems. The software was written in PDL (an extension of Pascal supporting concurrent programming). Unlike the experiment conducted on TCN, this experiment dealt with a load-sharing multinode computing station.

Fig. 8 depicts the distributed real-time system adopted as the baseline configuration for this experimental study. The system is a distributed implementation of a closed-loop radar control system. One processor is dedicated to simulation of a radar and assimilation of data returned from the radar (RRA), four processors to a set of three analysis processes ($A1$, $A2$, $A3$), and a sixth processor to the radar scheduling process. The software architecture makes use of the multiported shared memory modules of the CMS. A shared database is maintained in shared memory and contains data on the objects tracked by the radar.

The Application Program scheduler (APP scheduler) in each of the four processors schedules analysis processes for execution. It polls the three Input Data Queues ($A1Q$, $A2Q$, $A3Q$), which contain radar return data sets, in a round-robin fashion looking for work. When an entry is found, the scheduler activates the appropriate analysis process to work on the data set. After processing the data set, an analysis process places a tracking request for a tracking pulse in the radar request queue (RRQ) connected to the radar scheduler (RS). The radar scheduler honors such requests by placing them in appropriate slots within the radar schedule queue (RSQ) connected to the radar.

The system clock is accessible to all processors as a time base. Every processor in the CMS is a Z8001 processor capable of performing about 0.350 MIPS (million instructions per second). The target processor should be much more powerful. Therefore, the radar control simulation is run at a much slower rate. The scale factor is based on the ratio of the target machine instruction execution rate to the Z8001 instruction execution rate. This time scaling approach enabled us to evaluate the time cost of DRB in an application context very close to the real operating environment.

Fig. 9 depicts the fault-tolerant distributed system configuration with the DRB scheme incorporated. RB was incorporated only for the analysis process $A3$. The process AT determines if the result computed by $A3$ is within reasonable bounds based on flight dynamics. $BA3$ is a backup independently coded analysis process.

All data sets produced by the radar return assimilator and other processes are kept in the shared database. In fact, data sets never really enter any of the queues (Fig. 9); only the pointers to the data sets enter the queues. For example, the radar return assimilation process places a pointer to a data set into $A3Q$. If an APP scheduler picks the pointer and activates the analysis process $A3$, then $A3$ makes a copy of the data set pointed to by the pointer for its processing and places its result into TRQ. Later, a certain APP scheduler picks this result from TRQ and executes the acceptance test (AT). If the result is acceptable, an update of the database with the accepted result follows. If the result is not acceptable, the AT places a pointer to the original (unprocessed)

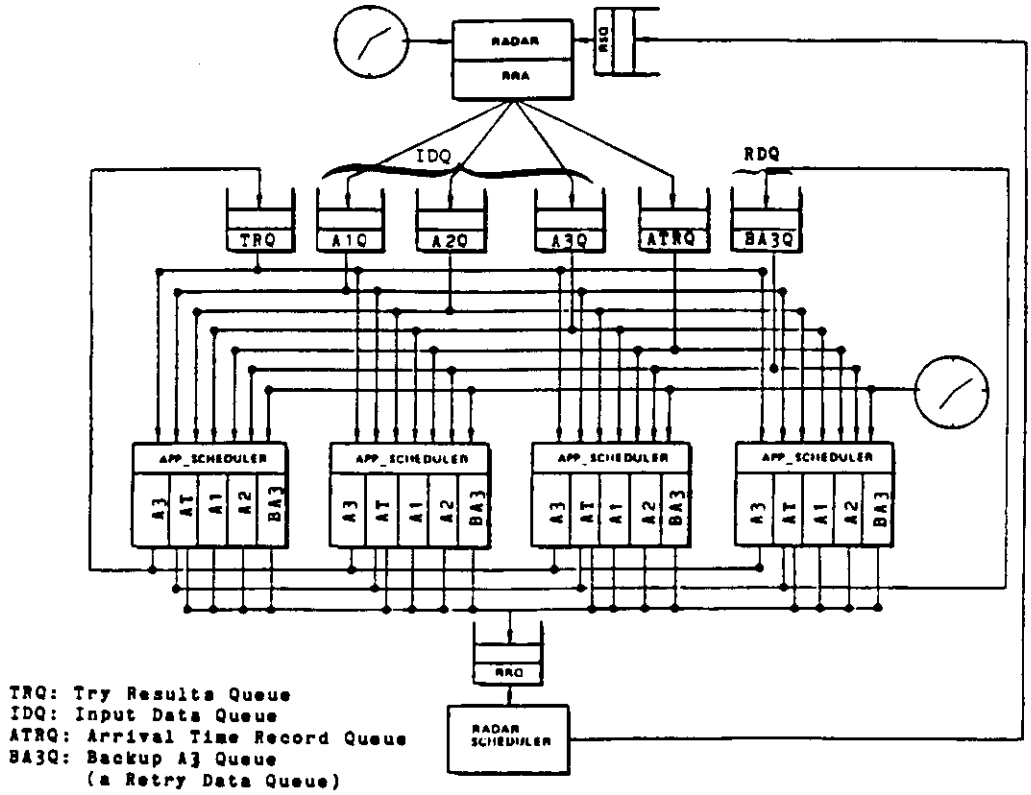


Fig. 9. Fault-tolerant network configuration used at CMS for experimentation with DRB.

data set into RDQ, thereby causing the backup analysis process *BA3* to process the data set again. Also, when a pointer to a data set is originally entered into *A3Q*, a deadline for the analysis process *A3* to process the data set is placed in ATRQ. ATRQ is processed by the APP scheduler to detect data sets for which the processing by *A3* is overdue. A deadline of 30 ms was used in this experiment to ensure that no false alarms (reporting fault detection when there are no faults) would occur.

This fault-tolerant network configuration (Fig. 9) was compared with the baseline network configuration (Fig. 8) by running it against the same radar load and measuring a number of performance parameters. Node failures were injected several times during the analysis *A3*, but each time a successful recovery was accomplished for all cases.

Track response time is the time from the entry of a data set into *A3Q* to the insertion of a corresponding radar request into RSQ by the radar scheduling process. It was used as a measure of real-time computer system effectiveness in this study.

Fig. 10 is a track response time histogram. The track response times are increased by inclusion of the DRBs because the task sequence in the fault-tolerant configuration includes the acceptance test (AT) and because of the lengthened polling loop of the APP scheduler. The mean track response time (shown in the figure by a vertical line) rises from 1.76 to 2.6 ms which is still considerably below the allowable maximum, 40 ms, for this particular application. As mentioned earlier, these numbers represent the performance expected in the real systems built with the tools and components required by the applications.

Fig. 11 presents maximum track response times for the data sets processed by *A1*, *A2*, and *A3* in the fault-tolerant network configuration. The figure shows the maximum track response times over every 50-ms interval during the experiment. The large spike of 32 ms of *A3* is caused by recovery from an injected processor failure in a computing node. For false alarm control, the timeout value for ATRQ was set at 30 ms. Seven additional small spikes are visible for injected algorithm errors detected by the acceptance test function.

IV. Database Locking Schemes and their Experimental Evaluation [15]

In a distributed processing system, application tasks residing at several processors often require sharing common information files. As a result, read/write conflicts of the common file may occur. Locking is commonly used for maintaining data consistency in tightly coupled distributed systems.

The performance of different concurrency control techniques used for a given set of application tasks depends on the data access pattern, their invocation rates, and the system operating environment. Because of the complex nature of the problem, we often have to evaluate the cost/performance of various techniques via experimentation. In this section, we shall present testbed methods for experimental evaluation and selection of the appropriate locking protocol for a given set of real-time radar tracking application tasks.

The database for the radar tracking application consists primarily of a dynamic object track file. The size of the track file varies as new objects are acquired, tracked, and subsequently dropped as they pass through the engagement space. The track file is composed of 512-byte records (one for each object in track) containing object state information. There are two types of records: early tracks and precision tracks. Each record in the file is updated periodically at the track rate (normally 20 Hz). Typical profiles of early and precision track sizes (number of records) are shown in Fig. 12(a) and (b) for the database locking experiments.

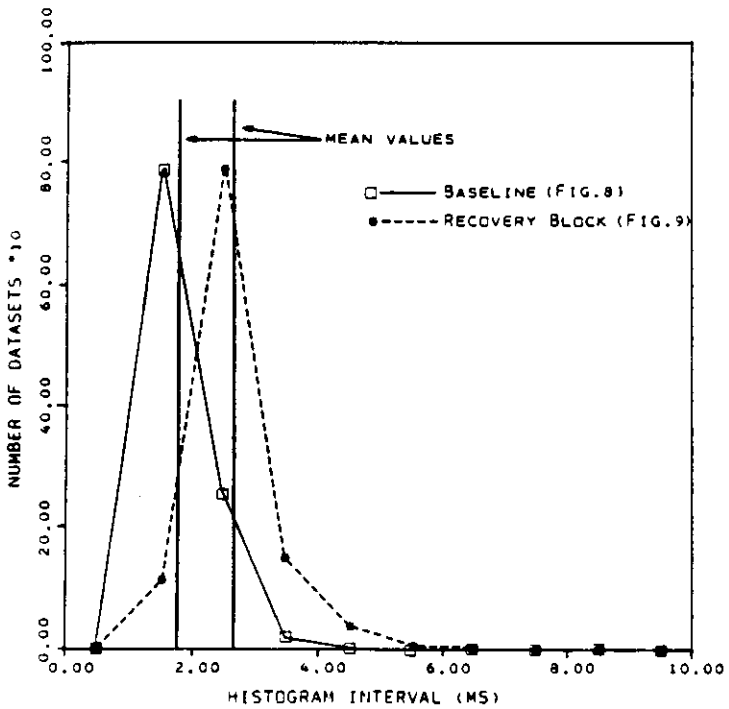


Fig. 10. Track response time histogram.

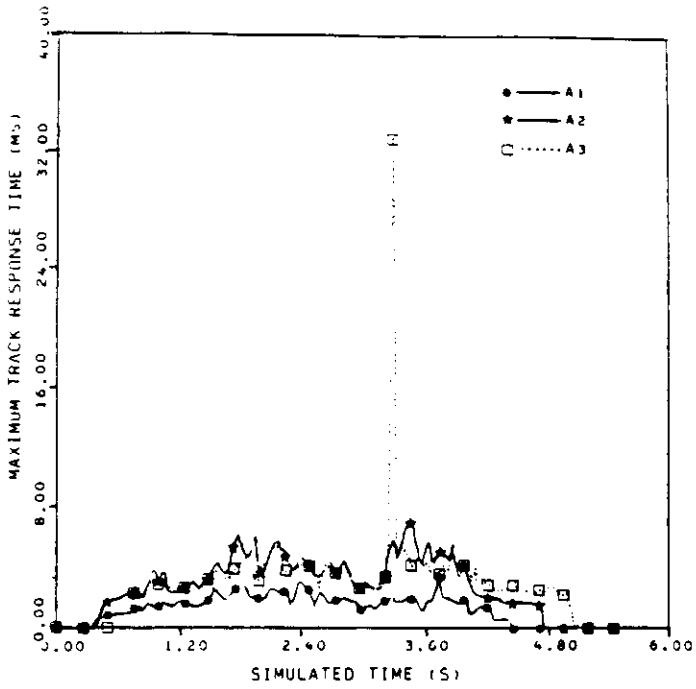


Fig. 11. Maximum track response times.

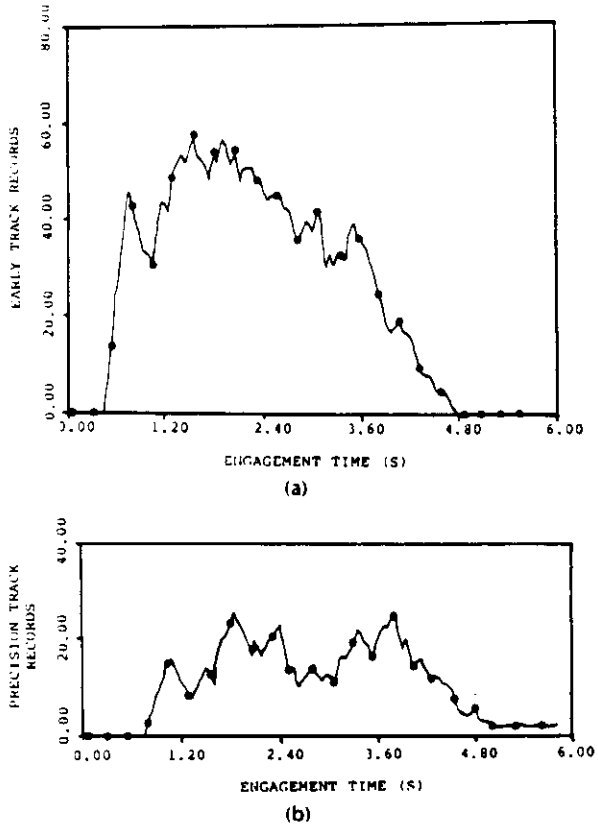


Fig. 12. Track file profile. (a) Early track profile. (b) Precision track profile.

A. Locking Techniques

To manage shared data accesses, we introduce several database locking techniques. In all cases, the locking unit is referred to as a data item. A data item is locked prior to its being accessed for read or write. Once an accessing task has completed all its references to the data item, the data item is unlocked.

Although exclusive access is required to write a data item, it is possible to have a single writing function execute concurrently with multiple reads. This is accomplished by the writing function using the following technique:

1) Reserve the data item (which permits concurrent reads but no exclusive access or other reserves).
2) Make local copy of the data item.
3) Read and write the local copy.
4) Upgrade the reserve to exclusive access (requires that all readers finish).
5) Write the shared copy of the data item.
6) Unlock the data item.

This approach is referred to as *reserve upgrade locking*.

The simple locking protocol, on the other hand, guarantees sole access of a processor to a data item by specifying the data item as *USED* or *NOTUSED*. The reserve-upgrade locking protocol establishes data availability based on the type of access a processor is attempting. Three data item access modes are defined: write (W) access, read-for-update (RU) access, and read (R) access. Write access must be exclusive; no W, RU, or R permission is granted to a task in another processor when W permission is held on the data item. Read-for-update access may be granted if no other RU access is held on the data item. RU access allows any read access currently held to complete but does not allow any new reads to initiate. Read permission to a data item can be granted to as many tasks/processors as necessary, if a W or RU is not active on that data item.

Lock contention rate is defined as the number of iterations required to gain access to the record locking mechanism per LOCK PROCEDURE call per measurement period (50 ms was used for these experiments). The performance of the data locking techniques is influenced by two factors: lock contention for the data item and the CPU time required

by the file access routines to execute the locking protocol and the time for tasks holding the lock(s).

The reserve-upgrade locking scheme is primarily designed to reduce contention for data by allowing simultaneous reads and reads with intent to update on the same data item. This more complex file access method requires more calls to the file access procedure and more instructions within the procedure to provide data consistency. There is a tradeoff between reduction in contention and increase in CPU utilization, which is a function of the overall file activity and file access patterns of the application problem.

1) Experiments to Evaluate Locking Techniques: We shall present testbed experiments to explore the choice of locking granularity and the selection of lock protocol. Experiments were performed with the CMS. Response time and processor utilization are used as performance measures. Data were collected every 50 ms of the simulation time.

In these experiments, the application task resided at six processors that require read and write of the common object track files. Each Track File has a record for each object being processed. Since read/write conflict can occur among these application tasks, concurrency control is required.

a) Simple file locking: Simple file locking locks the entire track file when any type of access (read, write) is made. This is accomplished by making a call when access to the file is desired; then the correct number of instructions are executed using SYNTHETIC-LOAD to simulate the appropriate action taken on the file. Finally, a RELEASE call is made to free the Track File. The obvious problem associated with this experiment is that no two processors/tasks can work on the Track File at the same time. In fact, this experiment would not successfully run because of massive file contention problems. For this reason the simple file lock experiment was rejected. No useful data were collected.

b) Reserve-upgrade file locking: This approach locks the entire Track File using the W, RU, and R operations. To access the file, a call is made to a LOCK-PROCEDURE where control remains until access to the lock is granted. This technique has the advantage of allowing concurrent readers to access the Track File. Contention can occur only when a write or a read-for-update access is requested to the file that is being used by the other processor.

c) Reserve-upgrade record locking: This experiment is the same as experiment in item b) except individual task records were locked rather than files. To access a record, a call to a LOCK-PROCEDURE is made with an index into the Track File passed as a parameter, control remains there until access to the Track File record is granted. This technique has the advantage of allowing concurrent readers access to individual records, thus reducing contention to a minimum. Contention can occur only when a write or a read-for-update access to the current record is being used.

d) Simple record locking: The final experiment involved locking individual Track-initiated records and track records on a USED/NOTUSED basis. No two processors/tasks could gain access to the same record regardless of the type of access desired. To gain access to a record, a call to a lock procedure is made with an index into the Track File passed as a parameter. Contention occurred only when two processors/tasks required access to the same record.

2) Discussion of Locking Experiment Results:

a) File versus record locking: Both experiments b) and c) used the read, read-for-update, and write locks to minimize contention. A load near the point where maximum track response times are reached (full system load) was used as input. This corresponds to the Track File profile shown in Fig. 12(a) and (b). Experimental results reveal that the file locking implementation shows much larger lock contention rate than the record locking (Fig. 13). Because of the time spent in the contention, the file locking routine causes substantially greater CPU utilization. As a result, the maximum Track response time for the reserve-upgrade record locking is better than that of file locking (Fig. 14).

b) Reserve record locking versus simple record locking: Experiments c) and d) were run at the same load levels. The results show that the reserve-upgrade locking yields slightly less contention than that of the simple locking case (Fig. 15).

The reserve-upgrade record locking technique uses slightly more CPU resource throughout the engagement, as shown in Fig. 16. This is due to more subroutine calls to service the reserve-upgrade protocol and more executed instructions in its lock/grant loop. The read-for-update (RU) access is a precursor to the write (W) access for each update (radar return) in the track processing task, one to set RU, one to set W, and a third to clear RU and W. Only two file access subroutines are required for the simple record locking protocol. The experimental results reveal that the reserve-upgrade record lock enters the file access subroutine approximately 22 percent more often than the simple record locking technique.

In this testbed application task, because of its low lock contention rate, the reserve locking protocol did not contribute significantly to the time spent in the lock test/grant loop and also did not provide any response time improvement over the simple locking protocol (Fig. 14).

3) Conclusion of the Locking Experiments: Implementing the lock mechanisms at the file level induced too much contention for the application tasks. Record locking reduced contention significantly. For the tested applications, the lock contention is at such a low level that the extra processing load required by the reserve-upgrade locking scheme is not compensated by the contention improvements over the simple locking protocol. However, in a higher contention environment, the response time improvement from the reduction in contention provided by the reserve-upgrade lock protocol could very well outweigh the increase in CPU utilization.

B. Fault-Tolerant Locking

In a tightly coupled distributed processing system, multiple copies of shared files are maintained in different shared memory modules to provide high survivability. To assure *mutual consistency* among the copies, data updates should be applied to all file copies. However, if a processor fails during an update process, some file copies may have been updated while others have not, resulting in mutual inconsistency. To recover from this type of failure and assure the transaction is atomic [16], we propose to use the *Fault-Tolerant Locking (FTL)* [17] protocol that is installed on top of the conventional consistency-control protocols. FTL detects a processor failure, identifies and recovers inconsistent file copies, and releases the file lock so that other processors may lock and use the file again. FTL also prevents proces-

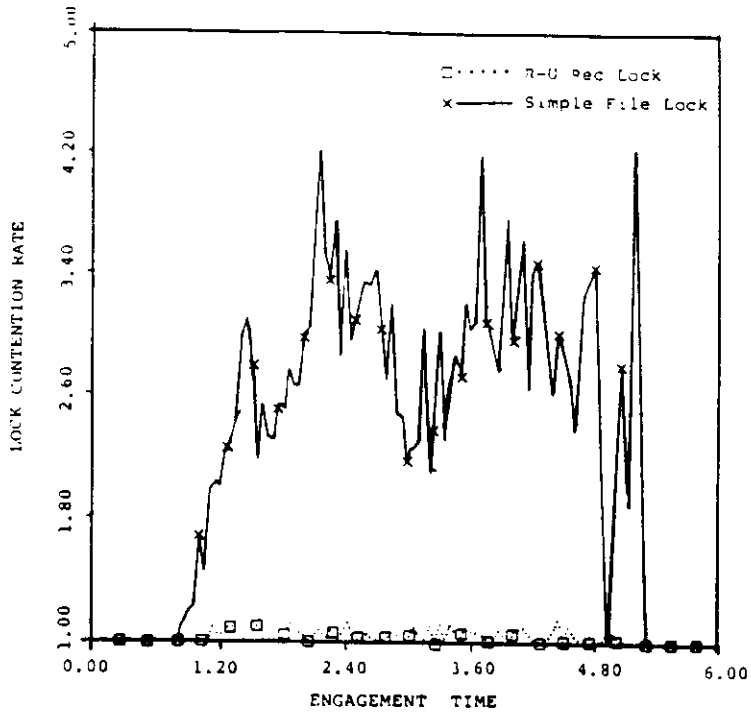


Fig. 13. Lock contention rate.

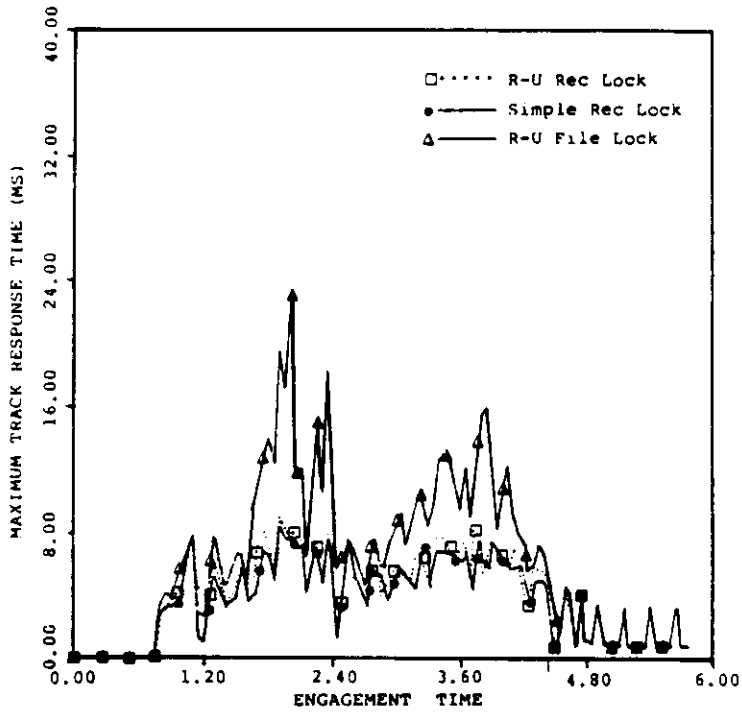


Fig. 14. Testbed measured maximum track response time for selected locking protocols.

sors from reading and updating out-of-date file copies in case of shared memory modules and/or path failures.

1) Concept: To provide the status of a file, a word that indicates the current state (free, locked, update-initiated, or failed) of that file as well as the accessible state (accessible, inaccessible) of the replicated file copies of each file copy files that are stored in the shared memory modules. The replicated file copies are updated one at a time according to a predefined sequence. In this manner, if a processor fails[1] during a file update, the update status (*completely* updated, *partially* updated, or *un-updated*) of all the copies of a file can be identified. Based on this status of the copies

[1]A failed processor is also assumed unable to issue any memory reference.

