DATABASE MANAGEMENT ALGORITHMS FOR ADVANCED BMD APPLICATIONS

Wesley Chu

1984
CSD-840031

DATABASE MANAGEMENT ALGORITHMS FOR ADVANCED BMD APPLICATIONS

FINAL REPORT FOR THE PERIOD

FROM: February 1, 1983

TO: January 31, 1984

Contract No. DASG 60-79C-0087

Prepared For:

Ballistic Missile Defense Advanced Technology Center

Huntsville, Alabama 35807

April 30, 1984

University of California, Los Angeles

Wesley W. Chu, Principal Investigator

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

CONTENTS

CHAPTER I

INTRODUCTION AND SUMMARY

I. INTRODUCTION AND SUMMARY

During the past year, we have focused our research efforts in the following areas: concurrency control for replicated shared files in BMD systems; heuristic task assignment for BMD applications; resilient commit protocol for shared files; and the fault-tolerant locking for shared-memory systems. Each of these is briefly discussed below.

1. Concurrency Control for Replicated Shared File in BMD Systems

We have developed the exclusive-writer protocol (EWP) for concurrency control for replicated shared files in BMD system. EWP avoids inter-computer synchronization delay and requires low implementation cost. However, when there are conflicting updates, EWP accepts only one update and discards the others. For many BMD application files (e.g. track files), this is acceptable. But, certain files may not tolerate with discarding of conflicting updates. To remedy this, the EWP is extended with a locking option to avoid discarding of conflicting updates; the new protocol is called EWL. Intercomputer synchronization delay is incurred when such an option is activated. The EWL works exactly like the EWP when there is no update conflict. When any update conflict exists, EWL behaves like the Primary Site Locking (PSL).

To compare the performance of various concurrency control protocols, we study the response delay in terms of the execution response time and update finalization response time of EWP, EWL, PSL, and basic timestamp. The key parameters are update conflict probability, cost of locking, sizes of control message and update message. The comparison results are presented.

EWP has no transaction restarts, database rollbacks, or deadlocks due to shared data access. But EWP ensures only a limited form of serializability. EWL is an extension of EWP that ensures full serializability. EWL has no database rollbacks, either. Also, EWL guarantees that a transaction will be restarted at most once. To further reduce restarts, each site can independently and dynamically switch between primary site locking and EWL.

We conclude that if limited serializability is acceptable, EWP should be used due to its simplicity and short response time. Otherwise, EWL has much appeal since it has good performance over a wide range of parameter values and permits dynamic switching with PSL to further improve performance.

2. Heuristic Module Assignment

Our previous research results on measurement and estimation of intermodule communication (IMC) provide us important insights into the task assignment problems. Module assignment problems usually require excessive computation and exact solution for a large number of modules (e.g. 25) and a large number of computers (e.g.10) is computationally infeasible. This motivates us to propose a heuristic technique for module assignment.

An objective function based on the concept of minimizing *bottleneck* (the utilization of the most heavily loaded processor) is proposed as a criterion for module assignment to achieve minimum port-to-port time. This function has been used to find a good assignment for the DPAD system, via an *exhaustive search* through the entire space of all possible module assignments.

To reduce computation, a technique based on IMC data is proposed to combine modules into clusters. Using the UCLA DPAD simulator we compare the module assignment generated from the heuristic algorithm with that generated from the exhaustive search. We noted that the heuristic algorithm yields good assignment. Currently, we are further investigating the effect of precedence relationship on port-to-port time and developing techniques to incorporate the precedence into our heuristic module assignment algorithm.

3. Resilient Commit Protocol

The process of posting an update is known as commit. Two phase commit is a well known method that ensures mutual consistency among file copies in case of failures during update. This protocol is classified as blocking commit because when a coordinator site (the site that originates the update) fails during the update broadcast, other sites are blocked from using the file copies until the failed coordinator recovers and completes the unfinished commit work. This technique has been implemented in several systems. However, for real-time systems, blocking commit is not acceptable. Therefore we propose a low cost non-blocking protocol that is resilient to multiple failures. In the proposed commit protocol, sites are linearly numbered and updates are broadcast in one phase according to this number sequence. Failures are recovered by the smallest numbered surviving site. Further, we also show how to incorporate this commit protocol into the existing concurrency control techniques such as EWP and PSL, and discuss the site recovery procedure.

4. Fault-Tolerant Locking Protocol For Shared-Memory Systems

To provide fault tolerance for shared-memory systems, the Fault-Tolerant Locking (FTL) protocol has been proposed. FTL maintains the mutual consistency of replicated file copies and the internal consistency of all shared files in the event of failures of computers, shared memory modules, and/or communication links. An implementation of FTL for the ARC multi-μprocessor testbed is presented. This implementation is based on the copy of the testbed application program dated November 4, 1982. Four experiments are proposed to assess the performance of the FTL protocol in the testbed. These experiments are currently being evaluated by the System Development Corporation (SDC) at Huntsville, Alabama.

CHAPTER II

THE EXCLUSIVE-WRITER APPROACH TO UPDATING REPLICATED FILES IN DISTRIBUTED PROCESSING SYSTEMS

THE EXCLUSIVE-WRITER APPROACH TO UPDATING REPLICATED FILES IN DISTRIBUTED PROCESSING SYSTEMS

1. INTRODUCTION

Distributed processing has commonly been used because of its flexibility, reliability, and cost effectiveness. Distributed processing has many design implications. One of these is preserving file consistency, which arises when two or more transactions (or *transaction modules, TM)* concurrently access the same file in a conflicting manner (i.e., at one access is a write). This problem is further complicated since files may be replicated to reduce read access times and to improve reliability.

To preserve file consistency, *consistency control protocols (CCP)* are used (see surveys in [BERN81] and [LIN83]). Distributed processing systems need CCP with short response times. Such response times are measured from the transaction's arrival to when its file updates are available. In some applications, file updates need only be tentative (e.g., radar tracking in which object positions are continuously changing). So, we use the following response time measure.

Execution Response Time (T_E): the time from the transaction's arrival until it has completed its first execution, thereby making available tentative file updates.

For other applications (e.g., banking transactions), a transaction must be assured that previous updates to its input files are finalized (i.e., will not be undone due to database rollbacks) before the files can be used. Here we use,

Update Finalization Response Time[1] (T_U): the time from the transaction's arrival

[1] The site that first knows an update is finalized depends on the CCP. In particular, for some CCP this site may not be the transaction's execution site. Our reason for not always using the transaction's execution site relates to the fact that distributed processing applications are often performed by a sequence of transactions (e.g., [GREE80]). So, communication delays are reduced by assigning a transaction to the site which first knows that the updates of its predecessor have been finalized.

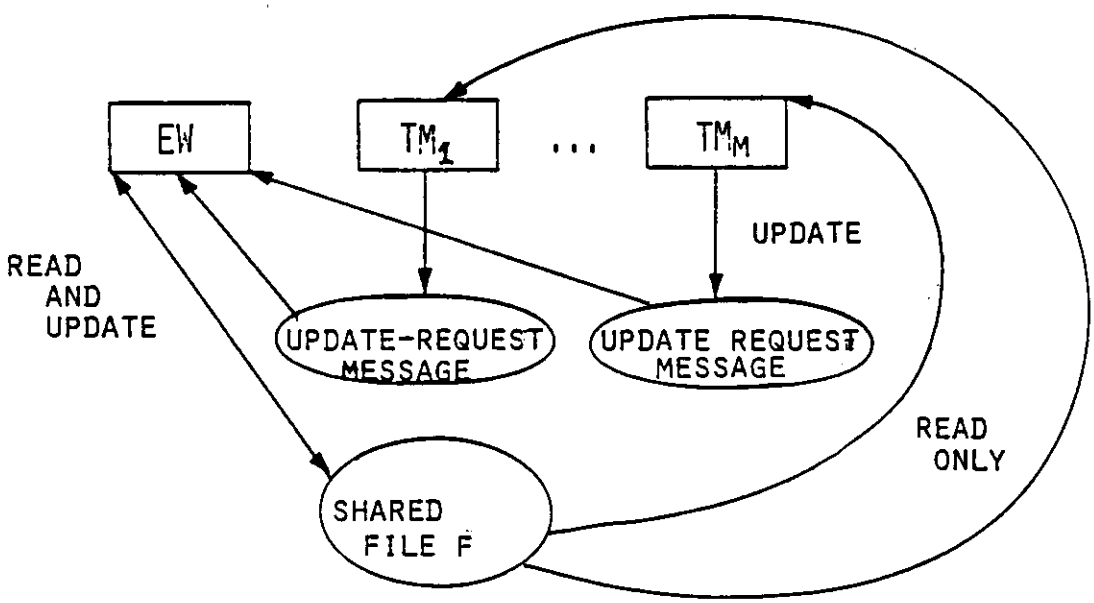
until its update is known to be finalized at some site.

CCP can be characterized by whether they check for conflicting file accesses *before* or *after* transactions access shared files. Protocols that check after shared files are accessed (e.g., [KUNG79], Basic Timestamps [BERN81], and the majority vote algorithm [THOM78]) are seen as *optimistic* since they are optimistic that a conflict will not occur. Conversely, protocols that check before permitting shared file accesses (e.g., primary site locking [STON79], SDD-1 timestamp protocols [BERN78], and centralized locking [GARC78]) are called *pessimistic*. Pessimistic protocols delay completing a transaction's first execution, or execution response time, T_E, until messages are exchanged for conflict checking which causes *inter-computer synchronization delays (ICSD)*. Optimistic protocols avoid such delays for T_E. But existing optimistic protocols repeatedly restart a transaction until it executes without conflict, which can cause long delays to finalize a transaction's update, T_U, and saturate the computing and communication resources.

In this paper, we present two new optimistic protocols: the exclusive-writer protocol (EWP) and the exclusive-writer protocol with a locking option (EWL). We describe the operation and implementation of EWP in section 2 and EWL in section 3. In section 4, we study the response times of EWP, EWL, primary site locking (PSL), and basic timestamps (BTS). Our conclusions are presented in section 5.

2. The EXCLUSIVE-WRITER PROTOCOL (EWP)

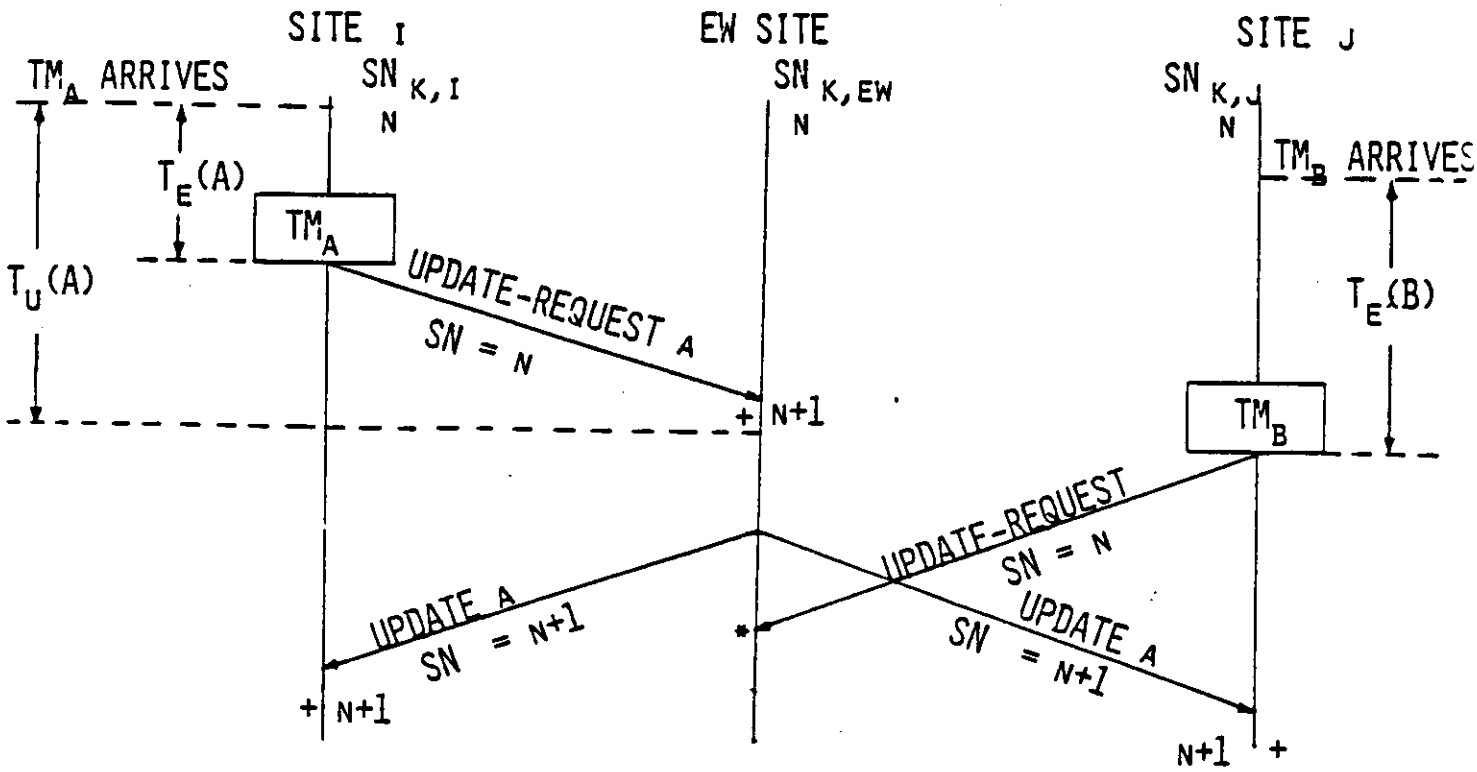
The *exclusive-writer protocol (EWP)* is a low-cost and special-purpose optimistic protocol for updating replicated files. Under EWP, each file has a single designated *exclusive-writer (EW)*, and all copies of a file have the same EW. Only a file's EW can write (update) the file. Other transactions send to the file's EW an update-request message which includes the proposed update (see fig. 1). Conflicts are detected by the use of *update sequence numbers (SN)*. An SN is attached to each file copy, and update-requests include the SN of the file copy read. If an EW accepts an update-request, it increments the SN of its file copy, and distributes the update with the new SN (see fig. 2). Updates are written in order of their SNs. A file's EW accepts an update-request if the SN in the update-request is identical to the SN of the EW's file copy, since the requesting transaction read the most current file copy. Otherwise, a conflict occurred,



- $TM_1, \ldots, TM_M$ ARE TRANSACTIONS

FIG. 1. THE EXCLUSIVE-WRITER APPROACH

- TRANSACTIONS TM_A AND TM_B ONLY ACCESS FILE F_K
- F_K IS REPLICATED AT ALL SITES



$SN_{K,I}$ = UPDATE SEQUENCE NUMBER FOR THE COPY OF FILE F_K AT SITE I

= TRANSACTION EXECUTION

T_E = TRANSACTION EXECUTION RESPONSE TIME

T_U = UPDATE CONFIRMATION RESPONSE TIME

\+ = UPDATE IS WRITTEN

* = UPDATE-REQUEST IS DISCARDED

FIG. 2. TIMING DIAGRAM FOR EXCLUSIVE-WRITER PROTOCOL (EWP)

and the transaction is said to have "lost the conflict". *An update-request that loses a conflict is discarded,* and, if required, the requesting transaction is notified.

Fig. 3 presents an algorithm for EWP operation at site i for file F_k. Let EW_k be F_k's EW, $SN_{k,i}$ be the SN of F_k at site i, and *RMSG* be an update-request message. We use the programming language notation for records to indicate fields in messages: *RMSG.SN* is the SN field in the update-request message and *RMSG.UP* is the proposed update. *UMSG* is defined similarly for an update message. The symbol ':=' is used to indicate an assignment statement, and text enclosed in '(* ... *)' are comments. The algorithm operates under the following assumptions: transactions only access F_k which is both read and written; at each site, transaction and protocol control processing are performed atomically (i.e., conflicting file accesses are prevented by using local consistency control techniques such as semaphores); SN fields have a sufficient number of bits so that an update to F_k will be written at all sites before the same SN value is assigned to another update for F_k; there are no site failures; the network guarantees that messages are eventually received without error; and when the system begins operation, copies of the same file are identical and their SNs have the same value.

EWP preserves mutual consistency and the serializability of successful transactions (i.e., those whose updates are distributed). The latter property is called *limited serializability*[1], since it is limited to transactions whose updates are distributed. We note that consecutive updates to F_k are assigned consecutive SNs. This is due to EW_k always incrementing its SN for F_k before a F_k update is broadcast. Mutual consistency follows from the fact that consecutive F_k update messages receive consecutive SNs, and updates are written in order of their SN. Hence, the same updates are written in the same order to all copies of F_k.

To establish limited serializability, we present a serial order for executing successful transactions which results in the same values for F_k as when transactions execute under EWP. Let $TM(m)$ be the m^{th} successful transaction for F_k. We show that $TM(m+1)$ read a copy of F_k to which $TM(m)$'s updates were the last to be written, or

[1] Limited serializability implies internal consistency, if the database is initially internally consistent and each transaction preserves internal consistency, since: only transactions whose updates are distributed affect the database, and these transactions are serializable.

TRANSACTION *TM* HAS JUST COMPLETED ITS EXECUTION

1. $RMSG.SN := SN_{k,i}$
2. $RMSG.UP :=$ *TM*'s update to F_k
3. send *RMSG* to EW_k

AN UPDATE MESSAGE (*UMSG*) WAS RECEIVED

1. once $UMSG.SN = SN_{k,i} + 1$
   (* Wait until in sequence *)
   a. $SN_{k,i} := UMSG.SN$
   b. update $F_{k,i}$ based on *UMSG.UP*

AN UPDATE-REQUEST MESSAGE (*RMSG*) WAS RECEIVED
(* EW_k resides at site i. *)

1. if $RMSG.SN = SN_{k,i}$ (* No Conflict *)
   a. $SN_{k,i} := SN_{k,i} + 1$
   b. update $F_{k,i}$ based on *RMSG.UP*
   c. $UMSG.SN := SN_{k,i}$
   d. $UMSG.UP := RMSG.UP$
   e. broadcast *UMSG* to all other sites with a copy of F_k
2. otherwise (* Conflict *)
   a. discard *RMSG*
   b. if required, notify the requesting transaction

Fig. 3. EWP Operation at Site i for file F_k

TM(*m*+1) *read from TM*(*m*). (This is equivalent to a serial execution in which *TM*(*m*) executed to completion then *TM*(*m*+1) began execution and read *TM*(*m*)'s updates.) Since *UMSG*(*m*+1) (*TM*(*m*+1)'s update message) is the $m+1^{st}$ F_k update, *UMSG*(*m*+1).*SN* = *m*+1. So, *TM*(*m*+1)'s update-request message (*RMSG*(*m*+1)) must have had *RMSG*(*m*+1).*SN* = *m*. Since a copy of F_k with an SN of *m* was last written by *TM*(*m*), *TM*(*m*+1) read from *TM*(*m*). Hence, EWP preserves limited serializability.

EWP has much appeal for distributed processing systems. Foremost, EWP is simple to implement. Since it does not use locking, *EWP has no deadlocks due to shared data access*. Also, because an update is not written until the EW accepts it, *EWP has no database rollbacks*. Since update-requests that lose a conflict are discarded, *EWP has no transaction restarts*. EWP has no ICSD for T_E since EWP is optimistic. In addition, $T_U(EWP)$ is small even when conflicts are frequent since: the only source of ICSD is checking update-requests at the EW's site; and T_U never includes a wait for transaction restart. However, EWP is not a general purpose protocol since it only ensures a limited form of serializability.

2.1 EWP Application Areas

For many applications, discarding update-requests that lose a conflict is not acceptable. However, EWP has much appeal for applications such as: real time feedback control in which a conflict occurs only when a real time constraint has been violated, and data collection in which occasionally discarding an update-request has little overall effect.

For example, EWP is currently used in the Distributed Processing Architecture Design (DPAD) [GREE80] for updating radar tracking data. In the DPAD, a record is maintained in the track file for each object detected by radar. Track file records are updated only as a result of radar returns for the object. The *real time constraint* of concern here is that *updates to an object's track record must be completed before a new radar beam is sent for the same object.* (The intent is to direct a radar beam based on the object's most recent sighting so that the beam encounters the correct object.) Thus a conflict occurs only when the real time constraint is violated (e.g., track processing for record *k* was not completed prior to scheduling a radar beam for the object associated

with record k). By discarding an update-request that loses a conflict for the track file, system load is reduced which facilitates meeting real time constraints. Of course, information is lost when an update-request is discarded. However, this information can be recovered from future radar returns which, hopefully, will arrive when the system is not so heavily loaded. Thus, under the very extreme conditions of violating a real time constraint, discarding update-requests that lose a conflict has appeal.

2.2 EWP With Basesets That Have Multiple Files

The files a transaction reads and writes are referred to as its *baseset*. When a transaction's baseset consists of several files which have the same EW, the update-request should include the SN for each file in the baseset. The EW accepts the update-request if the SN for each file in the update-request is identical to the SN of the EW's file copy; otherwise the update-request is discarded. When an update-request is accepted, the SN of each file written is incremented, and file updates are distributed.

If a transaction's baseset includes files with different EWs, a separate update-request is sent to *each* EW. An update-request is accepted only if the SN for each EW's file copy is the same as the file's SN in the update-request. For real time applications, the communication costs and time delays required for EWs to inform one another that an update-request has been accepted may be excessive. Thus, it is desirable for the transaction's entire baseset to have the same EW.

2.3 EWP Implementation Considerations

When a transaction accesses a file that is not replicated at its execution site, a copy of the required file data must be obtained along with the file copy's SN. The SN of the remote copy is inserted in the transaction's update-request in the same manner as if the site had a local file copy.

How transactions should be assigned to sites (referred to as task assignment in [CHU80]) is affected by utilization due to transaction executions and inter-transaction (or intermodule) communication [CHU84]. Assigning an EW to a transaction's execution site increases the site's utilization due to EW executions. However such an assignment can decrease communication overhead, since a transaction does not send an update-

request message for a file whose EW is assigned to the transaction's execution site.

In EWP, non-EW transactions can not directly modify files. Instead, writing updates is deferred until the EW has checked them for conflicts. Referred to as *deferred update writing,* this approach requires: providing a transaction with a temporary copy of the file data it modifies, and writing a transaction's update to the site's permanent file copy once the EW has accepted the transaction's update-request. Deferring file updates can be accomplished with little additional overhead if shadow file copies are used for site fault tolerance [GRAY81]. With this technique, the operating system creates a temporary copy of file data whenever a transaction first writes the data; after the transaction finishes executing, the temporary copy becomes the site's permanent file copy. Deferred update writing can be achieved by not modifying the permanent file copy until the EW has accepted the transaction's update-request.

Interrupt scheduling affects a transaction's ability to read a file copy at any time. For example, if input messages preempt transaction executions, receiving an update message might cause a file copy to be written while some transaction was reading it. In the DPAD, this problem is resolved by using a cyclic dispatcher which ensures non-preemptive execution of transactions. Systems that permit preemption require synchronization techniques such as semaphores to preserve data consistency during transaction execution.

EWP message volume can be reduced as follows. Site *i* retains a copy of all update-request messages it sends. If an EW accepts a site *i* update-request, an update-accepted *control message* is sent to site i (instead of an update message), and site *i* writes the update based on its saved update-request. Thus, message volume is reduced since update messages should always be larger than control messages. This approach can be applied to broadcast networks by having requesting sites broadcast their update-requests (which are saved at all sites), and EWs broadcast an update-accepted message when an update-request has been accepted.

Since sites may fail, it is desirable that EWP: 1) preserve database consistency when sites fail; and 2) permit resumption of database operations at sites that have recovered. Regarding 1), if sites write updates atomically, only the failure of an EW site

affects consistency, since the EW site for F_k could fail before an update message has been sent to all copies of F_k. This can be resolved by having for each EW a backup that assumes control if the current EW site fails.[1] For 2), we can either: maintain a log of updates missed by the failed site; or the site can, after recovery, obtain (via a remote read) a current copy of the data it requires.

3. EXCLUSIVE-WRITER PROTOCOL WITH LOCKING OPTION (EWL)

We extend EWP so that it does not discard update-requests that lose a conflict, and this is done in a way that minimizes transaction restarts. Our approach is to add a locking option which is used only when a conflict occurs. In essence, we combine EWP with primary site locking - PSL [STON79].

We begin by describing the operation of PSL. PSL ensures serializability by requiring transactions to get permission from the file's primary site (PS) before accessing the file as shown in fig. 4. Specifically when the transaction arrives, a lock-request message is sent to the file's PS. The transaction waits until the PS replies with a lock-grant message. This message includes the SN of the most current copy of the file. After the transaction executes, the file's SN is incremented, and the transaction's updates are distributed with the new SN. Updates are written in order of their SNs. The PS treats an update message as an implicit lock-release.

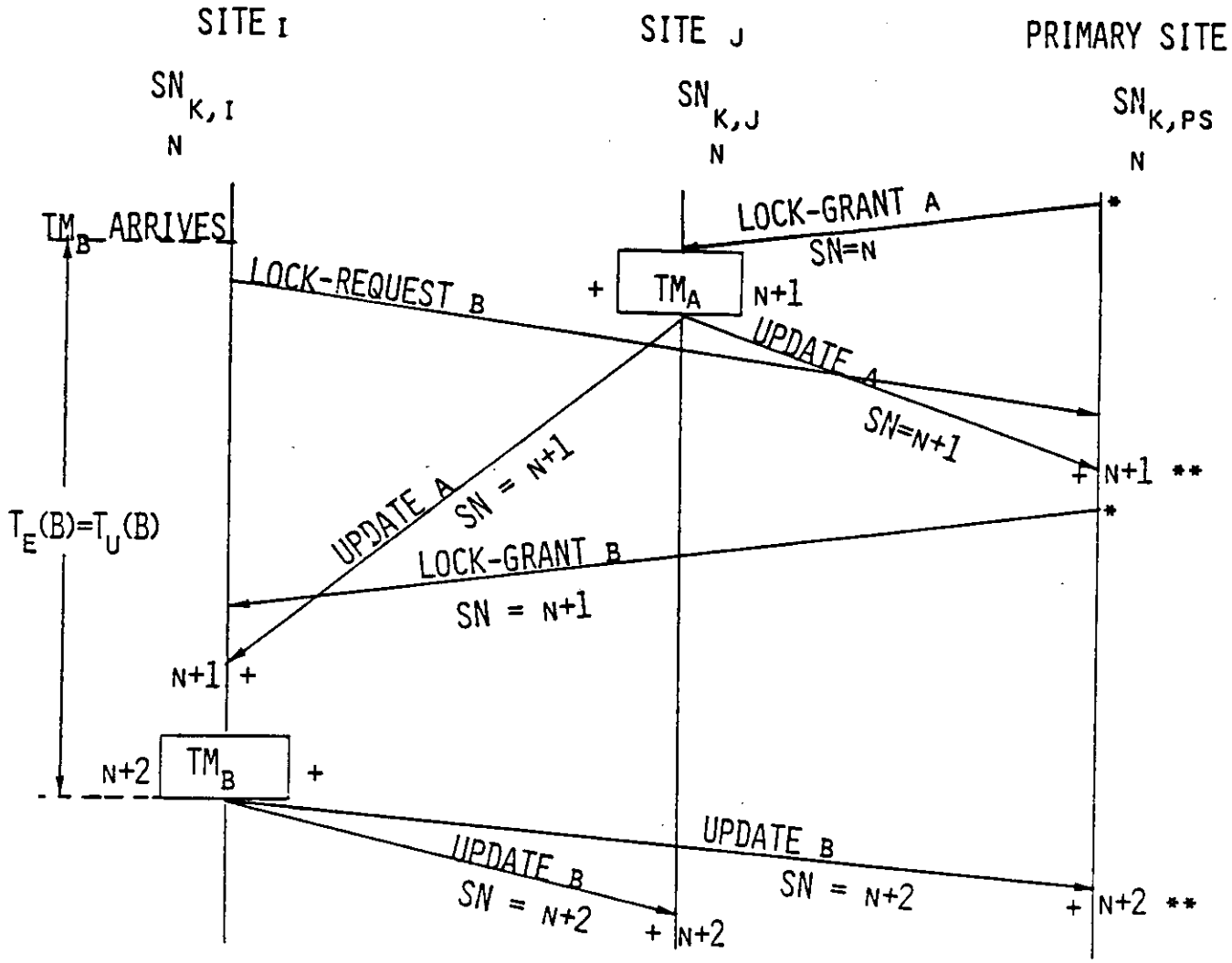
EWL combines EWP and PSL as shown in fig. 5. When a transaction arrives, it executes under EWP. EWL's operation is the same as EWP's if the transaction's update-request is accepted. Otherwise: 1) the file's EW site becomes its PS; 2) the update-request is treated as a PSL lock-request and is placed in a lock queue; and 3) the transaction is restarted under PSL.[2] The criteria for accepting an F_k update-request are:

1. The SN for F_k in the update-request is identical to the SN of the file copy at EW_k's site.

[1] A single backup can be extended to N-site resiliency as in [ALSB76].

[2] A site learns that a transaction will be restarted as follows. If transaction TM executes at site i and updates F_k with $SN = n$, then site i knows that TM will be restarted if site i receives an update for F_k with $SN = n+1$ but the update was not made by TM.

- TRANSACTIONS TM_A AND TM_B ONLY ACCESS FILE F_K
- F_K IS REPLICATED AT ALL SITES



$SN_{K,I}$ = UPDATE SEQUENCE NUMBER FOR THE COPY OF FILE K AT SITE I

□ = TRANSACTION (TM) EXECUTION

T_E = TRANSACTION EXECUTION RESPONSE TIME

T_U = UPDATE CONFIRMATION RESPONSE TIME

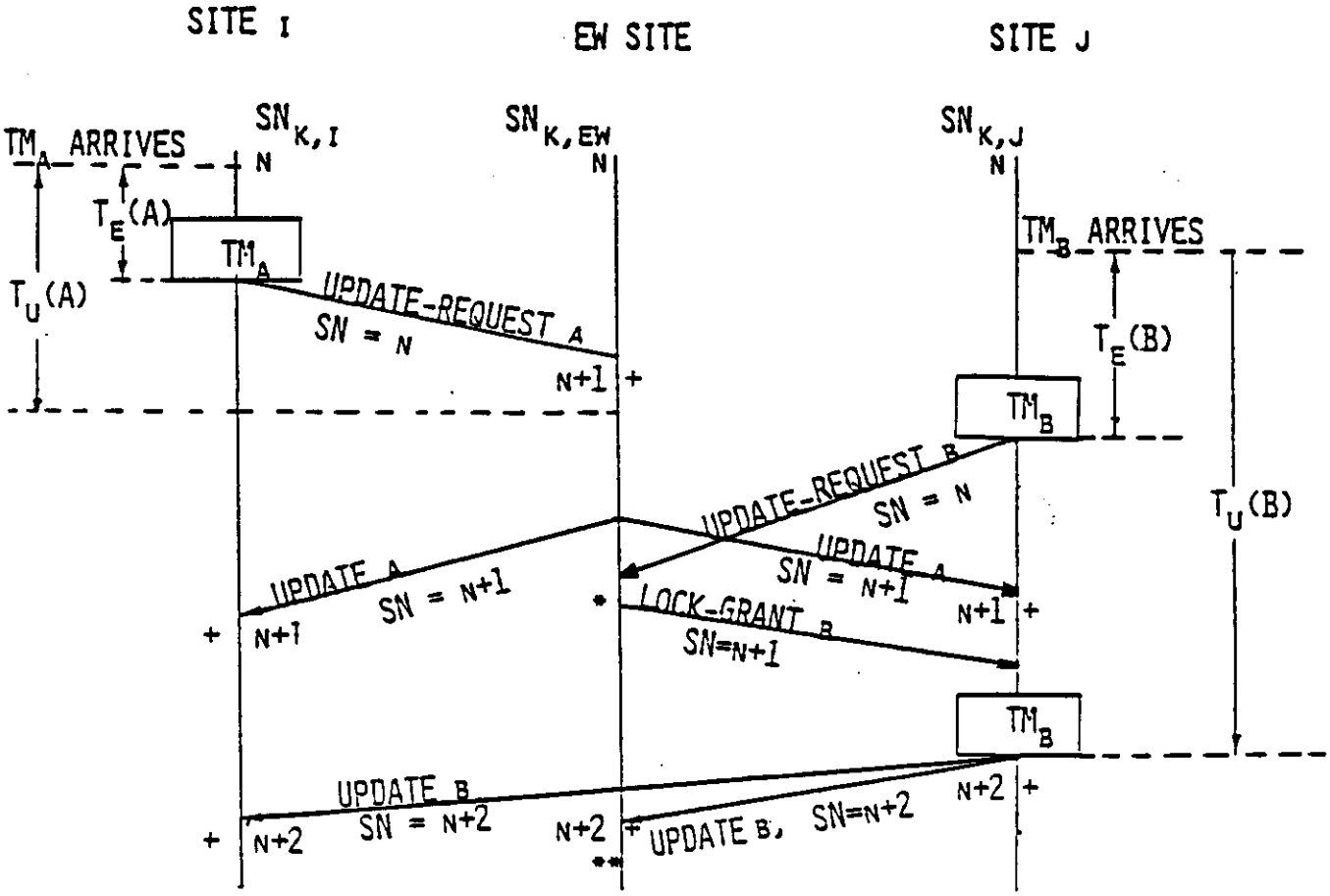
\+ = UPDATE IS WRITTEN

* = FILE IS LOCKED

** = FILE IS UNLOCKED

FIG. 4. TIMING DIAGRAM FOR PRIMARY SITE LOCKING (PSL)

- TRANSACTIONS TM_A AND TM_B ONLY ACCESS FILE F_K
- F_K IS REPLICATED AT ALL SITES



$SN_{K,I}$ = UPDATE SEQUENCE NUMBER FOR THE COPY OF F_K AT SITE I

▭ = TRANSACTION EXECUTION

T_E = TRANSACTION EXECUTION RESPONSE TIME

T_U = UPDATE CONFIRMATION RESPONSE TIME

\+ = UPDATE IS WRITTEN

* = FILE IS LOCKED

** = FILE IS UNLOCKED

FIG. 5. TIMING DIAGRAM FOR EXCLUSIVE WRITER PROTOCOL WITH LOCKING OPTION (EWL)

2. F_k is not locked.

Fig. 6 contains an algorithm for EWL operation at site i for file F_k. We use the same notation and assumptions as for the EWP algorithm with the additional assumption that a transaction's baseset does not change when it is restarted.

We show that the algorithm in fig. 6 preserves mutual consistency and serializability. First note that consecutive updates to F_k are assigned consecutive SNs. This is a consequence of: 1) at most one transaction has the authority to distribute updates at any instant (i.e., the EW distributes updates if F_k is not locked and the current lock holder distributes updates if F_k is locked), and this transaction keeps update distribution authority until its updates have been received by EW_k; 2) a transaction with update distribution authority does not execute until its copy of F_k (and hence F_k's SN) is identical to EW_k's copy; and 3) F_k's SN is always incremented before its updates are distributed. Mutual consistency follows since updates are written in the same order to all copies of a file.

To establish serializability, let $TM(m)$ be the m^{th} transaction for F_k to have its updates distributed. We show that the same value for F_k would have been obtained by a serial execution of the transactions in the order in which their updates were distributed. Specifically, $TM(m+1)$ read a copy of F_k to which $TM(m)$'s updates were the last to be written, or $TM(m+1)$ *read from* $TM(m)$. Let $RMSG(m+1)$ be $TM(m+1)$'s update-request message, $UMSG(m+1)$ be its update message, and i be its site of execution. By definition of $TM(m+1)$, $UMSG(m+1).SN = m+1$. If $TM(m+1)$ was not restarted, EW_k distributed $UMSG(m+1)$. So,

$$RMSG(m+1).SN = UMSG(m+1).SN - 1 = m$$

So, $TM(m+1)$ read from $TM(m)$. Now suppose that $TM(m+1)$ was restarted. Let $SN(k,i,t)$ be the SN of F_k at site i when $TM(m+1)$ was scheduled for execution. Since F_k's SN is incremented after $TM(m+1)$ executes and before $UMSG(m+1)$ is distributed,

$$m + 1 = UMSG(m+1).SN = SN(k,i,t) + 1$$

Hence, $SN(k,i,t) = m$. So, $TM(m+1)$ read from $TM(m)$.

TRANSACTION *TM* HAS JUST COMPLETED ITS EXECUTION

1. if *TM* was restarted
   a. $SN_{k,i} := SN_{k,i} + 1$
   b. update $F_{k,i}$ based on *TM*'s update
   c. $UMSG.SN := SN_{k,i}$
   d. $UMSG.UP :=$ *TM*'s update to F_k
   e. broadcast *UMSG* to all other sites with a copy of F_k
2. otherwise
   a. $RMSG.SN := SN_{k,i}$
   b. $RMSG.UP :=$ *TM*'s update to F_k
   c. send *RMSG* to EW_k

AN UPDATE MESSAGE (*UMSG*) WAS RECEIVED

1. once $UMSG.SN = SN_{k,i} + 1$
   (* Wait until in sequence *)
   a. $SN_{k,i} := UMSG.SN$
   b. update $F_{k,i}$ based on *UMSG.UP*
2. if EW_k resides at site i
   (* F_k is currently locked *)
   a. remove the first entry in F_k's lock queue and unlock F_k
   b. if F_k's lock queue is not empty
      i. lock F_k
      ii. $LMSG.SN := SN_{k,i}$ (* *LMSG* is a lock-grant message *)
      iii. send *LMSG* to the transaction with the first entry in F_k's lock queue
      (* Delegate Update Distribution Authority *)

Fig. 6. EWL Operation at Site i for file F_k

AN UPDATE-REQUEST (*RMSG*) MESSAGE WAS RECEIVED

1. if $RMSG.SN = SN_{k,i}$ and F_k is not locked
   (* No Conflict *)
   a. $SN_{k,i} := SN_{k,i} + 1$
   b. update $F_{k,i}$ based on *RMSG.UP*
   c. $UMSG.SN := SN_{k,i}$
   d. $UMSG.UP := RMSG.UP$
   e. broadcast *UMSG* to all other sites with a copy of F_k
2. otherwise (* Conflict *)
   a. put *RMSG* at the end of F_k's lock queue
   b. if *RMSG* is the first entry in F_k's lock queue
      i. lock F_k
      ii. $LMSG.SN := SN_{k,i}$
      iii. send *LMSG* to the transaction with the first entry in F_k's lock queue
      (* Delegate Update Distribution Authority *)

A LOCK-GRANT MESSAGE (*LMSG*) WAS RECEIVED FOR TRANSACTION *TM*

1. once $LMSG.SN = SN_{k,i}$
   (* Wait until $F_{k,i}$ is current *)
   a. schedule *TM* for restart execution

Fig. 6. Continued

EWL is more difficult to implement than EWP. However unlike EWP, EWL ensures serializability. EWL has no database rollbacks. Since it restarts transactions under a pessimistic protocol, *EWL restarts a transaction at most once* if the transaction's baseset does not change when it is restarted. $T_E(EWL)$ has no ICSD. The ICSD for $T_U(EWL)$ are similar to those for $T_U(EWP)$ if the transaction is not restarted, and similar to those for $T_U(PSL)$ if the transaction is restarted.

3.1 EWL With Basesets That Have Multiple Files

The criteria for granting a lock must consider the possibility of deadlock. Thus some form of deadlock management, such as avoidance, prevention, or detection, is required.

There may be files in a transaction's baseset that are read but not written. If the transaction is restarted, the EW for such a file will not receive an update to indicate that the file's lock should be released. So, after a transaction completes its restart execution, a lock-release message should be sent to the EW of each file which is read but not written.

If all files in the transaction's baseset have the same EW, the EW accepts the update-request if for each file: the SN in the update-request is identical to the SN of the EW's file copy, and the file is not locked. If a transaction's baseset includes files with different EWs, a separate update-request is sent to each EW. An update-request is accepted only if for each file in the baseset the file's EW has the same SN for its file copy as the SN in the update-request, and the file is not locked. For real time applications, the communication costs and time delays required for EWs to inform one another that an update-request has been accepted may be excessive. Thus as with EWP, each file in the transaction's baseset should have the same EW.

3.2 Dynamic Switching Between PSL And EWL

EWL's performance can degrade due to transaction restarts which increase load and lengthen T_U. EWL improves on existing optimistic protocols since EWL can guarantee that a transaction will be restarted at most once. However when restarts are

frequent, pessimistic protocols are preferred since they have no transaction restarts. Ideally, we should use an optimistic protocol when restarts are rare and not too costly and use a pessimistic protocol when restarts are frequent and/or costly. A very appealing aspect of EWL is that each site can independently and dynamically choose whether to execute a transaction under EWL (which is optimistic) or PSL (which is pessimistic). Further, unlike other approaches to dynamic protocol selection (e.g., [BERN78] and [MILE81]), ***dynamic switching between EWL and PSL is done without additional messages or delays to synchronize protocol selection.*** This feature results from being able to use EWL and PSL concurrently for the same shared file (see fig. 7). Recall that EWL already requires implementing PSL. So, a PSL lock-request is treated as a rejected EWL update-request. Specifically, a lock-request received by EW_k is placed in F_k's lock queue until EW_k selects it. Then, EW_k sends a lock-grant to the requesting transaction, and the transaction executes. EW_k treats the transaction's update message as an implicit lock-release. While switching between PSL and EWL requires no message or delay to synchronize protocol selection, overhead may be required for sites to gather the information necessary to choose which protocol to use.

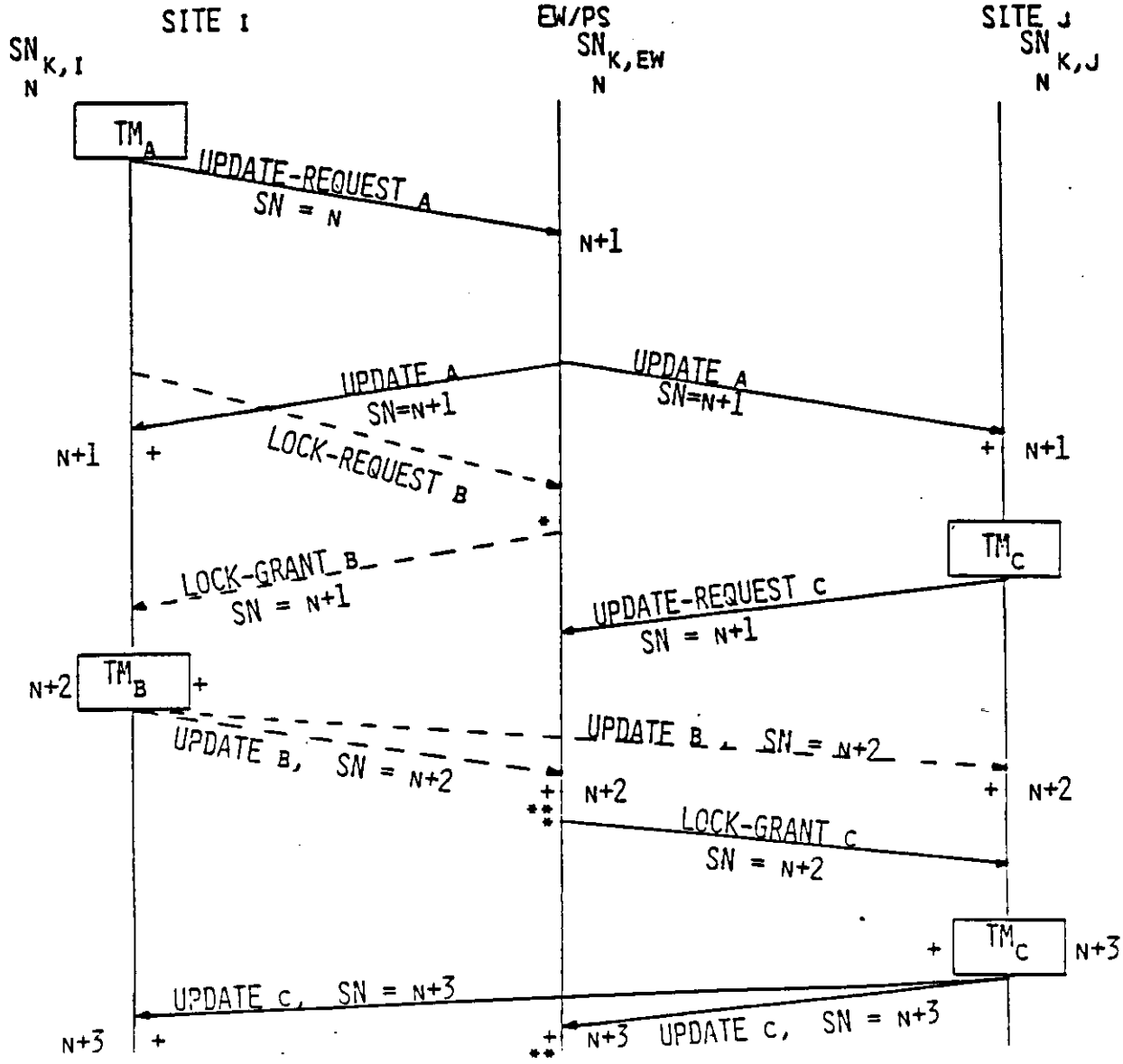
3.3 EWL Implementation Considerations

Except for fault tolerance, the same implementation considerations mentioned for EWP also apply to EWL. In addition, using EWL has implications on transactions whose basesets change when they are restarted.

EWL can be made fault tolerant by employing an existing scheme for fault tolerance which is applicable to PSL (e.g., [WALK83]). The EWL update-request is viewed as a PSL lock-request. If the update-request is accepted, this is immediately followed by a lock-release. If the update-request is not accepted, EWL's operation becomes the same as that of PSL.

In general, a transaction's baseset may be different when it is restarted, since file access patterns can be data dependent. This can be resolved by the transaction sending a lock-release message to all EWs from which it received a lock-grant and sending an update-request message to the EW of each file in its new baseset. Thus, if there is a high probability that a transaction's baseset changes when it is restarted, EWL might

- TRANSACTIONS TM_A, TM_B, AND TM_C ONLY ACCESS F_K
- F_K IS REPLICATED AT ALL SITES



$SN_{K,I}$ = UPDATE SEQUENCE NUMBER FOR THE COPY OF F_K AT SITE I

□ = TRANSACTION EXECUTION

\+ = UPDATE IS WRITTEN

* = FILE IS LOCKED

** = FILE IS UNLOCKED

——— = EWL

- - - - - = PSL

FIG. 7. CONCURRENT USE OF EWL AND PSL FOR THE SAME SHARED FILE

perform poorly since it would have delays for locking and repeated transaction restarts. Dynamic switching to PSL could alleviate this problem by using PSL for those transactions that have a high probability of being restarted with a different baseset.

4. RESPONSE TIME STUDIES

Here, we compare the response times of EWP and EWL with two existing protocols: PSL and basic timestamps, an optimistic timestamp protocol[1] [BERN81] . We first develop analytic models for T_E and T_U, then derive crossover points for the response times of the protocols, then perform numerical studies, and finally discuss our results.

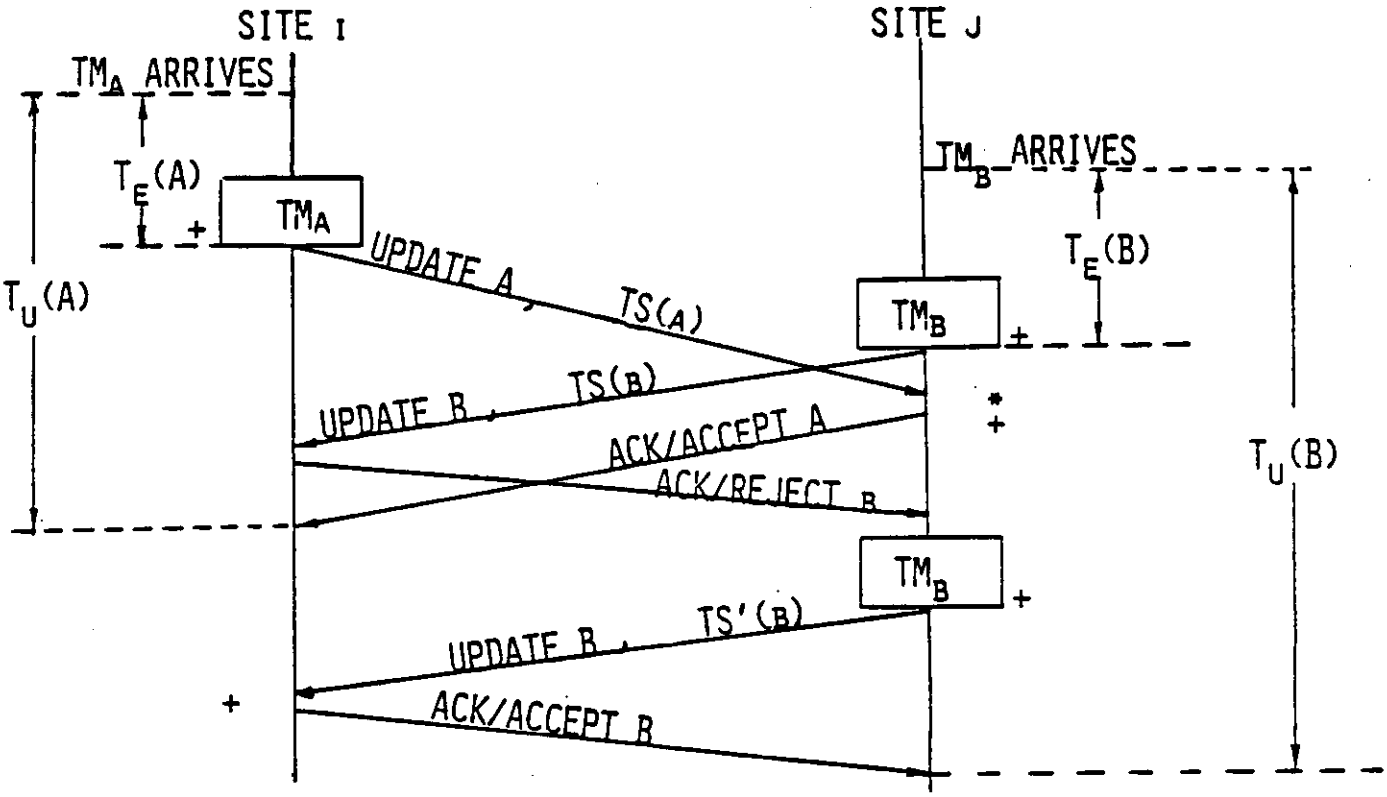
Let us describe the basic timestamps (BTS) protocol (see fig. 8). BTS preserves mutual consistency and serializability by assigning globally unique timestamps to transactions and their updates and by requiring updates to be written in order of their timestamp. Since BTS is optimistic, after a transaction arrives it executes without any ICSD. Then the transaction's updates are written to the database at its execution site. An *update-log* is maintained in case updates are not written in timestamp order and must be removed by performing a *database rollback*. BTS uses *distributed conflict checking*. That is, the transaction's updates are sent to all other sites, and each site checks that the updates are in timestamp order. If the updates are in timestamp order, an acknowledgement indicating acceptance is returned to the sending site. Otherwise, an acknowledgement indicating rejection is returned to the sending site. Transactions whose updates are rejected are restarted in the same manner as their original execution. The first site to know that an update has been finalized is the transaction's execution site when it has received an acknowledgement with acceptance from all other sites.

4.1 Response Time Models

To construct our response times models, we assume an environment similar to that for the DPAD System in which:

[1] The algorithm in [BERN81] only applies to single copy database systems. However, extensions to multiple copy databases are assumed in our analysis.

- TRANSACTIONS TM_A AND TM_B ONLY ACCESS FILE F_K
- F_K IS REPLICATED AT ALL SITES



- $TS(A) < TS(B) < TS'(B)$

= TRANSACTION EXECUTION

T_E = TRANSACTION EXECUTION RESPONSE TIME

T_U = UPDATE CONFIRMATION REPONSE TIME

\+ = UPDATE IS WRITTEN

* = DATABASE ROLLBACK

TS = TIMESTAMP

FIG. 8. TIMING DIAGRAM FOR BASIC TIMESTAMP (BTS)

1. A transaction's baseset does not change if the transaction is restarted.

2. All files in a transaction's baseset have the same PS for PSL and the same EW site for EWP and EWL.

3. Transaction file requirements are known in advance.

4. The communication network is reliable, and sites do not fail.

$T_E(PSL)$ includes time from the transaction's arrival until its lock-request is placed in the lock queue; wait in the lock queue; time from lock grant processing at the PS until the transaction is placed in its site execution queue; transaction's wait for execution; and execution of the transaction (see fig. 9). Thus,

$$T_E(PSL;m,i) = D_{PL}(m,i) + W_L(m) + D_{GE}(m,i) + W_E(m,i) + X_E(m,i) \quad (1)$$

where:

$T_E(PSL;m,i)$ = PSL execution response time for transaction m at site i

$D_{PL}(m,i)$ = Pre-lock processing delay for transaction m at site i: from transaction arrival at site i to enqueueing all of the transaction's lock requests

$W_L(m)$ = Waiting time for file locks for transaction m: starts when the transaction's last lock request is enqueued and ends when its last lock is granted

$D_{GE}(m,i)$ = Delay starting from lock grant processing of transaction m to when the transaction enters the execution queue at site i

$W_E(m,i)$ = Waiting time for executing transaction m at site i

$X_E(m,i)$ = Service time for executing transaction m at site i

Since PSL is a pessimistic protocol, updates are known to be finalized after the transaction's first execution.