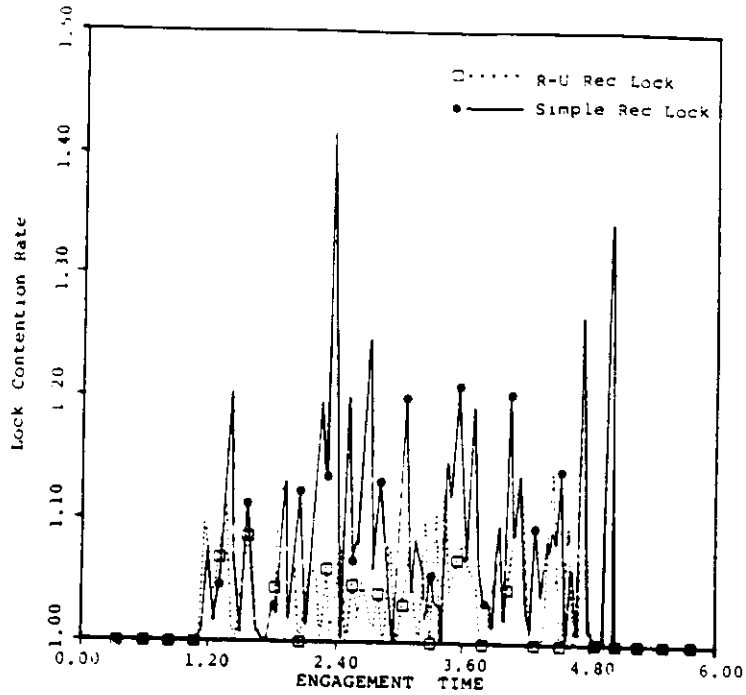


Fig. 15. Lock contention rate for record locking.

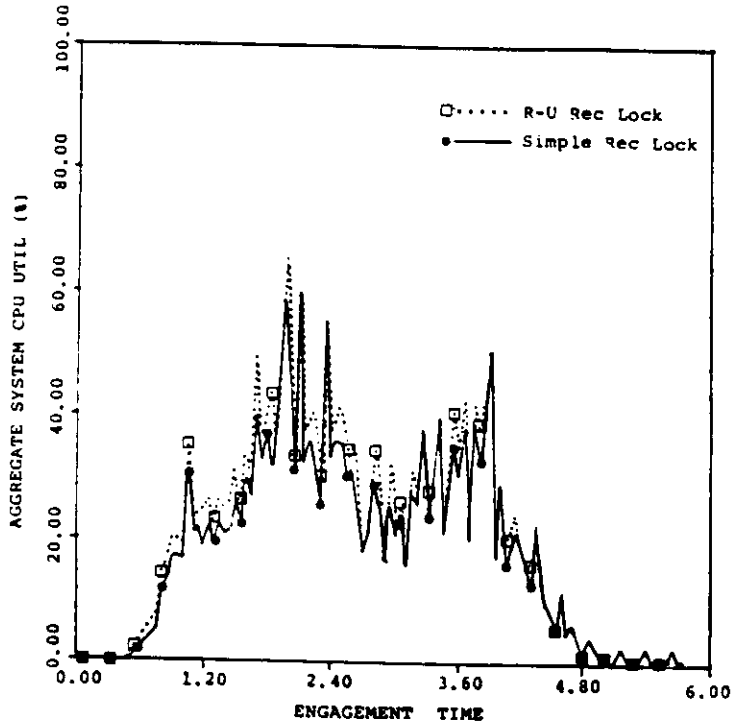


Fig. 16. Aggregate CPU usage for record locking experiment.

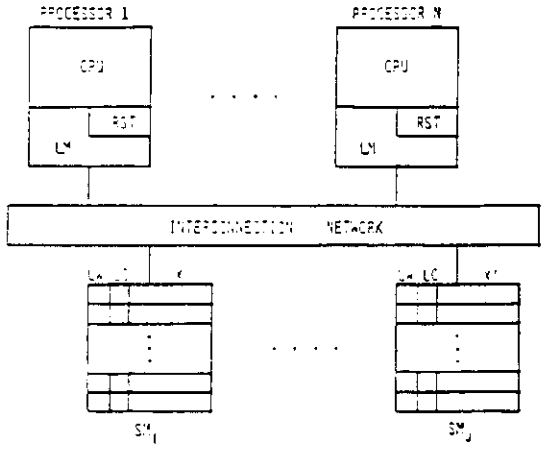
of a file, the inconsistent copy can be detected and recovered from any of the consistent copies.

When a processor that holds the lock fails, attempts from any other processors to lock this file will, of course, be unsuccessful. Such a processor failure will be detected after a prespecified unsuccessful number of repeated lock attempts (timeout) by the other processors. To prohibit further accesses to failed copies, each processor maintains a file copy *status table* in its local memory. When a processor experiences a memory and/or path failure while accessing a file copy, it marks the file inaccessible on its *status table*.

2) FTL Operations:

a) Implementation: To implement the FTL in a tightly coupled system, shared records are duplicated in different

shared memory modules (Fig. 17). Each processor maintains a *record status table* (RST) in its local memory which indicates the status (accessible or inaccessible) of each record copy, or segment, in the shared memory modules. Each rec-



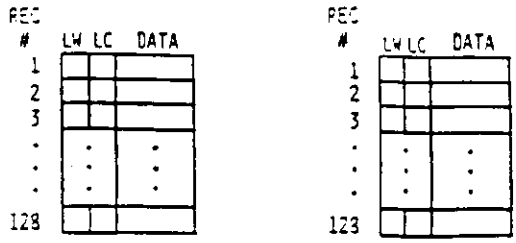
LM LOCAL MEMORY
SM SHARED MEMORY
X' SHADOW COPY OF RECORD X
LW LOCK WORD (FOUR POSSIBLE STATES)
LC LOCK COUNTER
RST RECORD STATUS TABLE

Fig. 17. A tightly coupled distributed system with FTL.

ord copy, or segment, has a *lock word* (*LW*) that indicates one of the four possible states of the copy (free, locked, update-initiated, or failed) and the states of the other copies (accessible or inaccessible) (Fig. 18). The status of the copy assures data consistency in case of path failure. Each processor is required to read the *lock word* before accessing the record copy and mark on its record *status table* if either of the record copies is inaccessible.

To simplify our discussion, we assume each file has two copies: a *primary* copy and a *shadow* copy. Before accessing a record copy,[2] a processor first checks the RST to determine if the copy is accessible. Then it reads the *LW* of the record copy. If the *LW* of the copy indicates "failed," the processor marks "failed" on the RST and tries the other copy. If the requested copy is being locked or update-initiated (by some other processor), the processor repeatedly checks the *LW* until the copy becomes free. When the processor finds the copy is free, it locks the copy and repeats the process for the second copy. Then, it prepares updates in its local memory. When all updates to the record are ready, the processor marks "update initiated" on the *LW* of the first copy and performs the update. After completing the update onto the second record copy in the same manner, the processor releases the lock for both copies. The FTL update procedures for normal operations with no failure and no lock contention are shown in Fig. 19.

[2]For simplicity in our discussion, we assumed the record as a unit of data items for locking and recovery. However, to reduce overhead, a group of records (segment) may be used as a unit of data item for locking and recovery.

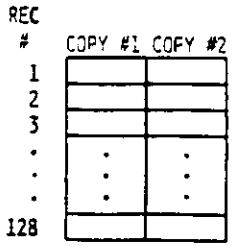


LW = LOCK WORD

LC = LOCK COUNTER

Each lock word indicates:
1) One of four states: *Free, Locked, Update initiated,* or *Failed.*
2) Status of the other record copies *(accessible or inaccessible).*

(a)



Each entry indicates one of two states: *Accessible* or *Inaccessible.*

(b)

Fig. 18. Duplicated records and record status table. (a) Duplicated records in shared memory modules. (b) Record status table (RST) in each processor.

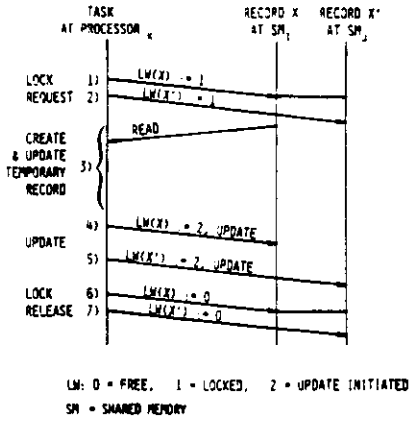


Fig. 19. FTL protocol for record update.

b) Detection of processor failure: When two or more processors request to lock the same record simultaneously, only one of them will obtain the lock grant. Other processors might experience timeout and initiate the recovery process which is undesirable. To prevent this from occurring, a *Lock Counter* (*LC*) is introduced for each record copy. After a processor successfully locks a record copy, the *LC* of the copy is incremented by one. When a record copy is currently locked, a processor trying to lock the same copy will repeatedly request to lock the copy until it succeeds. When the processor finds that the *LC* of the requested record copy has been incremented while waiting for a lock

grant, this implies that the record has been released by its holding processor and locked by some other processor again. The processor resets the *timeout counter* and continues requesting for the lock-grant. If the *LC* remains unchanged after a predetermined number of lock requests (i.e., timeout period), the processor currently holding the lock is considered failed. The processor that detects the timeout then increments the *LC* of the record copy by one. This prevents other processors from detecting the same failure. To prevent *false* failure detection, the timeout period (determined by the number of repeated lock requests) must be larger than the lock-holding time of any application program.

c) Recovery from a processor failure: The processor reads the *LW*s of all copies of the requested record when it detects a timeout for a record. Based on the *LW status table* the processor takes the appropriate recovery action: either discarding the inconsistent record copy and operating in a degraded mode, or copying from the consistent record copy into the inconsistent one.

d) Handling of a shared memory failure: When a processor detects a memory failure by hardware detection or diagnostic program, one approach is to notify all the other processors of the failure. Whenever the processor accesses a record copy in a shared memory, it needs to check message boxes which requires large processing overhead. Further, two or more processors may detect the same memory failure independently and may receive duplicate failure messages. Therefore, such a message passing technique requires high overhead.

To avoid such message passing overhead, we propose the following technique for handling memory failure. When a processor requests a record from a shared memory module and detects a memory failure, it marks "inaccessible" on its local RST (without notifying the other processors). It also marks "inaccessible" on the *LW* of the record copy if the *LW* is still accessible. When a second processor finds that the *LW* on its requested record copy is marked "inaccessible," this second processor marks the "inaccessible" of that record in its RST. If the *LW* of that record copy is inaccessible, then this record copy cannot be accessed by any processors which assures data consistency.

e) Handling of a path failure: A single point failure in the crossbar network or in the multiple bus system may prevent one or more processors from accessing a particular memory module.[3] This will prevent record updating in that memory module. However, the records may be accessed by other processors that are not blocked by that single point path failure, and cause data inconsistency. This can be avoided by maintaining the status of the other record copy in the *LW* of each record copy. When a processor detects a record is inaccessible, it marks "inaccessible" for that record on its RST and on the *LW* of the accessible copy. When another processor accesses the accessible copy, the *LW* will reflect the inaccessibility of the other copy. The processor should then mark that information on its RST to avoid further accesses of that inaccessible copy of that record.

[3]Assuming the failure does not cause network partitioning; that is, two processors having access to only one of the two copies, but not the same one.

C. FTL Experiments

In this section, we shall present results of experiments to characterize the behavior of the FTL protocol. The experiments for the FTL protocol were implemented on the CMS Testbed. The locking used by the application program is on the record level. This section describes the implementation of the protocol on the testbed and experimental results.

The Track File was duplicated and placed in shared-memory modules of the testbed. When a task needs to modify a record in a shared-memory, it first locks the record copies in sequence to avoid deadlock situations. A local copy is then created, and a synthetic load is executed simulating an update to the local copy. After the update is completed, the primary record copy is marked update-initiated and the local copy written to it. If a task cannot lock both copies of a record in the allotted time, the processor holding the lock is considered to have failed and a recovery procedure is initiated on the record. The allotted time to lock the record copies is measured by iterations of unsuccessful lock attempts in the lock procedure. When a predetermined number of iterations are attempted, the processor attempting to lock the record transfers control to a recovery procedure that examines the lock states from the *LW* of the two record copies to determine if they are consistent. If inconsistency exists, the lock states are used to determine which copy is inconsistent so the consistent copy may be written over the inconsistent copy. The lock states of both record copies are set to "free" and the processor returns to the thread of processing.

1) Experimental Results: A set of experiments was performed for evaluating the feasibility of the FTL protocol via the CMS Testbed. Three experiments characterize the FTL protocol under no-processor-failure situation, in terms of 1) overhead of the FTL protocol, 2) choice of lock-request retry period, and 3) choice of time-out period for processor failure detection. Another experiment studies the FTL protocol recovery time in the presence of processor failures.

a) Overhead of the FTL protocol: Concurrency control with the FTL protocol requires additional lock and update to the second copy of each record. Therefore, it requires more processing resources than without FTL (baseline system). For the particular application, we note from Fig. 20 that FTL requires 24.2 percent more CPU time of which 7.8 percent is used to lock the second record copy, and 16.4 percent is used to execute the FTL code. As a result, the lock-holding time of the FTL system is longer than the baseline system. Because of larger processor utilization for the system using FTL, the response times are also increased. From Fig. 21 we note that for this application, using FTL increases the Track Thread response time by 19.7 percent.

b) Lock-request retry period: When a processor fails to obtain a lock grant for a record, it retries repeatedly until it receives a grant or reaches a time-out (i.e., detects a processor failure). If the period between retries (retry period) is too short, the number of shared memory conflicts increases. If the retry period is too long, the processor may be waiting for a lock even though the record is "free." For the given application example a delay loop is inserted in the lock procedure. The retry period can be controlled by varying the number of the loop iterations. Each loop (retry) iteration is about 0.95 μs. Fig. 22 displays the lock-grant time as a function of the retry period. The lock-grant time varies

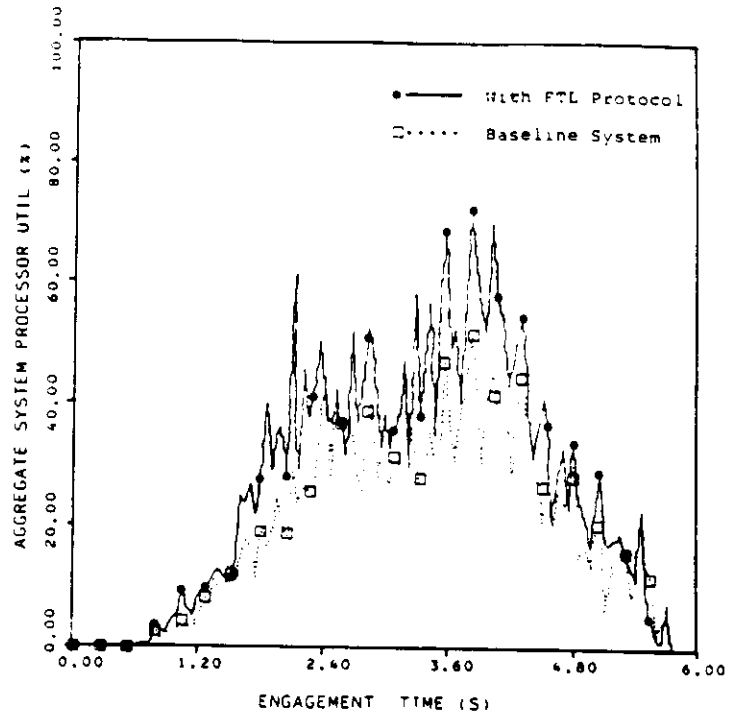


Fig. 20. Aggregate system processor utilization.

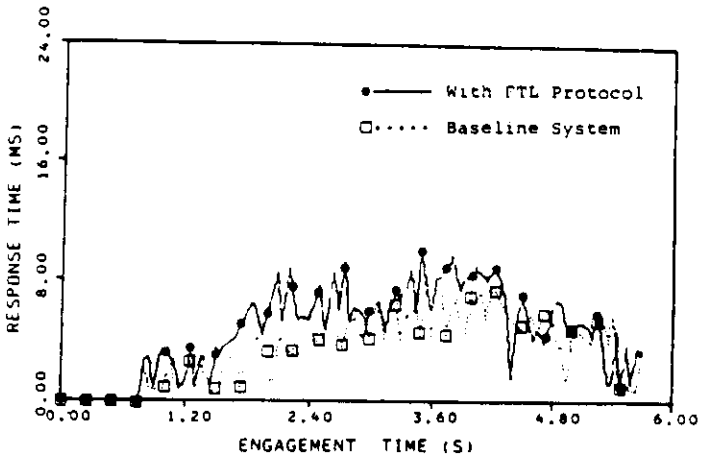


Fig. 21. Maximum response time for the Track Thread.

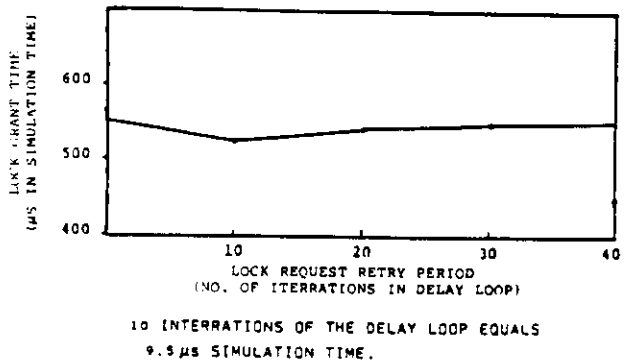


Fig. 22. Lock-request retry period versus lock-grant time.

with the level of lock conflict and memory conflict which depends on the number of locked records and the number of memory modules in the system. Futher, since reduces lock-retry period increases lock conflict, lock-grant time also depends on the lock-retry period [18]. Due to the fact that the application task has a very low lock conflict rate, the lock-grant time is rather insensitive to the retry period. A slightly lower average lock-grant time occurred at ten loop iterations (9.5-μs simulated time).

c) Time-out period for processor failure detection: The time-out period for detecting a processor failure during a record update should be longer than the maximum lock-holding time for any task. The time-out period is measured as the maximum number of lock requests for a record. If the time-out period is too long, a processor would issue unnecessary requests for a lock that is held by a failed processor. On the other hand, if the time-out period is too short, the processor would initiate unnecessary recovery processes. Again, the time-out period is implemented by executing loop iterations. Each iteration runs for about 54 μs.

For the application example, the experiment shows that 13 iterations (corresponding to 650 μs) is the lowest number that yields no false time-out detection as shown in Fig. 23.

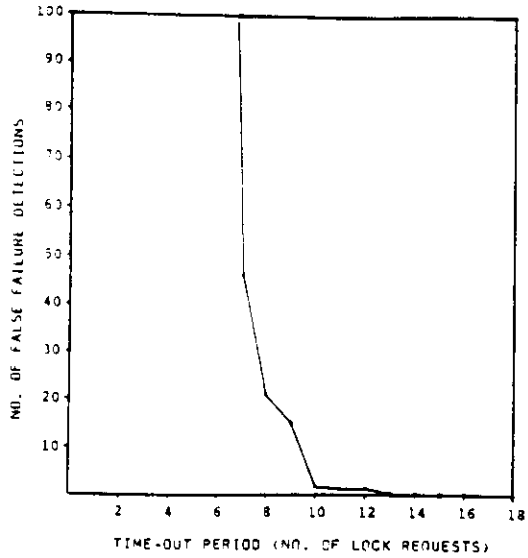


Fig. 23. Timeout period versus false failure detection.

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

D. Conclusion of the FTL Experiments

Experimental results reveal that the FTL is capable of detecting a processor failure during update and recovering from data inconsistency between record copies. The overhead for performing the fault-tolerant locking protocol depends on the lock frequency and its application. The parameters that may affect system performance are: time-out period, lock granularity (record or a group of records), and lock protocol (e.g., exclusive lock for write and shared for read, or reserve, upgrade, or exclusive lock).

V. The Interrelationship Between DRB and FTL Schemes in Real-Time Tightly Coupled Systems

The DRB scheme enhances the probability of correctly executing the data manipulation, whereas the FTL scheme enhances the probability of maintaining database consistency in volatile environments. Therefore, DRB and FTL play a complementary role in real-time tightly coupled systems. For example, the DRB scheme may be applied to compute a new value for a database item. In this case, the DRB scheme enables a real-time process to produce a correct value for the database item despite hardware and software failures. On the other hand, the FTL scheme maintains database consistency during the retrieval and updating of the item with a new value produced under the DRB scheme in spite of storage module failures or lockup of the database item by a failed processor.

VI. Conclusion

In this paper, we have used the multi-computer testbeds to demonstrate the feasibility of the distributed recovery block (DRB) scheme for handling hardware and software faults while meeting the real-time response requirements imposed by the radar tracking application. Testbed experiments also reveal that to maintain data consistency, applying a lock on the record level yields far less contention than on the file level for the radar tracking application. Furthermore, experiments demonstrate good performance of a new fault-tolerant locking (FTL) protocol in handling failure during data update in tightly coupled distributed systems with replicated copies of files in shared memory.

The CMS has proven to be an effective tool for evaluating distributed software technologies for real-time systems with target architecture similar to that of the CMS. The fundamental hardware and software features provide the necessary primitives to enable evaluation of a variety of concepts in software recovery, fault-tolerant data management, resource allocation and scheduling, and process control. Both message-based and shared-data-based concepts can be explored and their performance quantified in a cost-effective manner. The crossbar switch, private memory, shared memory, flag memory, full interrupt capability, and I/O processor, provide substantial flexibility in this regard. Target architectures that differ substantially from the CMS can also be effectively emulated; however, the possible effects on fidelity and overhead introduced by the emulation must be carefully evaluated.

The TCN has also proven to be an effective tool for evaluating distributed real-time software technologies. The TCN can closely match the network structure with the computation structure of a chosen real-time application. However, configuring the TCN to fit a new application sometimes involves manual hardware reconfiguration which can be tedious. The structure of the CMS, on the other hand, is more flexible and can represent a variety of network structures through software reconfiguration. However, there may be susbstantial differences between the CMS physical structure and that of the target application system. In such cases, the types of faults that can be effectively injected in the system may be limited. Therefore, a testbed that combines the facilities of both the TCN and the CMS would be desirable.

Although testbeds are ususally more time-consuming to construct and set up than pure software simulators, they are capable of representing the operating environment and input scenario more accurately than software simulators. As a result, testbed-based evaluation produces more accurate results than pure software simulation. Furthermore, testbeds can provide more specific detailed information than simulation. Thus testbeds provide greater insight into the characteristics and limitations of the concepts being explored. Our experience leads us to conclude that testbed-based experimentation is an effective approach to validation of system concepts and design techniques for real-time distributed systems.

Acknowledgment

The authors wish to thank C. Davis, D. Thomas, T. Smith, and T. Johnson of the U.S. Army Strategic Defense Command, Huntsville, AL, for their guidance, support, and

encouragement through the entire course of this research. We also acknowledge the efforts of those who contributed to this project; in particular, M. Beasley, C. Bryant, J. Dingeldine, C. Hall, M. Kurtti, and N. Vosbury for the work on the CMS hardware and software; W. Farrar, P. Rehm, and S. Yang for the work on the TCN hardware and software; and J. M. An, G. Barnett, J. Hellerstein, S. Heu, W. Moquin, H. Welch, and J. Yoon for the work in developing the experiments and the analysis tools.

References

[1] K. H. Kim, "Software techniques for fault tolerance in BMD computing systems, Vol. II, Testbed and tool development," Final Rep., U.S. Army BMDATC Contract, May 1985, available from the Defense Technical Information Center, Cat. No. B092639L.