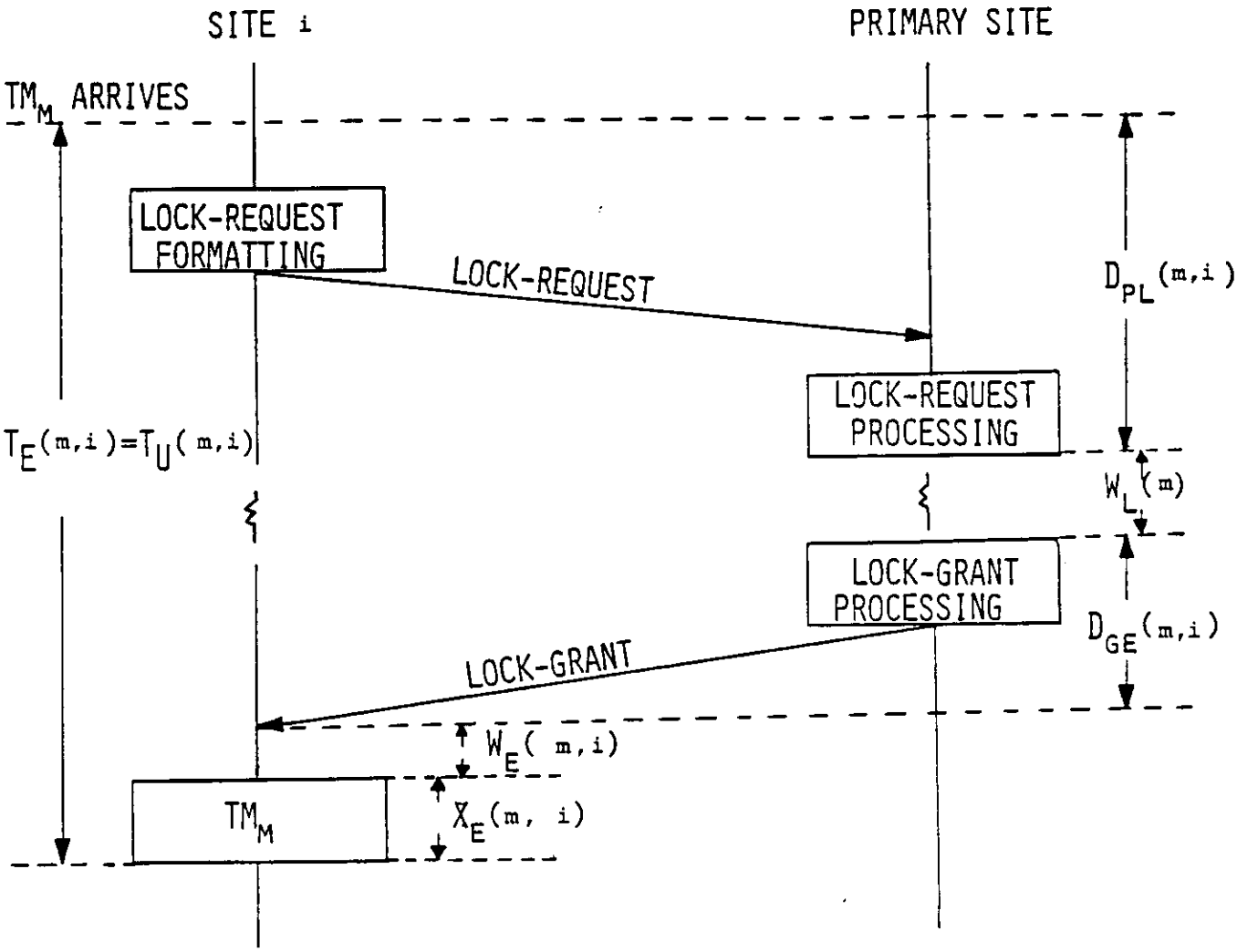


FIG. 9. TIMING DIAGRAM FOR PSL RESPONSE TIMES

$$T_U(PSL;m,i) = T_E(PSL;m,i) \tag{2}$$

where:

$T_U(PSL;m,i)$ = PSL update finalization response time for transaction m at site i

Next consider the response times for BTS. Since BTS has no ICSD for T_E, $T_E(BTS)$ consists of only the transaction's wait for execution, transaction execution time, and time for update-log maintenance (see fig. 10).

$$T_E(BTS;m,i) = W_E(m,i) + X_E(m,i) + Y_M(m,i) \tag{3}$$

where:

$Y_M(m,i)$ = Service time for update-log maintenance after transaction m executes at site i

Under BTS, transactions are repeatedly restarted until they execute without conflict. Let $q(m,i)$ be the probability of restarting transaction m at site i. The number of times this transaction executes for each arrival-instance is

$$N_E(m,i) = \frac{1}{1 - q(m,i)} \tag{4}$$

where:

$q(m,i)$ = Probability of restarting transaction m which executes at site i

To compute $T_U(BTS)$, note that each time a transaction executes there are delays for execution response time, conflict checking (i.e., wait for *all* sites to check the timestamp of the updates and return an acknowledgement), and database rollbacks if the transaction is restarted.

$$T_U(BTS;m,i) = N_E(m,i)\Big[T_E(BTS;m,i) + \max_j\{D_{EC}(m,i,j) + D_{CA}(m,j,i)\} + q(m,i)\,Y_R(m,i)\Big] \tag{5}$$

where:

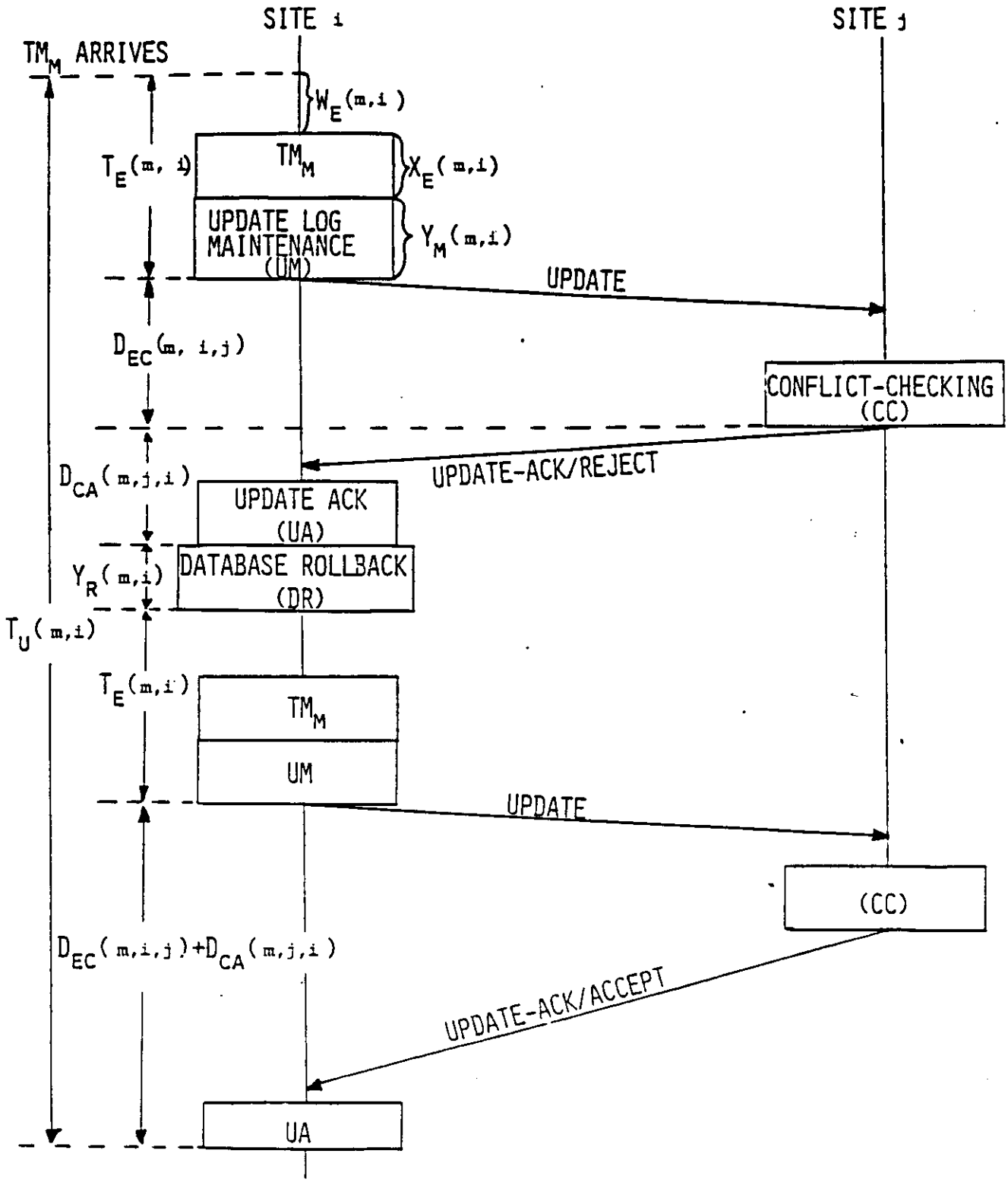


FIG. 10. TIMING DIAGRAM FOR BTS RESPONSE TIMES

$D_{EC}(m,i,j)$ = Delay from completing the execution of transaction m at site i to completion of checking the timestamp ordering of its updates at site j

$D_{CA}(m,j,i)$ = Delay from checking the timestamp ordering of transaction m's updates at site j until completion of processing of site j's acknowledgment at site i

$Y_R(m,i)$ = Service time for database rollback of updates made by transaction m at site i

We now consider response times for EWP. $T_E(EWP)$ only consists of the transaction's wait for execution and transaction execution time (see fig. 11).

$$T_E(EWP;m,i) = W_E(m,i) + X_E(m,i) \tag{6}$$

$T_U(EWP)$ has update request processing by the EW. Thus update finalization response time is

$$T_U(EWP;m,i) = T_E(EWP;m,i) + D_{ER}(m,i) \tag{7}$$

where:

$D_{ER}(m,i)$ = Delay from completing the execution of transaction m at site i until update request processing has been completed for all of the transaction's update-requests

From the discussion of EWL in section 3, $T_E(EWL)$ is identical to $T_E(EWP)$. So,

$$T_E(EWL;m,i) = T_E(EWP;m,i) \tag{8}$$

To estimate $T_U(EWL)$, there are two cases. If the transaction is not restarted, $T_U(EWL)$ is the same as $T_U(EWP)$. Otherwise, there are delays for: execution response time, time from the transaction's execution until update-requests have been placed in lock queues, wait for file locks, time from lock grant processing until the transaction enters the execution queue at its site, and another execution response time. (The time components for the restart case are shown in fig. 12.)

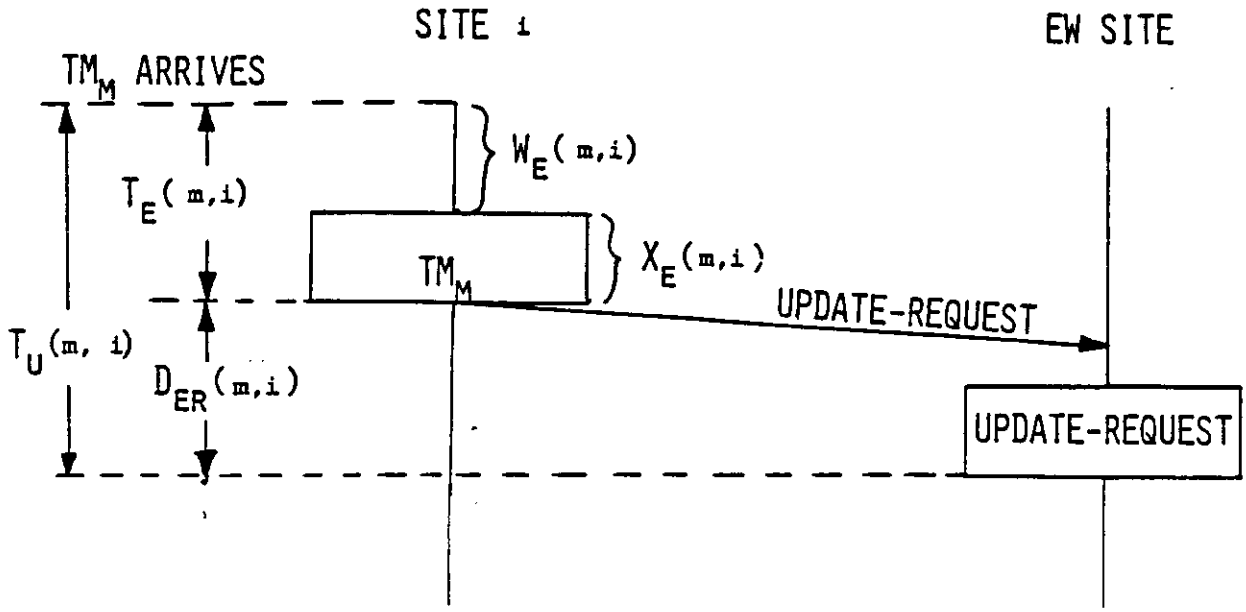


FIG. 11. TIMING DIAGRAM FOR EWP RESPONSE TIMES

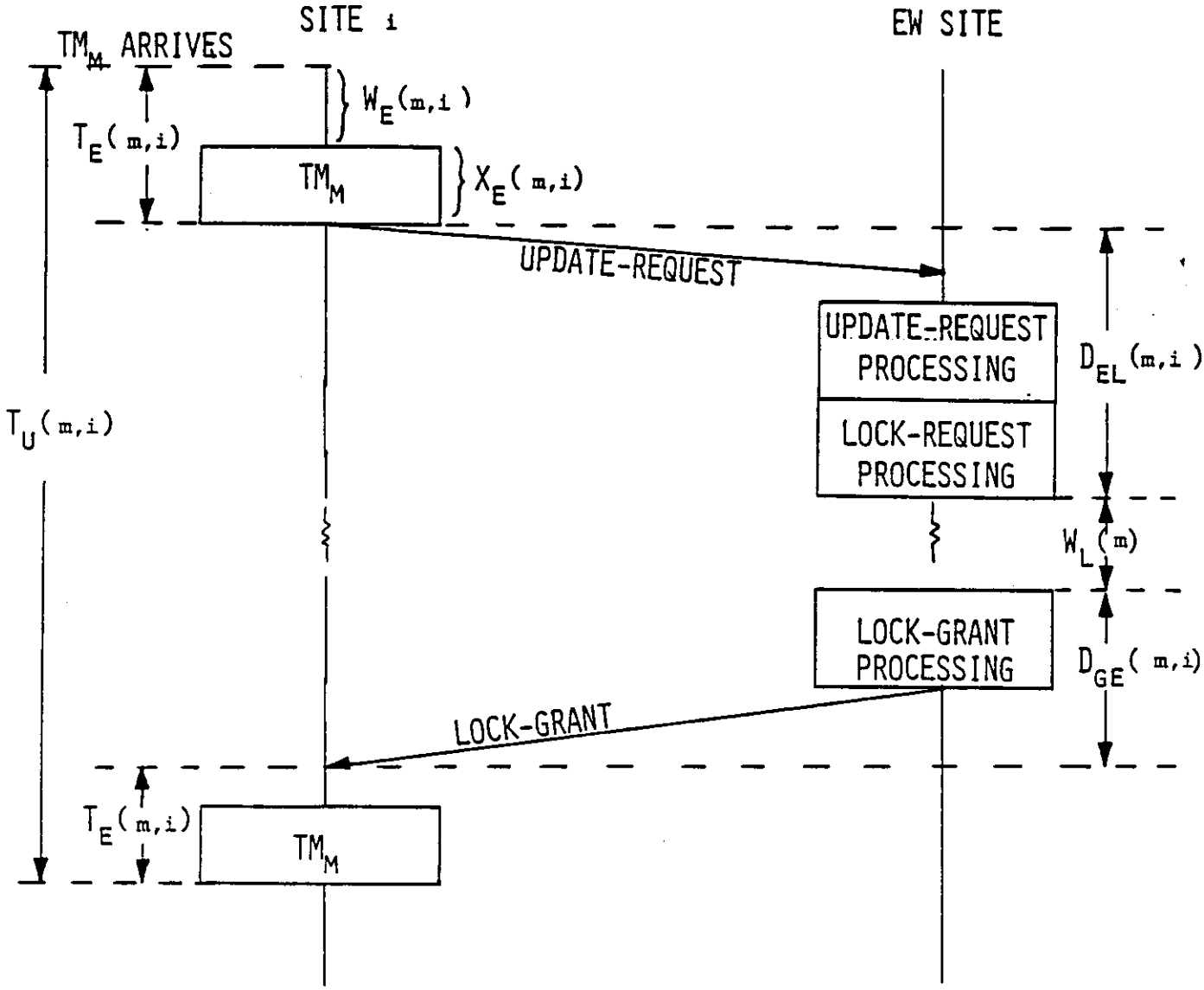


FIG. 12. TIMING DIAGRAM FOR EWL RESPONSE TIMES WHEN TRANSACTION IS RESTARTED

$$T_U(EWL;m,i) = T_E(EWL;m,i) + (1 - q(m,i))D_{ER}(m,i)$$
$$+ q(m,i)\Big[D_{EL}(m,i) + W_L(m) + D_{GE}(m,i) + T_E(EWL;m,i)\Big] \quad (9)$$

where:

$D_{EL}(m,i)$ = Delay from completing the execution of transaction m at site i until all its lock-requests have been enqueued

4.2 Crossover Points For Protocol Response Times

By comparing the protocol response time equations, we derive crossover points for the response time measures[1]. Let us first consider execution response times. From (8), EWP and EWL have the same execution response times, so it suffices to compare PSL, BTS, and EWL. From (1) and (8), we have $T_E(EWL;m,i) \leq T_E(PSL;m,i)$ when

$$0 \leq D_{PL}(m,i) + W_L(m) + D_{GE}(m,i) \quad (10)$$

From (3) and (8), we have $T_E(EWL;m,i) \leq T_E(BTS;m,i)$ when

$$0 \leq Y_M(m,i) \quad (11)$$

Comparing (1) and (3), we have $T_E(BTS;m,i) \leq T_E(PSL;m,i)$ when

$$0 \leq D_{PL}(m,i) + W_L(m) + D_{GE}(m,i) - Y_M(m,i) \quad (12)$$

We observe that optimistic protocols have much appeal for execution response times. Specifically, *EWP and EWL have the smallest* T_E. $T_E(BTS)$ will also be small if the cost of update-log maintenance, Y_M, is small.

Next, we consider update finalization response times. By comparing (2) and (9), we have $T_U(EWL;m,i) \leq T_U(PSL;m,i)$ when

$$q(m,i) \leq \frac{D_{PL}(m,i) + W_L(m) + D_{GE}(m,i) - D_{ER}(m,i)}{D_{EL}(m,i) - D_{ER}(m,i) + W_L(m) + D_{GE}(m,i) + W_E(m,i) + X_E(m,i)} \quad (13)$$

For EWL and BTS, we obtain a quadratic equation when comparing (5) and (9). Solving

[1] The parameters should be qualified by the protocol. For example when restarts are frequent, $W_E(BTS;m,i)$ may be considerably larger than $W_E(EWL;m,i)$ due to the additional site utilization caused by BTS repeated transaction restarts.

for both roots, we determine that $T_U(EWL;m,i) \leq T_U(BTS;m,i)$ when either (14a) or (14b) holds.

$$q(m,i) \leq \frac{b(m,i) - \left[b^2(m,i) - 4a(m,i)c(m,i)\right]^{1/2}}{2a(m,i)} \tag{14a}$$

$$q(m,i) \geq \frac{b(m,i) + \left[b^2(m,i) - 4a(m,i)c(m,i)\right]^{1/2}}{2a(m,i)} \tag{14b}$$

where:

$$a(m,i) = D_{EL}(m,i) - D_{ER}(m,i) + W_E(m,i) + X_E(m,i) + W_L(m) + D_{GE}(m,i)$$

$$b(m,i) = W_L(m) + D_{GE}(m,i) + D_{EL}(m,i) - 2D_{ER}(m,i) - Y_R(m,i)$$

$$c(m,i) = Y_M(m,i) + \max_j\{D_{EC}(m,i,j) + D_{CA}(m,j,i)\} - D_{ER}(m,i)$$

Comparing (2) and (5), we have $T_U(BTS;m,i) \leq T_U(PSL;m,i)$ when

$$q(m,i) \leq \frac{D_{PL}(m,i) + W_L(m) + D_{GE}(m,i) - Y_M(m,i) - \max_j\{D_{EC}(m,i,j) + D_{CA}(m,j,i)\}}{D_{PL}(m,i) + W_L(m) + D_{GE}(m,i) + W_E(m,i) + X_E(m,i) + Y_R(m,i)} \tag{15}$$

From (7) and (9), we have $T_U(EWP;m,i) \leq T_U(EWL;m,i)$ when

$$0 \leq q(m,i)\left[D_{EL}(m,i) - D_{ER}(m,i) + W_L(m) + D_{GE}(m,i) + W_E(m,i) + X_E(m,i)\right] \tag{16}$$

We observe from (13) and (15) that $T_U(PSL)$ is smaller than $T_U(BTS)$ or $T_U(EWL)$ when the restart probability, q, is large, since BTS and EWL have restarts but PSL does not. From (14b), $T_U(EWL)$ is smaller than $T_U(BTS)$ for large q, due to BTS repeated transaction restarts. Since $D_{EL} \approx D_{ER}$, (16) indicates that $T_U(EWP) \leq T_U(EWL)$.

4.3 Numerical Studies

To better illustrate the interrelationships of parameters for T_U, numerical studies are presented. To simplify our studies, we make several assumptions. First, all sites are identical, and all transactions are identical. Thus, $W_E = W_E(m,i)$, $X_E = X_E(m,i)$, $q = q(m,i)$, and $W_L = W_L(m)$. Second, the delay for communicating and processing an update message is the same for all types of update messages. Thus, $D = D_{ER}(m,i) = D_{EL}(m,i) = D_{EC}(m,i,j)$. Third, the delay for communicating and

processing a control message is the same for all types of control messages. Thus, $D' = D_{PL}(m,i) = D_{GE}(m,i) = D_{CA}(m,j,i)$. Fourth, the time for update-log maintenance is the same as the time for database rollbacks. Thus, $Y = Y_M(m,i) = Y_R(m,i)$. Fifth, we define R_{max} as the relative delay for conflict checking of the slowest site and assume that R_{max} is the same for all transactions and sites. Let D_{EC} be the average value for $D_{EC}(m,i,j)$ and D_{CA} be the average value for $D_{EC}(m,j,i)$. Since $D = D_{EC}$ and $D' = D_{CA}$, $R_{max} = \frac{\max_j\{D_{EC}(m,i,j) + D_{CA}(m,j,i)\}}{D + D'}$. We define site response time as the sum of the waiting time for transaction execution and the transaction execution time. To further simplify our studies, time is normalized with respect to site response time.

Fig. 13 presents T_U as a function of restart probability, q, for selected values of the times for processing and communicating update messages, D, and the times for processing and communicating and control messages, D' . $T_U(PSL)$ is unaffected by q since PSL has no restarts. Also, PSL is not affected by D, since under PSL no update message is sent prior to the update being finalized. $T_U(PSL)$ increases linearly with D' at a rate of 2, since for each transaction two control messages are required (i.e., lock-request and lock-grant). $T_U(BTS)$ increases geometrically with q due to repeated transaction restarts (see (4)). Also, $T_U(BTS)$ increases linearly at a rate of $\frac{R_{max}}{1-q}$ with both D and D' due to distributed conflict checking and repeated restarts. $T_U(EWL)$ increases linearly with q. $T_U(EWL)$ increases linearly with D at a rate of 1, due to the update-request message. $T_U(EWL)$ increases linearly with D' at a rate of q since a control message (i.e., lock-grant) is required only when a transaction is restarted.

We consider four cases for comparing $T_U(PSL)$, $T_U(BTS)$, and $T_U(EWL)$ in fig. 13. When D and D' are both small, T_U is primarily affected by the other parameters. If D is small and D' is large, EWL has the smallest T_U since EWL has the lowest rate of increase with D' . For D large and D' small, PSL has the smallest T_U, since $T_U(PSL)$ is unaffected by D. When both D and D' are large, $T_U(EWL)$ is the smallest for small q and $T_U(PSL)$ for large q.

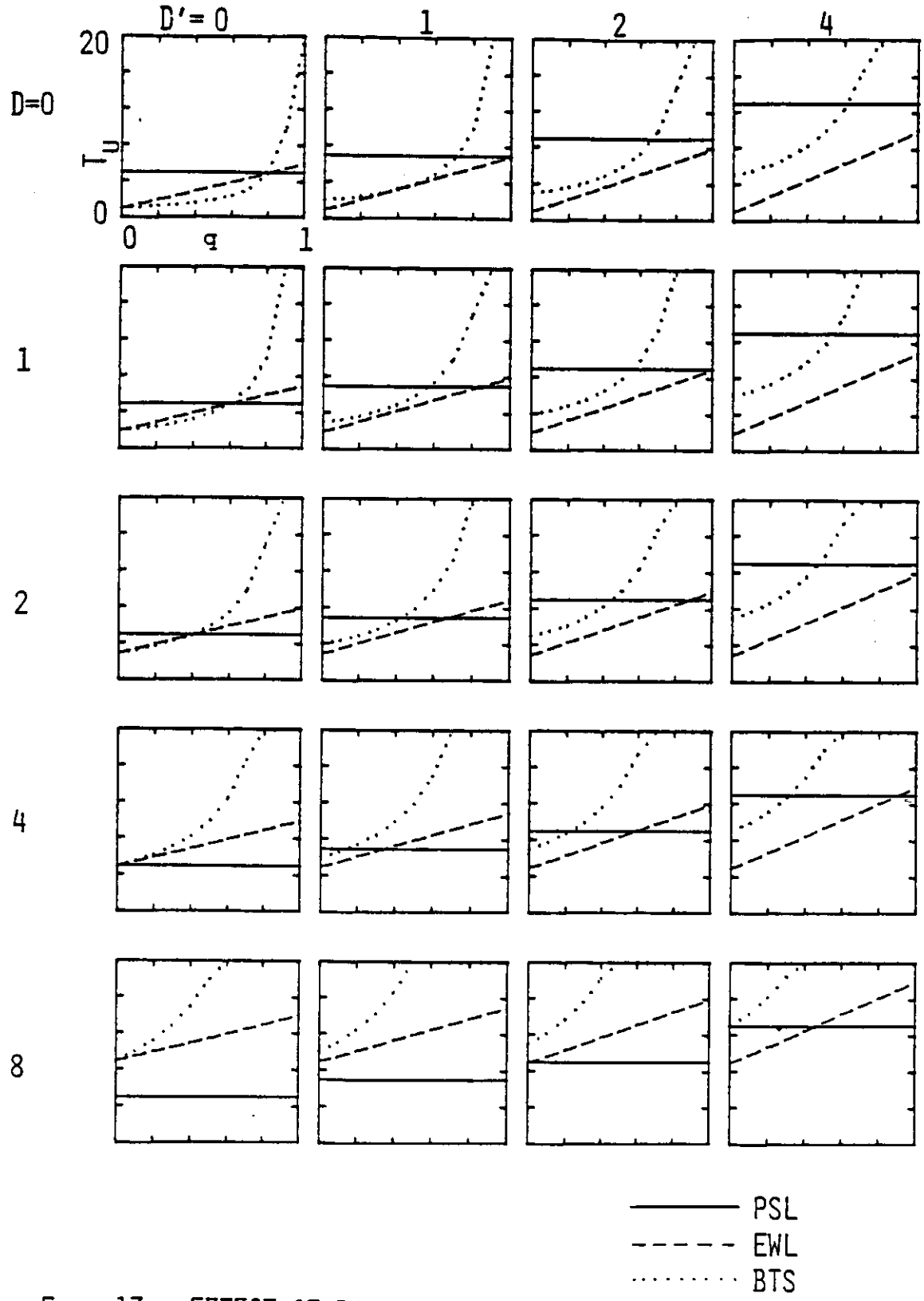


FIG. 13. EFFECT OF D' AND D ON T_U (W_L=4, Y=0, R_{MAX}=1)
(TIME IS NORMALIZED WITH RESPECT TO SITE RESPONSE TIME)

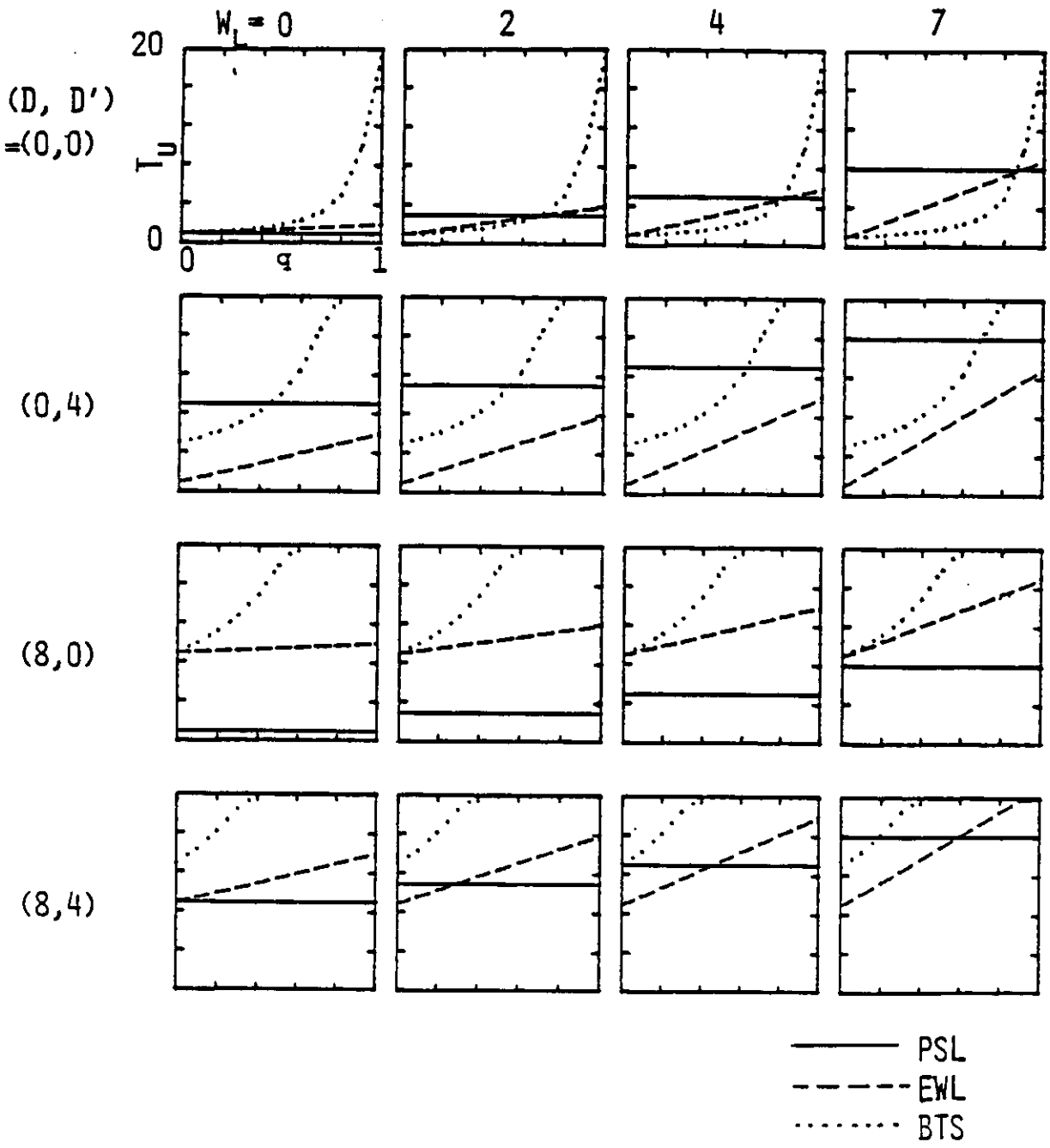


FIG. 14. EFFECT OF W_L ON T_U (R_{MAX}=1, Y=0)
(TIME IS NORMALIZED WITH RESPECT TO SITE RESPONSE TIME)

Fig. 14 plots T_U against q for selected values of the wait in the lock queue, W_L, with four selected pairs of D and D' values. $T_U(BTS)$ is unaffected by W_L; $T_U(PSL)$ increases linearly with W_L at a rate of 1, since all transactions wait in the lock queue; $T_U(EWL)$ increases linearly with W_L at a rate of q, since locking is used only when a transaction is restarted. Thus, BTS is preferred when W_L is large but q, D, and D' are small, since BTS is unaffected by W_L but $T_U(BTS)$ grows rapidly with the other parameters. Of the three protocols, PSL has the lowest T_U when W_L and D' are small and q is large, since both $T_U(BTS)$ and $T_U(EWL)$ increase with q and D, but PSL is unaffected by either q or D. EWL has a lower T_U than BTS for larger values of q and has lower T_U than PSL for larger values of W_L.

Fig. 15 plots T_U against q for selected values of update-log maintenance and database rollbacks, Y, as well as the relative delay for conflict checking of the slowest site, R_{max}, with four selected pairs of D and D' values. PSL and EWL are unaffected by Y and R_{max}, and thus are preferred as Y and R_{max} increase. $T_U(BTS)$ increases with both Y and R_{max}, especially with larger q (due to repeated restarts). Further, the effect of R_{max} is greatest with larger D and D', since the cost of distributed conflict checking increases with longer delays for message communication and processing.

4.4 Discussions

We observe that optimistic protocols, such as BTS, EWP, and EWL, have smaller execution response times, T_E, since they have no ICSD before the transaction's first execution. However, optimistic protocols which require an update-log, such as BTS, lengthen T_E for update-log maintenance (i.e., Y_M).

For update finalization response times, T_U, BTS performs poorly for several reasons. First, BTS has overhead for update-log maintenance and database rollbacks. Second, BTS uses distributed conflict checking, so $T_U(BTS)$ grows with the relative delay of conflict checking for the slowest site (i.e., R_{max}). Finally, BTS has repeated transaction restarts which further lengthens $T_U(BTS)$ and can saturate the computing and communication resources.

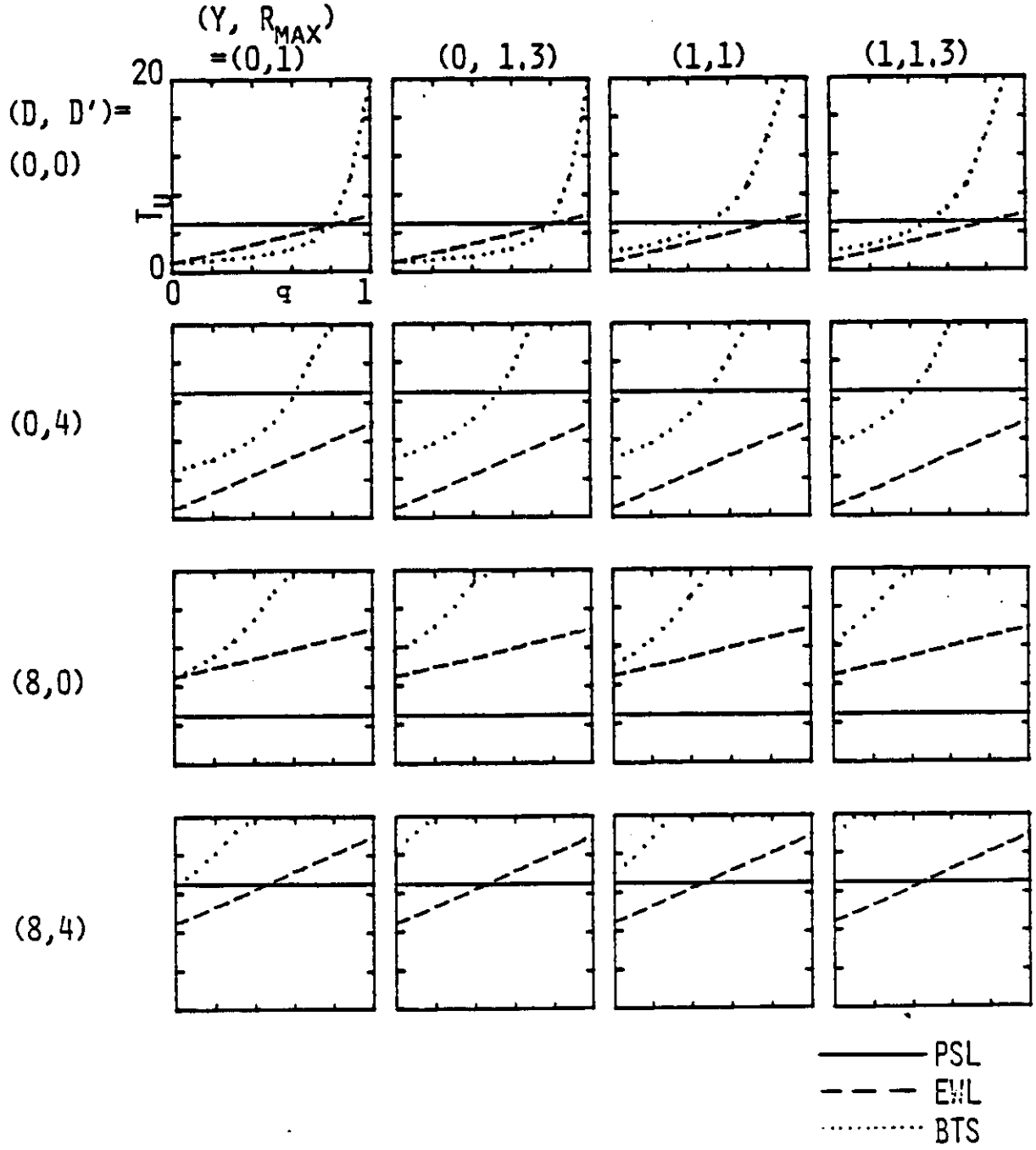


FIG. 15. EFFECT OF Y AND R_{MAX} ON T_U (W_L=4)

(TIME IS NORMALIZED WITH RESPECT TO SITE RESPONSE TIME)

Choosing between EWL and PSL for T_U depends on several parameters. $T_U(EWL)$ increases with delays for communicating and processing update messages, D, since EWL's update-request message contains the proposed update. Also, $T_U(EWL)$ increases with q, the probability of a restart. $T_U(PSL)$ is unaffected by both D and q. However, $T_U(PSL)$ increases more rapidly than $T_U(EWL)$ with delays for communicating and processing control messages, D' , since PSL exchanges two control messages (i.e., lock-request and lock-grant) and EWL only requires a single control message (i.e., lock-grant) when a transaction is restarted. Also, $T_U(PSL)$ increases more rapidly than $T_U(EWL)$ with waiting times for file locks, W_L, since PSL always has such waits while EWL incurs them only when a transaction is restarted.

Since EWP is a simplification of EWL, EWP's response times should never be larger than those for EWL. Specifically, $T_U(EWL)$ increases with q, W_L, and D' while $T_U(EWP)$ is unaffected by these parameters.

Choosing among PSL, BTS, and EWL requires knowledge of the relationships between the waiting times for file locks, W_L, and the probability of a transaction restart, q. Intuitively, BTS provides a distributed queueing system in which transactions are queued at their execution site but are delayed (via restarts) by transactions at other sites. Because a transaction accesses the same files regardless of the protocol used, one might expect a transaction in this distributed queueing system to wait for as many other transactions as if PSL had been used. However, the site waiting times, W_E, for BTS will be longer than those for PSL due to BTS repeated transaction restarts which increase site loads. Thus in general, $T_U(BTS)$ will be larger than $T_U(PSL)$. EWL has restarts but can guarantee that a transaction will be restarted at most once. Hence, site waiting times for EWL will be lower than those for BTS but higher than those for PSL. However, EWL waiting times for file locks may be smaller than those for PSL. Let λ_{TM} be the external arrival rate of transactions and λ_L be the arrival rate at the lock queue. Under PSL, $\lambda_L(PSL) = \lambda_{TM}$ since all transactions enter the lock queue. However, for EWL locking is required only when a transaction is restarted, so $\lambda_L(EWL) = q\lambda_{TM}$. Thus the utilization of the lock queue under EWL may be smaller than that for PSL, as a result $W_L(EWL)$ may be less than $W_L(PSL)$.

5. CONCLUSIONS

Optimistic protocols have appeal for distributed processing systems since they avoid inter-computer synchronization delays (ICSD) for execution response times, T_E. However, existing optimistic protocols have repeated transaction restarts which lengthen update finalization response times, T_U, and can saturate the computing and communication resources.

EWP and EWL are optimistic protocols that avoid repeated restarts. EWP has no restarts or database rollbacks and avoids deadlocks due to shared data access. However, EWP is not a general purpose protocols since it preserves only a limited form of serializability. EWL is a fully serializable extension of EWP. Although deadlocks are possible under EWL, EWL has no database rollbacks. EWL restarts a transaction at most once if the transaction's baseset does not change when it is restarted. To further reduce restarts, each site can independently and dynamically switch between primary site locking, PSL, which has no restarts and EWL. Such switching requires no additional messages or delays to synchronize protocol selection.

Analytic models were developed to study the response times, T_E and T_U, of EWP, EWL, PSL, and basic timestamps (BTS). The performance of these protocols depends on several parameters: q (the probability of restarting a transaction), D (delay for communicating and processing an update message), D' (delay for communicating and processing a control message), W_L (waiting time in the lock queue), Y (time for update-log maintenance and database rollbacks), and R_{max} (relative delay for the slowest site to do conflict checking). EWP and EWL have the smallest T_E, since neither requires update-log maintenance (unlike BTS) or has ICSD for T_E (unlike PSL). Unless D is quite large, $T_U(EWP)$ is smaller than T_U for the other protocols, since EWP has no transaction restarts or lock queue waits. $T_U(PSL)$ is unaffected by q, but it increases linearly with W_L and D' . $T_U(EWL)$ grows linearly with q and D, but is less affected than $T_U(PSL)$ by either W_L or D' , since these costs are incurred only when a transaction is restarted. $T_U(BTS)$ grows geometrically with q due to repeated transaction restarts, increases with Y since update-log maintenance is required for each transaction execution and database rollbacks occur whenever a transaction is retarted,

and increases with D, D' and R_{max} due to distributed conflict checking. Although $T_U(BTS)$ is unaffected by W_L, in general delays due to repeated transaction restarts exceed those for W_L.

We conclude that if limited serializability is acceptable, EWP should be used due to its simplicity and short response times which have great appeal for distributed processing systems. Otherwise, EWL has much appeal since it has good performance over a wide range of parameters and permits dynamic switching with PSL to further improve performance.

REFERENCES

ALSB76 Alsberg, Peter A. and John D. Day. "A Principle for Resilient Sharing of Distributed Resources," *Second Internation Conference On Software Engineering,* October 13-15, San Francisco, 1976, pp. 562-570.

BERN78 Bernstein, Philip A., James B. Rothnie, Nathan Goodman, and Christos A. Papadimitriou. "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)," *IEEE Transactions on Computers,* Vol. SE-4, NO. 3, May 1978, pp. 154-168.

BERN81 Bernstein, Philip A. and Nathan Goodman. "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys,* Vol. 13, No. 2, June 1981, pp. 185-222.

CHU80 Chu, Wesley W., Leslie J. Holloway, Min-Tsung Lan, and Kemal Efe, "Task allocation in distributed data processing," *IEEE Computer,* November 1980, pp. 57-69.

CHU84 Chu, Wesley W., Min-Tsung Lan, and Joseph Hellerstein, "Estimation of Intermodule Communication (IMC) and its Applications in Distributed Processing Systems," *IEEE Transactions On Computers,* to appear June 1984.

GARC78 Garcia-Molina, Hector. "Performance comparison of two update algorithms for distributed databases," *Third Berkeley Workshop on Distributed Data Management and Computer Networks,* August 29-31, 1978, pp. 108-119.

GRAY81 Gray, Jim, Paul McJones, Mike Balsgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger, "The Recovery Manager of the System R Database Manager," *Computing Surveys,* Vol. 13, No. 2, June 1981, pp. 223-242.

GREE80 Green, Michael L. et al., "A Distributed Real Time Operating System," *Proceedings of the Symposium on Distributed Data Acquisition, Computing, and Control,* December 1980.

KUNG79 Kung, H.T., and J.T. Robinson, "On Optimistic Methods for Concurrency Control," *Proceedings of the Very Large Database Conference,* Rio de Janeiro, October 1979.

LIN83 Lin, Wen-Te, Jerry Nolte, Philip Bernstein, and Nathan Goodman. "Distributed Database System Designer Handbook," Computer Corporation of America, contract number F30602-81-C-0028.

MILE81 Milenkovic, Milan. *Update Synchronization in Multiaccess Systems,* University of Michigan Research Press, 1981.

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, May 1979, pp. 188-194.

THOM78 Thomas, Robert. "A Solution to the Concurrency Control Problem for Multiple Copy Data Bases," *COMPCON78,* February 28 - March 3, 1978, pp. 56-62.

WALK83 Walker, Bruce, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. "The LOCUS Distributed Operating System," *Proceedings of the 9th Symposium in operating system principles,* October 10-13, 1983, pp. 49-70.

CHAPTER III

HEURISTIC MODULE ASSIGNMENT

HEURISTIC MODULE ASSIGNMENT

We have documented our research results on the measurement and estimation of *intermodule communication* (IMC) in our previous annual report [CHU82]. During the year of 1983, we continued the study on module assignment problems. The results are reported here in this chapter. Section 1 reviews existing methods for module assignment. Section 2 identifies key parameters that should be considered in a module assignment. Next, the concepts of module assignment tree and enumeration procedure are described (Section 3). An objective function based on the concept of *minimum bottleneck* is proposed as a criterion to be used in the search for good module assignments. Then an example is presented (Section 4) where the objective function was applied to the DPAD system to select good module assignments. Those selected assignments were simulated via the UCLA DPAD simulator and they did yield excellent port-to-port time performance. However, due to the complexity of the problem, an optimal solution is difficult, if not impossible, to obtain. A heuristic algorithm for module assignment is thus proposed in Section 5 and it is shown to generate good assignments.

1. EXISTING MODULE ASSIGNMENT METHODS

We can divide existing module assignment methods into 4 categories: queueing model approach, graph theoretic approach, integer 0-1 programming approach, and heuristic approach. Existing queueing models do not consider IMC and therefore do not provide good assignments.

1.1 Graph Theoretic Approach

In general, this approach can handle only 2-processor module assignment problems. Each module is represented by a node in a graph and IMC cost between each

pair of modules is represented by the weight of a nondirected arc connecting the two nodes [JENN77, RAO79, STON77, STON78]. The arc weight then becomes the *interprocessor communication* (IPC) if the pair of modules are not coresident on a processor. Any pair of coresident modules is assumed to have zero IPC cost. An additional node is also provided for each processor; an arc between a module node and *one* of the two processor nodes represents the processing cost of running the module on *the other* processor.

The module assignment strategy in this model is to minimize total cost, defined as the sum of processing cost and IPC cost. In order to represent the assignment of modules to processors an assignment matrix X is defined such that

$$x_{i,k} = \begin{cases} 1 & \text{if } \textit{module } M_i \textit{ is assigned processor } P_k \\ 0 & \textit{otherwise} \end{cases}$$

Processing cost is given by the Q matrix,

$$Q = \left\{q_{i,k}\right\}, \qquad i=1, \ldots m, \qquad k=1, \ldots n$$

where $q_{i,k}$ represents the processing cost for module M_i on processor P_k. A value of infinity, $q_{i,k} = \infty$, implies that module M_i cannot be executed at processor P_k.

Let $v_{i,j}$ represent the IMC volume between M_i and M_j. The total cost for processing a given task can then be expressed as an objective function of the assignment X.

$$Cost(X) = \sum_k \sum_i \left\{ q_{i,k}x_{i,k} + \sum_{l \neq k} \sum_{j<i} w v_{i,j} x_{i,k} x_{j,l} \right\} \qquad (1)$$

The first term of eq. (1) represents the processing cost for each module on its assigned processor. The second term represents the IPC cost between non-coresident modules.

The normalization constant w is used to scale processing cost and IPC cost to account for any differences in measuring units. The minimum-cost module assignment is obtained by performing a *min-cut algorithm* on the graph [HARA69].

While this method is conceptually simple, it has several limitations. First, an extension of the min-cut algorithm to an arbitrary number of processors quickly becomes computationally intractable. An extension to four or more processors has been proposed for cases where the IMC pattern can be constrainted to be a tree [STON78]. Second, this method provides neither a mechanism for representing limited resources in memory size or processor capacity, nor a mechanism for *load balancing among processors.* It might result in a quite unbalanced module assignment where one processor becomes the bottleneck and the response time becomes unacceptable. Third, the method assumes static or one-time execution of modules. But, in almost all real-time systems, program modules reside in the systems during entire mission time and a module is invoked (i.e., *enabled)* to execute by each occurrence of certain type(s) of events. Finally, an assignment with the minimum cost for eq. (1) does not guarantee a good response time.

1.2 Integer 0-1 Programming Approach

This method stems from the file allocation problem where the model is formulated as an optimization problem and is solved via a mathematical programming technique [CHU69]. As with the graph theoretic approach, the goal is to achieve optimal system performance by minimizing the total cost defined in eq. (1) over the module assignment X. In addition, the minimization is done subject to some constraints which may be imposed by a given environment or the design specifications. For example, a limited-memory constraint is represented by

$$\sum_i s_i x_{i,k} \leq R_k , \quad k=1, \ldots n$$

where s_i represents the amount of memory storage required by module M_i and R_k represents the memory capacity at processor P_k.

Similar to the graphic-theoretic approach, the integer 0-1 programming approach has the following disadvantages: it is time-consuming to solve and can only solve problems of limited size, it assumes one-time execution of modules, and it fails to guarantee good response time.

1.3 Heuristic Approach

Gylys and Edwards proposed a heuristic algorithm for module clustering [GYLY76]. The process of assigning two modules to the same processor is called *fusion*. The algorithm searches for a pair of modules with the largest IMC and checks to see whether the fusion of the two modules satisfies the real-time and memory constraints. If it does, that pair is fused. Otherwise, the pair with the next largest IMC is chosen as a possible candidate. The fusing process continues until all eligible pairs are allocated. Although simple and fast, this approach does not yield load-balanced module assignments and suffers from the bottleneck in the heavily loaded processor.

After examining the above approaches, we are motivated to develop a heuristic algorithm that can remedy these shortcomings. We shall first define in the following the key parameters that profoundly affect module assignments.

2. KEY PARAMETERS FOR MODULE ASSIGNMENT

The two parameters that play important roles in module assignment are *accumulative execution time* (AET) and IMC [CHU82]. AET for a module M_j is the total execution time of this module within a time interval, (t_h, t_{h+1}). That is,

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

where $N_j(t_h,t_{h+1})$ = number of times module M_j executes during (t_h,t_{h+1}), and $X_j(t_h,t_{h+1})$ = average execution time (or average ET) per execution of M_j during (t_h,t_{h+1}). Both the average ET and the AET can be expressed in terms of machine-language instructions (MLI) executed. Although the execution time of one machine-language instruction varies from instruction to instruction, we can find the *mean* instruction execution time given the mix ratios for various different instructions. Our study shows that both the number of module executions and the AET are almost independent of module assignments if a *fixed* offered load is input to the distributed system. Let us see an example with the Distributed Processing Architecture Design (DPAD) system which was developed to manage the data processing and radar resources for a space defense application [GREE80, HOFF80]. This DPAD system will be used as an example throughout the chapter. A portion of its control-and-data-flow graph is given in Fig. 1. Fig. 2 shows the number of times module M_8 executes during 100-msec intervals under a scenario with 40 objects. And Fig. 3 displays the AET for M_8 during the intervals. Note that in both Figs. 2 and 3, the five curves corresponding to five different module assignments are so close to each other. Fig. 4 plots the AET for all 20 modules in the DPAD; this information will be used later as an input to our module assignment algorithms.

Our second parameter, IMC, is also characterized by little variation for different module assignments if a constant load is offered to the system. For example, Fig. 5 exhibits the measured IMC from M_9 to M_{13} from the DPAD simulation; five curves representing five different module assignments are almost identical. Fig. 6 displays all IMC existing in the DPAD. Each plot $IMC(i,j)$ represents the IMC sent from M_i to M_j. This $IMC(i,j)$ involves all the files which are updated by M_i and read afterwards by M_j. An alternative way to present the IMC is shown in Fig. 7, where each plot $V(i,k)$ shows

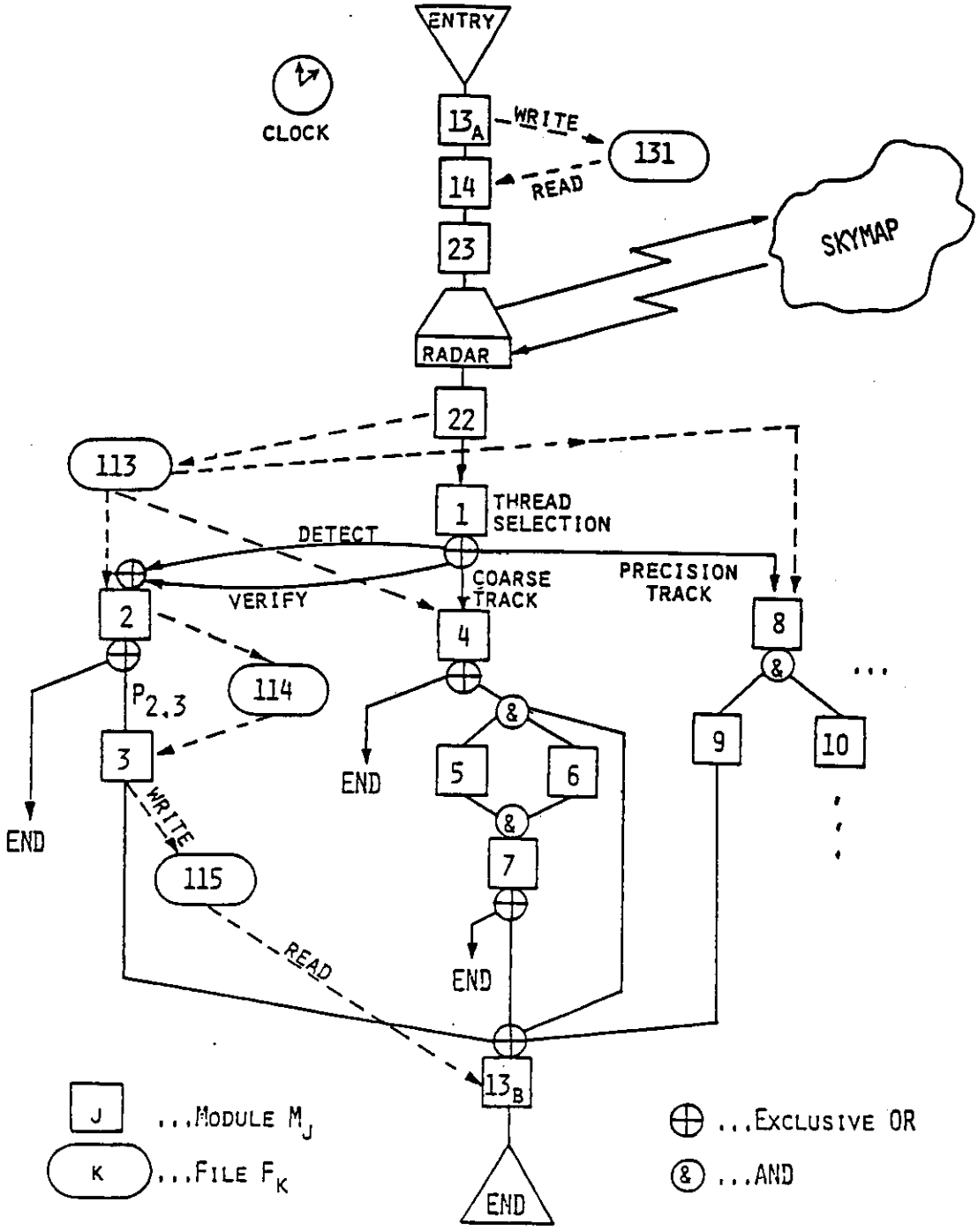


FIG. 1. A CONTROL-AND-DATA-FLOW GRAPH FOR A SPACE DEFENSE APPLICATION TASK

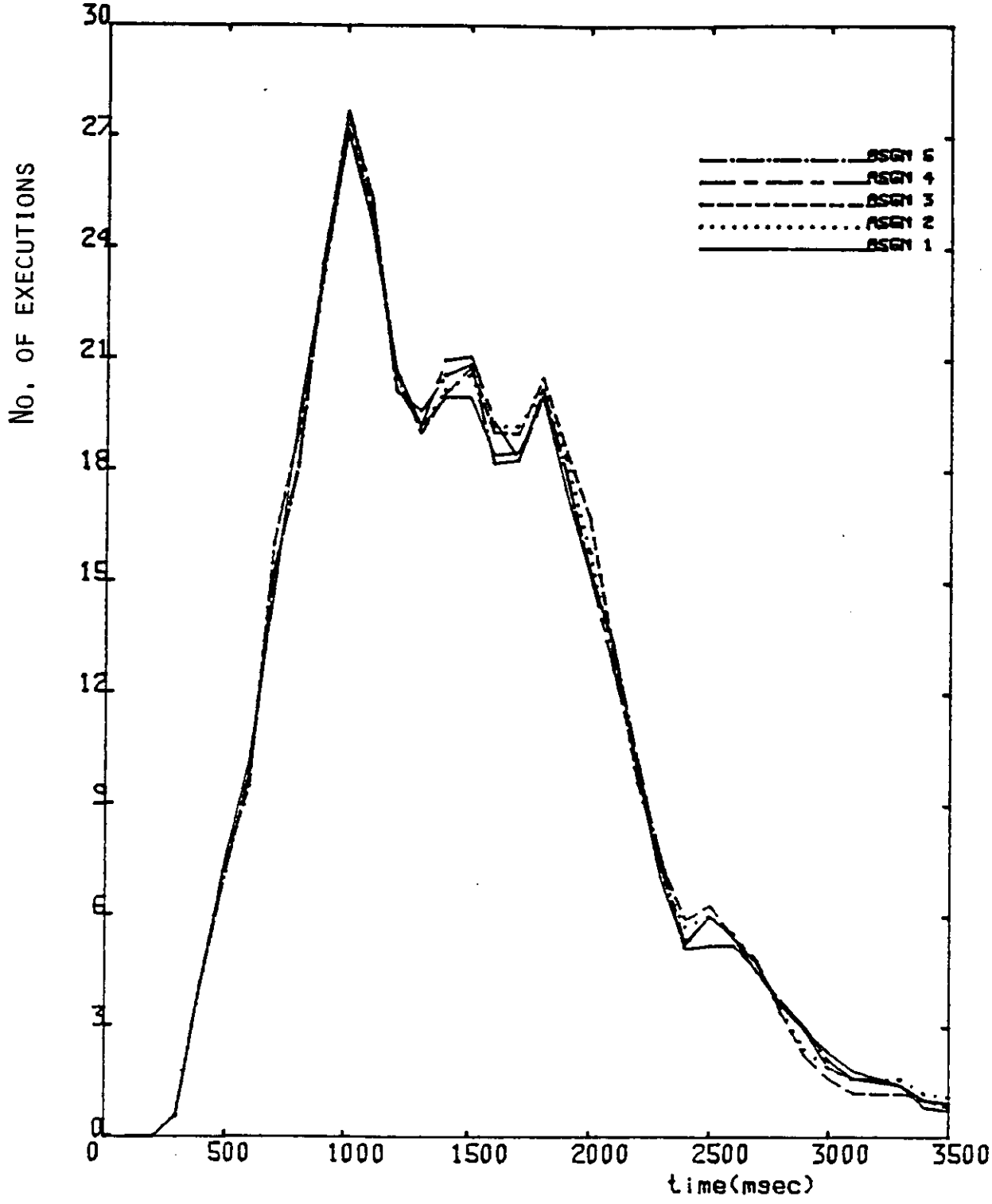


FIG. 2 NO. OF MODULE M_8 EXECUTIONS, N_8 (T, T + 100MSEC)