[2] P. B. Hansen, *The Architecture of Concurrent Programs*. Englewood Cliffs, NJ: Prentice-Hall, 1977.

[3] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in hard-real-time environment," *J. ACM*, pp. 46–61, Jan. 1973.

[4] W. C. McDonald and R. W. Smith, "A flexible distributed testbed for real time applications," *IEEE Computer*, vol. 15, no. 10, pp. 25–39, Oct. 1982.

[5] W. C. McDonald and M. W. Beasley, "A real-time multi-microcomputer architecture employing a fully parallel crossbar switch," in *Proc. ICCD 83*, pp. 255–258, Oct. 1983.

[6] W. C. McDonald, "A flexible multicomputer testbed for research in real-time distributed software technologies," in *Proc. IEEE EASCON 84*, pp. 269–275, Sept. 1984.

[7] T. G. Williams, W. C. McDonald, M. W. Beasley, and G. W. Cox, "A hardware architecture for a flexible distributed computing testbed," in *Proc. 3rd Int. Conf. on Distributed Computing Systems*, pp. 404–409, Oct. 1982.

[8] T. G. Williams, M. W. Beasley, and W. C. McDonald, "CMS—A testbed for evaluating distributed architectures," in *Proc. 15th Southeastern Symp. on System Theory*, pp. 153–156, Mar. 1983.

[9] N. Vosbury and C. Bryant, "System software for experiments in distributed computing on a distributed testbed," in *Proc. 3rd Int. Conf. on Distributed Computing Systems*, pp. 410–415, Oct. 1982.

[10] N. A. Vosbury, "The process design system," in *Proc. Computer Software and Applications Conf.*, pp. 374–379, Nov. 1979.

[11] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell, "A program structure for error detection and recovery," *Lecture Notes in Computer Science*, vol. 16. New York, NY: Springer-Verlag, 1974, pp. 171–187.

[12] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 220–232, June 1975.

[13] K. H. Kim, "Distributed execution of recovery blocks: An approach to uniform treatment of hardware and software faults," in *Proc. 4th Int. Conf. on Distributed Computing Systems*, pp. 526–532, May 1984.

[14] H. O. Welch, "Distributed recovery block performance in a real-time control loop," in *Proc. Real-Time Systems Symp.*, pp. 268–276, Dec. 1983.

[15] W. W. Chu *et al.*, "Database management algorithm for advanced BMD applications," UCLA Tech. Rep. ENG-83-20, CSD 830430, Apr. 30, 1983.

[16] B. W. Lampson, "Atomic transactions," in *Distributed Systems: Architecture and Implementation*. New York, NY: Springer-Verlag, 1981, ch. 11.

[17] W. W. Chu and J. M. An, "Fault tolerant locking (FTL) for tightly coupled systems," in *Proc. 5th Symp. on Reliability in Distributed Software and Database Systems* (Los Angeles, CA), 1986.

[18] W. W. Chu *et al.*, "Distributed database management for real time BMD applications," UCLA Tech. Rep. CSD 860040, ch. 4, July 1986.

CHAPTER V

PERFORMANCE OF CONCURRENCY CONTROL ALGORITHM FOR REAL-TIME DISTRIBUTED DATABASE SYSTEMS

PERFORMANCE OF CONCURRENCY CONTROL ALGORITHMS FOR REAL TIME DISTRIBUTED DATABASE SYSTEMS

1. Introduction

In a distributed database system, such as the BM/C^3 environment, replication of data is used to improve response time and data availability. In a network of processors or *sites*, each site stores some or all of the data for the application. To maintain consistency between copies of the data, alterations to the database at one site should be reflected in all replicated copies of the data. In addition, if one processor fails, the other processors should take over the load of the failed computer. Although there are several available protocols which maintain file consistency in a distributed database, most of them are not suitable for real-time applications.

In this chapter we will present resilient versions of the Exclusive Writer Protocol (EWP) [Chu85] and a protocol which uses Timestamping with Modified Rollback (TMR) [Fisc82] , [Allc83]. We will evaluate the usefulness of these protocols in real-time applications and compare them with the resilient version of Primary Site Locking (PSL) [Ston79] , [An85].

2. Operating Environment

A key to anticipating the behavior of any concurrency control protocol is understanding the environment in which it must operate. If site failures cause network instability, then we cannot expect protocols to operate at peak efficiency. Our initial assumptions are that communication and hardware failures are not malicious. If a message is sent, it either arrives intact or is lost in transit. We assume that message corruption does not occur. Similarly, we assume that all units in the system are either working or failed.

The above restrictions assume that the hardware and environment of our system has merely benign faults. If malicious faults occur in our system, then they may go undetected. Undetectable faults in the system can lead concurrency protocols to operate incorrectly.

3. Protocol Descriptions

For comparison, we will use the well-known protocol, Primary Site Locking (PSL) [Ston79] (Figure 1) as a baseline protocol. PSL is a *centralized* algorithm with respect to any given file; for each file there is one site which controls access to that file. This site, called the *Primary Site (PS)*, may be different for different files. When a site needs to update the data, it requests a lock from the PS. If the data is currently unlocked, then the PS locks the data and grants the lock to the requesting site, which may then update the data. If the data is already locked, then the requesting site must wait for the lock to be released to continue. PSL guarantees serializability, since only one transaction on a given file can take place at any time. However, due to the wait for synchronization, locking delay takes too long for real-time applications. Another problem that may be encountered with locking is *deadlock*. Therefore, in addition to PSL, a deadlock detection algorithm must be implemented.

To reduce overhead messages and eliminate deadlock, the Exclusive Writer Protocol (EWP) is introduced [Chu85] (Figure 2). Like PSL, EWP is centralized with respect to each file, but it is an *optimistic* method. When an update is required, the initiating site sends the update to the Exclusive Writer Site (EW). The EW uses the *sequence number* of the file to determine whether the update was made with the most recent value of the data. Accepted updates are sent to all sites, while rejected updates are discarded. After

each site receives the update, it performs the update locally and increments the sequence number for the file.

EWP guarantees data consistency but does not guarantee serializability. Should there be an update conflict, one of the updates will be discarded. In many applications, such as radar tracking, data are non-volatile and new readings are constantly coming in, so occasional updates may be discarded. By eliminating locks, we lose some control over the data. However, deadlocks cannot occur and we gain a great improvement in speed by avoiding synchronization delay.

For those applications where discarding updates is not acceptable, the Exclusive Writer Locking Protocol (EWL) can be used (Figure 3). EWL works like EWP until an update conflict occurs. Then, the first update to arrive is accepted, and the next update is treated as a lock request. The EW site becomes a PS, and processing continues using PSL. In EWL, EWP is used during conflict-free periods and PSL is used when update/lock conflicts arise. Thus, EWL is serializable. The difference between EWL and PSL is that EWL is an optimistic protocol; it does not incur synchronization costs when there are no update conflicts.

There are extensions to EWP and EWL providing for site and communication failures [An85] using one phase and a backup coordinator in case of EW failure. If network partitioning occurs, to insure data consistency, only sites that remain connected to the EW will be able to continue updating. Sites that cannot communicate with the EW may not update the file. Such protocols that suspend EWP operation during network partitioning are too restrictive, especially in critical environments such as flight control and battle management. More flexible concurrency control protocols are needed.

In order to continue processing during network partitioning, a time-stamp protocol with modified rollback (TMR) has been proposed. [Fisc82] , [Allc83] , [Jajo87] (Figure 4). Like EWP, TMR is optimistic, but there is no controlling site; each site computes updates locally. Normally, when an update occurs, it is executed and logged at the originating site and sent to other sites that contain replicated copies of the file being updated. To reduce communication costs, several updates may be batched and transmitted to other sites together. Likewise, after a partitioned network rejoins and communication is possible again, the batched updates are sent out to other sites. Incoming messages are merged into the local log in the proper order based on the timestamps associated with the update. TMR sacrifices data consistency and serializability, since the logs may be held locally for some time between broadcasts and messages can be received out of order. However, network partitioning will not halt updates; inconsistencies in the data are rectified when the network is rejoined, if possible.

The table below summarizes the differences, advantages and disadvantages of the different protocols.

Protocol	Features	Advantages	Disadvantages
PSL	Centralized Locking w.r.t file	Serializable Consistent Data Balanced Processing Load	Synchronization Delays Partitioning Problems Deadlock Possible
EWP	Centralized w.r.t file	Consistent Data Balanced Processing Load No Synchronization Delay	Update Conflicts Partition Problems
TMR	Distributed	Fewer Partitioning Problems Message Logging No Synchronization Delay	Data Inconsistent Merge Costs Extra Communication

Figure 5
Summary of Protocols

To detect network failure, the resilient EWP and PSL must periodically send I-AM-UP messages to monitor network connections. TMR must communicate a

connectivity vector - an *n by n* array of time stamps specifying the status of the connection to site *i* from site *j* - informing the receiving site about the sending site's view of the network. This information can be used for detecting network partitioning and for recovering lost messages and data. TMR will have higher overall communication costs because the connectivity vector must be transmitted with each update batch.

4. Performance Measures

There are different measures to compare the performance of concurrency control protocols such as data availability, data consistency and accuracy, response time, and throughput. In addition to normal operating conditions, system behavior during failure must be considered, especially in unstable environments.

Different protocols provide varying degrees of data consistency. We define the *True Database* as the portion of the database which is exactly the same as if all transactions had been run on a single infinitely fast processor. *Total Consistency*, as the name implies, is the strictest degree of consistency. A database is totally consistent if all data copies at different sites are the same as the true database for both read and write operations. A database has *Strong Consistency* if, during a write operation, all data copies at different sites are the same as the true database. According to our definitions, PSL is strongly consistent. Since there is no read locking in PSL, though, it is not totally consistent. EWP is not strongly consistent, since updates may be discarded. When an update is discarded, the EWP database varies from the true database, but all the copies of the data are mutually consistent. We say that a database has *Weak Consistency* if all copies at different sites are the same, but may be different from the true database. Clearly, this is true for EWP, but not for TMR, since the TMR database may be updated

at different sites based on different data values. These inconsistencies are later rectified during the merge phase of TMR. Therefore, TMR is *Temporarily Inconsistent.*

If the database is in the true state, then we say that the concurrency control protocol used is *Serializable*. The value difference between the true state of the database and the state after a write conflict is called *Inaccuracy*. When conflicts occur, serializability, accuracy, and weak consistency are affected. Figure 6 shows database consistency measures as read and write conflicts occur. The database slowly deteriorates with each update conflict until a database correction occurs. The cost and frequency of corrections to the data is one measure of the cost of the concurrency control protocol.

From Figure 6, it is evident that PSL provides a higher degree of data protection at the expense of throughput and response time as shown in Figures 7 and 8. As the number of update requests is increased, the total number of updates in the system increases. This increase in throughput continues until the network becomes congested with messages. When used without message batching, TMR requires more communication than other protocols, and, therefore, can cause rapid network congestion. In addition, as communication overhead increases, the number of updates possible (throughput) decreases. Therefore, message batching will reduce costs, making TMR behave better.

The batching of messages in TMR increases throughput, but also increases response time. In situations where the update rate is low, it may take too long to collect enough updates to reach maximum batch size. A timeout period can be specified. If the timeout is exceeded, then the incomplete batch is sent anyway. The timeout period should be selected so that updates will not become stale while waiting to form a batch.

As the update rate increases (see Figure 8), batching keeps the communication system from overloading, and thus response time improves. *Global Response Time* is the total time from initiation of an update to completion at the final site. *Local Response Time* is the time for an update to complete at the initiating site regardless of other sites. For PSL and EWP, local response and global response are almost equivalent. Since updates are executed immediately, TMR (with or without batching) provides excellent local response time. Global response time for TMR may vary dramatically based on batch size, timeout, communication overhead, and other parameters.

If network partitioning occurs, data which is separate from the EW site cannot be updated, reducing availability to zero. Since the data is not available, effective response time of EWP and PSL will be infinite during partitioning, as shown in Figure 9. TMR response time will remain low at the cost of expected data inconsistency. During network partitioning, there global response time is infinite for all protocols if copies of the data are divided. If all copies of the data are in one partition, then global response time will be slightly faster than normal due to the reduction in communication traffic.

When comparing the performance of concurrency control protocols, we must consider response time, throughput, consistency, and behavior during failure. The protocol selection should depend on the application; we may choose a protective protocol for sensitive data, and a different protocol to improve data availability.

5. Simulation

Simulation is used to study protocol behavior. The simulation program is written in the Pascal programming language. A typical run takes anywhere from 10 seconds to 1 minute of CPU time on a Sun 3.

The simulation program (Figure 10) consists of a generic driver along with system routines, atomic actions, and specifications for handling all possible events for each of the different protocols. Figure 11 shows a schematic of the driver. The simulation keeps a list of *events* which are active in the network. Initially, the active events are all update initializations for the length of the simulation. When the global simulation clock exceeds an event time, that event is executed at the appropriate site. If the site is not free, then the event is reinserted into the event queue, to execute at a later time. If the event is executed, then the time required is calculated, and the site involved is assigned to that event until the event is complete. The event is executed, possibly spawning more events for the future. The simulation ends when any site's local time exceeds the specified end time of the simulation. Figure 13 presents a flowchart of the event driver.

Figure 13 shows the protocol dependent event handler for generic EWP. After a 'make-update' event occurs, causing an update to be initiated, the database is read. The EW site for the data item is calculated, and an 'update-request' event is sent to the EW site. In the case of the 'read' operation, like other atomic actions, no real action occurs, but the simulation results are updated.

When the 'update-request' arrives at the specified EW site, the sequence number of the record to be updated is compared with the sequence number sent by the requesting site. If the sequence number is up-to-date, then the update is performed by sending 'do-update' messages to all sites containing a copy of the file in question. The file is also updated at the EW site, and the sequence number is incremented.

When a 'do-update' event arrives at a site, that site updates the database if the sequence number is correct. In a fault free environment, there will be no out-of-order

messages. However, if messages can be lost or delayed in transmission, the sequence numbers will protect the database from corruption.

The event handling mechanisms for PSL and TMR are in Figures 14 and 15 respectively. The 'forward' box in each handler is intended for use in networks which are not fully connected. If site A, B, and C are connected in a line, then, when A sends a message to C, it must pass through B. Rather than separate message handling from the rest of the system, we chose to include it as part of an integrated event handling system. Thus, when an event arrives, and it is not for the site at which it has arrived, then it is sent to the next node on the path towards the site.

For clarity several events were not included in the handler diagrams. These events, such as site failure, I-AM-UP messages, site recovery, and external read from a site without data, are not part of our discussion at this time.

Input to the simulation (Figure 16) is derived from the application domain, including time to write and read a record, time to send a message, time between updates (Update Rate), processing speed of the computers used, network topology and length of the simulation. There are also such protocol dependent parameters as time between I-AM-UP messages for EWP and merge costs for TMR. The output of the simulation (Figure 17) is a list of the number of different operations performed at each site, number of updates accepted and rejected, the level of data consistency and availability, merge and database reconciliation costs. The Run Information listed near the top of the results is a description of the distribution of the data amongst the sites in the network. The numbers (one through six, in this case) represent the different data replication possibilities. The numbers in parentheses are a list of the sites containing data of that type, the first number

being the EW or PS site. Thus, data type 4 is at all three sites with site 2 being the EW site. Just below that, there is a distribution of the number of records of each different type. In this example, there are 5000 total records distributed unevenly.

Using simulation experiments, we can compare the cost and performance of different protocols and the level of data consistency that the protocols provide before, during, and after network partitioning.

6. Examples

We shall use a hypothetical example from the BM/C^3 environment as input to the simulation. There are three battle managers which must respond to updates of 5000 objects in a time period of 900 seconds. There is some overlap between the regions controlled by the managers which creates replicated data regions. For this example, five thousand update initiations were generated and uniformly distributed in the zero to 900 second interval. For each of the initiations a site assignment was made based on conflict probabilities previously provided. Thus, some updates were selected for execution on single sites, while others were run at multiple sites.

Using this data, we varied several key parameters, including update rate, interprocess communication (IPC) cost, batch size for TMR, and coverage overlap between the sites. The parameter values shown in Figure 16 were used as baseline values. Response time, throughput, and the number of messages generated were measured. Figure 18 shows different distributions of the 5000 updates. The top diagram shows the baseline conditions, with darker lines representing EW control over data. For example, site 1 is the EW for records 0-3000 and also has replicated copies of 3000-3500. The middle figure is a low overlap coverage, while the bottom figure is a high overlap

coverage.

Figure 19 shows the response time of PSL, EWP and TMR as a function of IPC for baseline coverage. As expected, the response time for TMR was higher than EWP, but lower than PSL. It is interesting to note that although an increase in IPC causes a linear increase in response time, the rate of increase in response is quite different because of the varying degrees of communication required by different protocols.

Our investigation of site coverage reveals that when the data replication between sites is low, network traffic is reduced which improves response time (Figure 20). The volume of communication increases as the degree of replication increases, which increases response time (Figure 21). The volume of communication between sites is reduced when there is less data replication. As a result, response time for low replication databases is less sensitive to communication cost. For a given update rate, PSL is more sensitive to interprocess communication cost than TMR or EWP. The response time of PSL increases significantly as data replication increases.

Figure 22 shows the effect of update rate on response time. As the update rate increases, the network begins to congest. The large response delay of PSL due to waits for locking will cause PSL response time to become prohibitive as update rates increase. This illustrates that the added communication and delay due to locking required by PSL make it unsuitable for real-time systems. Since locking is not required for EWP, the rate of response time for EWP rises as update rates increase, but not as dramatically as the rate of PSL.

Figure 23 shows the effect of batching messages in TMR for various timeout periods. When communication costs are high, we expect to gain by batching messages,

since fewer messages will be transmitted between sites. However, the cost of batching is demonstrated in Figure 24, where an increase in batch size causes a corresponding increase in response time. For our initial results, we assumed that the communication costs for a message do not vary with the size of the message. Thus, response time results for TMR without batching are better than TMR with batching.

In addition to batch size, timeout period affects the behavior of TMR. As the timeout period increases, the number of messages in the system decreases and the global response time increases (Figure 25). * By selecting an appropriate maximum batch size and timeout period for TMR, we can minimize the communication overhead, while maximizing system throughput.

7. Future Work

Currently, we are continuing our experiments studying system behavior by varying other key parameters, such as I-AM-UP frequency. In addition, we are developing a methodology for measuring data consistency and accuracy for different protocols. We plan to measure response time during failure and reconciliation costs.

To study the behavior during failure, we will inject faults into the system and observe system response during and after the fault. We plan to investigate the fault recovery period, system degradation, merge period during network rejoin, cost of reconciliation and its effectiveness. When network partitioning occurs, database availability will be greatly reduced. We will quantify how the decrease in availability affects overall systemwide decision making.

* Due to our assumption of constant message size, the actual increase in response time for larger batches will be less than is shown.

Several approaches may be used to combat the reduction in data availability during partitioning. TMR allows data handling at all working sites, but then must spend extra time repairing the database when the network rejoins. *Blocking* restricts access to sensitive data during partitioning, and *Voting* allows dynamic EW site reconfiguration. Our plans include an analysis of the merits and pitfalls of different strategies.

Since blocking is too restrictive, voting is costly and restrictive, and continuing normally will result in inconsistent data, we plan to use information about the data to improve database access during partioning. Specifically, previous usage patterns, data volatility, and data repair costs will guide us. We can also use a knowledge base to provide information and trigger or alert the system in case of failure. The knowledge base and semantic information can be used to improve system behavior and provide fault avoidance, detection, inspection, isolation, repair and recovery.

Based on our study, we hope to develop a methodology that uses application requirements to guide the selection of appropriate concurrency control protocols for distributed real-time systems.

References

[Allc83] Allchin, J. E., "A Suite of Robust Algorithms for Maintaining Replicated Data Using Weak Consitency Conditions," *Proc. Symposium on Reliability in Distributed Software and Database Systems*, vol. 3, pp. 47-56, IEEE, 1983.

[An85] An, Jung Min and Wesley W. Chu, "A Resilient Commit Protocol for Real Time Systems," *Proceedings of the 1985 Real Time Systems Symposium*, San Diego, CA, December, 1985.

[Chu85] Chu, Wesley W. and Joseph Hellerstein, "The Exclusive Writer Approach to Updating Replicate Files in Distributed Processing Systems," *IEEE Transactions on Computers*, pp. 489-500, June, 1985.

[Fisc82] Fischer, Michael J. and Alan Michael, "Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network," *Symposium on Principles of Database Systems*, pp. 70-75, ACM, 1982.

[Ston79] Stonebraker, Michael, "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.

[Jajo87] Jajodia, Sushil and Catherine A. Meadows, "Mutual Consistency in Decentralized Distributed Systems," *Proc. Third International Conference on Data Engineering*, vol. 3, pp. 396-404, IEEE, 1987.

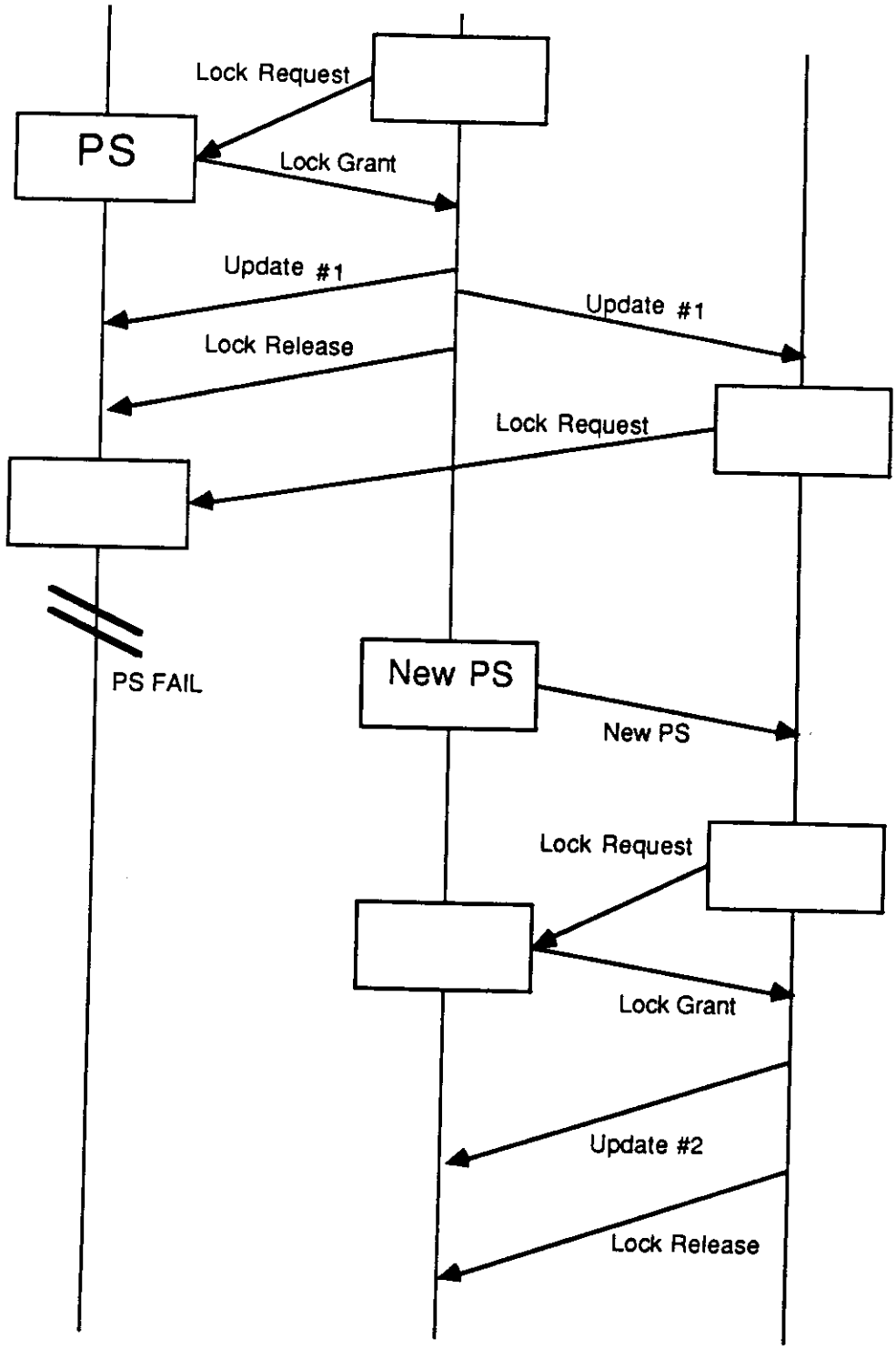


Figure 1
Resilient Primary Site Locking

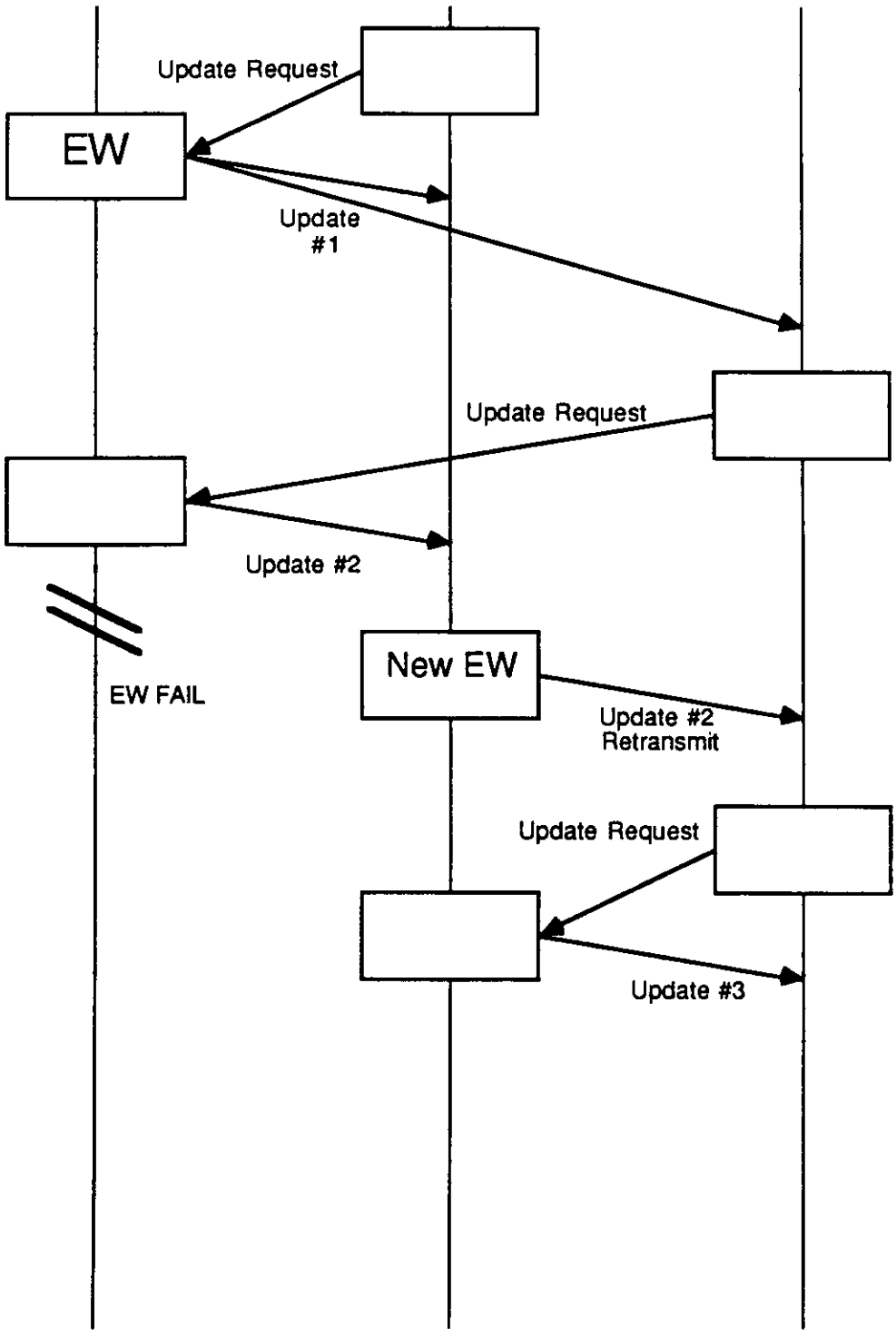


Figure 2

Resilient Exclusive Writer Protocol

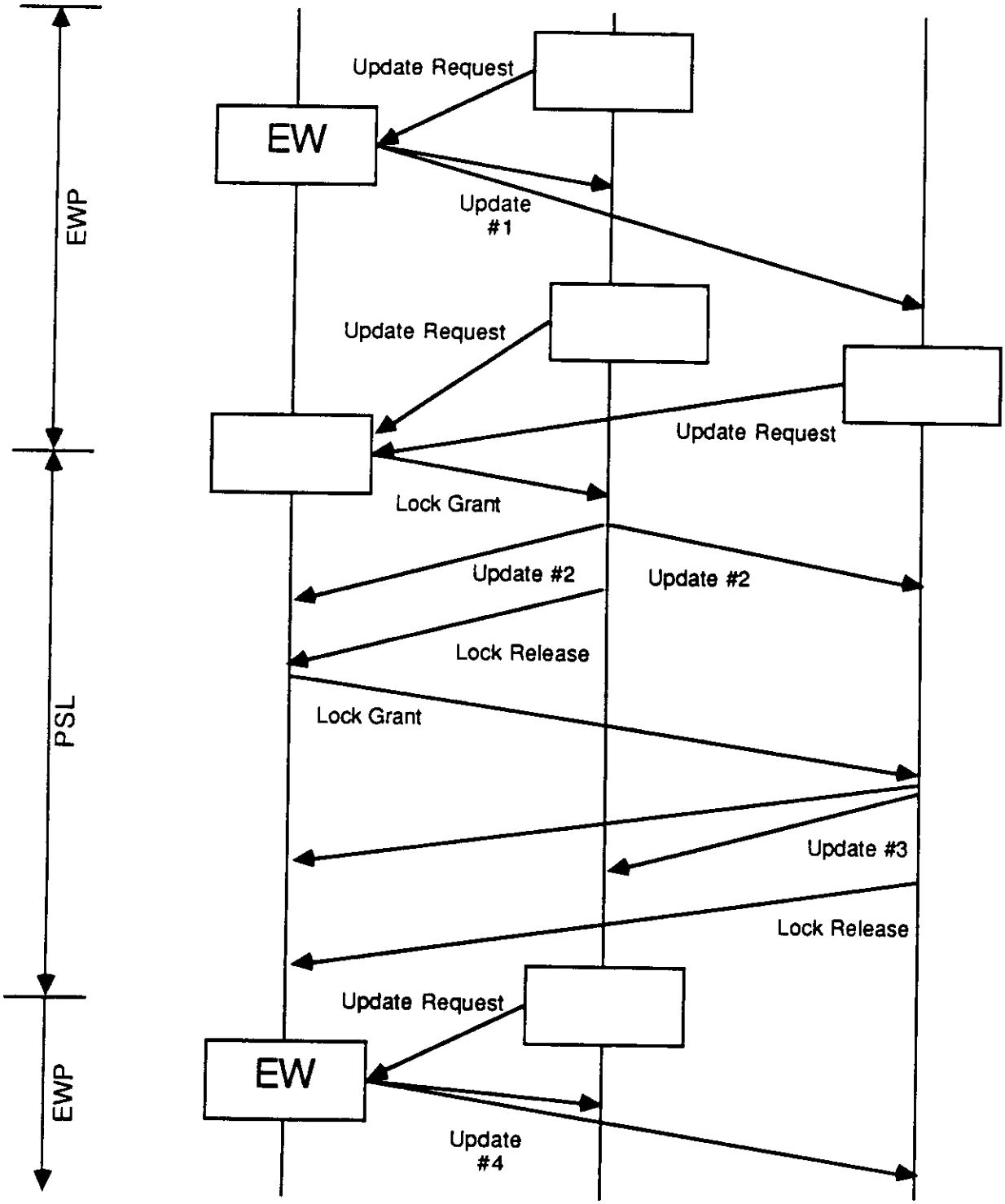


Figure 3
EWL Protocol

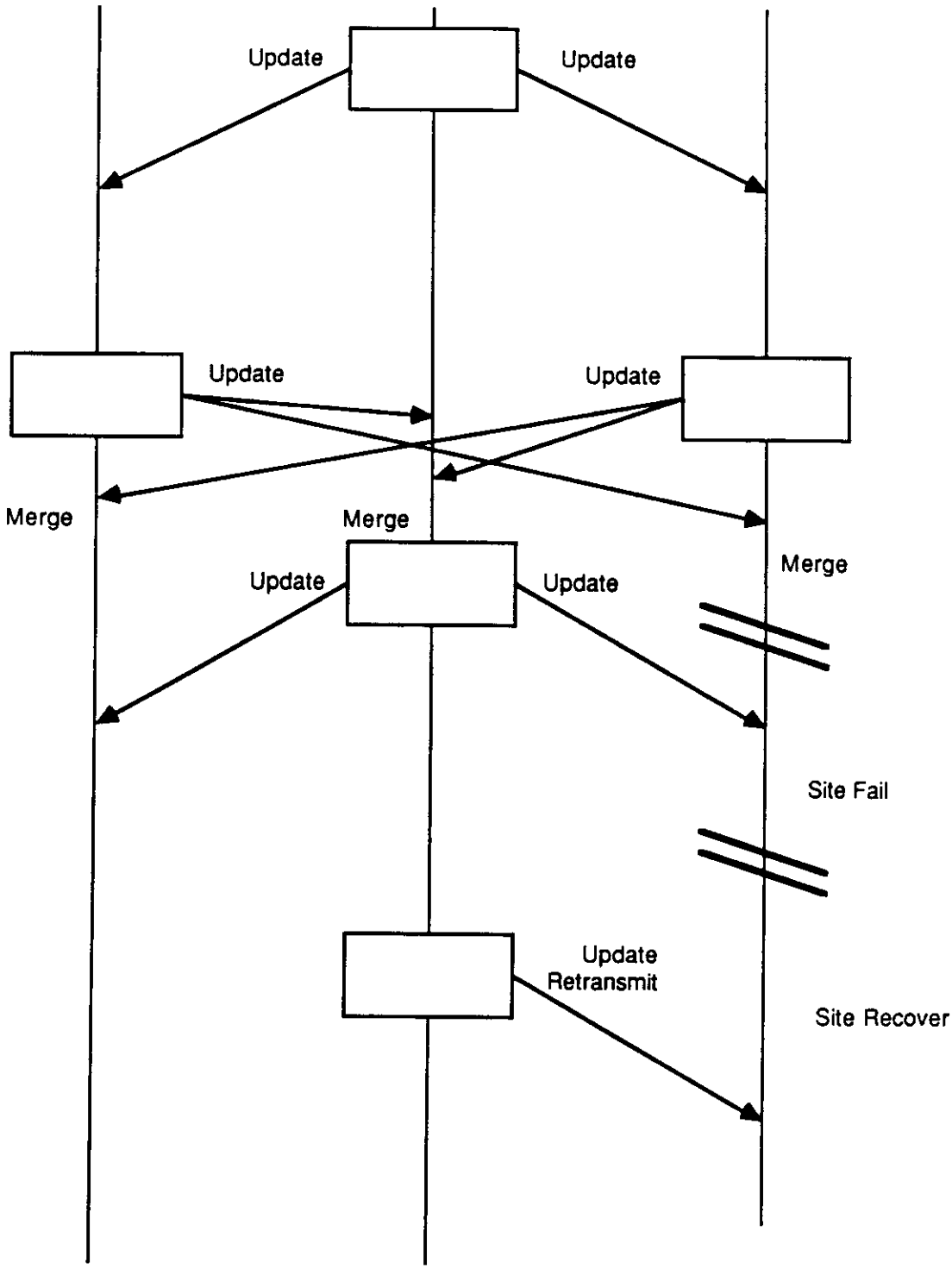


Figure 4
TMR Protocol

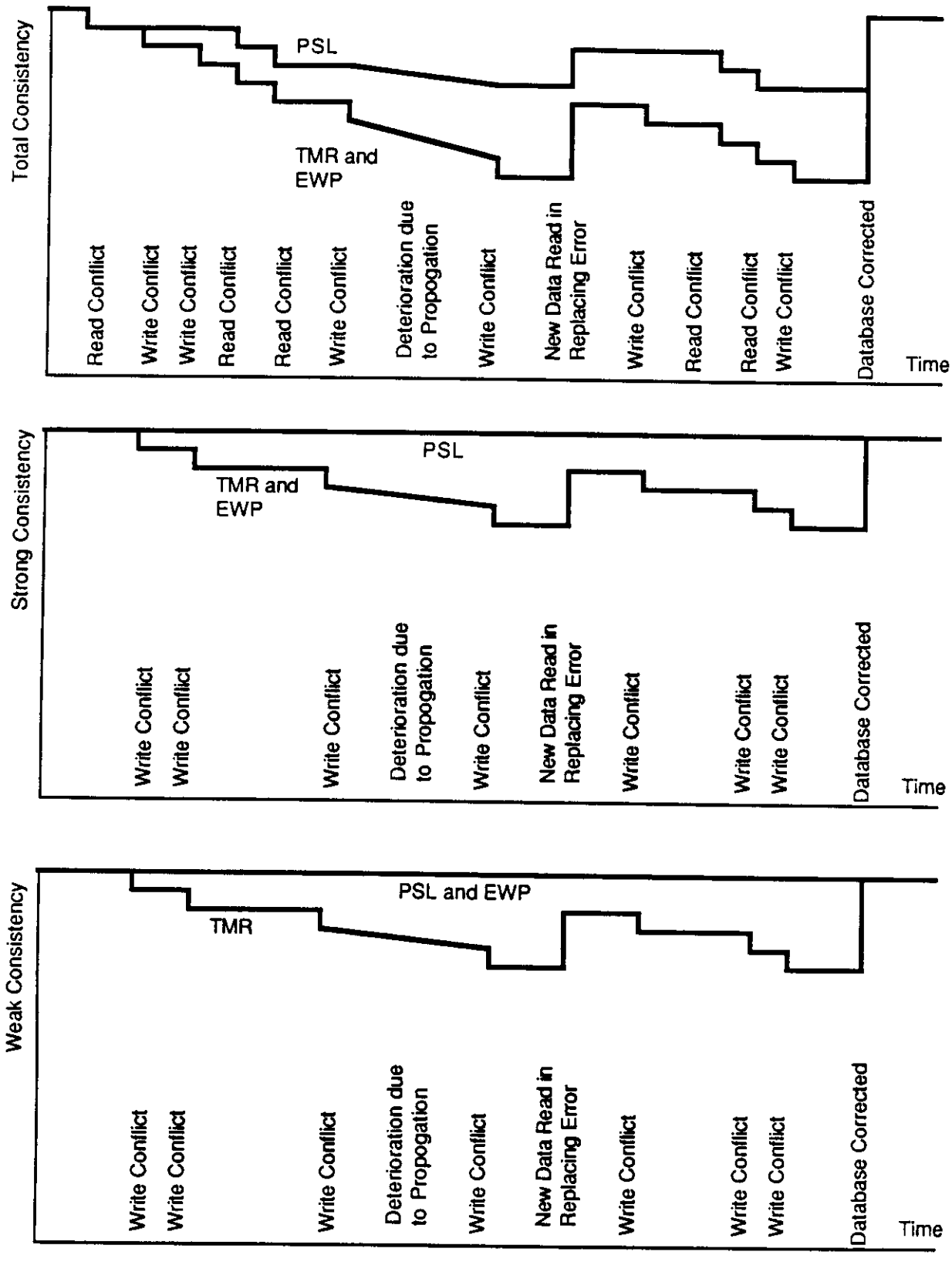


Figure 6

Consistency Deterioration and Correction for PSL, EWP & TMR

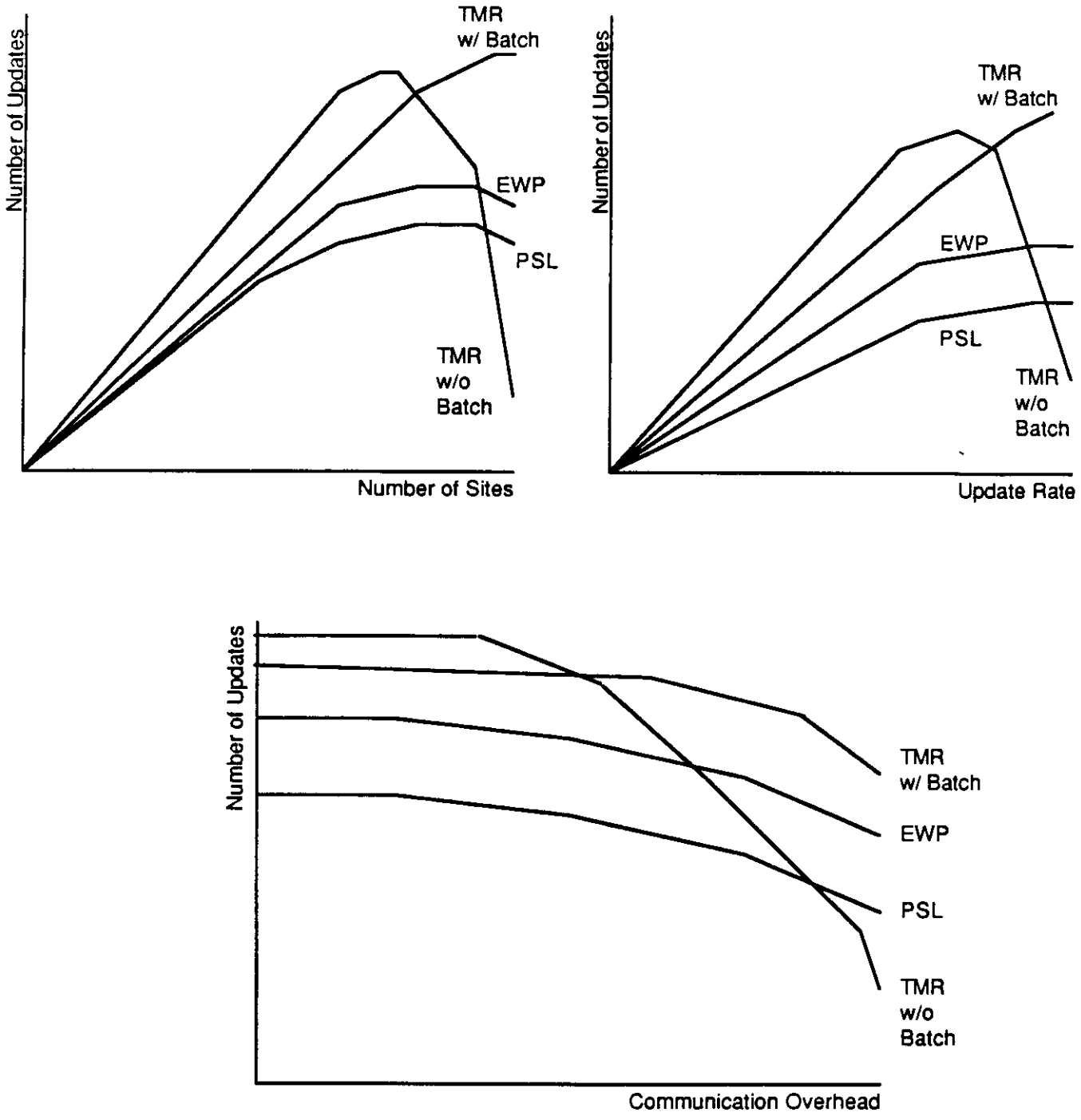


Figure 7

Throughput Behavior of PSL, EWP & TMR

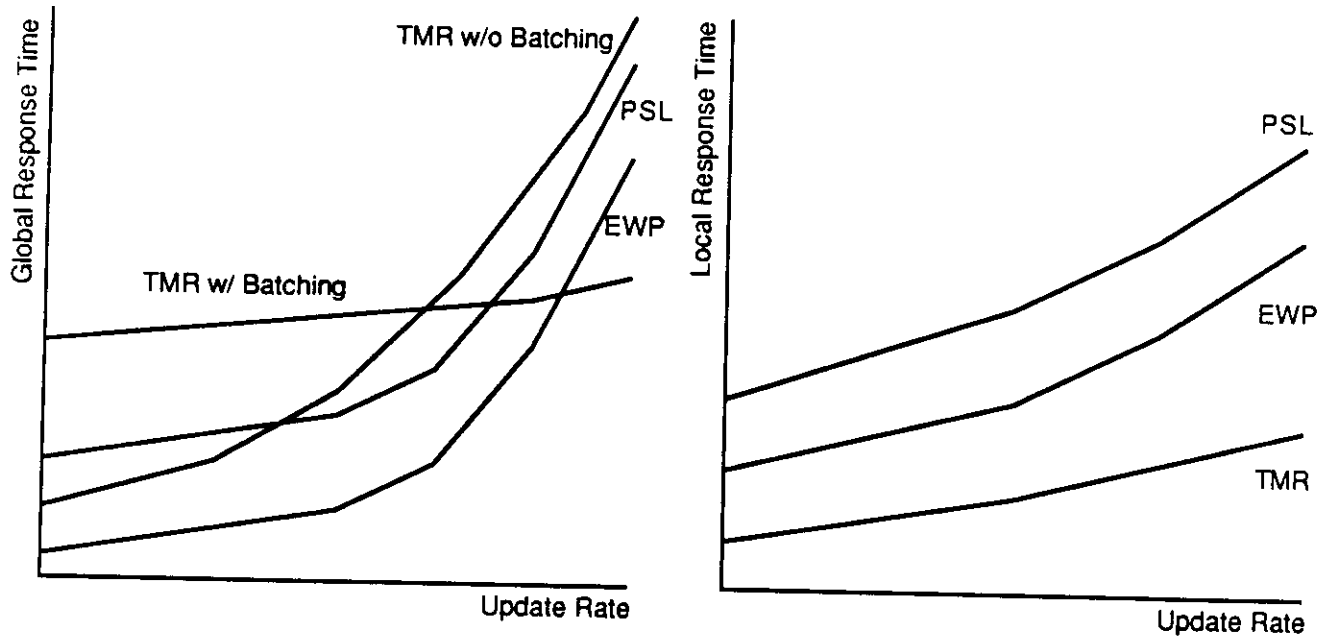
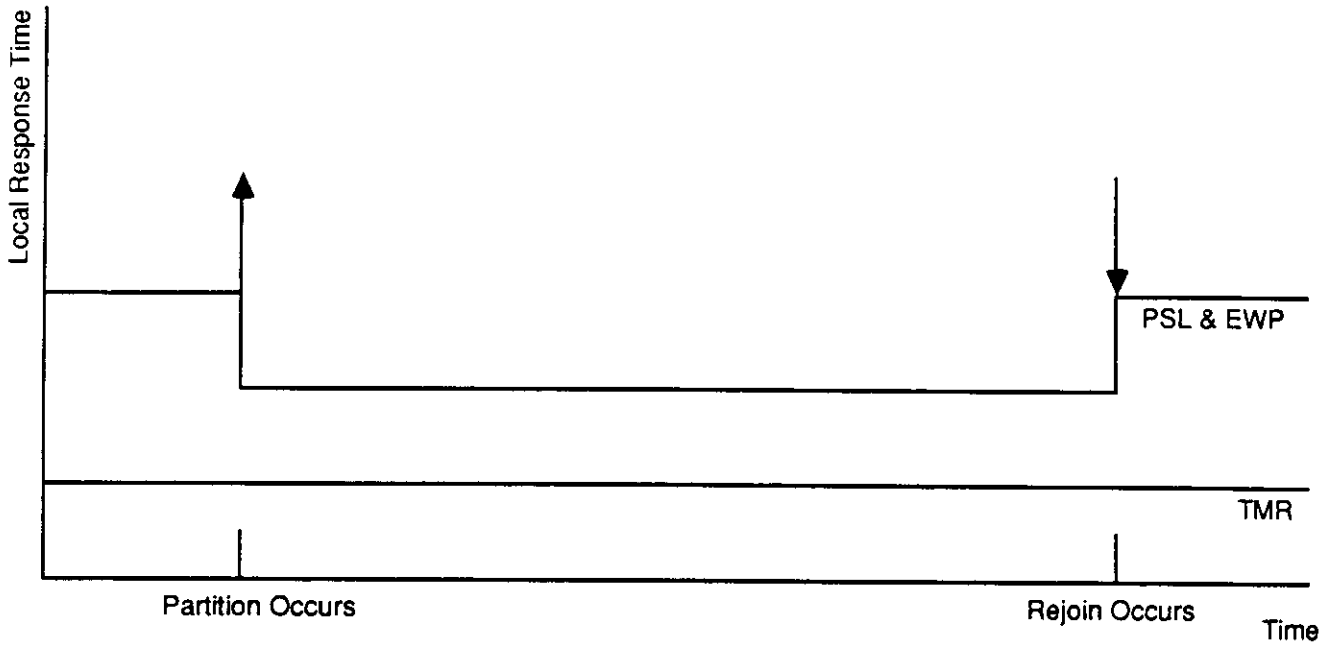