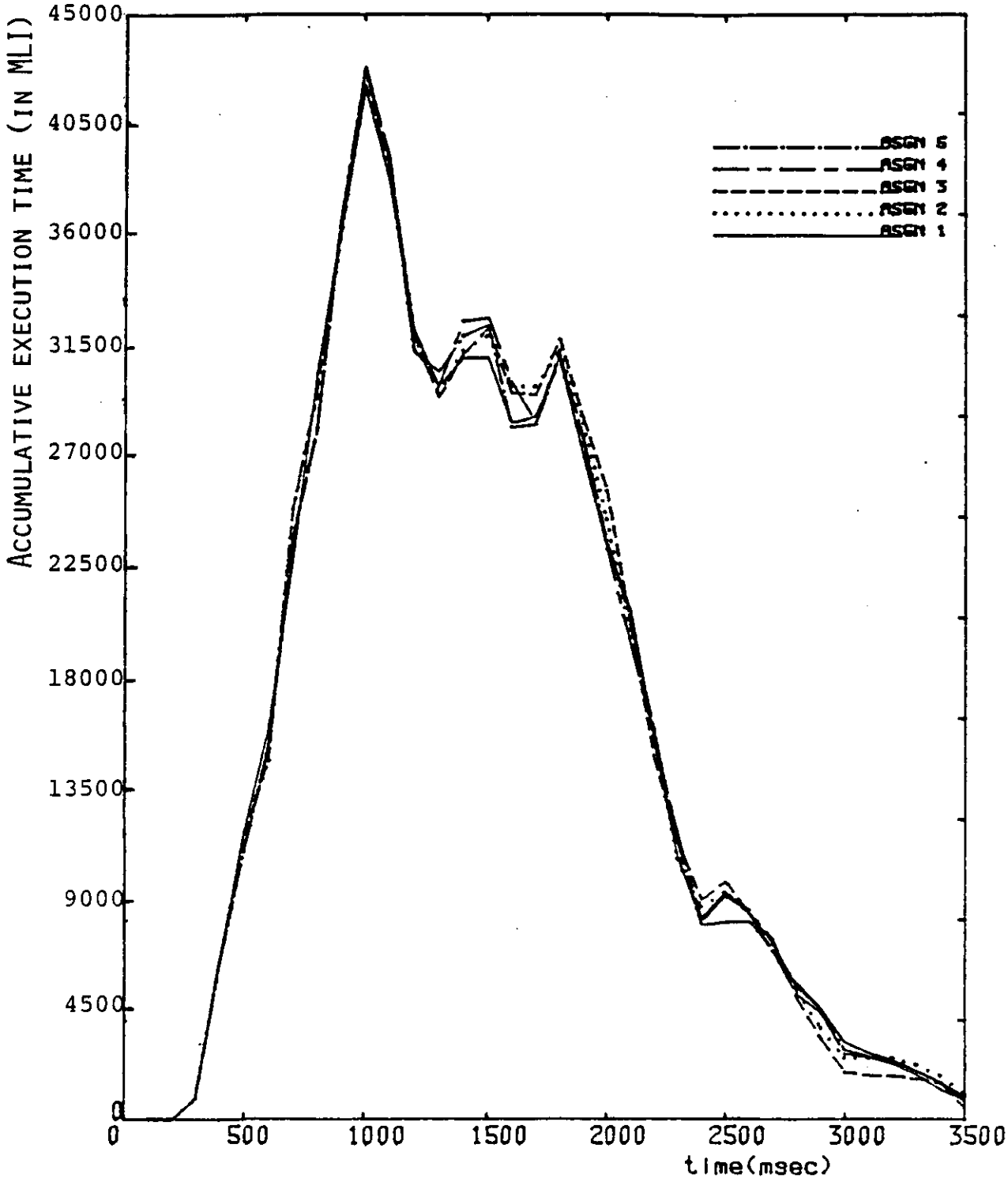


FIG. 3. MODULE M_8 ACCUMULATIVE EXECUTION TIME, $T_8(T,T+100\text{MSEC})$

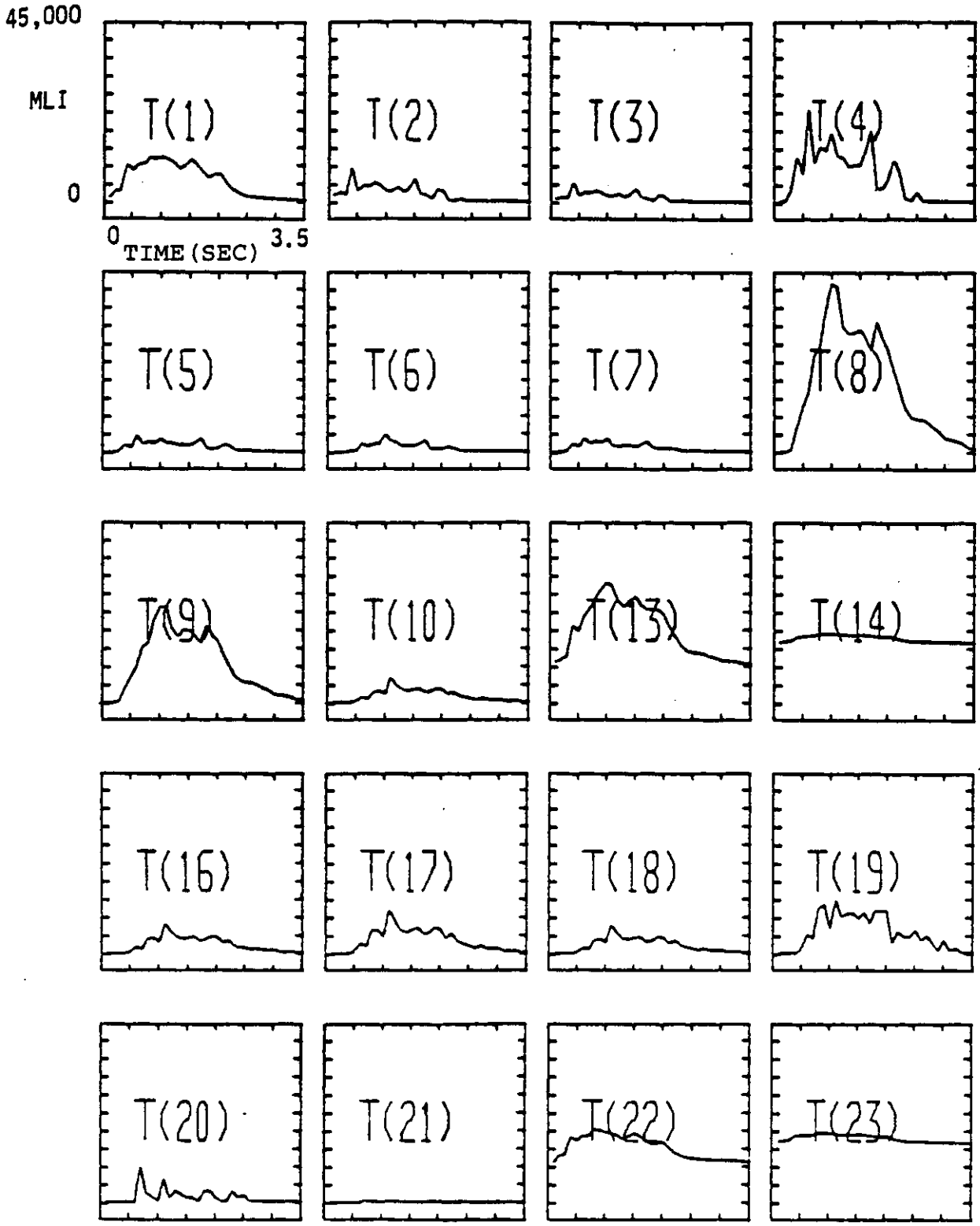


FIG. 4 . ACCUMULATIVE MODULE EXECUTION TIME

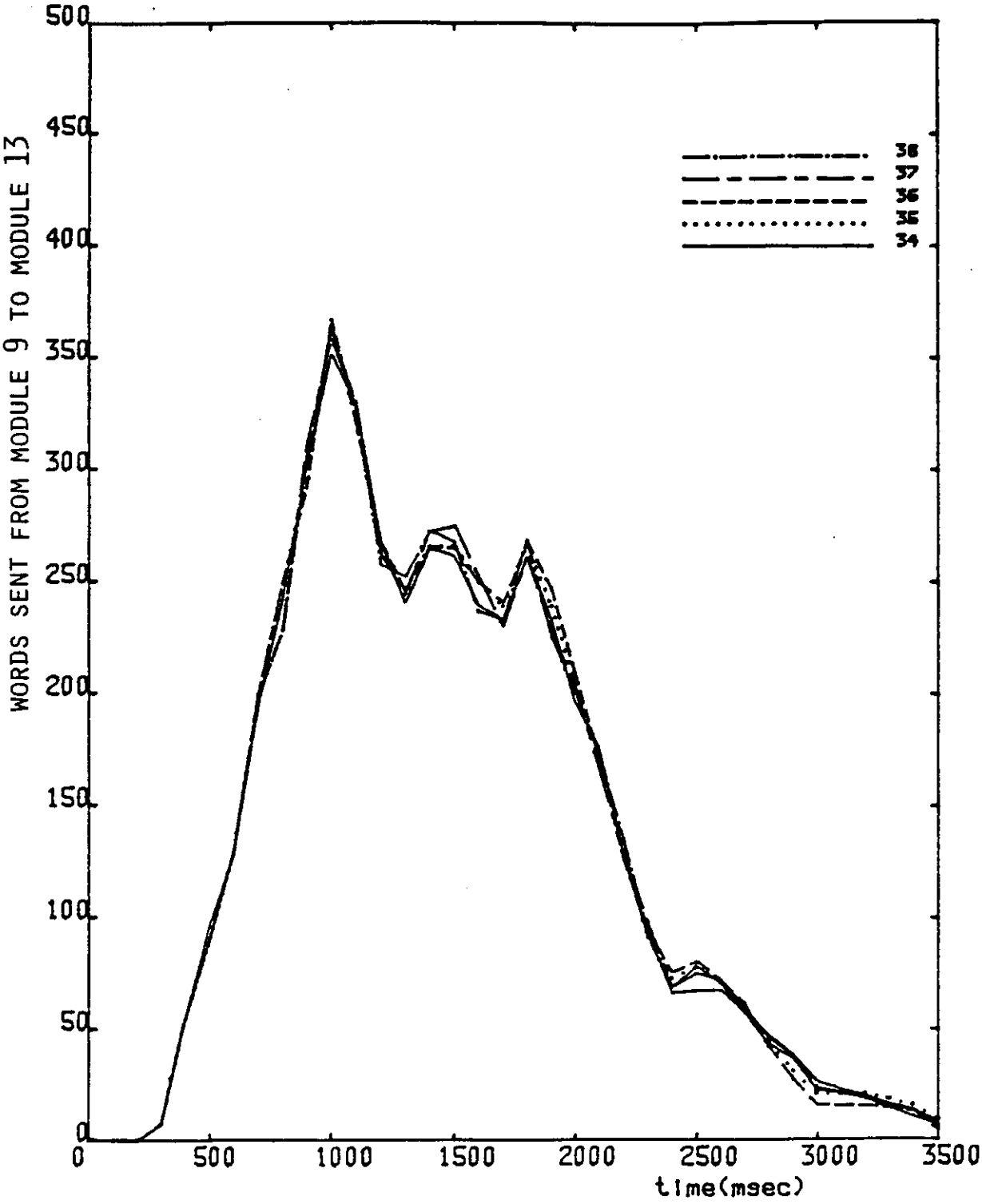


FIG. 5. IMC FROM M_9 TO M_{13}, $IMC_{9,13}$ (T, T + 100MSEC)

the amount of updates (in words) to a file F_k by a module M_i.

IPC, on the other hand, varies closely with module assignments because the occurrence of IPC between two communicating modules depends on whether these two modules are assigned to different processors. In the DPAD, modules communicate through shared files. If two modules reside on different processors, the file is replicated on each processor. When a processor updates the file, it updates the copy on its local processor. It then sends the updates to remote processors, resulting in IPC which contributes to CPU load on both the sending and receiving processors. The IPC is eliminated when a pair of communicating modules are assigned to the same processor because they use the same local file copy.

3. PROPOSED OBJECTIVE FUNCTION

Let us first define the concept of *module assignment tree*. Consider a software system which has been partitioned into a fixed number of program modules. Given the control and data flow graph, the problem of module assignment is to *assign* the program modules into a smaller number of processors in a way to meet the performance requirements. The main design requirement in real-time systems is to meet the response time constraint, or port-to-port (P-T-P) time in BMD terminology.

Since each module can be assigned to any of the m processors, there are m^n different ways to assign n modules to m processors, assuming that each module is assigned to one and only one processor. This can be represented by an assignment tree with m^n leaves, each leaf corresponding to a possible assignment. This tree has n levels, each standing for a module. At each non-leaf node there are m downward branches, each representing the choice of a possible processor to host the particular module. Fig. 8 shows an example with $n = 23$ and $m = 3$.

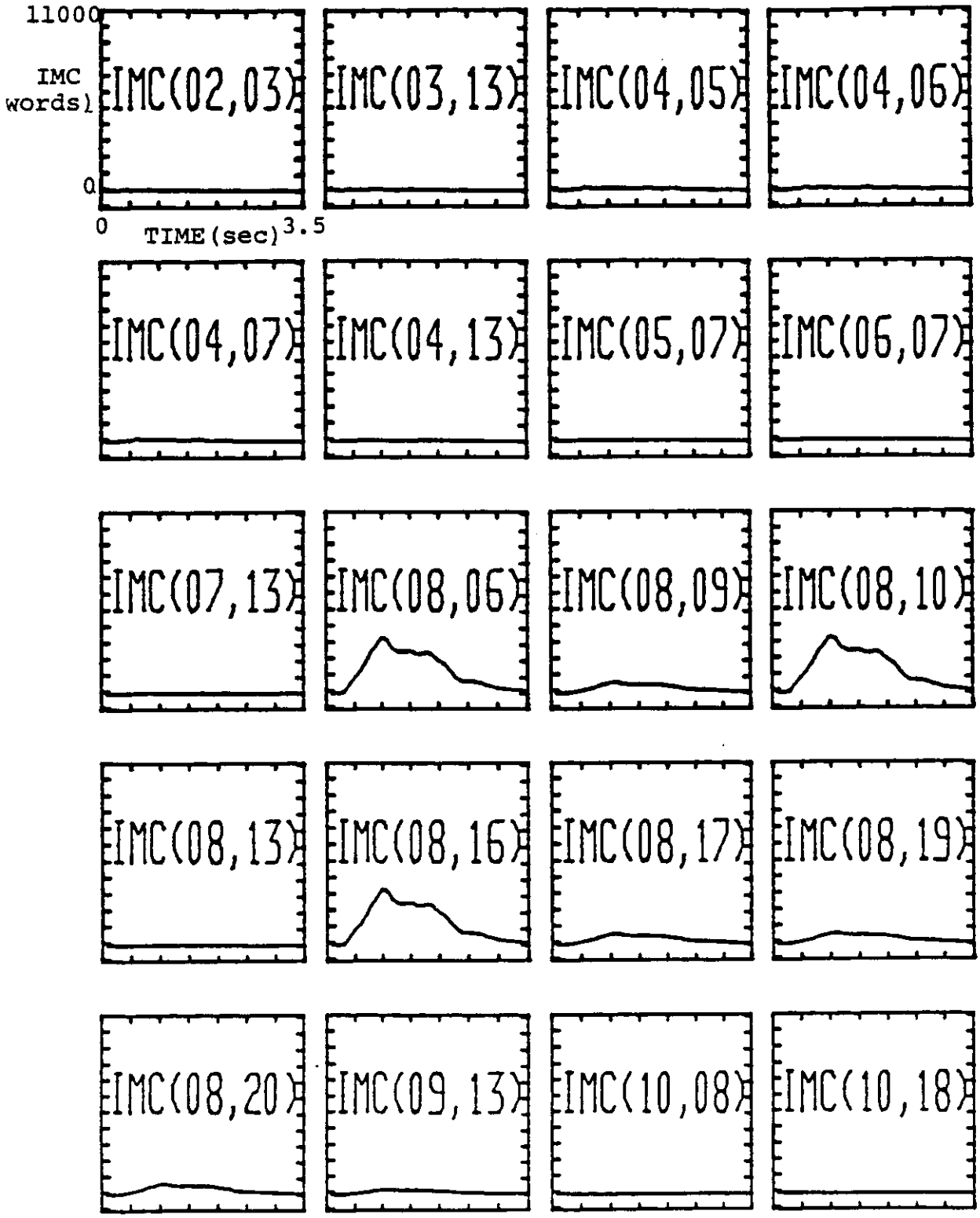


FIG. 6. IMC BETWEEN MODULE PAIRS

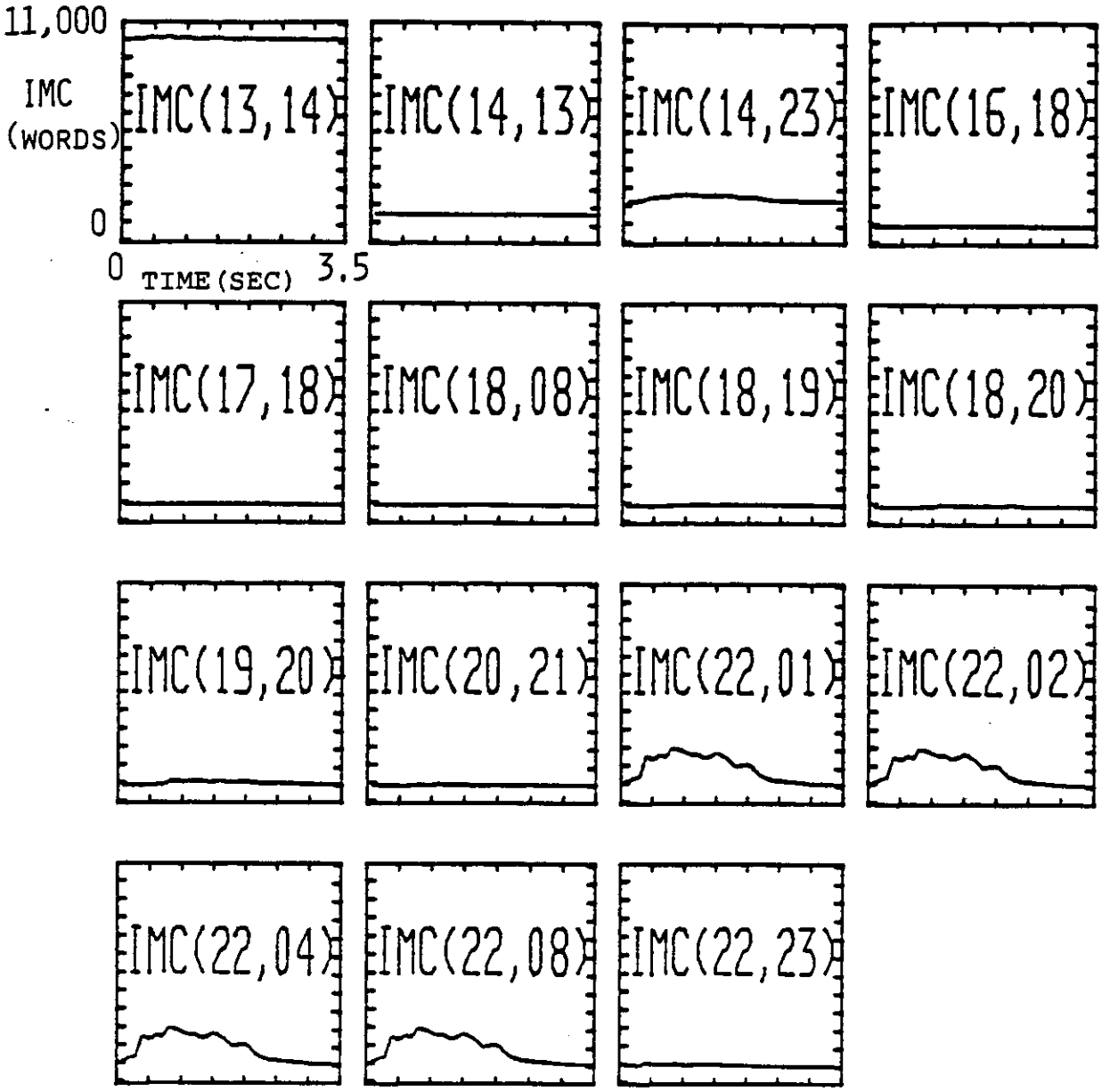

11,000
IMC
(WORDS)
0
0
TIME(SEC)
3.5
IMC(13,14)
IMC(14,13)
IMC(14,23)
IMC(16,18)
IMC(17,18)
IMC(18,08)
IMC(18,19)
IMC(18,20)
IMC(19,20)
IMC(20,21)
IMC(22,01)
IMC(22,02)
IMC(22,04)
IMC(22,08)
IMC(22,23)

FIG. 6. (CONT'D)

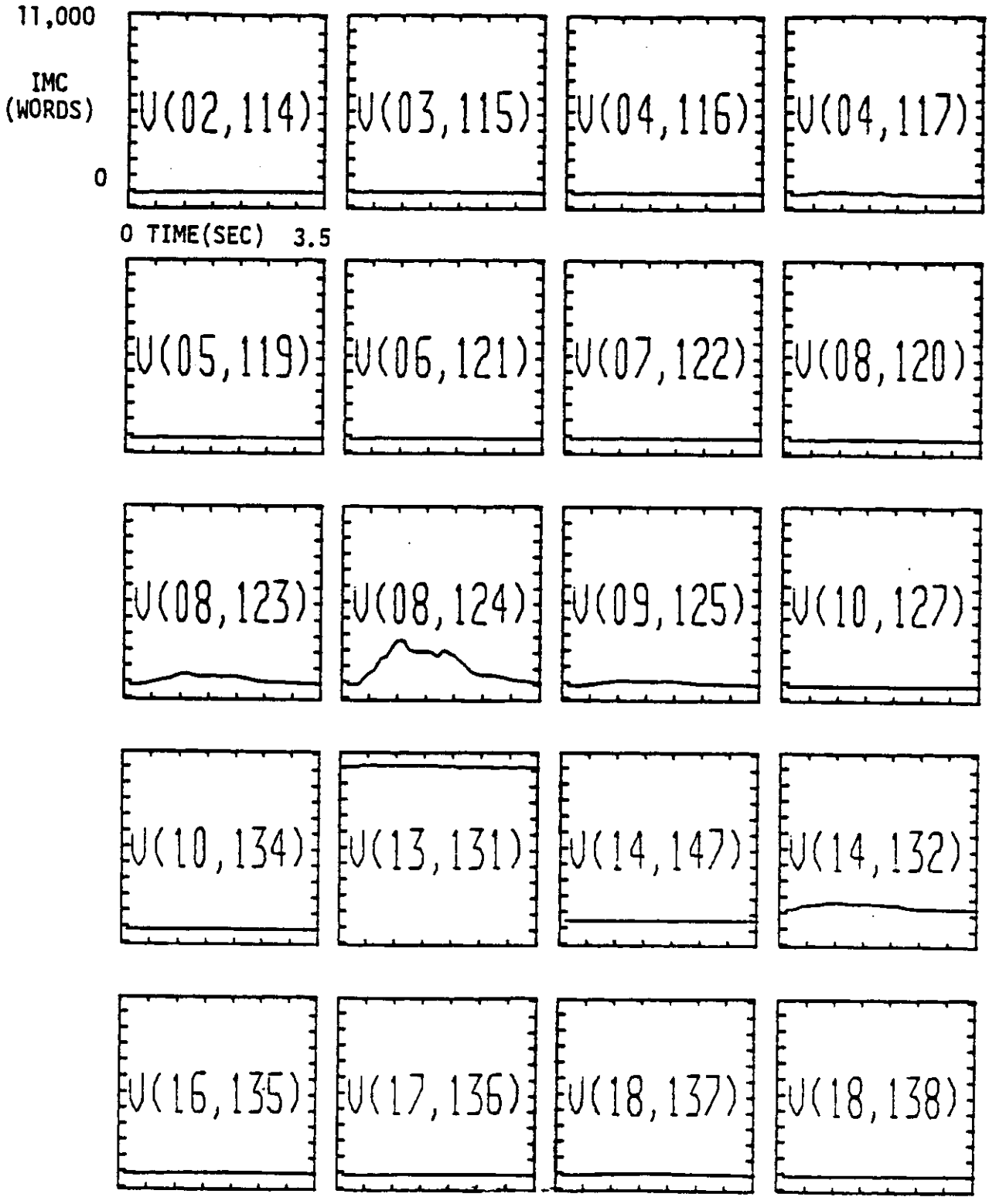


FIG. 7. MODULE-FILE (M-F) IMC FOR THE EXAMPLE

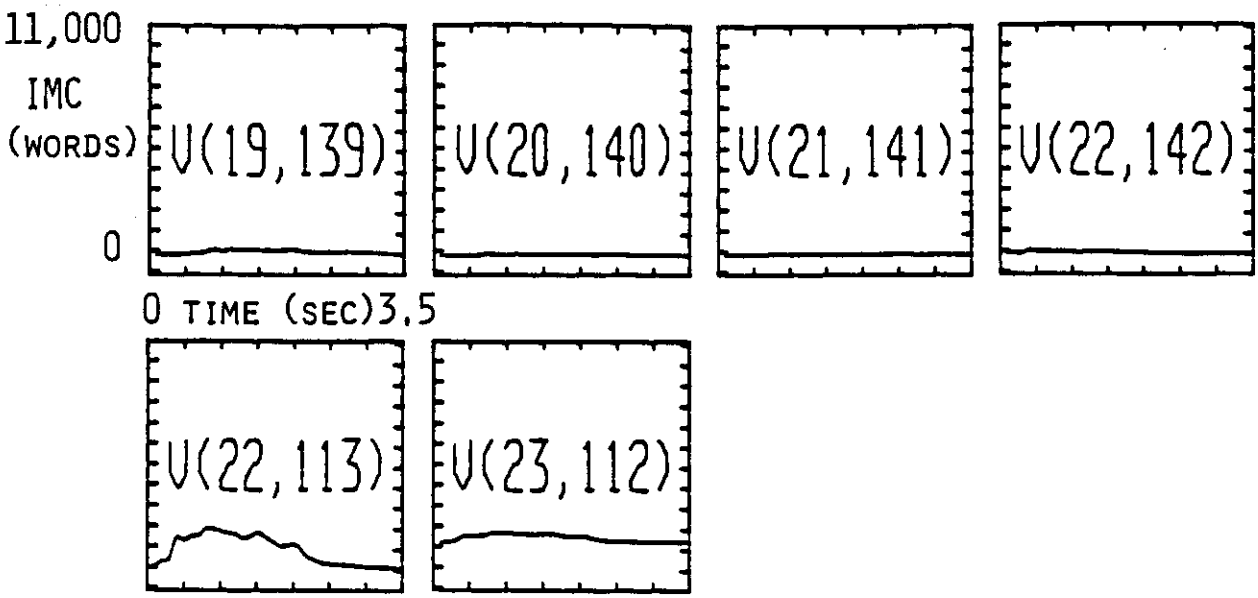
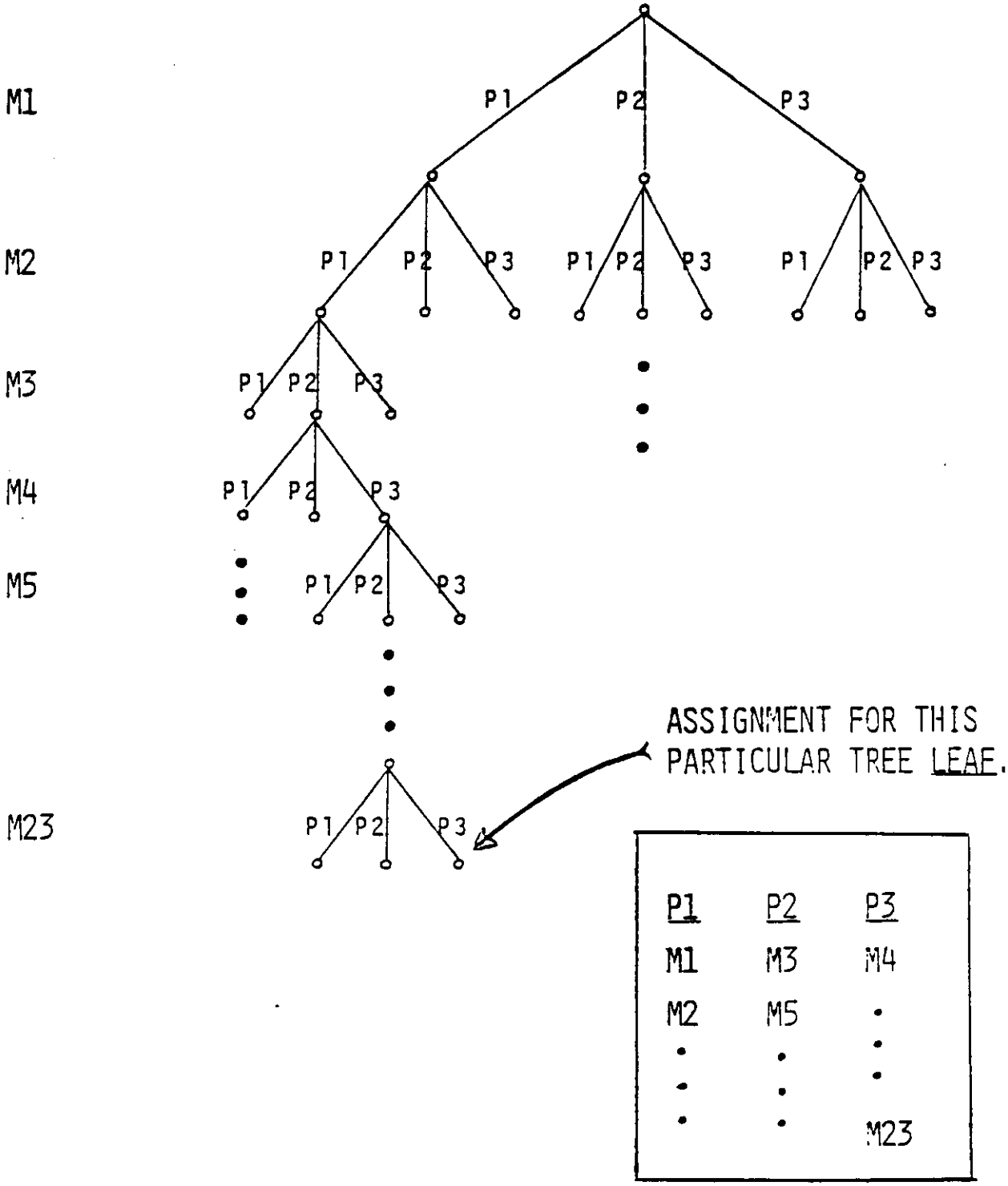


FIG. 7 (CONTINUED)

FIG. 8. AN EXAMPLE OF AN ASSIGNMENT TREE



An objective function for module assignment is a mathematical function which yields a numerical value when evaluated for a candidate module assignment. This value might represent a performance measure such as total system cost or expected response time. The module assignment problem is to search through the leaves of the assignment tree for the particular assignment which yields the ***minimum*** function value (or maximum values in some cases, e.g., maximum system throughput). An ***exhaustive search*** through all leaves (called an ***enumeration)*** is usually undesirable because of the enormous amount of time involved. For example, enumeration for a tree with 3^{20} leaves takes around 10 days if each leaf (i.e., each assignment) requires 0.25 msec of evaluation time.