Figure 8
Response Time vs. Update Rate



LRT is smaller if site is in partition containing EW/PS
LRT is infinite if site is not in partition containing EW/PS

Figure 9
Local Response Time
(before, during, and after Network Partitioning)

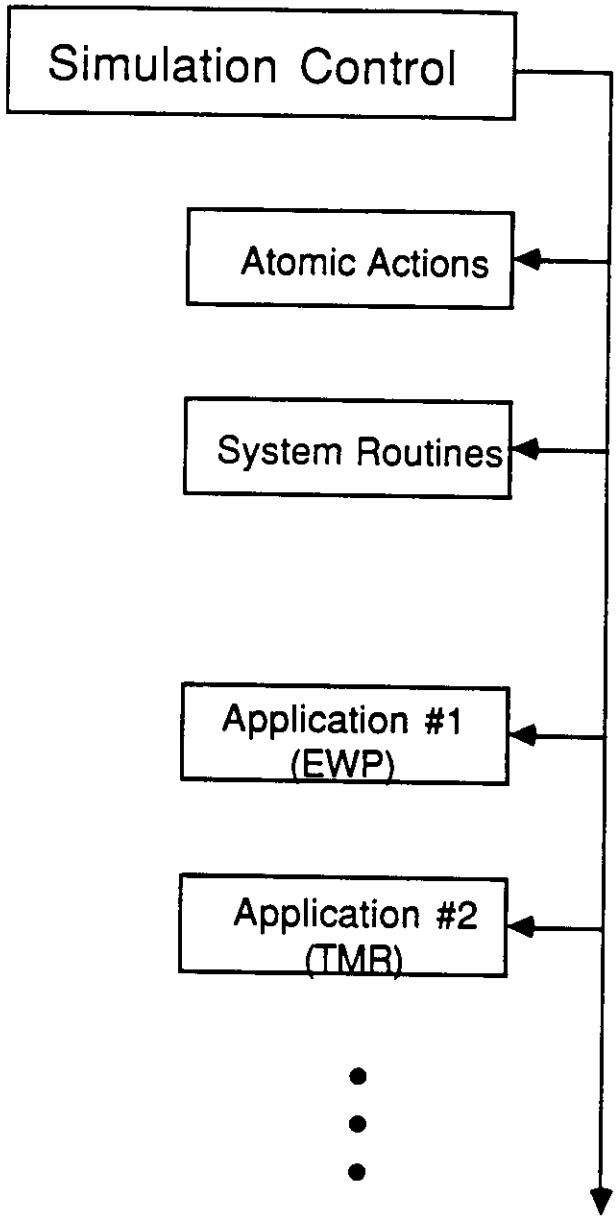
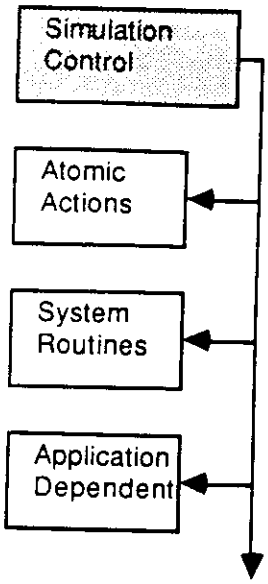


Figure 10
Simulation System Organization



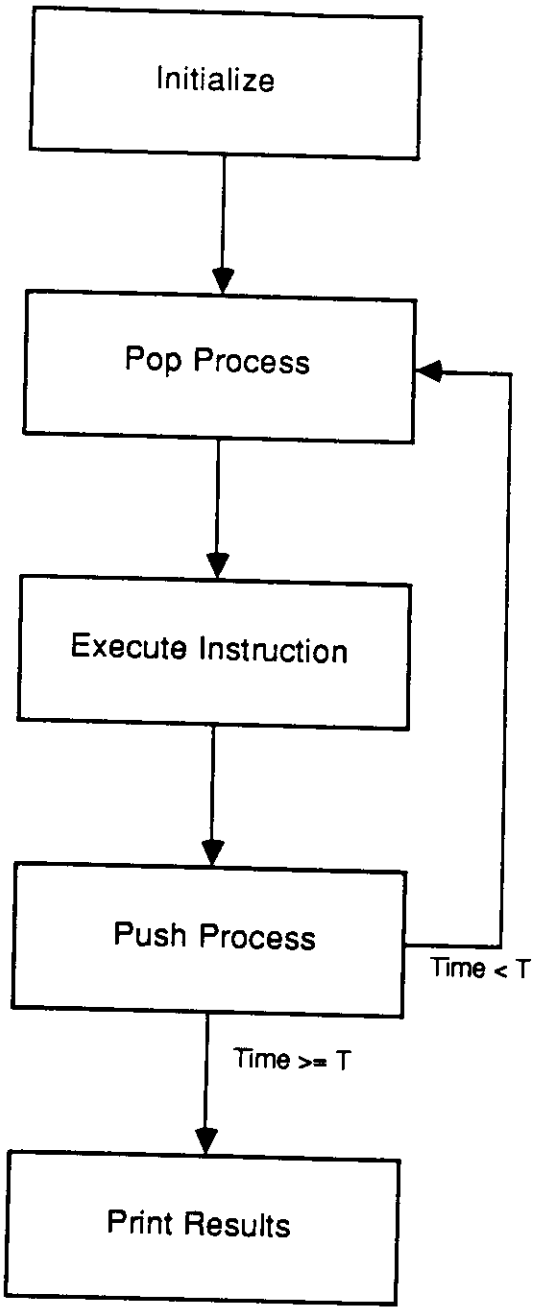


Figure 11

Simulation Control

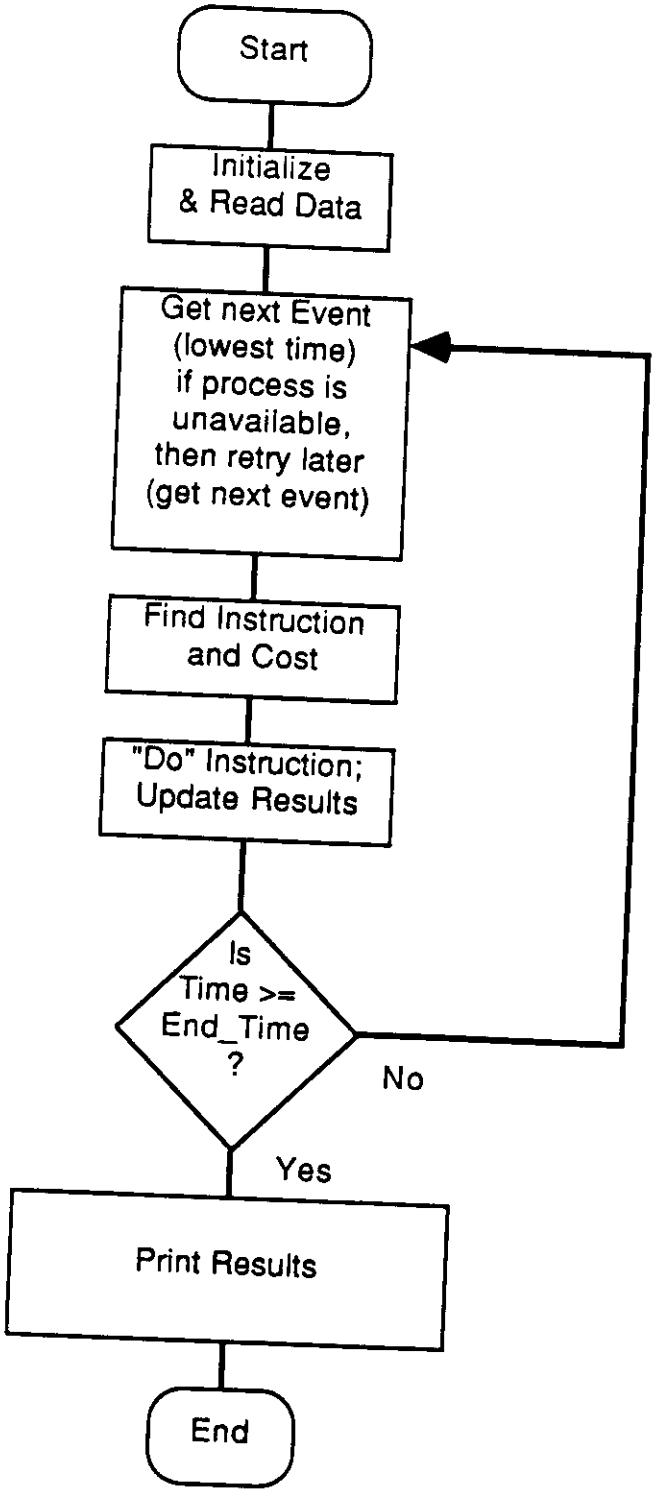


Figure 12

Flowchart of Simulation

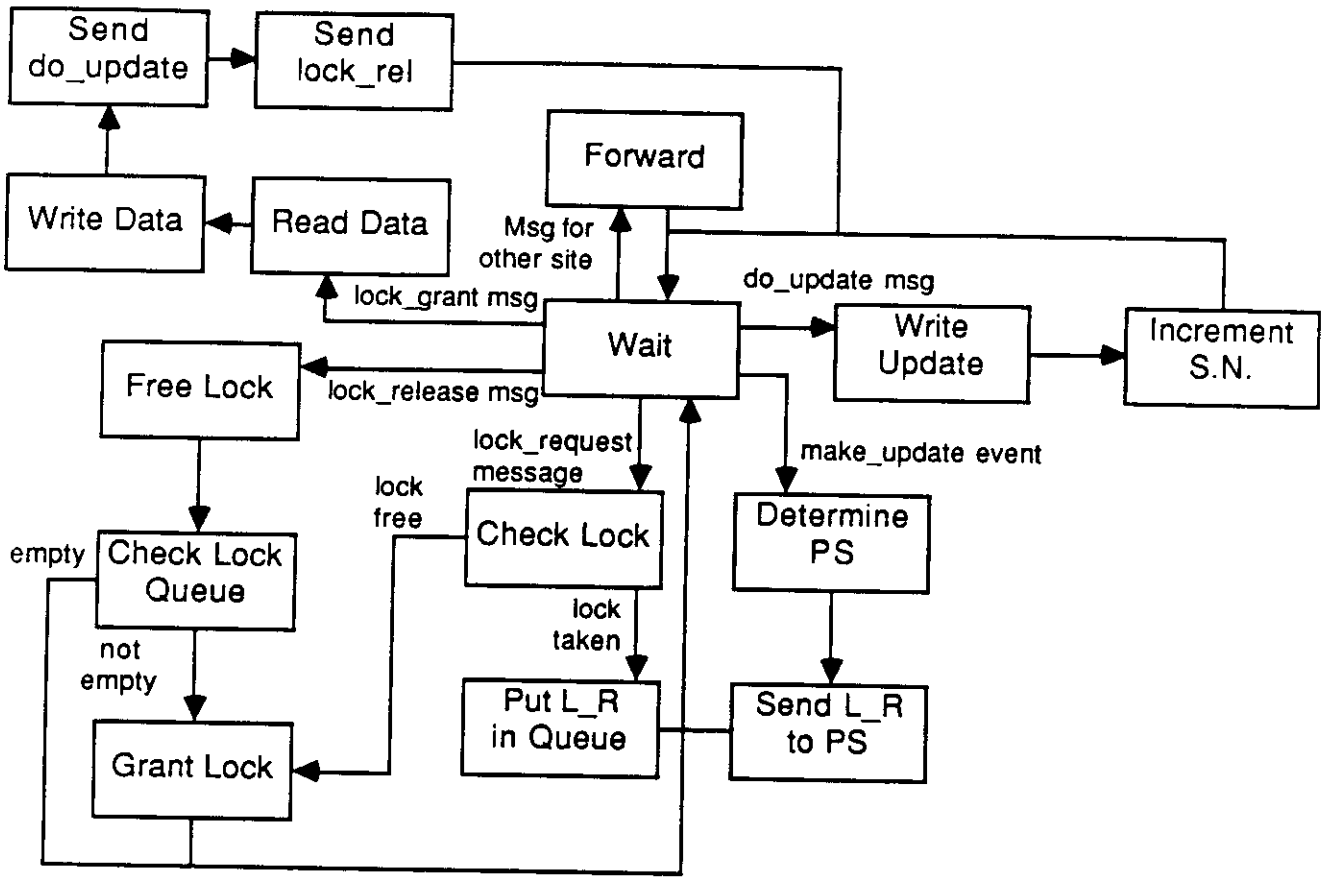


Figure 13

Event Handler for PSL

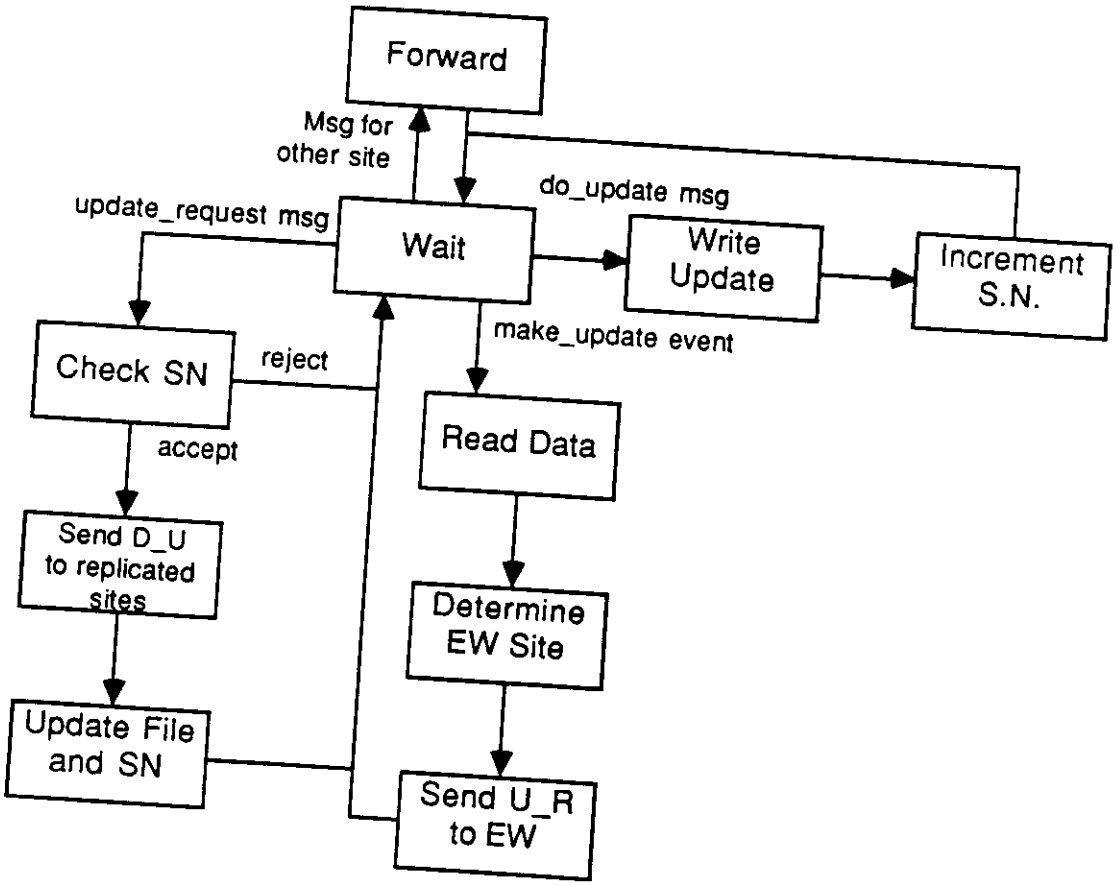


Figure 14

Event Handler for EWP

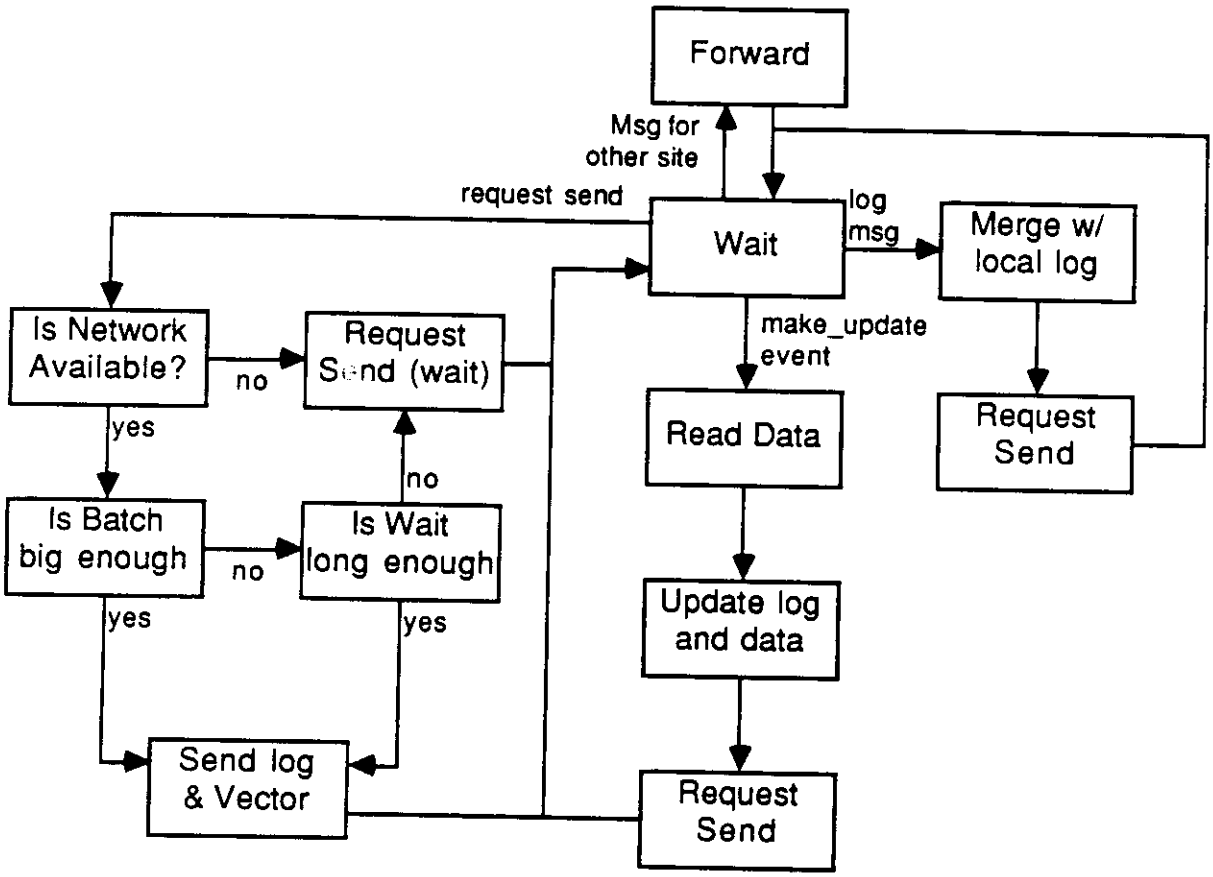


Figure 15

Event Handler for TMR

```
3       Number of Sites
5000    Objects
900.0   End Time (in Seconds)
1       Processor Rate - Speedup over 1 MIP machine
1       Network Topology
3       Protocol
5       Batch size (1 = No Batching) in TMR
0.25    Max Time between batches (0 = No Batching) in TMR
0.0003  costs[READ] in Seconds
0.0030  costs[WRITE] in Seconds
0.0002  costs[SEND] in Seconds
0.0500  costs[IPC] in Seconds
44      debugging level
```

Figure 16

Parameter Specification for Simulation
(baseline example)

```
Initializing
Running for 3 Sites until Time = 900.00
5000 Records; Protocol = EWP; Topology = FULL

Costs (adjusted for a 1.00 MIP machine):
Read:  0.0003
Write: 0.0030
Send:  0.0002
IPC:   0.0500

Run Information:  (Baseline Data Replication)
1:(1) 2:(1,2) 3:(2,1) 4:(2,1,3) 5:(3,2) 6:(3)
          1      2040
          2       989
          3       504
          4       481
          5       487
          6       499

Total Conflicts  - 2461
Conflict Rate    - 32.98%
Total Rejections - 2942
Rejection Rate   - 37.04%

Simulating

Simulation complete at: 899.85

Processor    Inits    Writes   Messages   Down
        1     4014      4014       1974   0.00000
        2     2461      2461       2942   0.00000
        3     1467      1467        968   0.00000

Number of updates Completed:      5000
Number of updates Rejected:       2942
Percentage of updates Completed: 62.96

Average Global Response Time: 0.02874
Maximum Global Response Time: 0.06059
Minimum Global Response Time: 0.00370


     30866 milleseconds processing time

End of Simulation Run
```

Figure 17

Sample of Simulation Output
(for baseline example)