Let us define the *processor work load* (or simply, "processor load") as the sum of: a) load due to program module execution, and b) load due to IPC. These two components should be expressed in the same measurement unit. We take a simple approach to this by *converting* the number of words transferred in IPC into the machine-language instructions (MLI) spent by the processor for transferring or receiving the IPC.

After identifying the module execution time and IPC as the key parameters for module assignment, we are motivated to propose an objective function for module assignment. IPC minimization by itself will not produce a good assignment. In fact, a minimum-IPC assignment will assign all program modules to a single processor; this processor would be saturated while others are idle. According to queueing theory, a saturated processor results in long, unacceptable delay for the modules running on it. A saturated processor in a distributed system is a *bottleneck* which causes blockage to overall system data flows [TSUC80]. This suggests that a 3-processor assignment resulting in processors being 58%, 60%, and 61% utilized might have a better response time than an assignment with 20%, 40% and 90%-utilized processors although the former has a higher total processor load (due to more IPC).

For an assignment, let us call the processor with the highest load a ***bottleneck processor***. The objective function proposed here is the work load (MLI's) of the bottleneck processor. And the purpose of the tree enumeration is to find the assignment with the ***minimum bottleneck load***.

Under a given assignment X, the work load L(s;X) on a processor P_s (s = 1, 2, ... , m) is

$$L(s;X) = \sum_{j=1}^{n} x_{js} T_j + \sum_{\substack{t=1 \\ t \neq s}}^{m} \left[IPC(s,t;X) + IPC(t,s;X) \right] \tag{2}$$

The first term accounts for each module's AET for all modules assigned to processor P_s. The second term is IPC overhead which consists of two parts: the first part is the overhead due to the P_s–originated IPC to be sent to other processors, while the second part is for incoming messages from other processors. In the DPAD example, file-update messages dominate the IPC traffic. We therefore ignore other types of IPC such as module enablement messages and system control messages. The total overhead due to outgoing IPC at P_s is thus

$$\sum_{\substack{t=1 \\ t \neq s}}^{m} IPC(s,t;X) = w \sum_{j=1}^{n} x_{js} \sum_{k=1}^{K} V_{jk} \sum_{\substack{t=1 \\ t \neq s}}^{m} \delta_{kt} \tag{3}$$

where K is the number of files used in the distributed system; V_{jk} is the M-F IMC message volume sent from M_j to update the replicated file F_k at a remote processor P_t; δ_{kt} indicates whether a replicated copy of F_k resides at P_t; the term $\sum_{\substack{t=1 \\ t \neq s}}^{m} \delta_{kt}$ gives the number of F_k's remote copies that must be updated; and w is a weighting constant to convert the message volume into MLI's. Similarly, the total overhead at P_s due to incoming IPC from all remote sites P_t's is

$$\sum_{\substack{t=1 \\ t \neq s}}^{m} IPC(t,s;X) = w \sum_{\substack{t=1 \\ t \neq s}}^{m} \sum_{j=1}^{n} x_{jt} \sum_{k=1}^{K} V_{jk}\, \delta_{ks} \tag{4}$$

In a system with message-broadcasting capability, a file update need only be sent out once; the term $\sum_{\substack{t=1 \\ t \neq s}}^{m} \delta_{kt}$ in eq. (3) then reduces to 1.

The objective function is the load of the bottleneck processor, i.e.,

$$Bottleneck(X) = \max_{s=1}^{m} \left\{ L(s;X) \right\} \tag{5}$$

The module assignment problem is to search through the assignment tree for the assignment that yields the *minimum* bottleneck load *among all possible assignments*, i.e.,

$$\underset{X}{Minimize} \left\{ Bottleneck(X) \right\} \tag{6}$$

or,

$$\underset{X}{minimize} \left\{ \max_{s=1}^{m} \left[PT(s) + IPC(s) \right] \right\} \tag{7}$$

where PT(s) and IPC(s) are the total module execution time and the total IPC overhead incurred at site s. Eq. (7) is different from the equation proposed in [STON77] which is to minimize the sum of processor loads, i.e.,

$$\underset{X}{minimize} \left\{ \sum_{s=1}^{m} \left[PT(s) + IPC(s) \right] \right\} \tag{8}$$

4. PERFORMANCE OF PROPOSED OBJECTIVE FUNCTION

In this section we evaluate the performance of the proposed objective function for selecting a module assignment. A FORTRAN program was written to compute the proposed objective function for every assignment in the DPAD assignment tree. Ten

good assignments were selected (see Appendix A). These assignments were then simulated with the UCLA DPAD simulator to verify if they yield top port-to-port time performance.

In that FORTRAN program, values for a module's AET and the IMC terms used in the proposed objective function are derived from the measurements shown in Figs. 4 and 7. AET for each module M_j is represented as a single value T_j in the objective function (see eq. (2)). IPC information between a module and a file is also represented as a single value $V_{j,k}$ (see eq. (3)). However, the measurement results show the AET for each 100-msec interval and this value varies from interval to interval (see Fig. 4). Information contained in such a series of time-varying AET must be *compressed* into a single value for the objective function. Since we are most concerned with system performance during the peak-load time, we examine the measurement curves, identify the peak-load period, and compute an average from all those measured values during that period. Table 1 shows the average AET obtained in this manner for every module in the DPAD experiment where the identified peak-load period is from 1.0 sec to 2.0 sec of mission time. That is, each value T_j for module M_j is an average of ten measured values $T_j(1.0sec, 1.1sec), T_j(1.1sec, 1.2sec), \ldots, T_j(1.9sec, 2.0sec)$.

The same procedure is used to derive IMC information for the objective function from the IMC measurements. The results are shown in column 3 of Table 2. Column 2 shows the file(s) updated by the write module. Column 4 lists all the modules which read the updated file. If a read module for a file and its associated write module are separated on different processors, both processors would have a copy of the file and the IPC occurs for updating the replicated file copy.

TABLE 1

ACCUMULATIVE EXECUTION TIME (AET) PER 100 MSEC

(UNIT: MLI)

Module	AET
M1	8865
M2	2700
M3	1590
M4	10410
M5	1860
M6	1950
M7	1680
M8	32055
M9	18600
M10	3360
M11	0
M12	0
M13	25305
M14	16860
M15	0
M16	4170
M17	6240
M18	3975
M19	9705
M20	2010
M21	195
M22	16410
M23	10725

MLI = Machine Language Instructions

EACH AET IS AN AVERAGE ACROSS THE PEAK-LOAD PERIOD, FROM 1.0 SECOND TO 2.0 SECONDS.

TABLE 2

FILE UPDATE IMC (in MLI) PER 100 MSEC

Write Module	File Updated	IMC Size	Read Modules
M1	none		
M2	F114	124	M3
M3	F115	144	M13
M4	F116	112	M13
	F117	314	M5, M6, M7
M5	F119	68	M7
M6	F121	68	M7
M7	F122	67	M13
M8	F120	62	M13
	F123	1568	M6, M10, M16, M9, M17, M19, M20
	F124	6387	M6, M10, M16
M9	F125	806	M13
M10	F127	1	M8
	F134	1	M18
(M11)	Module Not Implemented		
(M12)	Module Not Implemented		
M13	F131	30371	M14
M14	F147	1800	M13
	F132	5019	M23
(M15)	Module Not Implemented		
M16	F135	100	M18
M17	F136	100	M18
M18	F137	229	M19
	F138	36	M8
	F139	244	M20
M19	F139	599	M20
M20	F140	62	M21
M21	F141	32	Radar
M22	F142	242	M23
	F113	4593	M1, M2, M4, M8
M23	F112	5112	Radar
Radar	F111	14737	M22

* EACH TRANSFERRED WORD = 3 MLIs

EACH IMC SIZE IS AN AVERAGE ACROSS THE PEAK-LOAD PERIOD, FROM 1.0 SECOND TO 2.0 SECONDS.

The ten assignments (Fig. 9) selected (see Appendix A) are simulated with the DPAD simulator and the results are presented here. Figs. 10(a) and (b) show the CPU utilization for the first two assignments. Note that in each figure, loads of the 3 processors are quite balanced during the peak-load period between 1.0 sec. and 2.0 sec. Assignments #3 through #10 in Fig. 9 exhibit similar load-balanced behavior. This coincides with our expectation and thus suggests that our *processor load and bottleneck model* (eqs. (2) thru (5)) is a good approximation to the processor loads under a given module assignment. On the other hand, Figs. 11(a) and (b) show the processor loads for an arbitrary assignment and Holloway's knowledge-guessed manually generated assignment [HOLL82] which up to the time had been the best assignment (for port-to-port time) known to the author. (The knowledge-guessed assignment is obtained by a combination of intuitive insight and trial and error.) These last two assignments are less load-balanced and their bottleneck loads are much higher.

Fig. 12 shows the Precision-Tracking P-T-P time for the top 9 assignments and the arbitrary assignment. We note that the performance difference between a good and a bad assignment can be very large.

Fig. 13 compares the port-to-port time for the Detect/Verify thread between the top 9 assignments and Holloway's assignment, and Fig. 14 does the same for the Precision Tracking thread. Our experiments confirm that the proposed objective function generates good module assignments.

5. HEURISTIC MODULE ASSIGNMENT

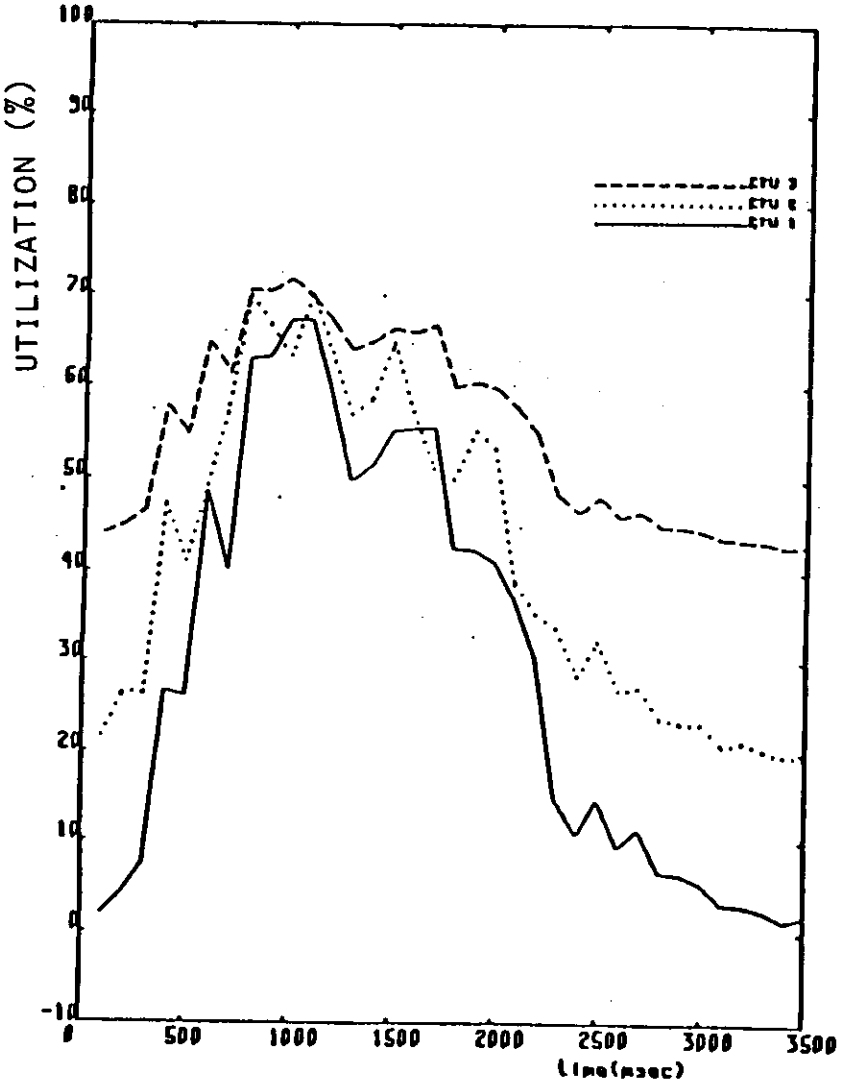
In Section 4 we show the minimum bottleneck is an effective objective function for selecting module assignments that yield minimum P-T-P time. However, an exhaustive search through the entire tree is prohibitively time-consuming. For example, there are 3^{20}

FIG. 9 ENUMERATION RESULTS: 10 GOOD ASSIGNMENTS

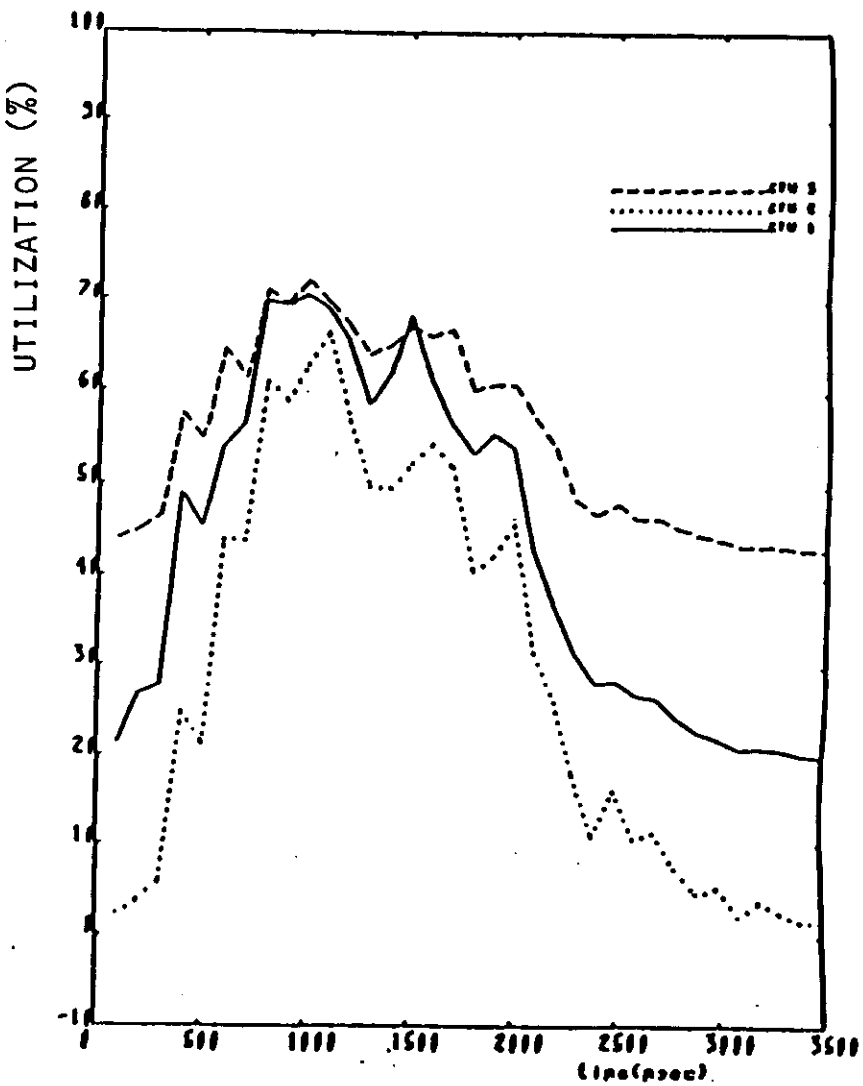
	ASSIGNMENT	LOAD-1	LOAD-2	LOAD-3	BOTTLENECK	TOTAL LOAD
10th	11111 11121 00330 12322 123	75612	75546	70420	75612	221578
9th	11111 11121 00330 12322 323	75323	75546	70709	75546	221578
8th	11112 11121 00330 13222 223	75413	75352	73643	75413	224408
7th	11112 12131 00220 13231 132	75178	74275	73829	75178	223282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

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

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

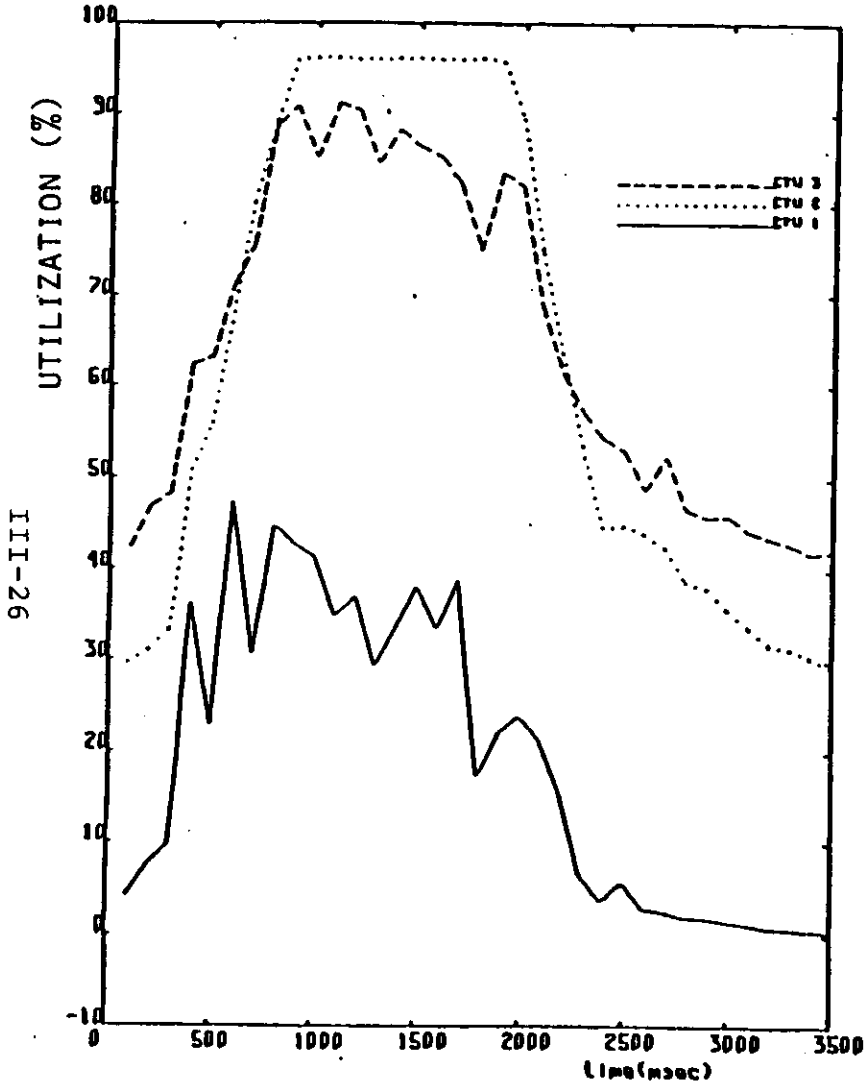


(A) FIRST SELECTED ASSIGNMENT

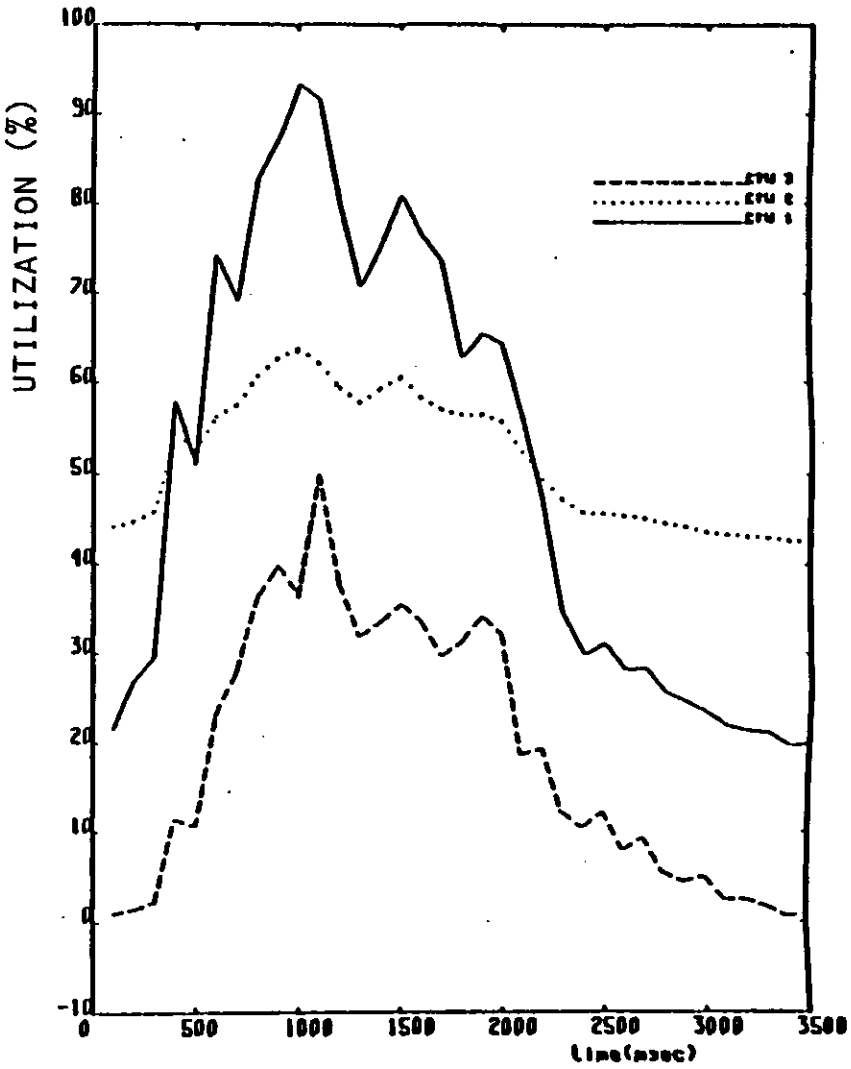


(B) SECOND SELECTED ASSIGNMENT

FIG. 10 CPU UTILIZATION IN THREE PROCESSORS FOR MODULE ASSIGNMENTS SELECTED BY EXHAUSTIVE SEARCH



(A) AN ARBITRARY ASSIGNMENT



(B) A KNOWLEDGE-GUESSED ASSIGNMENT

FIG. 11 CPU UTILIZATION IN THREE PROCESSORS FOR TWO OTHER ASSIGNMENTS

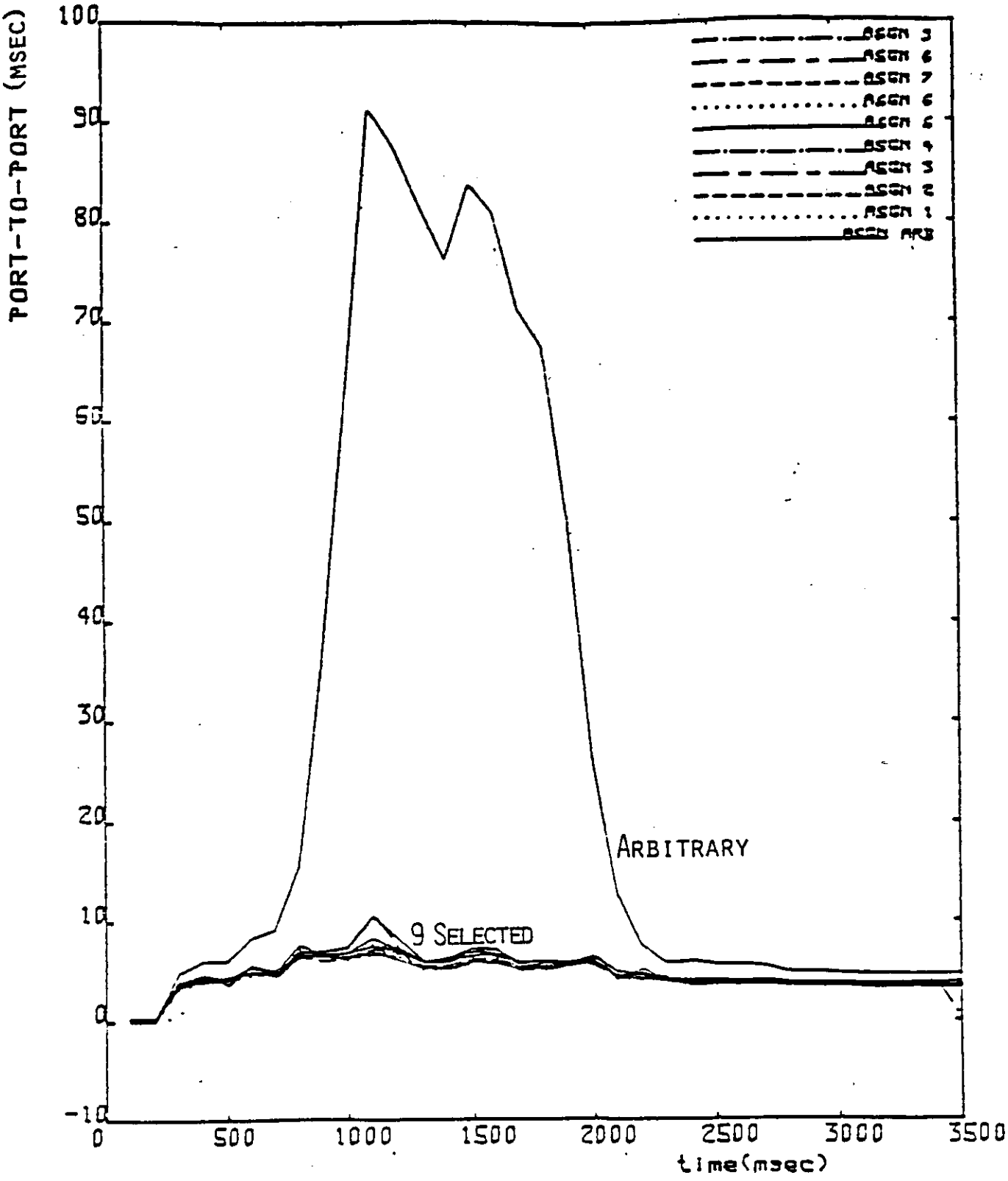


FIG. 12 PTP TIME FOR THE PRECISION-TRACKING THREAD ---COMPARE AN ARBITRARY ASSIGNMENT AND 9 ASSIGNMENTS SELECTED BY EXHAUSTIVE SEARCH