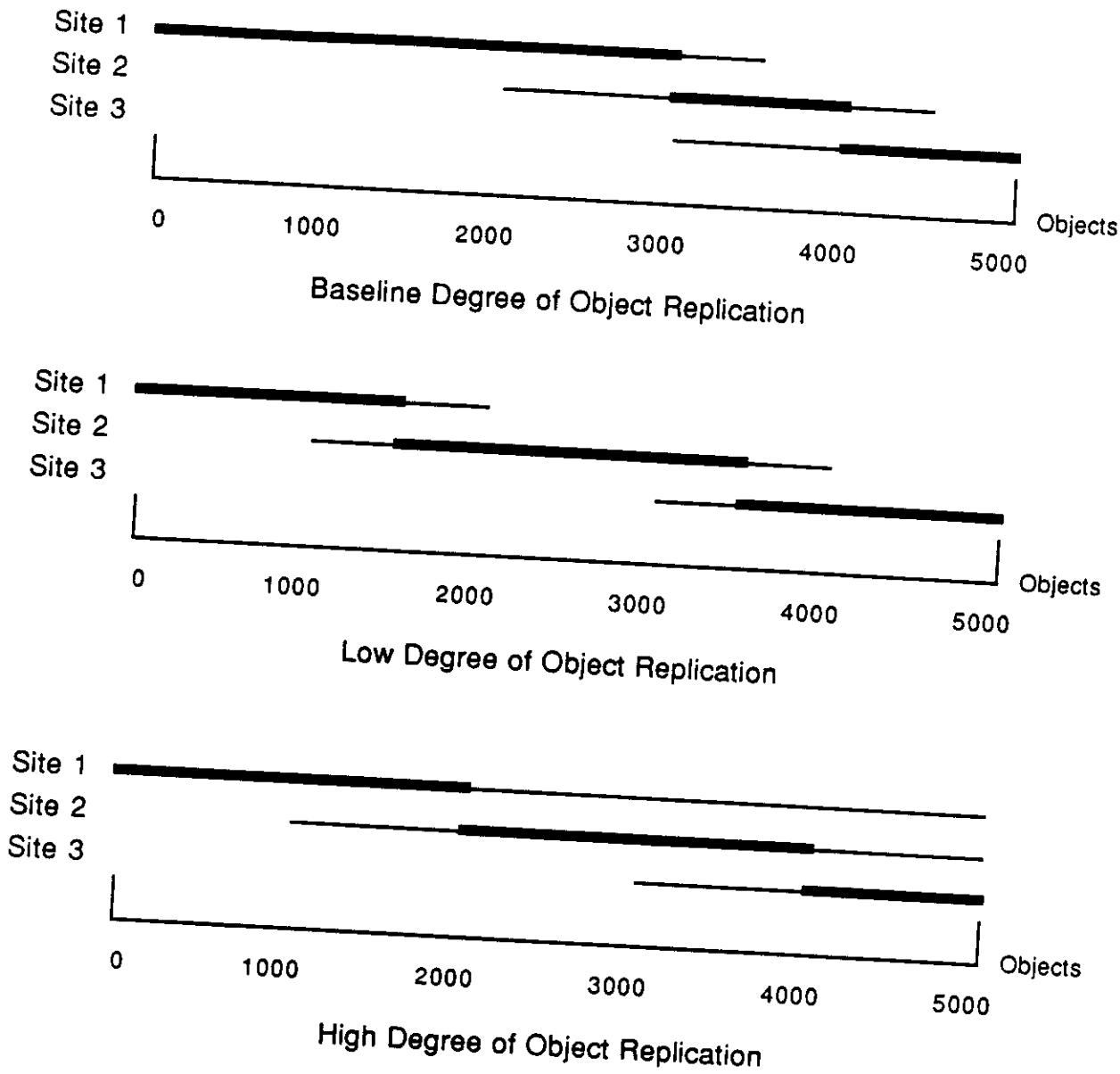


Figure 18

Object Replication
for a 3 Site Battle Management Example

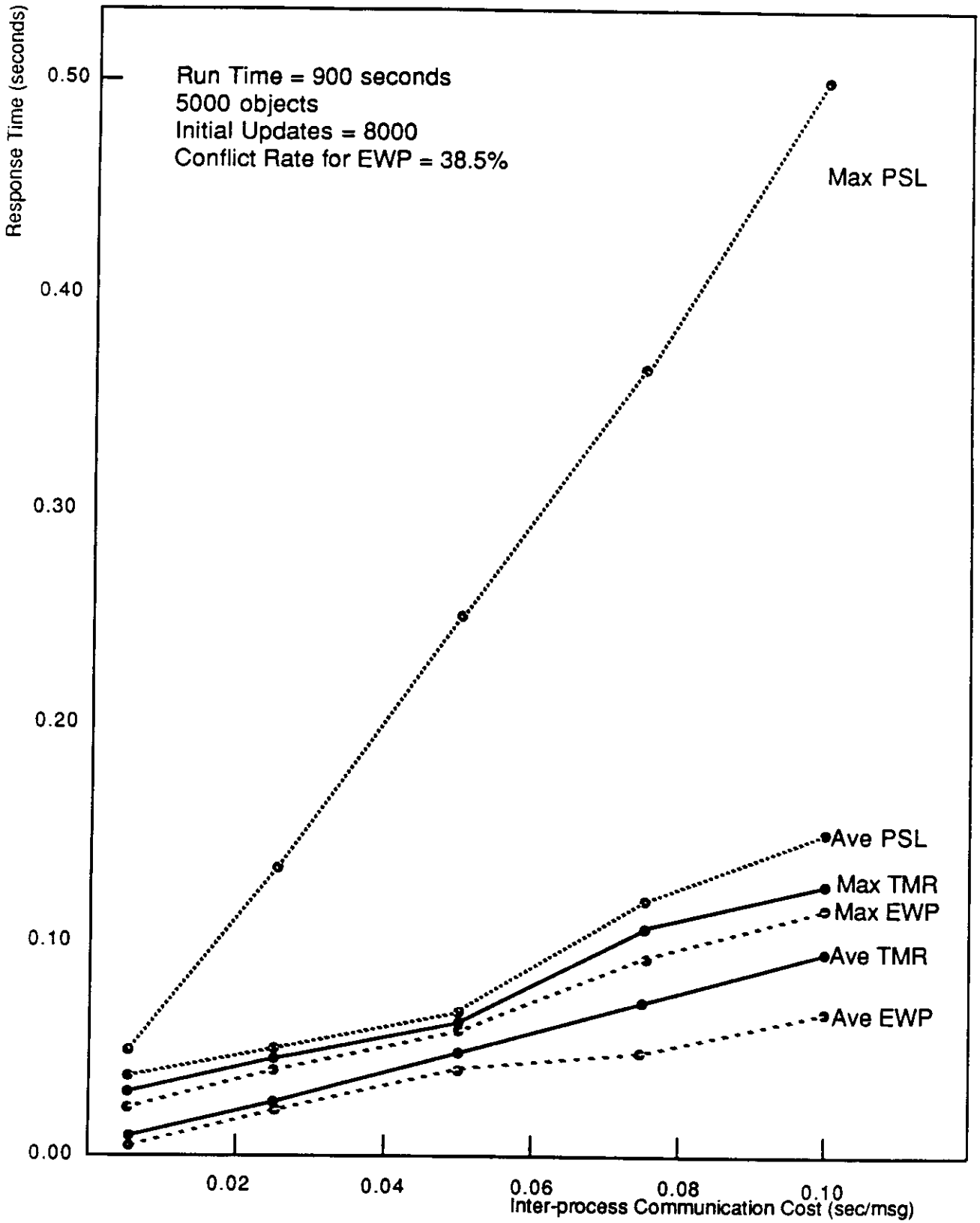


Figure 19

Response Time of PSL, EWP &TMR vs IPC for Baseline Replication

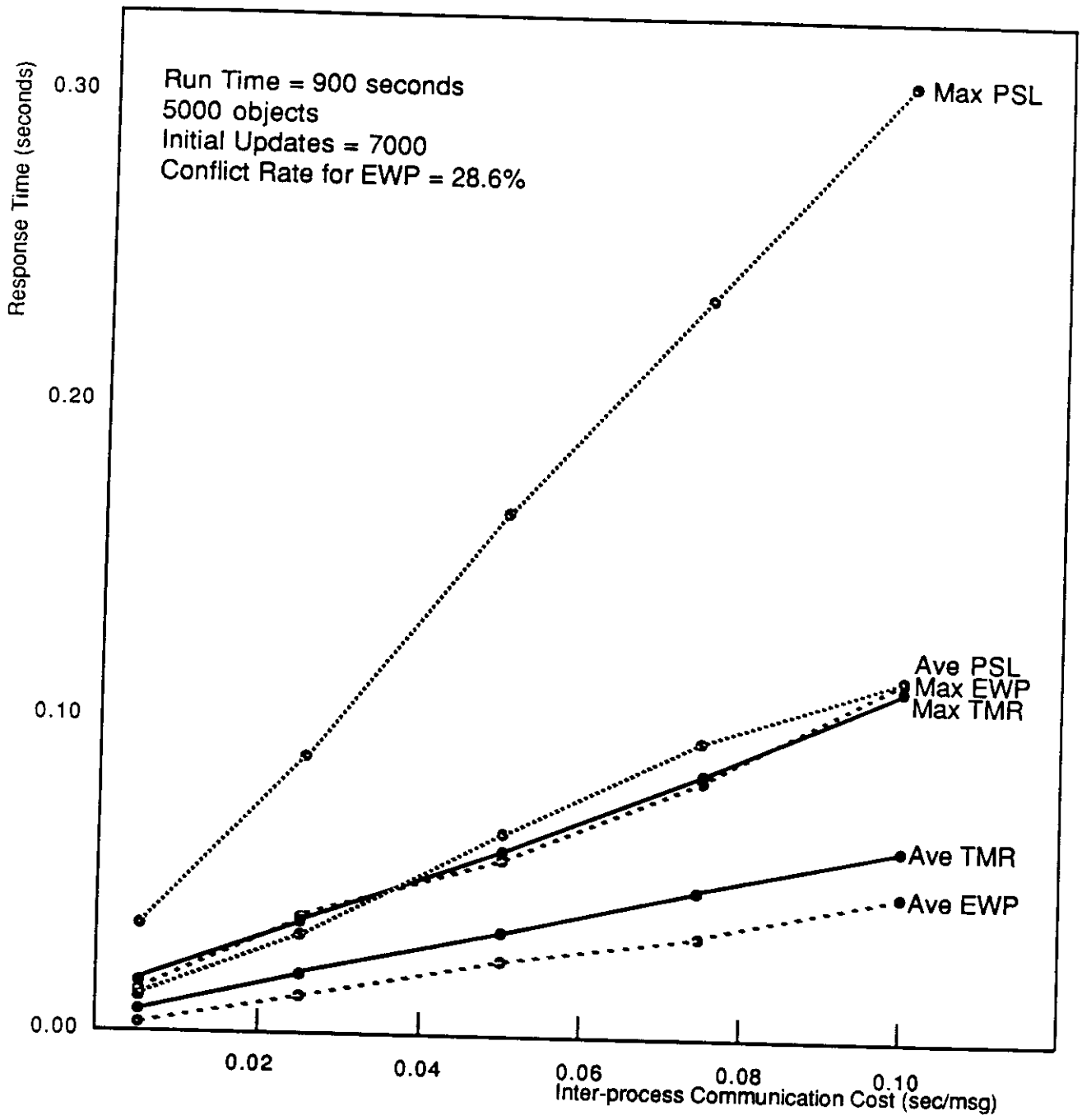


Figure 20

Response Time of PSL, EWP &TMR vs IPC for Low Data Replication

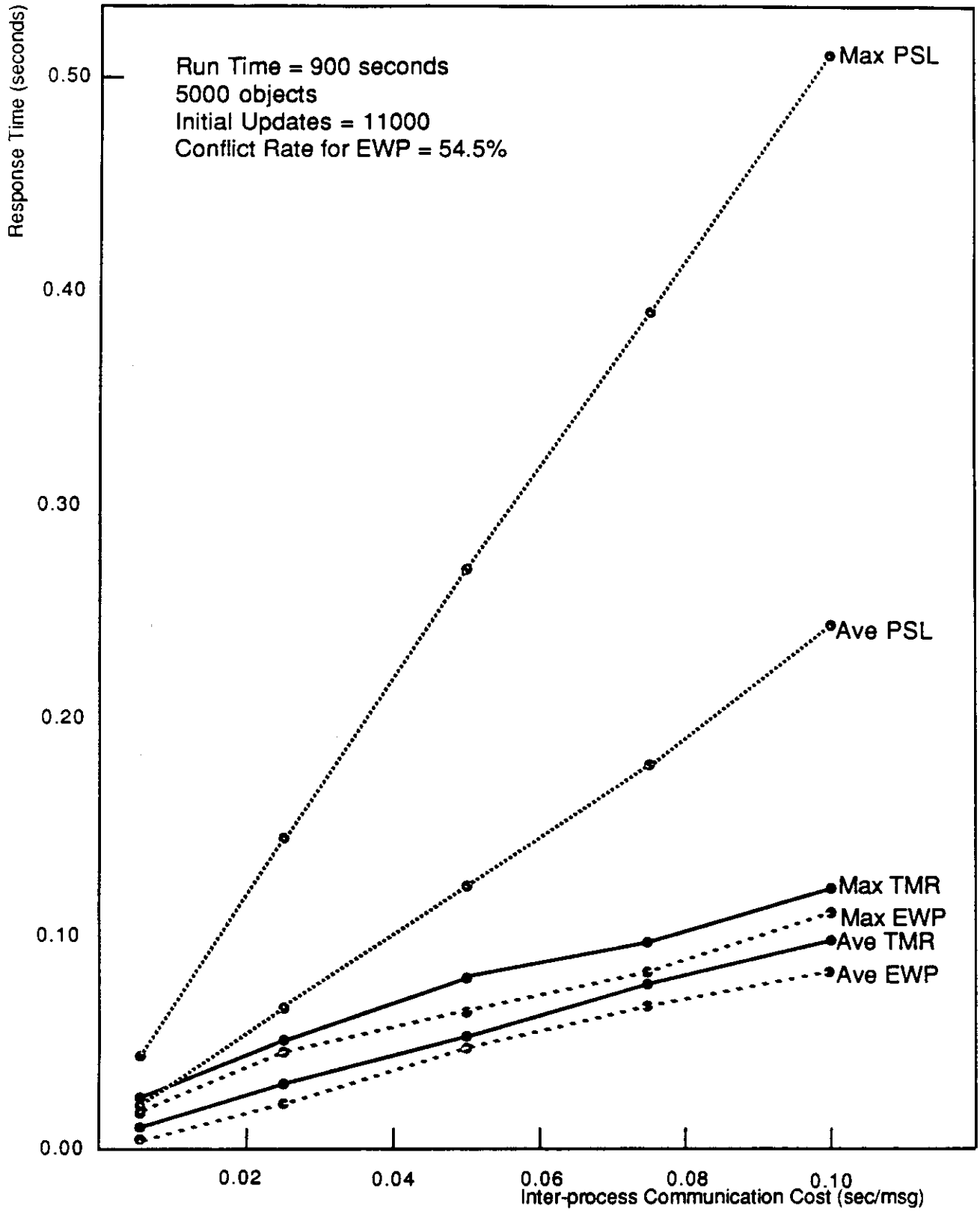


Figure 21

Response Time of PSL, EWP &TMR vs IPC
for High Data Replication

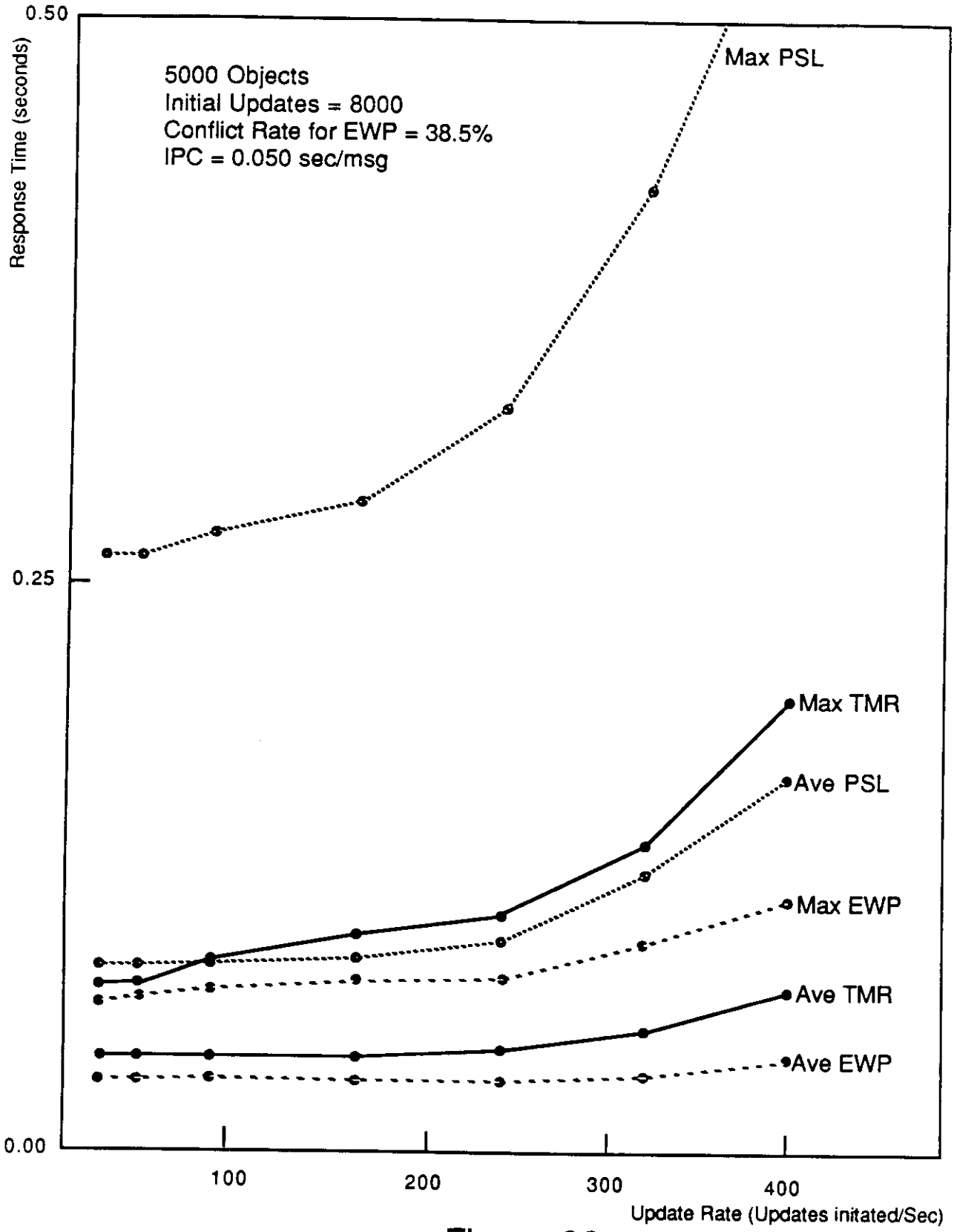


Figure 22
Response Time for PSL EWP & TMR vs. Update Rate using Baseline Data Replication

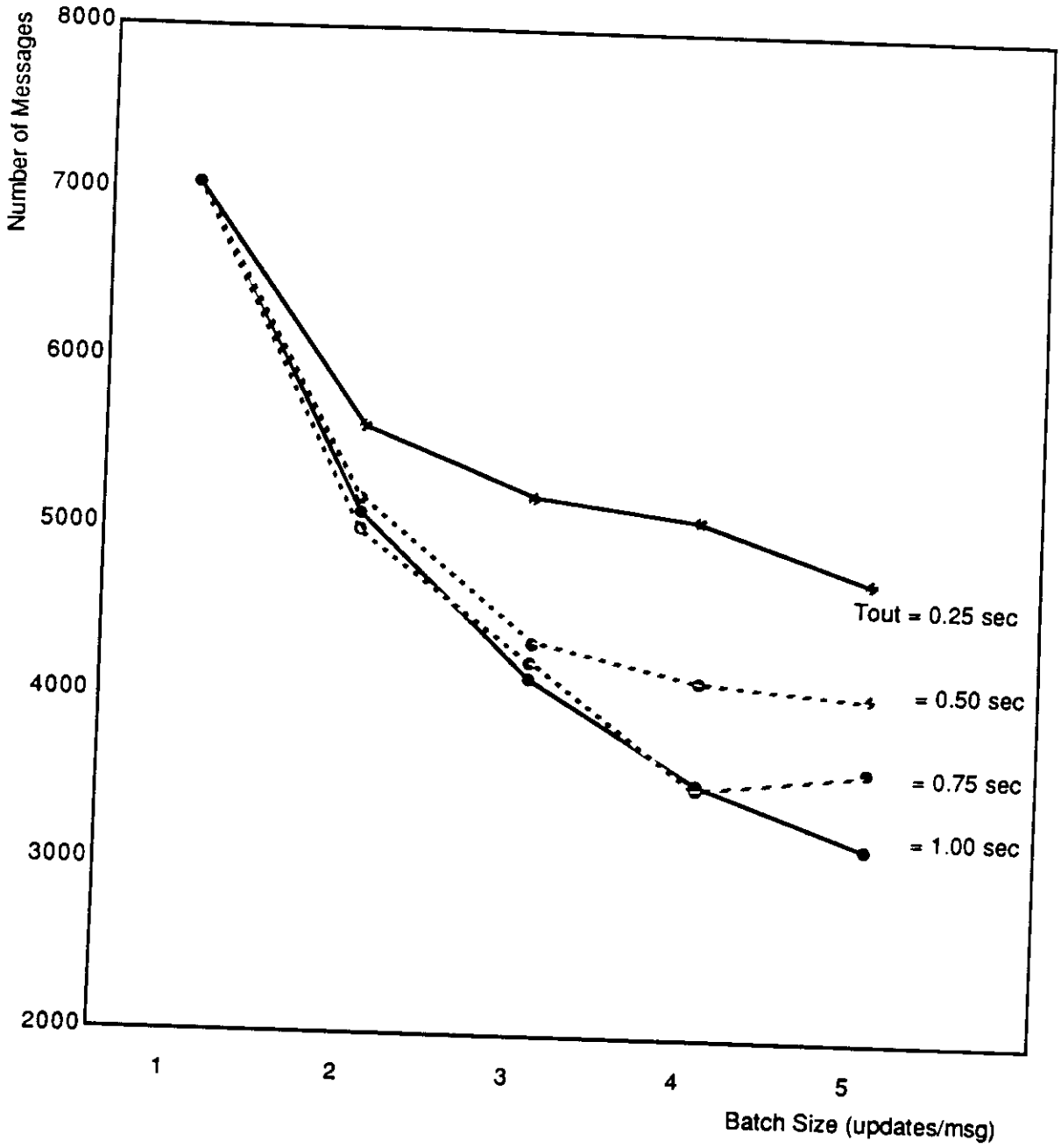


Figure 23

The Number of Messages for TMR
as a Function of Batch Size
and Selected Timeout Periods

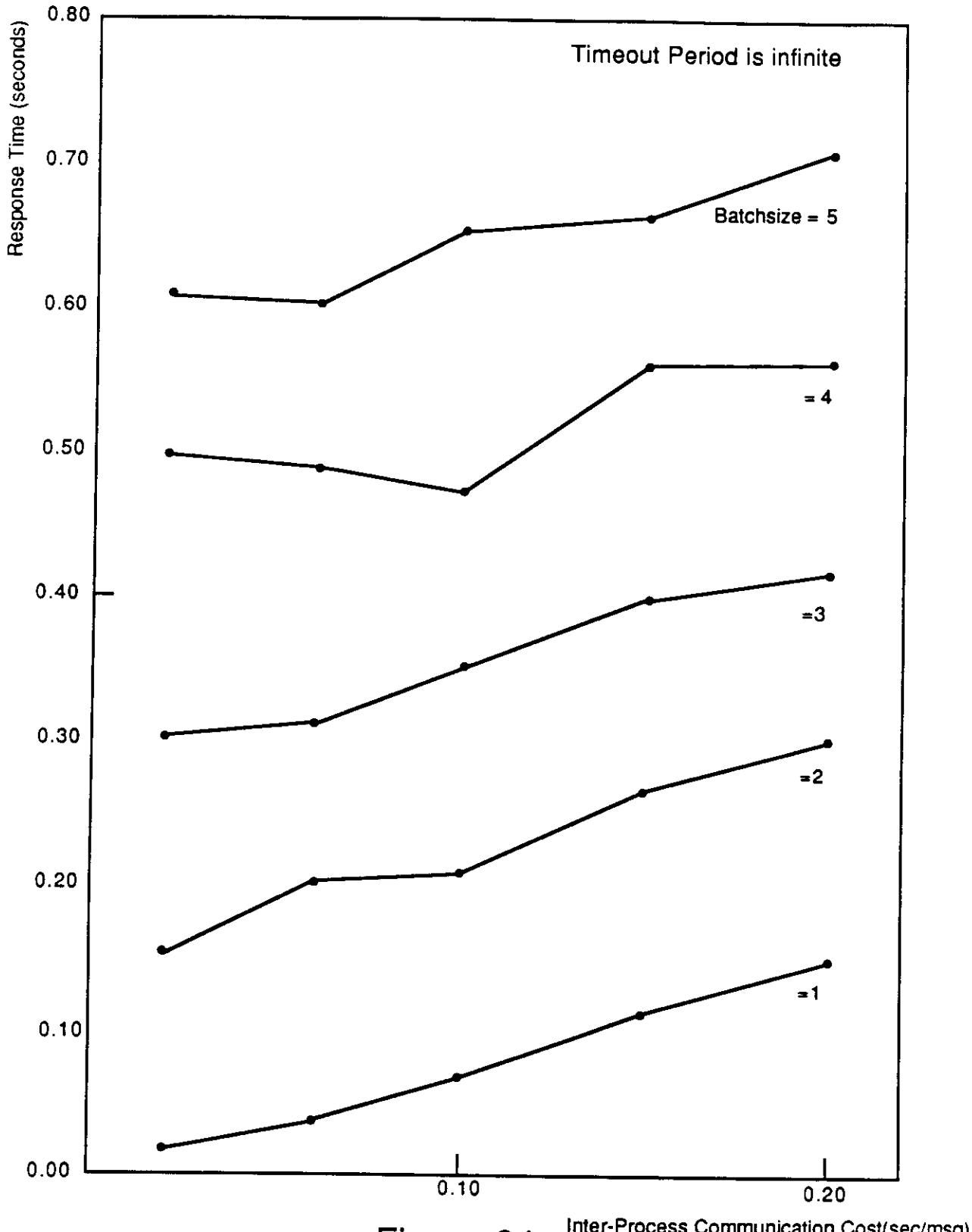


Figure 24

Response Time effect of TMR as a function of IPC

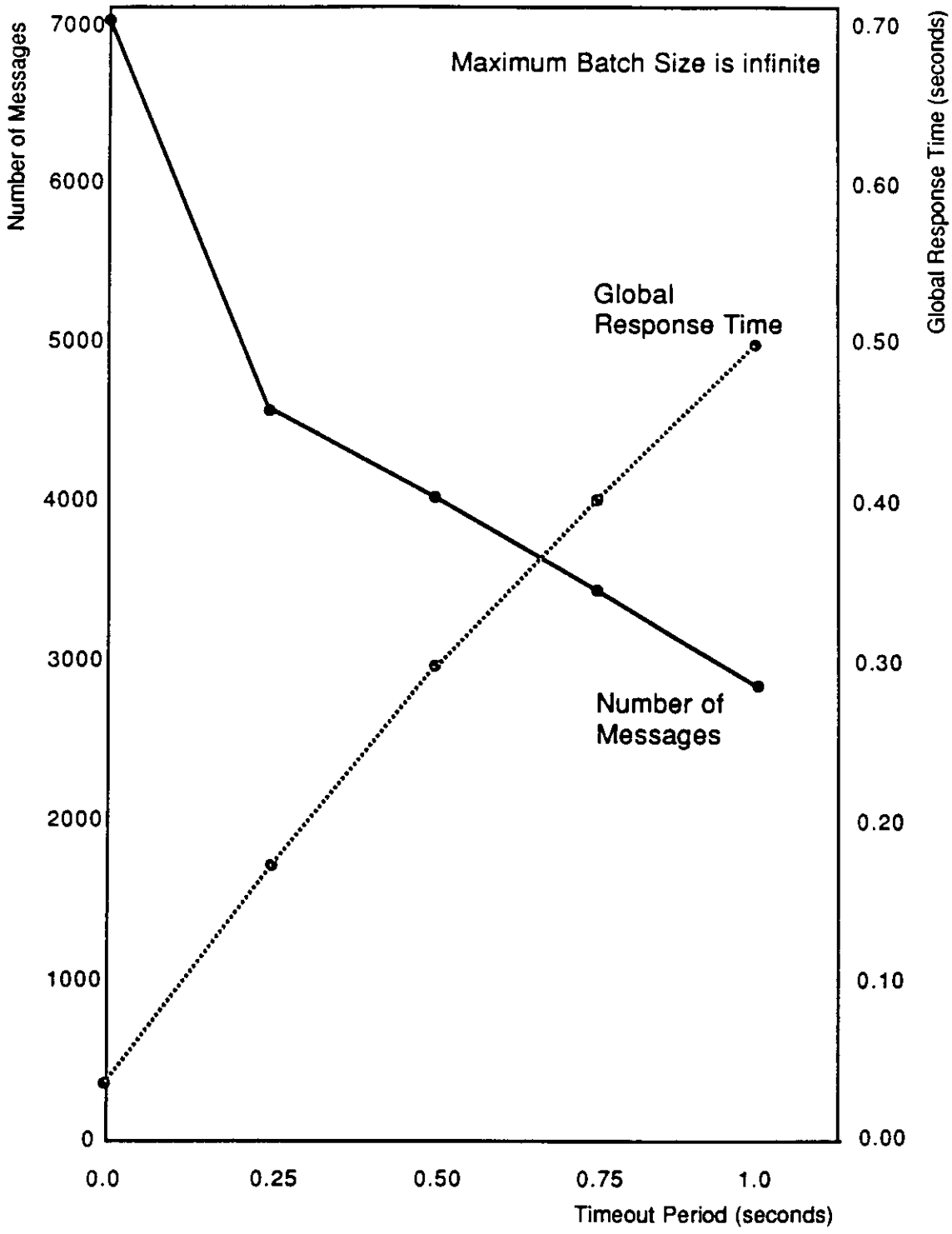


Figure 25

Global Response Time and Number of Messages of TMR as a Function of Timeout Period

CHAPTER VI

A KNOWLEDGE ACQUISITION METHODOLOGY FOR SEMANTIC QUERY PROCESSING

A KNOWLEDGE ACQUISITION METHODOLOGY FOR SEMANTIC QUERY OPTIMIZATION

1 INTRODUCTION

Query processing is a key consideration in database management systems. Conventional approach to query optimization mainly uses domain-independent techniques to transfer the original query to a set of sequences of operations. Optimization usually involves determining the optimal sequence such that the objective function (e.g., response time, operating cost) for processing the given query is minimized [CHU82, JARK84b].

In contrast to conventional query optimization, semantic query processing uses knowledge of application domains to transform the original query into an equivalent one that is cheaper to process yet yielding the same result as the original query. The transformations are limited to those that yield a semantically equivalent query; that is, a query that results in the same answer as the original regardless of the database state.

Much work has been done in developing heuristics and reasoning techniques in semantic query optimization [KING81, XU83, CHAK84, CHAK85]. However, an important area that has been neglected is knowledge acquisition. Integrity constraints are commonly used as the knowledge in semantic query processing. Although integrity constraints have been useful in improving certain queries, their effectiveness is quite limited. A main problem is that they are often too general to be useful in query transformation. This is because integrity constraints are used to ensure that every allowable state of the database is a valid instance of the application, while queries are only concerned with the current database state. Therefore, knowledge about the current database state is more useful than the knowledge of the applications.

Therefore, a new approach for acquiring knowledge of database state is proposed in this research. However, due to the limitation of conventional data models, for example, failure to distinguish different generic relationships among application objects in the record-based database models, the knowledge that can be collected is quite limited. Therefore, we propose to use a semantic data model to facilitate the knowledge acquisition process. Since most semantic data models are interested only in providing various semantic constructs to specify the structural properties of database applications, it usually lacks the knowledge specification in most data models. To remedy this problem, a data model based on ER Model is developed to provide both a set of semantic constructs and the knowledge specification capability. This allows us to automatically acquire semantic knowledge for semantic query processing.

In the sections that follow, we first provide an overview of the semantic query optimization in Section 2. Next, we then discuss the inadequacy of the previous approaches. Section 4 presents the scope of our research proposal using semantic data model to automatically acquire state knowledge of database and also its integration with the semantic query processing. Finally, we outline the development and experimental plans to evaluate the effectiveness of this new approach as compared with the integrity constraints approach.

2 SEMANTIC QUERY OPTIMIZATION

Semantic query optimization is a technique that uses the domain knowledge to improve the performance of query processing. The general approach to semantic query optimization is to transform an original query intelligently into a *semantically equivalent* but more efficient form for processing. Two queries are considered to be semantically equivalent if they produce the same answer in any database state.

Basically, the technique uses a forward-chaining reasoning to induce a new set of constraints from a given query expression, or checks if certain constraints can be induced from other

constraints in the same query expression. By adding or dropping certain constraints, the original query can be transformed into a set of different queries. All of these queries yield the same answer as the original, which are by definition semantically equivalent to the original. Semantic query processor then computes the processing cost for each of the equivalent queries; determines the optimal expression which has the lowest processing cost; and then forwards the optimal one to the conventional query processor to retrieve the answer.

2.1 An Example

Consider a SHIP database management system that monitors the movements of about 10,000 ships. Each ship visits about 20 ports around the world yearly. Assuming such a database contains only three relations: two entities SHIP and PORT and one relationship VISIT. Entities SHIP and PORT have attributes describing the characteristics of each ship and port. The relationship VISIT keeps track of each visit of the ship. The database schema is given as follows:

SHIP = (ShipID, ShipType, Draft, DeadWeight, Registry)

PORT = (PortID, PortType, Depth, Country).

VISIT = (ShipID, Date, PortID, Reason).

The keys for the relations SHIP and PORT are ShipID and PortID respectively. ShipID and Date together provide the key for the relation VISIT.

The database supports 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 1986.*"

Q1 can be answered very quickly by using X and *1986* as the (partial) key to access the VISIT relation. However, other queries (e.g. **Q2**) may need to scan the entire relation.

Q2: "*List the name of the ship with a deadweight greater than 200 thousand tons.*"

To answer this query, we need to access all the 10,000 ship records and check the deadweight of each ship to find which ships satisfy this constraint. This amounts to 10,000 record accesses.

If a clustering index is provided on ShipType, then SHIP can be accessed by the index ShipType. A ship that has a deadweight greater than 150 thousand tons is a supertanker. With this knowledge, the database administrator can transform the query and access the SHIP relation indexing on supertanker. The transformed query is then:

Q2': "*List the name of the ship which is a supertanker with a deadweight greater than 200 thousand tons.*"

If less than ten percent of the ships are supertankers, answering the transformed query will be much faster than answering the original one. Adding the extra constraint replaces a scan of 10,000 ships with accesses of less than a thousand ships, which results in an order of magnitude reduction in retrieval cost. This type of improvement is called *knowledge-based query optimization* or *semantic query optimization.*

2.2 Semantic Equivalence of Queries

Two queries are considered to be semantically equivalent if they yield 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.

Semantic equivalence can be explained with mathematical logic. The idea is to treat the database as a model with the integrity constraint as the set of axioms. User's query is considered as a theorem to be proved. A theorem is true if the query has an answer from the database. Assuming that the theorem is true, new theorems can be generated by applying different axioms. All these new theorems are equivalent to the original theorem; that is, answers satisfying the original query will also satisfy new theorems and vice versa.

For example, suppose there is a semantic integrity constraint that if a ship is a supertanker, its deadweight must be greater than 150 thousand tons. If the database conforms to this condition, then the query **Q2** is semantically equivalent to the query **Q2'** in the previous section. 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, say, a tanker having a deadweight of 180 thousand tons, then these two queries will produce different results. The tanker will be in the answer of **Q2** but not **Q2'**.

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. A violation of integrity constraints means that database contains some values which cannot be attained in the application. Thus, if integrity constraints are valid at all instances of database states, queries transformed with integrity constraints are semantically equivalent.

2.3 Semantic Query Transformation

Two approaches have been proposed to semantic query optimization for query transformation: *interactive* and *compiled* approach. The interactive approach [KING81, XU82] starts with user's query expression from where new constraints are induced and added onto. It uses a forward-chaining reasoning technique, which is similar to term-rewriting in theorem proving [CHAN73]. The method uses the constraints in the query expression to induce new constraints

and adds to the query expression. 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,

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

The compiled approach, on the other hand, started with the definition of a database, induces a set of constraint residues and adds onto the definition of each database relation. The query is then transformed by the constraint residues of the relations involved in query expression. The work was first done by Chakarvarthy [CHAK84] in deductive database. However, it can also be applied to conventional databases.

The method is based on *subsumption* of mathematical logic combining semantic reasoning with the compiled method of accessing techniques of deductive database. In general, a (deductive) database is divided into two components: the *extension database* (EDB) and the *intension database* (IDB). EDB contains the elementary facts while IDB contains the general laws (rules) which define how new facts can be derived from the elementary facts in EDB. EDB may change with time through updates while IDB contains a relatively permanent portion of the database. The compiled access method of deductive databases is to transform each rule in IDB into a form containing only EDB definitions. Each EDB definition is actually a retrieval statement providing a direct access to facts of EDB.

The compiled approach of semantic query optimization consists of two parts: *semantic compilation* and *query transformation*. Semantic compilation is done only once and the procedure is given below:

1. Each rule in IDB is compiled to a form containing only EDB definition.

2. Determine if an integrity constraint is *merge compatible* with a rule in IDB. Merge Compatibility means that the integrity constraint partially subsumes the rule and at least one of the residues (results of partial subsumption) is non-trivial.

3. For each rule, a set of semantically constrained rules is then generated and merged as the form of:

$$H \leftarrow P_1, ..., P_m \{C_1, ..., C_n\}$$

where H is the head of the rule, each P_i is either an EDB definition or an evaluable constraint, and each C_j is the residue which is the result of integrity constraints merging with the rules.

For each user's query, the semantically constrained rules are applied, and the query expression is then merged and factored and produces a set of new queries. With the same reason of semantic equivalence, the new set of queries produces the same answer as the original one.

3 Inadequacies of Previous Approaches

The previous works on semantic query optimization showed its usefulness and provided a good framework for semantic query optimization. However, one of the major problems of semantic query processing is *knowledge acquisition*. That is, how to effectively acquire knowledge to construct the knowledge base for semantic query processing. Currently, integrity constraints are used as the knowledge and is provided by the domain experts which is a time-consuming manual process. To reduce cost and speed up the knowledge acquisition process, an automated approach would be desirable. In the following, we shall discuss the inadequacy of the current approaches and develop a new methodology for acquiring knowledge that is more effective than the integrity constraints and provides facilities for automated knowledge acquisition.

3.1 Inadequacy of Using Integrity Constraints

Integrity Constraints are used as knowledge in the previous semantic query optimization. Integrity constraints must be specified by the database administrators or domain experts who have the expertise of the database applications. Because of the complexity and/or diversity of database applications, it may not be easy for the database designer to provide a complete and valid specification of integrity constraints at the database design stage. Even if the domain expert has the expertise, he or she may not know how to exactly describe it. Due to the dynamic features of the database domains, the validity of the constraints may change with time or environment. To reflect this situation, integrity constraints are usually specified in a general way to cover all the possible values of database domains, which as a result, limits their usefulness as the knowledge for semantic query processing.

Let us consider the following example. Suppose the labor law requires that companies hire only people older than 18 years but does not set any upper limit for the employees' age. An integrity constraint **IC1:** "All employees must be older than 18." may be added to the company's personnel database. Let us consider processing the following query **Q3**:

Q3: "*Which employee is older than 60?*"

Clearly, **IC1** will not be able to speed up in answering **Q3**. However, if the oldest employee in a company X is 60 years old, which limits the age range of the employees in company X to be 18 to 60. If we have this knowledge about the current state of the database, then query **Q3** can be answered "*none*" immediately. Since this knowledge depends on the contents of the database at different times, it cannot be specified as the integrity constraint. This is because integrity constraints is the set of knowledge about the "enterprise" while queries are only concerned with the current instance of the "database". This motivates us to consider using the up-to-date knowledge (*state knowledge*) of the database instead of the integrity constraints for semantic query optimi-

zation. Such knowledge can be acquired through inductive learning from the database contents.

3.2 Limitation of Knowledge Acquisition in Conventional Data Modelling

3.2.1 Machine Learning Techniques

The acquisition of knowledge is one of the most difficult problems in the development of a knowledge-based system. Currently, the acquisition of knowledge is still largely a manual process as follows: A knowledge engineer using the expert system tools transforms the available knowledge into the internal form (knowledge representation) that is understandable by the expert system. It usually involves [MICH83]:

1. studying application literature to obtain fundamental background information,

2. interacting with the domain expert to obtain the expert level knowledge,

3. after all the information is collected, the knowledge engineer translates what he has learned into the knowledge representation for the expert system,

4. through testing and further interaction with the domain expert, the knowledge base is iteratively refined.

Such a manual process is very time-consuming. Furthermore, even if the domain experts have the expertise, they may not be able to describe their own expertise to others. Useful knowledge may not be easy to collect.

A different approach is to use the machine learning technique to construct the knowledge base. Rather than the knowledge engineer learning the application, or the domain experts learning the expert system tools and using their understanding of the application to construct the knowledge base, machine learning technique can provide a means by which the understanding of the application and the creation of the knowledge base are accomplished automatically.

Inductive learning is a technique of machine learning that has been used in different areas of AI research. The problem is the following: Given a concept and a set of training examples representing the concept (the set of examples may include counter examples), find a description for the concept such that all the positive examples satisfy the description (and all the negative examples contradict the description, if counter examples exist). There are two approaches to inductive learning: *interactive approach* and *taxonomical* approach. The interactive approach looks at each training example in sequence and modifies the concept description if the new training example violates the description [MICH83, WINS84]. The approach is to keep a current hypothesis of the concept description. When a positive instance does not match the description, a generalization technique is used to generalize the hypothesis by either dropping a condition (e.g., A & B $\rightarrow$ C is replaced by A $\rightarrow$ C), or adding alternative rules (e.g., B $\rightarrow$ C is merged with A $\rightarrow$ C to get A | B $\rightarrow$ C). When a negative example matches the hypothesis, a technique called specialization is used to add more conditions. The interactive learning technique has been used in modelling the human learning activities.

Taxonomical approach looks at all the training examples at the same time to determine which descriptors are most significant in identifying the concept from other related concepts. This approach recursively determines a set of descriptors that classify each example [QUIN79, MICH83]. The approach is as the following: Select the best descriptor from a set of examples, based on a statistical estimation or theoretical information content. The set of examples is then partitioned into subsets S_1, S_2, ..., S_n according to the values of the descriptor for each example. For each S_i, recursively apply this technique unless each subset contains only positive examples, in which case, the set of descriptors describes the examples set.

Although the automated approach speeds up the knowledge acquisition process, it has been used mainly in applications with a smaller set of training examples. To a database in which a very large volume of data is stored, abusively applying this technique would be a disaster. An

informative database schema would be a useful guide for the knowledge acquisition process. However, because of the lack of semantic expressiveness of the conventional data models, the types of useful knowledge that can be collected would be very limited. This will be discussed in the next section.

3.2.2 Limitation Due to Conventional Data Modelling

Conventional data models, such as *hierarchical*, *network*, and *relational* models, provide record-based data structures for describing the database. A fundamental problem of these models is their limited *semantic expressiveness* [HAMM79]. The record-oriented structure of conventional data models implies the limited modelling capability, and inevitably results in loss of information. Therefore, only a limited portion of the database designer's knowledge of the application will be captured.

To illustrate the limitation of the conventional data models, let us consider the same example as in the previous section. Suppose the constraint **IC1** still holds stating that the ages of employees must be at least 18. **IC1** is still useless in answering the query:

Q4:" *Who is older than 50 in the company?*"

If, however, the employees in the company are divided into three categories: *engineers*, *managers*, and *secretaries*, each category can have a different age constraint associated with it as follows:

(1) *all engineers must be younger than 50*;
(2) *all secretaries must be younger than 40*;
(3) *no restriction on the ages of managers*,

with the additional global constraint **IC1**: "*All employees must be older than 18*". Using these kinds of constraints, the answer to the query **Q4** would be "certain managers" in the company. If

a direct access to tuples of managers is provided, the query can then be processed faster than the original one.

This example shows how the knowledge acquisition is limited by the conventional data model. The limitation arises from the record-based database models failure to distinguish different generic kinds of relationships among application objects. Three generic kinds of semantic relationships have been recognized and should be expressed in a data model [SMIT78, SMIT80, MCLE82, KING86]: *generalization*, *aggregation*, and *classification*. In recent years, there has been much effort devoted to the development of semantic data models [HAMM78, SMIT78, BORD84b, KING84, STON84]. However, most of the data models focus only on structure information which provides semantic constructs for designers to specify the database according to their knowledge about the applications.

The understanding of the enterprise is the designer's knowledge about the database application. At the database designing stage, semantic data modelling provides a tool in developing and using database systems. Intuitively, systems with more knowledge should perform better than those systems without. Although developers of semantic data models have claimed that systems designed with semantic data models are more efficient, seldom have they discussed how the system performance is improved with the use of semantic information. They leave the improvement for users to determine.

This is because during the construction of the database, knowledge other than the structural semantics (database schema) is not used by the database management systems and therefore not saved. However, domain knowledge is useful to query processing. Semantic data models should provide facility for database designers to specify and store their expertise in data dictionary or knowledge base to allow query processor to utilize this knowledge to improve the performance.

Let us use the ship database as an example to illustrate the usefulness of the knowledge used in constructing the database. Ships in the database are divided into different categories according to certain characteristics. One distinguishable characteristic is the deadweight of the ships. The following table lists the range of deadweight for each ship type:

Characteristics Table	
Type	Deadweight (tons)
cruisers	over 10,000
light cruisers	7,000 - 10,000
destroyers	3,000 - 7,000
frigates	1,100 - 3,000
corvettes	500 - 1,100

The characteristics that are used to divide the ships into different types are also part of semantic knowledge. These characteristics which are used to define the schema for the ship database are also useful in answering queries like "Which ship has deadweight over 7,000 tons?" The answer must be some cruisers (including light cruisers). Without this knowledge, a scan of database is unavoidable. This example illustrate the usefulness of saving the semantic knowledge that was used in constructing the database. A complete modelling tool is needed to provide the knowledge specification with the schema specification.

4 SCOPE OF RESEARCH

Semantic query optimization using record-based conventional data modelling is rather restricting because of the limited expressive capability. There is no good technique provided in utilizing the semantic knowledge for semantic data modelling. Semantic data models provides a useful tool and also a friendly user interface for modelling and the use of the database systems. Works are needed to extend and develop a semantic database management system that uses semantic information to improve system performance.