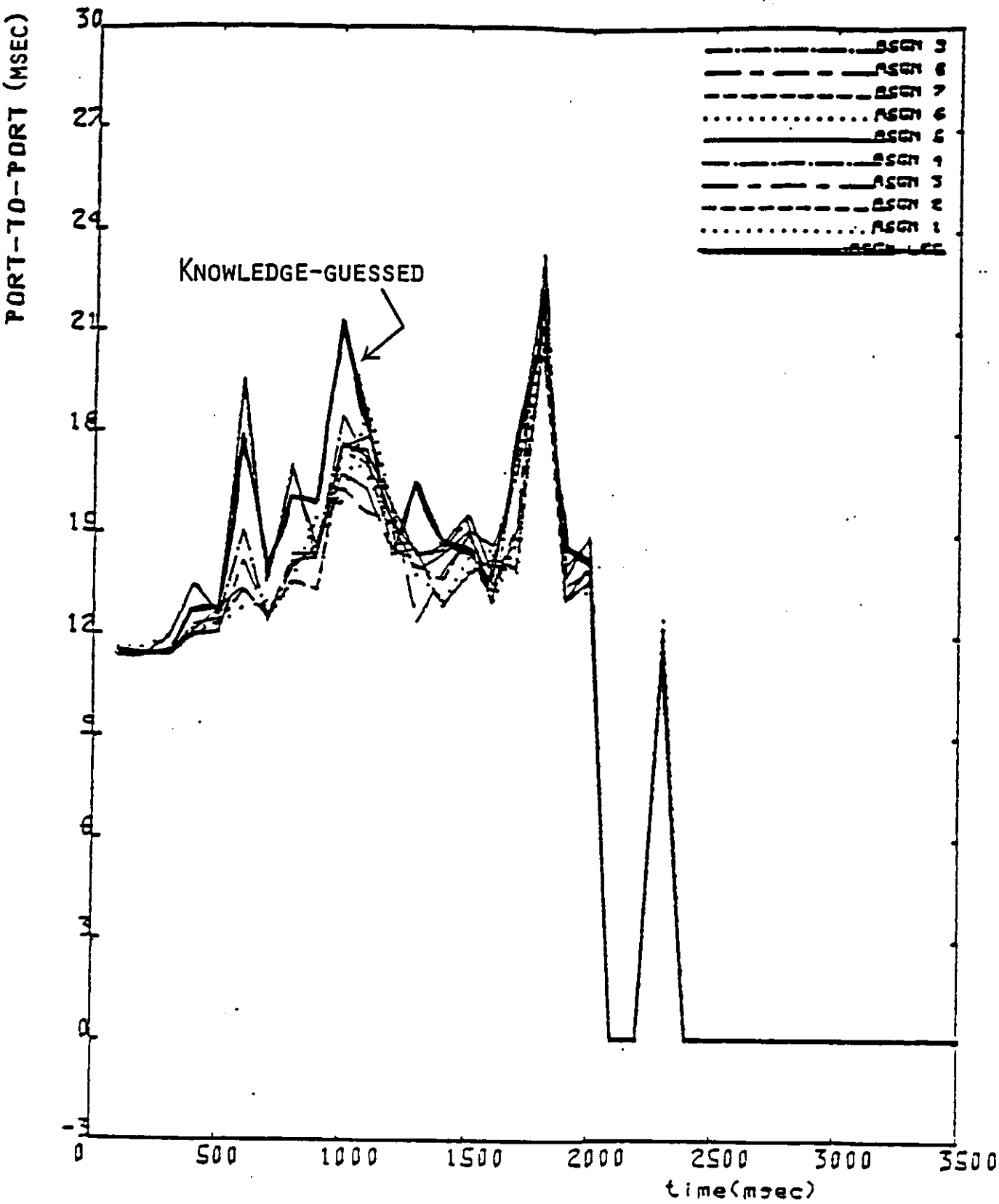


FIG. 13 PTP TIME FOR DETECT/VERIFY THREAD---COMPARE THE KNOWLEDGE-GUESSED ASSIGNMENT AND 9 ASSIGNMENTS SELECTED BY EXHAUSTIVE SEARCH

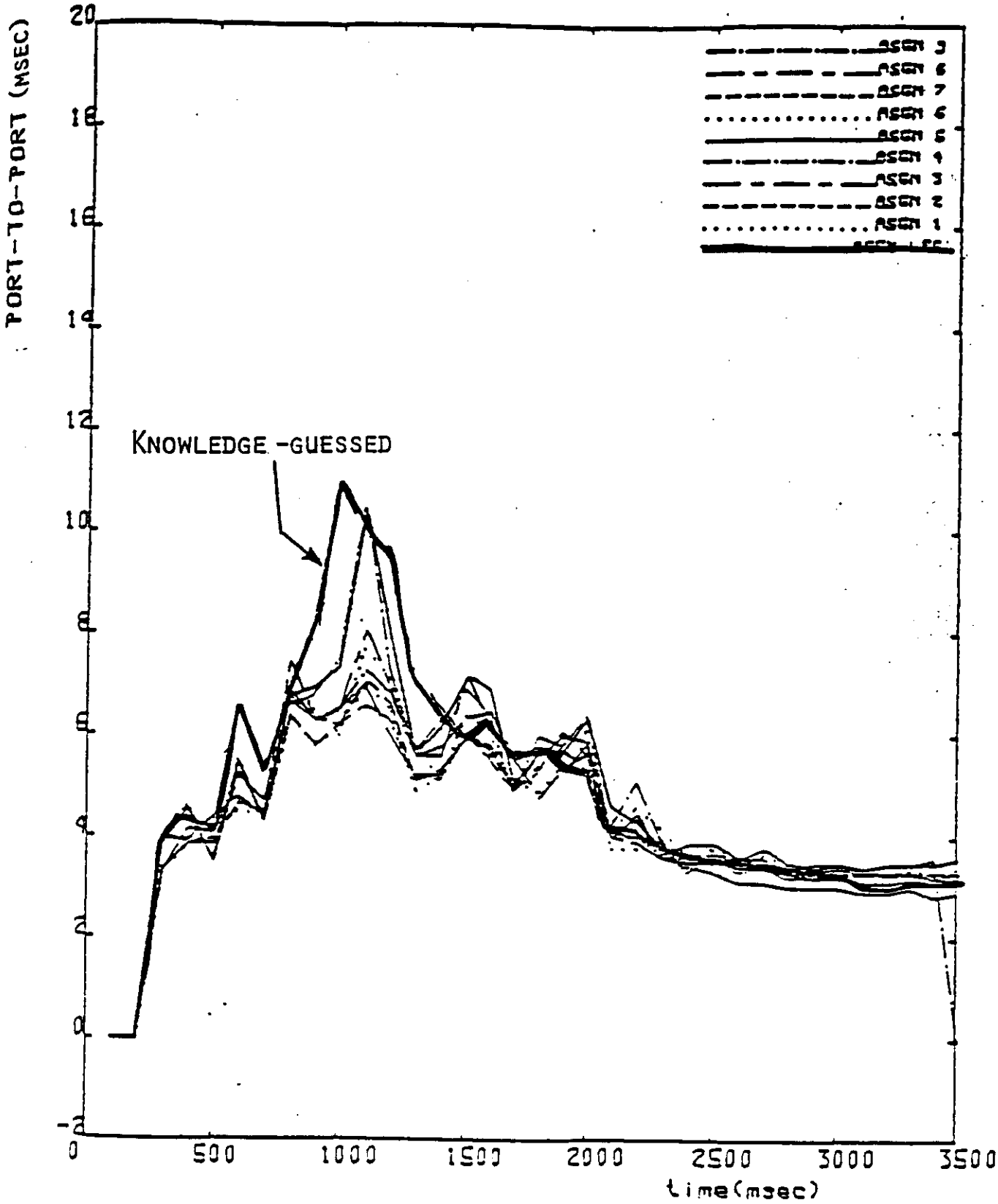


FIG. 14. PTP TIME FOR THE PRECISION-TRACKING THREAD --COMPARE THE KNOWLEDGE-GUESSED ASSIGNMENT AND 9 ASSIGNMENTS SELECTED BY EXHAUSTIVE SEARCH

possible assignments in the assignment tree for the DPAD system. In order to greatly reduce the computation time, we shall develop a heuristic algorithm for selecting good assignments from a huge problem space.

As stated in Appendix A, the ten assignments selected from the objective function are *not* the best ten. *A good many more* assignments better than the ten selected ones were left out from the list on Fig. A-1. Although not the best ten, the ten selected assignments give similar good performance. Therefore, a good *heuristic* module-assignment algorithm will be able to find a sub-optimal assignment that yields close-to-best P-T-P performance.

We have previously identified AET and IPC as two important parameters which should be considered for module assignment. In the following we propose a heuristic algorithm for module assignment taking these two parameters into account. This algorithm consists of two phases. In order to avoid heavy IPC, Phase I merges heavily communicating modules into groups if the resulting group does not have too large an AET. This phase can be done in a very short time. Phase II assigns the module groups resulted from Phase I to the available processors such that the bottleneck (in the most heavily utilized processor) is minimized. Our algorithm assumes that

1. there are n modules, $M_1, M_2, \ldots, M_n$, and m processors, $P_1, P_2, \ldots, P_m$;

2. peak-load average of AET for each module M_i is given, denoted as T_i; and

3. peak-load average of IMC between each module pair M_i and M_j is given, denoted as $IMC_{i,j}$.

ALGORITHM:

Phase I

(Merge modules with large IMC into groups to reduce total system load)

1. Initially list all module pairs along with their associated IMC volume, in the descending order of IMC volume.

   Assign each program module to one individual group:
   $G_i \leftarrow \{M_i\} \quad i = 1,\ldots,n$

   Set an upper bound for group size:
   $ideal\_processor\_load \leftarrow \sum_{i=1}^{n} T_i / m$
   $threshold \leftarrow ideal\_processor\_load \times \beta\ \%$

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

3. If $IMC_{a,b} < ideal\_processor\_load \times \alpha\ \%$
   (* IMC too small. Won't benefit from further merging of modules. *)
   go to Phase II.

4. Find the group, G_s, which contains M_a and the group, G_t, which contains M_b.
   (i.e., $M_a \in G_s$, $M_b \in G_t$)

5. If $s = t$
   (* M_a and M_b were already in the same group. *)
   go to Step 2.

6. If $T_s + T_t > threshold$
   (* The otherwise merged group would be too large; it will make the load balancing impossible in Phase II. *)
   go to Step 2.

7. Merge G_s and G_t:
   $G_s \leftarrow G_s \cup G_t$
   $G_t \leftarrow \emptyset$
   $T_s \leftarrow T_s + T_t$
   $T_t \leftarrow 0$
   go to Step 2.

Phase II

(Assign merged groups to processors to minimize the bottleneck)

1. (* We now have q groups, $q < n$. Therefore, we have a much smaller assignment tree with m^q, instead of m^n, possible assignments. *)

   Perform a search through the tree using a slightly modified version of the exhaustive search method (described in Appendix A) to find good assignments — instead of generating module assignments with *strictly decreasing* bottlenecks, this

modified version keeps at any instant the best ten assignments obtained so far during the search.

2. Stop.

Steps 3 and 6 in Phase I require some discussion. When merging modules, pair by pair, we might reach a pair of modules with "small" IMC and merging the two associated modules gives little benefit in terms of the IPC saved. The "small" here is defined as being less than a small α percentage (e.g., 5%) of the *ideal_processor_load*. (*Ideal_processor_load* has been defined to be the sum of AET for all modules, divided by m, the number of processors.) In fact, the entire Step 3 can be deleted from the algorithm and the merging process continues until all IMC's have been examined. Of course, this will take a little more time to execute Phase I.

Discussion regarding Step 6 follows. We try to eliminate as much IPC as possible, but we generally can not eliminate all of it. Therefore, when merging modules into a group, we should leave some room in this group for accommodating the residual IPC (that IPC still not eliminated). This motivates us to introduce the factor β % (e.g., 75%) into the threshold. Otherwise, Phase II might not be able to produce a balanced-load assignment.

Let us now apply this heuristic algorithm to the DPAD module assignment problem. Information in Table 2 is reorganized into Table 3 which provides clearly the order of IMC size among module pairs. (Table 3 can also be obtained from Fig. 6 by the compression method described previously.) This table makes Phase I of the algorithm an easier task. (Phase II will still use Table 2). Fig. 15 shows the merging process of Phase I where 5% and 75% are adopted for the α and β respectively. Column 1 is the descending IMC selected from Table 3, columns 2 and 3 are the associated module pairs, column 4 displays the modules merged into one group, and column 5 calculates the total

TABLE 3

FILE UPDATE IMC (in MLI) PER 100 MSEC FOR MODULE PAIRS

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

* EACH TRANSFERRED WORD = 3 MLIs

$IMC_{a,b}$ (MLI)	Writing Module, M_a	Reading Module, M_b	Modules in the Merged Group	Exec. Time of the Group
30371	13	14	13-14	25305+16860=42165
7955	8	6	6-8	1950+32055=34005
7955	8	10	6-8-10	34005+3360=37365
7955	8	16	6-8-10-16	37365+4170=41535
5019	14	23	Can't group 13-14-23 Otherwise,	42165+10725=52890 > 75% of 59555
4593	22	1	1-22	8865+16410=25275
4593	22	2	1-2-22	25275+2700=27975
4593	22	4	1-2-4-22	27975+10410=38385
4593	22	8	Can't group* 1-2-4-6-8-10-16-22 Otherwise,	38385+41535=79920 > 75% of 59555
1800	14	13		

[Phase I finishes because the IMC 1800 is less than 5% of 59555.]

* Two large groups would otherwise be merged into one.

Figure 15. Example of Phase-I Evaluation of the Algorithm

AET for all modules within the group.

Since we have three processors in this example, the *ideal_processor_load* is $\sum_{i=1}^{23} T_i / 3 = 59555$ MLI. Phase I finishes when it reaches $IMC_{14,13}$ because this IMC (1800 MLI) is smaller than 5% of the *ideal_processor_load*. The resultant groups are:

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

* These 3 modules are not implemented in the DPAD; each has a zero AET, T_i.

We have merged 20 modules into 13 groups. This means a reduction from 3^{20} possible module assignments to 3^{13} possible group assignments for Phase II. The savings is $3^7 = 2187$-fold, reducing the algorithm's run time from approximately 3 days down to 2 minutes on a VAX-11/780.

Two of the three best assignments obtained from Phase II are among the ten assignments shown in Fig. 9, which were obtained previously by the exhaustive search method:

a. The first assignment — identical to the fourth assignment in Fig. 9

b. The second assignment — identical to the fifth assignment in Fig. 9

c. The third assignment was never selected before. It is 11212 31333 00220 33211 212 (see Fig. 9 for notation), with 3 processor loads being 74522, 74521, and 74023.

To evaluate the effectiveness of our heuristic algorithm, we compare the best assignment obtained by this algorithm with the best assignment obtained by the exhaustive search. The performance of these two assignments, along with eight other assignments, have been shown in Figs. 13 and 14. For clarity we show these two assignments alone in Figs. 16 and 17. We note that the proposed heuristic algorithm generates module assignments about as good as those generated by the exhaustive search method.

6. CONCLUSIONS

We have proposed and verified a good objective function for selection of module assignments. This objective function is to minimize the CPU load on the most heavily utilized processor (i.e., *minimize the bottleneck).* We have also presented a heuristic algorithm as a quick approach to finding a good module assignment. Simulation experiments indicate that our heuristic algorithm produces very good module assignments in terms of the port-to-port times of the Detect/Verify and Precision Track threads.

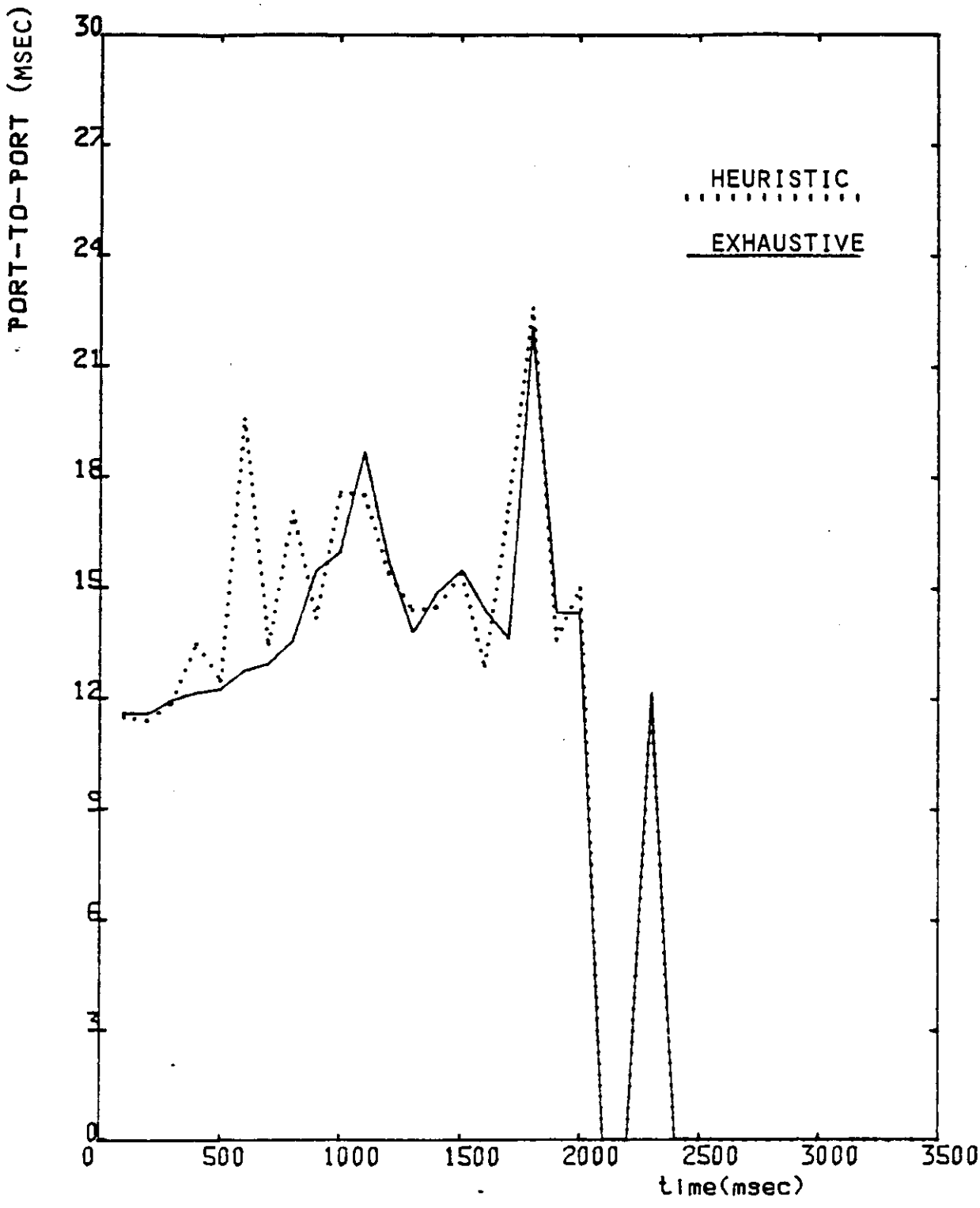


FIG. 16. PTP TIME FOR DETECT/VERIFY THREAD--COMPARE BEST ASSIGNMENT SELECTED BY HEURISTIC ALGORITHM AND BEST ASSIGNMENT SELECTED BY EXHAUSTIVE SEARCH

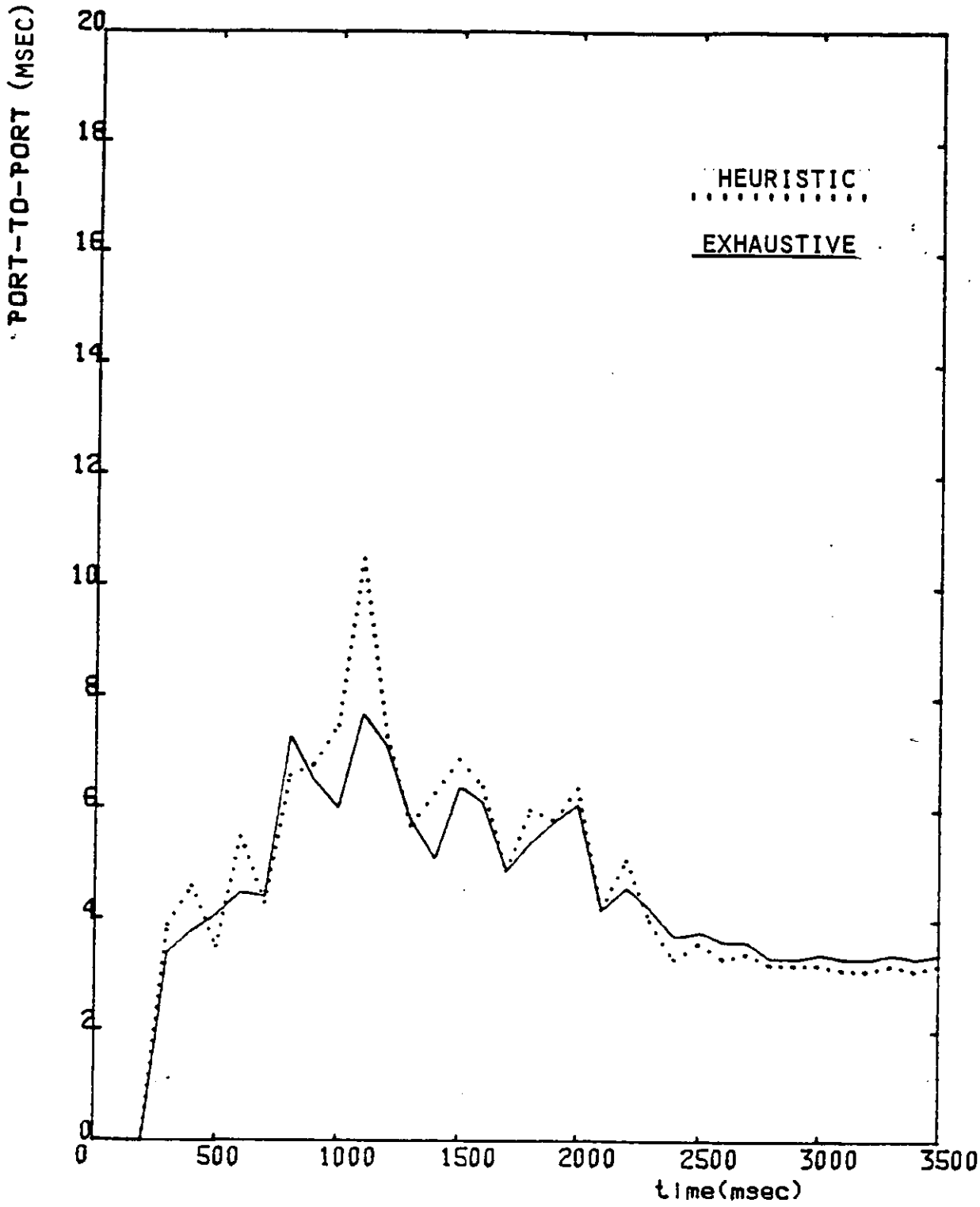


FIG. 17. PTP TIME FOR PRECISION-TRACKING THREAD---COMPARE BEST ASSIGNMENT SELECTED BY HEURISTIC ALGORITHM AND BEST ASSIGNMENT SELECTED BY EXHAUSTIVE SEARCH

APPENDIX A. MODULE-ASSIGNMENT SELECTION PROGRAM

The program selects the module assignments from the enumeration tree *according to the objective function of eq. (5).* Variable MINMAX keeps the minimum "bottleneck load" evaluated so far in the tree search. It is initialized with a large value, 999999 MLI's. Whenever a module assignment is evaluated to have a bottleneck smaller than MINMAX, the bottleneck value replaces the old MINMAX value and a line is printed to log this particular assignment. Fig. A-1 shows the printout from this program. Progressively better assignments are obtained. The first column displays the module assignment with the minimum bottleneck found so far and the next column shows the associated bottleneck value. Of the 23 numbers shown in the assignment, the j-th number with a value of 1, 2, or 3 means module M_j being assigned to processor 1, 2, or 3 in the particular assignment. Within each row, the bottleneck is the largest of those three load values in columns 3, 4, and 5, each column representing a processor in the distributed system. The rightmost column shows the total load of the 3 processors, i.e., the total system load.

As mentioned in section 3, it takes several days to enumerate the entire tree. So the program is designed to have a checkpoint written out in a temporary output file for every 3^{11} assignments evaluated. When a computer system failure occurs, it can continue the enumeration from the most recent checkpoint.

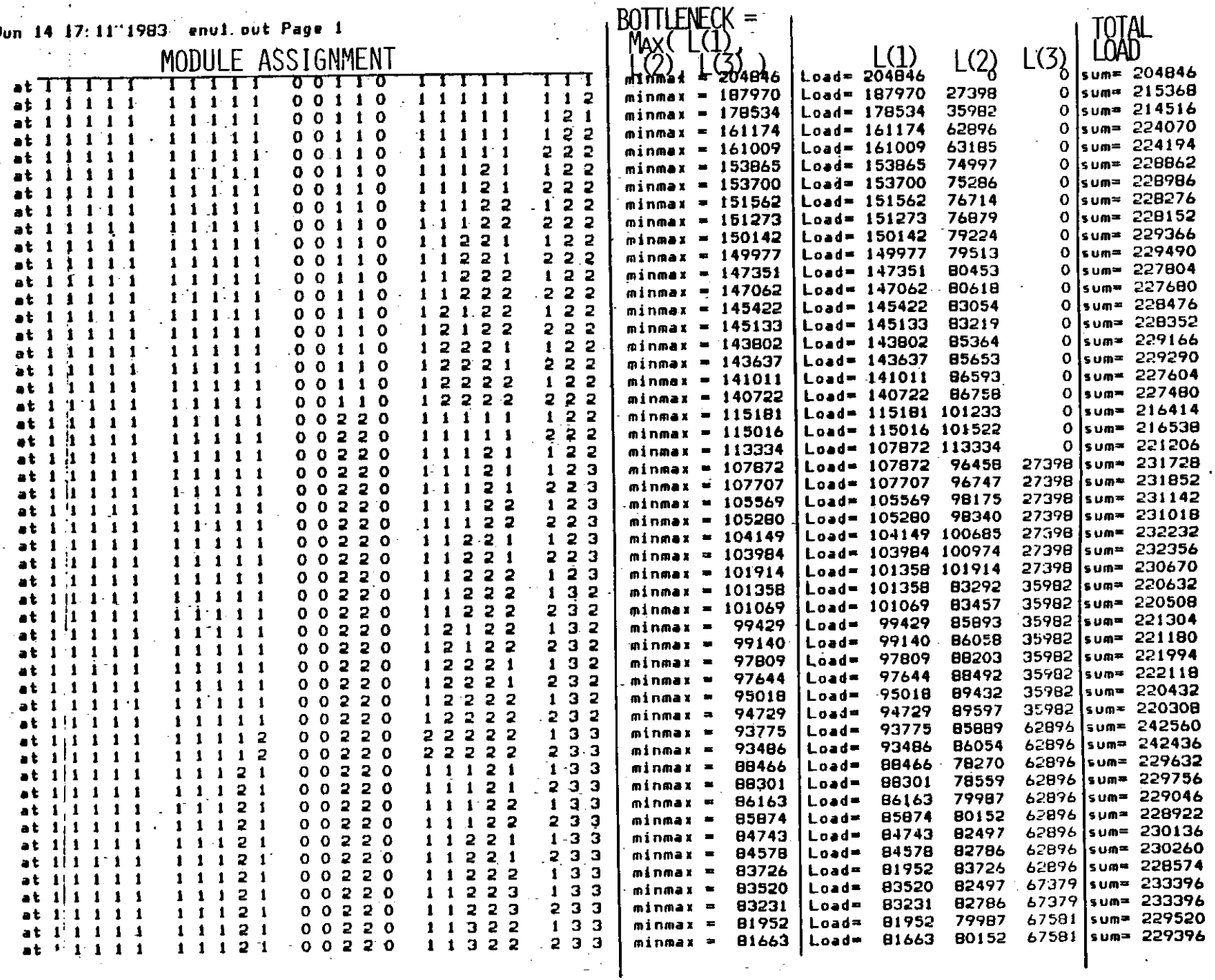


FIG. A-1. ENUMERATION RESULTS FROM EXHAUSTIVE SEARCH

```
at 1 1 1 1 1 1   1 1 1 2 1   0 0 2 2 0   1 2 1 3 2   1 3 3 | minmax = 81591 | Load= 81591 76992 74997 | sum= 233580
at 1 1 1 1 1 1   1 1 1 2 1   0 0 2 2 0   1 2 1 3 2   2 3 3 | minmax = 81302 | Load= 81302 77157 74997 | sum= 233456
at 1 1 1 1 1 1   1 1 1 2 1   0 0 2 2 0   1 2 2 3 1   1 3 3 | minmax = 79971 | Load= 79971 78562 74997 | sum= 233530
at 1 1 1 1 1 1   1 1 1 2 1   0 0 2 2 0   1 2 2 3 1   2 3 3 | minmax = 79806 | Load= 79806 78851 74997 | sum= 233654
at 1 1 1 1 1 1   1 1 1 2 1   0 0 2 2 0   1 2 2 3 3   1 3 3 | minmax = 78562 | Load= 77180 78562 76714 | sum= 232456
at 1 1 1 1 1 1   1 1 1 2 1   0 0 2 2 0   1 3 3 2 3   1 3 3 | minmax = 78270 | Load= 77180 78270 77716 | sum= 233166
at 1 1 1 1 1 1   1 1 1 2 1   0 0 3 3 0   1 2 2 2 1   2 2 3 | minmax = 78238 | Load= 78238 78145 65735 | sum= 222118
at 1 1 1 1 1 1   1 1 1 2 1   0 0 3 3 0   1 2 2 2 3   1 2 3 | minmax = 77856 | Load= 77180 77856 70218 | sum= 225254
at 1 1 1 1 1 1   1 1 1 2 1   0 0 3 3 0   1 2 2 3 2   1 2 3 | minmax = 77836 | Load= 77180 70208 77836 | sum= 225224
at 1 1 1 1 1 1   1 1 1 2 1   0 0 3 3 0   1 2 3 2 2   1 2 3 | minmax = 75612 | Load= 75612 75546 70420 | sum= 221578
at 1 1 1 1 1 1   1 1 1 2 1   0 0 3 3 0   1 2 3 2 2   3 2 3 | minmax = 75546 | Load= 75323 75546 70709 | sum= 221578
at 1 1 1 1 1 2   1 1 1 2 1   0 0 3 3 0   1 3 2 2 2   2 2 3 | minmax = 75413 | Load= 75413 75352 73643 | sum= 224408
at 1 1 1 1 1 2   1 2 1 3 1   0 0 2 2 0   1 3 2 3 1   1 3 2 | minmax = 75178 | Load= 75178 74275 73829 | sum= 223282
at 1 1 1 1 1 2   1 2 1 3 1   0 0 2 2 0   1 3 2 3 1   2 3 2 | minmax = 75013 | Load= 75013 74564 73829 | sum= 223406
at 1 1 1 1 1 2   3 2 3 3 3   0 0 2 2 0   3 3 2 1 1   1 1 2 | minmax = 74414 | Load= 74414 74275 74023 | sum= 222712
at 1 1 1 1 1 2   3 2 3 3 3   0 0 2 2 0   3 3 2 1 1   3 1 2 | minmax = 74312 | Load= 74249 74275 74312 | sum= 222836
at 1 2 1 2 3   2 3 2 1 2   0 0 3 3 0   2 1 3 2 2   1 1 3 | minmax = 74308 | Load= 74308 73873 74275 | sum= 222456
at 1 2 1 2 3   2 3 2 1 2   0 0 3 3 0   2 1 3 2 2   2 1 3 | minmax = 74275 | Load= 74019 74038 74275 | sum= 222332
at 1 2 2 1 3   1 3 1 2 1   0 0 3 3 0   1 1 3 2 2   2 2 3 | minmax = 74275 | Load= 74004 73805 74275 | sum= 222084
```


FIG. A-1. (CONTINUED)

From the program output we picked the last ten assignments (shown in Fig.9) and simulated each of them with the DPAD simulator. See Section 4 for the results.