A semantic database management system (SDBMS) is intended in this research to combine the semantic query processing and semantic data modelling to gather knowledge and carry out query optimization. At the data modelling level, a knowledge-based ER model is provided for specifying and using the database. At the system level, Semantic Query Processor utilizes reasoning and semantic knowledge to optimize the queries. An automated knowledge acquisition mechanism is proposed which uses the database schema as a guidance to induce a set of useful knowledge from database contents through inductive machine learning. SDBMS also provides a knowledge editor (KED) to allow domain experts to refine the knowledge base. Combining *semantic query processing* and *semantic data modelling* with *automated knowledge acquisition mechanism* will yield substantial improvement in query processing performance. We plan to develop a prototype SDBMS system and measure the performance improvement over the conventional database management systems.

In the following, we first introduce the KER Model and show that it not only is a tool for specifying the database applications, but also provides a way to systematically acquire knowledge. Next, we then describe a knowledge acquisition methodology based on KER and finally present the proposed SDBMS architecture.

4.1 A Knowledge-based E-R Model (KER)

A data model is for specifying the structure of database and operations for performing on the data. Record-based data models, limited due to the simple data structure, fail to distinguish different generic relationships among application objects. Three fundamental generic semantic relationships that should be provided in data models are: *generalization/specialization* , *aggregation*, and *classification*. Generalization defines an object type from a set of objects (subtypes) which corresponds to a bottom-up construction of a hierarchy (e.g., ANIMAL is a union of DOG, CAT, HUMAN, etc.); while specialization defines a subtype of an object which corresponds to a top-down construction of a hierarchy. *Aggregation* defines an object as a *rela-*

tionship among objects. Therefore, generalization/specialization and aggregation correspond to the set theoretic operations of "union" and "cartesian product" respectively.

4.1.1 The Need of Knowledge Specification

To facilitate the knowledge acquisition process, we need a tool to systematically collect useful knowledge. While dealing with semantic query processing, data models with the above-mentioned constructs are not enough. An important property that should be made explicit but generally implicit in data modelling is the *with-constraint* knowledge specification. For example, one can define a PROFESSOR as a subtype of PERSON (specialization) as:

PROFESSOR **isa** PERSON **with** duty = "teaching".

The **with** clause defines the constraint associated with the subtype PROFESSOR which is the knowledge to distinguish professors from other types of persons.

The *with-constraint* information is actually the database designer's knowledge used to construct the database schema. Our KER model is an extended E-R Model with the extension of *with-constraint* knowledge specification. The specification of *with-constraint* is optional. However, it can be served as integrity constraints to enforce the integrity checking. Or, it can be filled with the induced knowledge by the knowledge acquisition mechanism discussed in Section 4.2. In both cases, the specified or induced knowledge will be used by the query processor to reduce the query response time.

4.1.2 Schema Specification in KER

The basic constructs in KER are: **isa-with** and **has-with**. Isa defines an Entity hierarchy (e.g., ENGINEER **isa** EMPLOYEE, EMPLOYEE **isa** PERSON); while **has** defines an aggregation relationship (e.g., DEPARTMENT **has** MANAGER). **With** clause specifies the property that has to be satisfied with the definition of **isa** or **has**. In KER Model, *entities* and *relation-*

ships are the principal concepts. An entity is an object which can be distinctly identified. A specific person, a department, a course are examples of entities. An entity set is a collection of entities where each entity is distinguished by a unique identifier. The set of unique identifiers is called the primary key to the entity set.

The entities are classified into different entity types E_i, and each type is associated with some constraint predicate that tests whether an arbitrary entity belongs to this entity type. An entity set E_1 can also be a subset of another entity set E_2 satisfying certain constraint Ψ, which is defined as:

$$E_1 \textbf{ isa } E_2 \textbf{ with } \Psi.$$

A hierarchy example is "*PERSON consists of subtypes PROFESSOR, STUDENT, STAFF.*" That is, the personnel in a department is divided into three disjoint categories: professors, students, and supporting staff. For each subtype, there is also a constraint restricting allowable objects of this subtype. For example, a professor must have a Ph.D. degree, so there is a predicate stating that degree is "Ph.D." for the subtype PROFESSOR.

A relationship is an aggregation among entities. For example, the relationship TEACHING is defined as an aggregation of entities PROFESSOR and COURSE. The relationship type is a relation among *n* entities which is defined as:

$$R_i \textbf{ has } e_1 \in E_1, e_2 \in E_2, ..., e_n \in E_n \textbf{ with } \Psi.$$

An entity in a relationship expresses the role the entity performs in the relationship. In a relationship set, TEACHING is defined between entities from the entity sets PROFESSOR and COURSE, e.g.

$$\text{TEACHING } \textbf{has } p \in \text{PROFESSOR}, c \in \text{COURSE } \textbf{with } p.\text{DEPT} = c.\text{DEPT}.$$

The first element in the tuple may appear in the role PROFESSOR and the second in the role COURSE, while *with portion* states that both departments have to be the same.

The definitions of entity and relationship types and subtypes as well as the associated constraints will be kept in the knowledge database (KB) for semantic query processing. Figure 1 shows a school database example.

A relational table form is used to represent the entity relations and the relationship relations. One possible implementation of type hierarchy is to provide indexing on its supertype. For example, the type hierarchy "PERSON with subtypes PROFESSOR, STUDENT, and STAFF" is realized by providing a maximal space for the subtypes so that a type hierarchy is squeezed into a single table. An index on PERSON_TYPE is provided to access each subtype. Alternative implementation is to use a table for each subtype entity. The key of the entity type will also be the key to the subtype entities, which provides a link between the entity type and its subtypes. We plan to compare the performance tradeoff between these two approaches.

4.2 A Model-based Knowledge Acquisition Methodology (KAM)

Database semantics can be divided into two categories: *database enterprise knowledge* and *database state knowledge*. Enterprise knowledge refers to the semantics of the database application. Integrity constraints are part of the enterprise knowledge. Database knowledge, on the other hand, is an instance of the enterprise knowledge which is more concerned with the current database contents. For example, enterprise knowledge may specify that the ages of the employees must be older than 18, while database state knowledge contains the knowledge that the ages are in the range of 18 to 60.

Entity/Relationship Sets	*Attributes*	*Domains*	*Constraints*
PERSON	name	NAME	
	address	STREET × CITY	
	phone_number	NUMBER	ψ_1
	birth_date	DATE	
	status	STATUS	
	degree	DEGREE	
COURSE	number	COURSE_NUMBER	
	dept	DEPARTMENT	
	classroom	ROOM	ψ_2
	schedule	TIME	
TEACHING	instructor	PROFESSOR	
	course	COURSE	ψ_3
	grader	STUDENT	
PROFESSOR	name	NAME	
	paper_published	NUMBER	PROFESSOR isa PERSON
	office	ROOM	with ψ_4
	DEPT	DEPARTMENT	

PERSON(name, address, phone_number, birth_date, status, degree).
COURSE(number, dept, classroom, schedule).
TEACHING(instructor, course, grader).

PROFESSOR(name, paper_published, office, dept).

PROFESSOR **isa** *PERSON* **with** *degree = "Ph.D." and status = "teaching"*

STAFF(name, office, salary, rank).

STAFF **isa** *PERSON* **with** *status = "accounting" or "personnel" or "administrating"*

STUDENT(name, advisor, program, years).

STUDENT **isa** *PERSON* **with** *status = "studying" and (degree = "BS" or "MS")*

FULL_PROFESSOR(name).
ASSOCIATE_PROFESSOR(name).
ASSISTANT_PROFESSOR(name).

FULL_PROFESSOR **isa** *PROFESSOR* **with** *paper_published > 30*
ASSOCIATE_PROFESSOR **isa** *PROFESSOR* **with** *10 ≤ paper_published ≤ 30*
ASSISTANT_PROFESSOR **isa** *PROFESSOR* **with** *paper_published < 10*

Figure 1. A Database Example.

In general, queries are only concerned with the current database state. It is unreasonable to restrict the query transformation only to those yielding semantically equivalent queries. In this research, we relax the definition of *semantic equivalence* by allowing two queries to be

"loosely" semantically equivalent if they produce the same answer at the *current* database instance. That is, instead of using the *enterprise* as the scope of semantical equivalence, we use the current *database state* instead. Intuitively, if the knowledge induced from the database is consistent with the current database state, the query transformed with this knowledge should produce the same answer at the current state. The advantage of using the induced knowledge is that it is more specific in describing current database instance. Induced knowledge should be more effective than integrity constraints for semantic query optimization.

Since the database state may change by the updates, it may need to update the knowledge base accordingly to reflect the state change. However, there is also cost associated for integrity checking when using integrity constraints for query transformation. The tradeoff between these two approaches depends on the update/query ratio and the cost of updating knowledge base and that of integrity checking. If the cost for updating the knowledge base is comparatively small as compared to integrity checking, or the ratio of update/query is relatively small, then using state knowledge in query transformation yields better performance improvement for query processing than using integrity constraints.

We propose to use the database schema as a guide to induce semantic knowledge from database contents. Knowledge acquisition can be divided into two steps. The first step is to induce a set of knowledge from current database state by *taxonomical* inductive learning. The second step is to refine the knowledge base either modifying the rules by the domain experts or by the *interactive* machine learning when update occurs.

Using the taxonomical approach without guidance may generate meaningless knowledge. Heuristics is needed to guide the knowledge acquisition process. In accessing a database, key is faster than index and index is faster than a sequential search. Semantic query processing explores different opportunities that reduce the search space for query processing. We define a *target attribute* as an attribute which is a key or partial key to some entity or relationship.

Knowledge is acquired based on target attributes to provide opportunities in restricting the search space of a query, which is the basic concept of our knowledge acquisition methodology.

In a schema based on KER model, there are two types of objects: entities and relationships. There are two types of semantic knowledge which are useful to query processing: *domain knowledge* and *structural knowledge*. Domain knowledge is related to the attribute domains, whi(restricts the allowable objects of the entity or relationship sets. Each attribute in an entity or a relationship is bounded by a certain range. For example, the age of a person is in the range of (0 - 120), and the salary of an employee is in the range of (10,000 - 100,000), etc. Thus, *domain knowledge* specifies the static properties of entities and relationships.

Structural knowledge specifies the structural properties of the database schema. A database schema consists of entities and relationships. Each entity is either a stand-alone entity or an entity in an entity hierarchy. Each relationship is an aggregation of entities while each entity plays a role in the relationship. Structural knowledge specifies the semantics among these objects which can be further divided into *intra-* and *inter-structure knowledge*.

Intra-structure knowledge specifies the relationship between attributes within the object (an entity or a relationship). Functional dependency is an intra-structure knowledge example. Several entities can be aggregated into a relationship according to certain semantic constraints which is specified as the inter-structure knowledge. For example, the relationship TEACHING aggregated by the entities INSTRUCTOR, STUDENT, and COURSE contains a constraint that the course offered by the instructor must be in his department. This inter-structure knowledge is induced from the inter-relationship between INSTRUCTOR and COURSE linked by the TEACHING relationship.

After classifying different types of knowledge and defining the target attributes for knowledge acquisition, we shall now describe the Knowledge Acquisition Methodology (KAM)

which consists of three stages: *schema generating*, *automated knowledge acquisition*, and *knowledge base refinement* as follows:

1. Schema Generating:

   In this step, DBA uses KER to define database schema which includes:

   a. Identify entities and associated attributes.

   b. Identify entity hierarchies. The key of each entity is designated as a target attribute. If the database already exists, use the clustering indexes to define subtype entities. The indexes are the target attributes.

   c. Define aggregation relationships. Designate each of the referential keys as the target attributes. A referential key is the attribute of a relationship which is a key to some entity.

2. Automated Knowledge Acquisition:

   a. For each entity/subtype entity determine the *domain constraint* for each attribute.

   b. Use the *taxonomical* machine learning approach to induce *inter-structure* and *intra-structure* knowledge related to the target attributes from the database.

3. Knowledge Base Refinement:

   a. Whenever there is an update to the database, a verification will be made against the knowledge base to see if the update violates the rules. *Interactive* machine learning modifies the violated rules by generalizing them. If the system has gone through quite a few updates, DBAs may repeat the taxonomical machine learning (Step 2) to reconstruct the knowledge base.

b. Domain experts use the knowledge editor (KED) to refine the state knowledge in the knowledge base to improve the system performance. Since integrity constraints represent the most general knowledge, the refinement cannot violate the set of integrity constraints.

Unlike the manual approach to knowledge acquisition, KAM uses the database schema to guide the taxonomical learning process and induces knowledge from database contents. Such automated process reduces the time for knowledge acquisition. Furthermore, the knowledge base provides an up-to-date state information about the database which is more effective than integrity constraints for semantic query processing. Using our SHIP database as a testbed, experiments will be performed using the KAM to collect knowledge. We plan to measure the cost of acquiring knowledge using such an approach, the effectiveness of the induced knowledge for improving the query processing performance, and the cost for updating contents of the knowledge base.

4.3 SDBMS Architecture

After introducing the KER model, the semantic query processor, inductive learning, and the knowledge acquisition methodology, we now present the SDBMS architecture that integrates these components as shown in Figure 2.

The functionality of each component is listed below:

DB (*database*): the physical database that contains a large amount of facts.

KB (*knowledge base*): stores the up-to-date knowledge induced from the database contents or provided by the domain experts. Knowledge is represented in the form of Horn Clause.

SQP (*Semantic Query Processor*): performs the task of semantic query optimization based on the knowledge from KB.

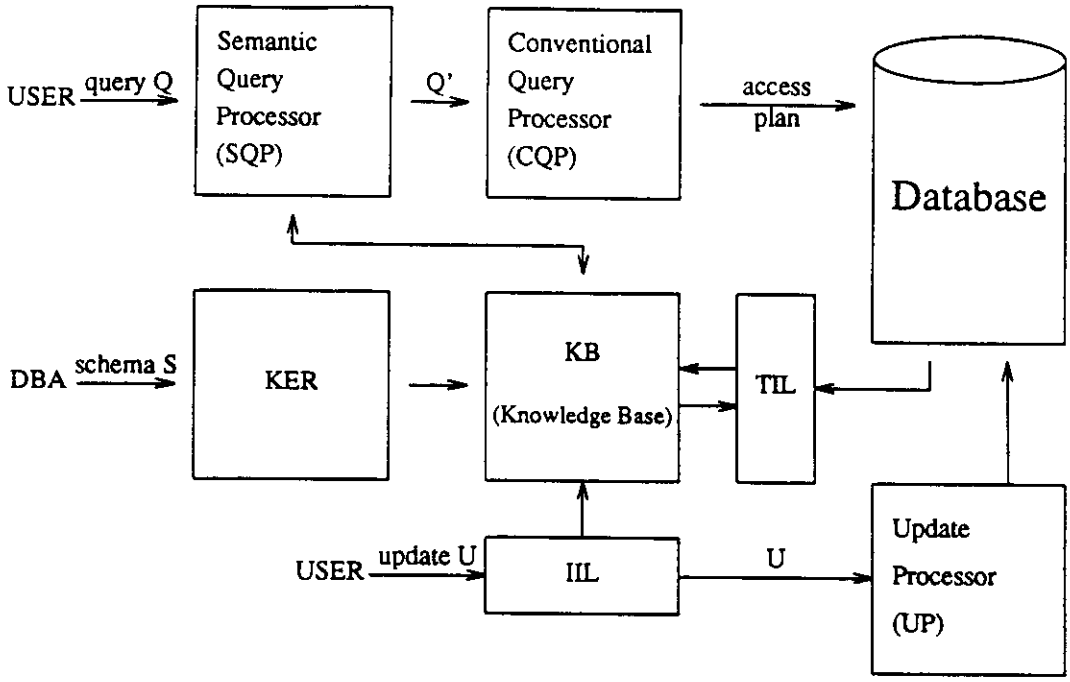


Figure 2. Architecture of Semantic Database Management Systems

Figure 2. Architecture of Semantic Database Management Systems.

CQP (*Conventional Query Processor*): uses a domain independent approach to transform a query into a set of sub-queries and determines an optimal access plan for retrieving the answer.

UP (*Update Processor*): updates the database contents.

KER (*Schema Definition Tools*): provides DBA a set of semantic constructs to define the database schema and also is the model used by TIL (Taxonomical Inductive Learning Subsystem) to induce knowledge.

TIL (*Taxonomical Inductive Learning Subsystem*): induces sets of knowledge from data-

base content according to KER schema definition.

IIL (*Interactively Inductive Learning Subsystem*): updates KB whenever an update violates certain rules in KB. IIL provides a partial functionality of integrity checking, which can be enhanced as a complete integrity checking subsystem. IIL also provides a knowledge editor (KED) to allow domain experts to refine the content of KB interactively.

DBA (*Database Administrator*): defines the database schema and verifies the meaning and effectiveness of the knowledge in KB. DBA also refines the knowledge in KB.

USER (*Person using the database*): issues queries to retrieve answers from or to update database contents.

According to the KAM methodology, the acquisition of knowledge is done in three steps: In the Schema Generating Step, the DBAs (Database Administrators or domain experts) uses the proposed KER specification facility to define the database schema. The *with-constraint* portion can be ignored at this point. The schema specification is stored in KB (Knowledge Base or data dictionary) for later use by TIL to induce knowledge. After data is read into the database, it enters into the Automated Knowledge Acquisition Step. In this step, TIL (the Taxonomical Inductive Learning subsystem) induces a set of knowledge from database contents with the help of the database schema. This set of knowledge will be stored in KB as part of the knowledge specification of KER schema specificaion, and used by the SQP (Semantic Query Processor) for transforming users' queries.

The knowledge base (KB) can be refined/updated automatically or by domain experts. This is done through IIL (Interactive Inductive Learning subsystem). IIL verifies each update is-

sued to see if it violates any knowledge specification in KB and modifies the specification if violation occurs. IIL also provides domain experts a knowledge editor KED to refine the knowledge specification in KB. This is the Knowledge Base Refinement Step in KAM methodology.

4.4 Implementation and Future Research

At UCLA, we have a naval SHIP database. Ships are the central entities in the database. The database contains information about 1,000 ships. It was developed by Unisys SDC, Santa Monica, California, through collaboration with the Naval Oceanic System Center in San Diego. The database provides an unclassified yet fairly realistic naval database operational characteristics for experimental study. The following is some of the operational characteristics: Ships may be assigned to battle groups. The warfare roles of ships depend on their class characteristics and the installed weapons systems. The ship and/or its weapon systems may be overhauled in a shipyard. Ships have a base port and may visit ports for purposes other than scheduled overhauls. Ships also have positions which are reported (for US ships) or sighted (for Russian ships). The movements of the ships are generated by the computer programs. The database consists of 17 relations and about 30,000 tuples and is in INGRES format. The database is currently running on a MicroVax machine using Ultrix V1.2 operating system (a UNIX 4.2 compatible system).

To study the performance improvement of query processing from SDBMS, we are using the SHIP database as a testbed. To carry out this research, we are developing a SDBMS prototype and plan to set up an experimental environment to measure the behavior and performance. The implementation of the prototype is divided into six tasks:

1. Developing the *Data Definition* facility of KER which provides a language for designer to specify database schema.

2. Developing a Taxonomical Machine Learning subsystem and coupling it with KER to

provide a knowledge acquisition tool.

3. Applying the knowledge acquisition tool to induce semantic knowledge from the SHIP database.

4. Developing a Semantic Query Processor by using both interactive and compiled approaches and comparing their performance.

5. Developing an Interactive Machine Learning subsystem to provide a learning capability.

6. Developing a Knowledge Editor (KED) to allow domain experts to verify and modify the knowledge base.

Currently we are implementing the Semantic Query Processor and using the integrity constraints to build the knowledge base to study the performance improvement of semantic query processing. In the next stage of research, we plan to develop the KER specification language and the machine learning subsystem and then apply the KAM methodology to build the knowledge base. We shall then measure the performance improvement of query processing by the induced knowledge over integrity constraints. Different types of queries will be used to measure the query processing performance from simple single-relation queries to more complex queries containing multiple relations and selected combinations of disjunctive/conjunctive constraints. We shall also measure the cost for updating the knowledge base for selected query/update ratio. The result of these experiments will help us understand the cost of acquiring knowledge, the performance gain in using the induced knowledge, and identify the types of knowledge that can be used to improve the performance of query processing.

5 REFERENCES

[BROD84a] Brodie, M., Mylopoulos, J., and Schmidt, J. W., (eds.) *On Conceptual Modelling. Perspectives from Artificial Intelligence, Databases, and Programming Languages*, Springer, New York, 1984.

[BROD84] M. Brodie and D. Ridjanovic, "On the Design of Database Transactions," in *On Conceptual Modelling*, Spring-Verlag, New York, 1984, pp. 277-312.

[CHAK84] Chakravarthy, U. S., Fishman, D. H., and Minker, J., "Semantic Query Optimization in Expert Systems and Database Systems," *Proceeding First International Workshop on Expert Database Systems*, Kiawah Island, October 1984.

[CHAK85] Chakravarthy, U. S., *Semantic Query Optimization in Deductive Databases*, Ph.D. Thesis, Department of Computer Science, University of Maryland, College Park, August 1985.

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

[CHEN76] Chen, P.P.S., "The Entity-Relationship Model: Toward a Unified View of Data," *ACM Transactio on Database Systems*, Vo. 1, No. 1, March 1976.

[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.

[GALL78] Gallaire, H. and Minker, J. (eds.) *Logic and Data Bases*, Plenum Press, New York, 1978.

[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.

[HAMM81] Hammer, M., and McLeod, D., "Database Description with SDM: A Semantic Database Model," *ACM Transactions on Database Systems*, Vol. 6, No. 3, September 1981.

[JARK84a] Jarke, M., Clifford, J., and Vassillou, Y., "An Optimizing Prolog Front-End to a Relational Query System," *Proc of ACM SIGMOD*, 14, 2, Boston, pp. 296-306, June 1984.

[JARK84b] 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.

[KING84] King, R. and McLeod, D., "A Unified Model and Methodology for Conceptual Database Design," in *On Conceptual Modelling*, Spring-Verlag, New York, 1984, pp. 313-331.

[KING86] King, R. and McLeod, D., "Semantic Database Models," in S. B. Yao (ed.) *Principles of Database Design*, Prentice-Hall, Englewood Cliffs, N. J. 1986.

[MCKE82] McLeod, D., and Smith, J. M., "Abstraction in Database," *Proc. Workshop on Data Abstraction, Databases, and Conceptual Modelling, SIGMOD Record*, Vol. 11, No. 2, February 1981.

[MICH83] Michalski, R. S., et al, (eds.) *Machine Learning: An Artificial Intelligence Approach*, Tioga Press, Palo Alto, 1983.

[NICO78] Nicolas, J. M., and Gallaire, H. "Data Base: Theory vs. Interpretation," in [GALL78].

[QUIN79] Quinlan, J. R., "Induction Over Large Data Bases", STAN-CS-79-739, Stanford University, 1979.

[SMIT78] Smith, J. M. and Smith, D. C. P., "Principles of Conceptual Database Design," *Proc. NYU Symposium on Database Design*, New York, May 1978.

[STON84] Stonebraker, M., "Adding Semantic Knowledge to a Relational Database System," in *On Conceptual Modeling*, ed. M. Brodie, J. Mylopoulos, and J. Schmidt, Sprintger-Verlag, 1984.

[WINS84] Winston, P. H., *Artificial Intelligence*, Addison-Wesley, Massachusettes, 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.

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