Although the 10 selected assignments have progressively smaller bottlenecks, they are *not* exactly the 10 assignments in the search tree which have the 10 *smallest* bottlenecks. For example, after the assignment with the bottleneck 74308 was printed on Fig.A-1 (the 3rd line from the bottom), an assignment with a bottleneck 74310 was not printed. Nor was an assignment with a bottleneck 74400. But, both 74310 and 74400 might well be among the 10 smallest bottlenecks. Therefore, there are many more assignments that have better (or comparable) performance than the 10 selected ones.

REFERENCES

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

CHU82 Chu, W. W., J. Hellerstein, M. T. Lan, and L. Holloway, Research on the Shared Database Kernel for the BMD Application, UCLA Computer Science Department, Report # CSD-820430, April 30, 1982.

GREE80 M. L. Green, E. Y. S. Lee, S. Majumdar, and D. C. Shannon, A Distributed Real-Time Operating System, *Proceedings of the Symposium on Distributed Data Acquisition, Computing, and Control*, Dec. 1980, pp.175-184.

GYLY76 Gylys, V. B. and J. A Edwards, Optimal Partitioning of Workload for Distributed Systems, *Digest of Papers, COMPCON Fall 76*, Sep. 1976, pp.353-357.

HARA69 F. Harary, *Graph Theory*, Addison-Wesley, New York, N. Y., 1969.

HOFF80 Hoffman, R. H., R. W. Smith, and J. T. Ellis, Simulation Software Development for the BMDATC DDP Underlay Experiment, *4th Intl. Computer Software and Applications Conf. (COMPSAC)*, Oct. 1980, Chicago, pp.569-577.

HOLL82 Holloway, L. J., Task Assignment in a Resource Limited Distributed Processing Environment, Ph.D Dissertation, Computer Science Dept., UCLA, 1982.

JENN77 Jenny, C. J., Process Partitioning in Distributed Systems, *Digest of Papers, NTC '77*, 1977, pp.31:1-1 — 31:1-10.

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

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

STON78 Stone, H. S., and S. H. Bokhari, Control of Distributed Processing, *IEEE Computer* Magazine, Vol. 11, No. 7, July 1978, pp.97-106.

TSUC80 M. Tsuchiya, Considerations for Requirements Engineering of Distributed Processing Systems, *Proceedings of the Symposium on Distributed Data Acquisition, Computing, and Control,* Dec. 1980, pp.61-65.

CHAPTER IV

A RESILIENT COMMIT PROTOCOL FOR REAL TIME SYSTEMS

A RESILIENT COMMIT PROTOCOL FOR REAL TIME SYSTEMS

1. Introduction

The survivability of distributed systems can be improved with multiple copies of files. When an update is performed at a site, this update should be delivered to those sites that have a replicated file. All file copies should be identical referred to as *mutual consistency*. File copies may differ temporarily during update propagation. If a site fails during an update broadcast, only a part of file copies may be updated, resulting in mutual inconsistency.

Posting a file update in a system is known as a *commit* which implies that the posted update will not be backed out [GARC82]. The site that broadcasts an update to other sites is known as the *coordinator* and the receiving sites of the update are the *participants*.

Two-phase commit is a well known method which ensures mutual consistency among file copies in case of failures [KOHL81]. It is also called a *blocking commit* because when a coordinator site fails during an update broadcast, other sites are blocked from using the file copies until the failed coordinator recovers and completes the unfinished commit work [SKEE81]. The time interval required for the commit process is relatively short as compared with the time needed to prepare update data at a coordinator site, thus it is less probable that a failure occurs during the commit interval resulting in blocking to the file. This technique has been implemented in several prototype systems such as INGRES [STON79] and LOCUS [WALK83]. Both of these are non-real time systems.

For a non-blocking commit, Skeen [SKEE81] proposed a *three-phase commit* protocol by adding another buffer state to the two-phase commit. When a coordinator fails, one of the participants is elected to be the new coordinator and completes the suspended commit process based on its local state. In the *back-up coordinator* method, introduced by Hammer [HAMM80], each file has a preassigned set of back-up coordinators. These back-up coordinators communicate with the coordinator during commit processes and take its place in case of coordinator failure. Both of the above methods require excessive communications and therefore are not suitable for real time applications.

In this chapter, we present a low cost resilient commit protocol that is *non-blocking* and *tolerant to multiple failures*. In this commit protocol, sites are linearly numbered and updates are broadcast in one phase according to this number sequence. Failures are recovered by the smallest numbered surviving site. Next, we show how to incorporate this commit protocol into existing concurrency control techniques such as EWP [CHU82] and PSL [STON79]. Finally, we discuss the procedure for site recovery.

2. Assumptions

We restrict our discussion to distributed systems that consist of multiple computers connected via an interconnection network. Further, we assume the following:

1. *The topology of the communication network is designed such that site partitioning does not occur even in the presence of link failures.*

2. *A message sent by a site will eventually be received by a destination site as long as the destination is alive.* This can be assured by exchanging a positive acknowledgement in the network subsystem for every message delivery. Based on this assumption, there would be no lost or out-of-sequence messages.

3. *"I am alive" messages are periodically exchanged among the sites for failure detection.* With this assumption, acknowledgement messages and time-out detection mechanisms are eliminated from our resilient commit protocol.

4. *Failed sites are not allowed to rejoin the system.* In real time systems, files are usually stored in volatile memories (e.g., MOS dynamic RAMs) and a site failure causes a total loss of data, making the site recovery infeasible within real time constraints. However, in section 5, the site recovery issues will be addressed under the assumption that all files are stored in non-volatile media.

3. Resilient Commit Protocol

The basic concept of our resilient commit protocol is that if an update is posted at any operating site all other operating sites that keep a copy of the file will eventually receive the update regardless of multiple failures. Thus mutual consistency among the file copies is preserved for an update. (When more than one site perform updates simultaneously, the commit protocol does not necessarily maintain mutual consistency. Concurrency control protocols provide file consistency by controlling simultaneous updates from different sites.)

In the commit protocol, sites are numbered for a given file. Updates are broadcast *according to this number sequence*. If only a subset of operating sites receives an update after a failure occurs, mutual inconsistency is resolved by having the smallest numbered operating site retransmit the last update received from the failed site. This retransmission must be done *in the number sequence* so that a failure of the smallest numbered site will have the next smallest numbered site redo the retransmission. Thus the commit protocol is resilient to multiple failures.

In the commit protocol, exchanges of acknowledgement messages are eliminated by relying on the network subsystem at each site for failure detection. However, network subsystems require an acknowledgement for every update delivery.

Let us now summarize the operation of the commit protocol:

1. For each file, sites are numbered and updates are sent in this number sequence.

2. Updates are posted immediately after being received. Further, each site should save the last updates *from all other sites* for update retransmissions in the event that it should become the smallest numbered surviving site.

3. When a site failure is detected, the smallest numbered surviving site retransmits the last update received from the failed site *in the number sequence*.

During the recovery from a failure, duplicate updates may be sent to a site, which would be detected by examining the update sequence number (SN). [1] The operations for the SN are listed as follows:

- For each file, every copy maintains a SN. Initially, all copies have the same SN (e.g., SN=n as shown in figure 1).

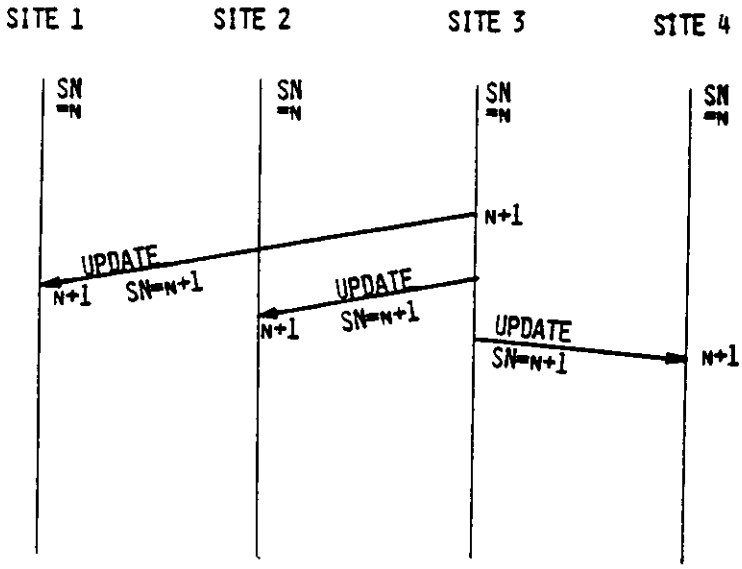
- When a coordinator finalizes an update, it increments its local SN by one and broadcasts the update with the new SN to other sites.

- An incoming update with SN less than or equal to the local SN of the receiving site is discarded.

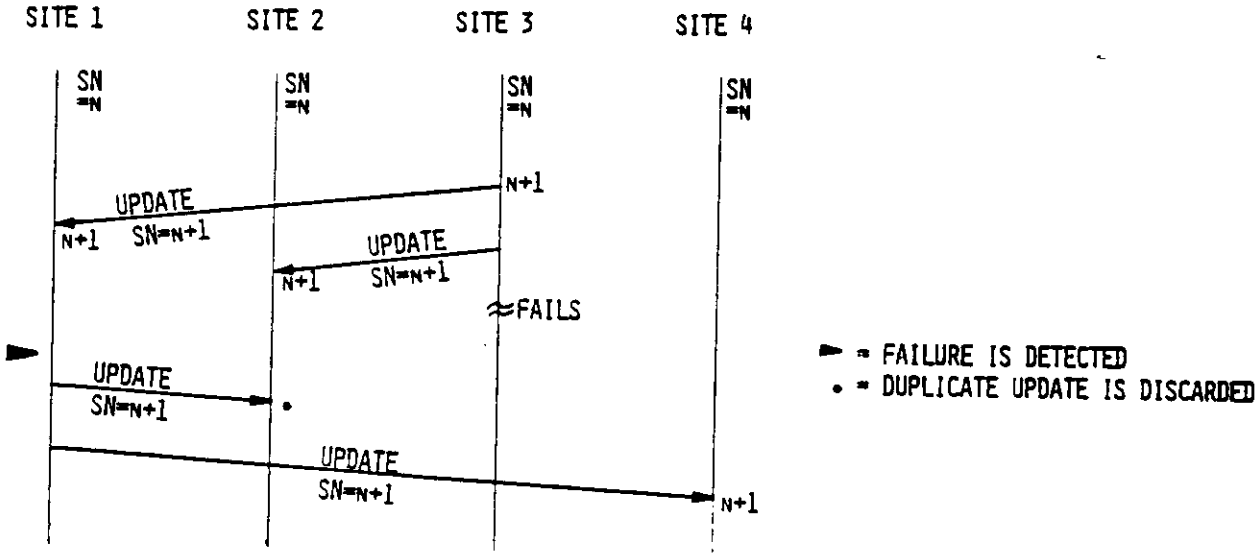
- When an update is posted, the local SN is replaced with the new value.

Figure 1a shows the no-failure case where site 3 broadcasts an update according to the number sequence. Note that no extra communication is required. In figure 1b, site 3 fails after sending an update to the first two sites. At the detection of the failure, the smallest numbered site (site 1) retransmits the update. By comparing the SNs, site 2 detects and discards the duplicate update.

[1] The SN concept was used in the EWP to detect conflicts among concurrent update requests (refer to chapter II).



1a. No-Failure Case



1b. Recovery from a site failure

Figure 1. The Resilient Commit Protocol

Reducing Messages for Failure Recovery

Suppose a site fails after all other sites have received its last update. Our commit protocol requires that the smallest numbered surviving site always retransmits the last update from the failed site. In this case, all sites will receive this update twice. To eliminate these duplicate updates during the recovery, a coordinator could indicate the completion of each update broadcast. This procedure is described as follows:

1. A coordinator sends an update-complete (UC) message to the smallest numbered site after completing an update broadcast.

2. When a site receives the UC message, it discards the saved update.

For systems with infrequent updates, the above approach is appealing since the additional overhead due to UC messages will be small and it is less probable that a site fails during an update broadcast. Moreover, real time systems usually require that real time constraints should be met even when failures occur and therefore it is desirable to reduce unnecessary overhead during a recovery.

4. Resilient Concurrency Control Techniques

For a set of concurrent updates generated from different sites, the commit protocol itself does not necessarily maintain mutual consistency. Concurrency control produces the relative order for multiple accesses to a file and provides internal and mutual consistency among file copies [BERN81].

Many concurrency control protocols have been introduced. The *primary site locking (PSL)* protocol [STON79] is accepted to be a low cost locking method. The *exclusive writer protocol (EWP)* has a very low overhead with no transaction restarts, database rollbacks, or deadlock. Although it provides limited serializability, the EWP has great appeal for distributed real time systems since strict serializability is usually not required in real time systems (refer to chapter II).

In this section, we shall discuss how to incorporate the resilient commit protocol into EWP and PSL to preserve internal and mutual consistency in the presence of failures.

4.1 Resilient Exclusive Writer Protocol

In the *Exclusive Writer Protocol (EWP)*, a file is written by only one predetermined site, called the *exclusive writer (EW)*, which does not change during system operation. Non-EW sites send update-request messages to the EW. Then the EW broadcasts the update to all other sites. (see figure 2a)

For the EWP to be resilient, sites are numbered and the site with the smallest number is designated as the EW. Since updates are broadcast only by the EW site, failures of non-EW sites do not cause any file inconsistency. When the EW site failure is detected, the next smallest numbered site becomes the new EW and retransmits the last update received from the old EW since the old EW may have failed during the broadcast

of the last update.

The operations for the resilient EWP are summarized as follows:

1. Sites are numbered and the site with the smallest number is selected as the EW.

2. EW sends an update to other sites *in the number sequence (i.e., lowest numbered site first, highest* numbered site last).

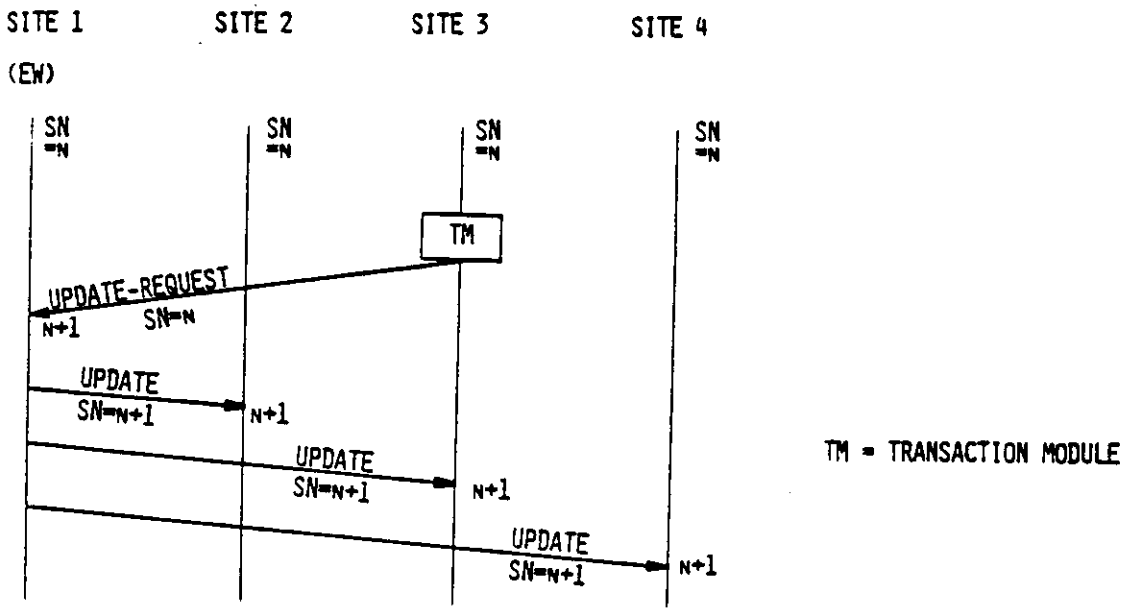
3. Each non-EW site should save the last update received from the EW.

4. When EW fails, the site with the next smallest number becomes the new EW and retransmits the last update received from the old EW *in the number sequence*.

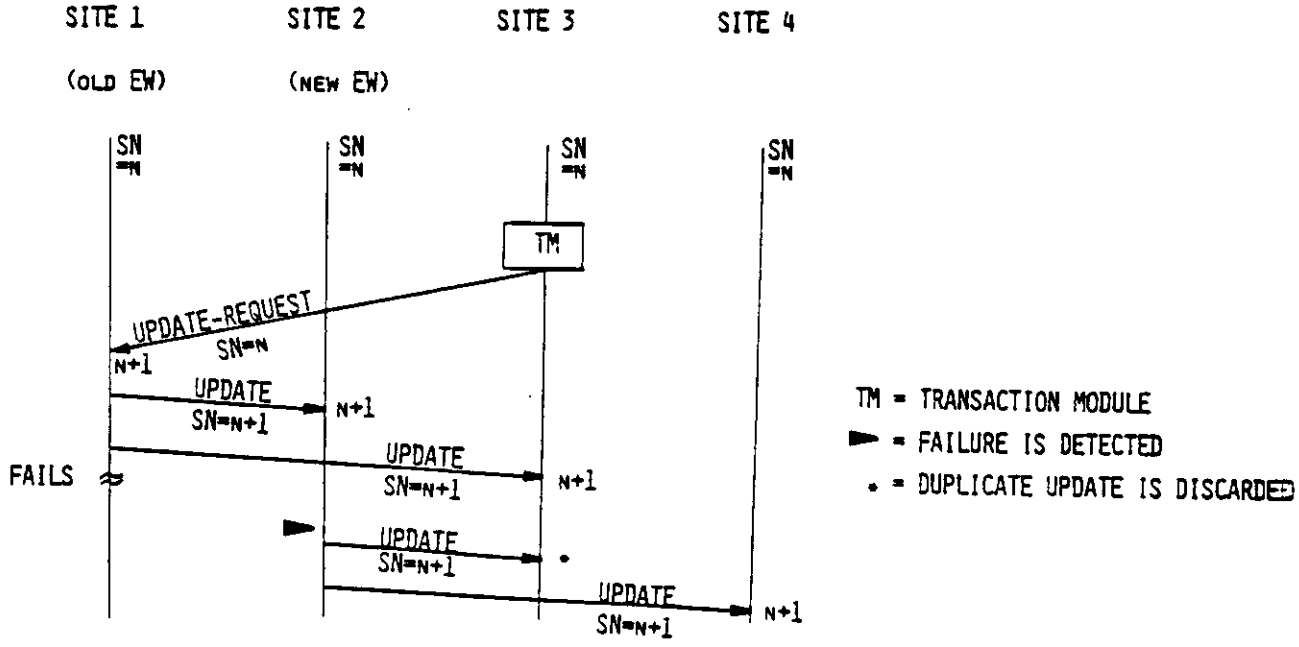
Figure 2a shows the no-failure case. Site 3 sends an update-request message to the EW which then broadcasts the update. The EW site fails during an update broadcast in figure 2b. The next smallest numbered site (site 2) becomes the new EW and resends the last update. A duplicate update is received and discarded by site 3.

If the EW site fails right after receiving an update-request but before sending out the update, the update-request will be lost. This is similar to a conflicting update request that is discarded in the EW site.

As explained in section 3, to eliminate unnecessary update retransmissions during the recovery from the EW failure, the EW may send an UC message to the next smallest numbered site after each update broadcast.



2a. No-Failure Case



2b. Recovery from the EW Failure

Figure 2. The Resilient EWP

4.2 Resilient Primary Site Locking Protocol

In the resilient PSL protocol, similar to the resilient EWP case, sites are numbered and the smallest numbered site is designated as the primary site (PS). When the PS fails, the next smallest numbered site becomes the new PS. Since the new PS does not know whether the lock was being held by any other site and whether any lock-request was queued in the old PS, the new PS requests the file status of all other site. Also a site that issued a lock-request but has not received a lock-grant from the old PS must resend a lock-request to the new PS.

In contrast to the EWP case, failures of non-PSs must be considered since non-PSs broadcast updates directly. In addition, non-PSs may fail while holding a lock or while its lock-request is being queued in the PS, which must be resolved by the PS. The operations for the resilient PSL protocol are summarized below:

1. Sites are numbered and the site with the smallest number is the PS.

2. Updates are broadcast *in the number sequence*.

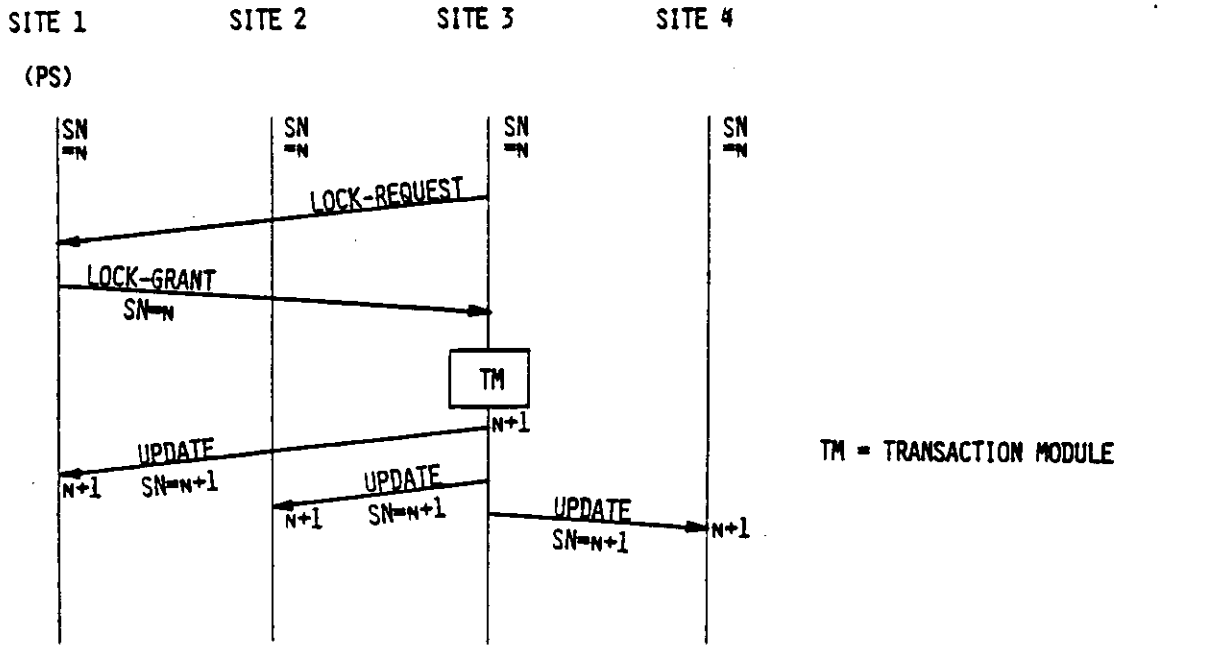
3. Each site saves the last updates *from all other sites*.

4. When a non-PS failure is detected: if the failed site is holding a lock, the lock is released by the PS; if the failed site has made a lock-request, the lock-request is discarded by the PS; otherwise the PS broadcasts the last update received from the failed site *in the number sequence*.

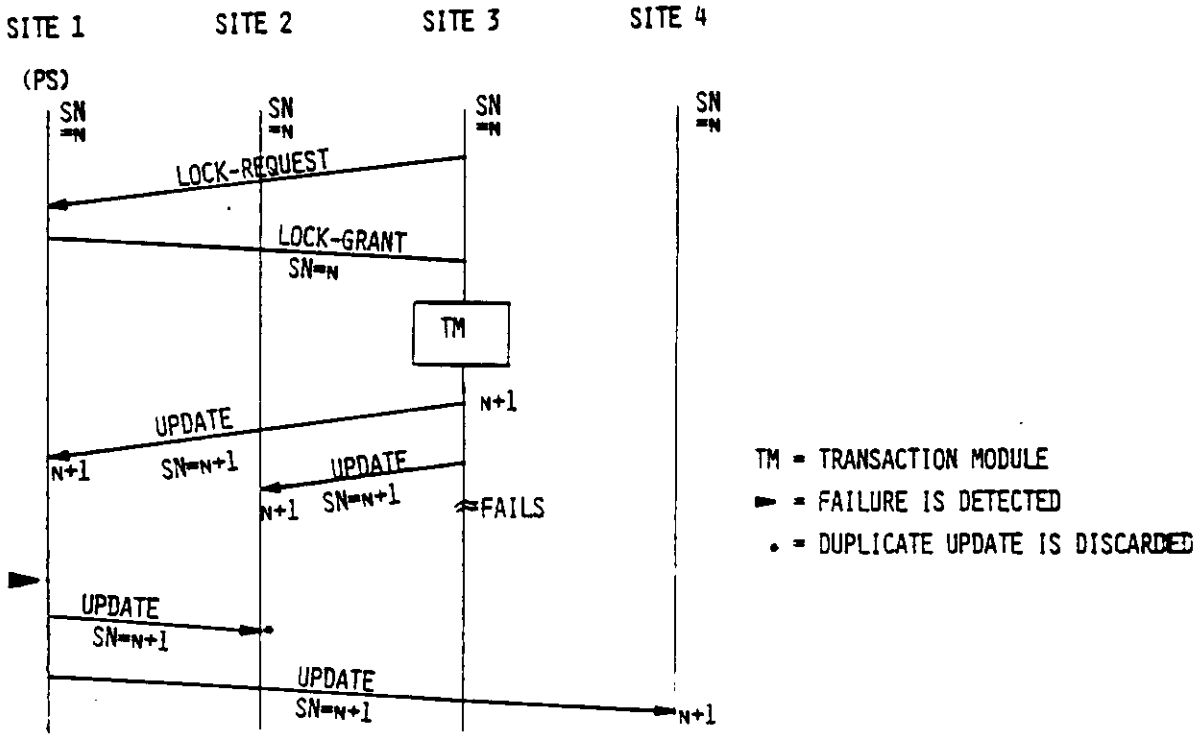
5. When the PS fails: the site with the next smallest number becomes the new PS; the new PS broadcasts the last update received from the old PS *in the number sequence* and requests the lock-status of other sites; if a site was waiting for a lock-grant from the old PS, it makes another lock-request to the new PS.

Figure 3a shows the no-failure case for the resilient PSL protocol. Site 3 issues a lock-request to the PS (site 1). After receiving a lock-grant, site 3 broadcasts an update. Figure 3b depicts the recovery procedure when a non-PS fails while broadcasting an update. The last update from the failed site is re-broadcast by the PS to ensure mutual consistency among the file copies. When the PS fails in figure 3c, site 2 becomes the new PS and requests the lock-status of other sites and site 3 responds that it is holding a lock. The dotted lines show the case where another site (site 4) which has made a lock-request to the old PS makes another lock-request to the new PS. After receiving an update from site 3, the new PS grants the lock to site 4.

In the resilient PSL protocol, the update-complete (UC) message can also be employed to reduce messages for the recovery from a failure. The non-PSs send an UC message to the PS each time an update broadcast is completed and the PS sends the UC to the next smallest numbered site.

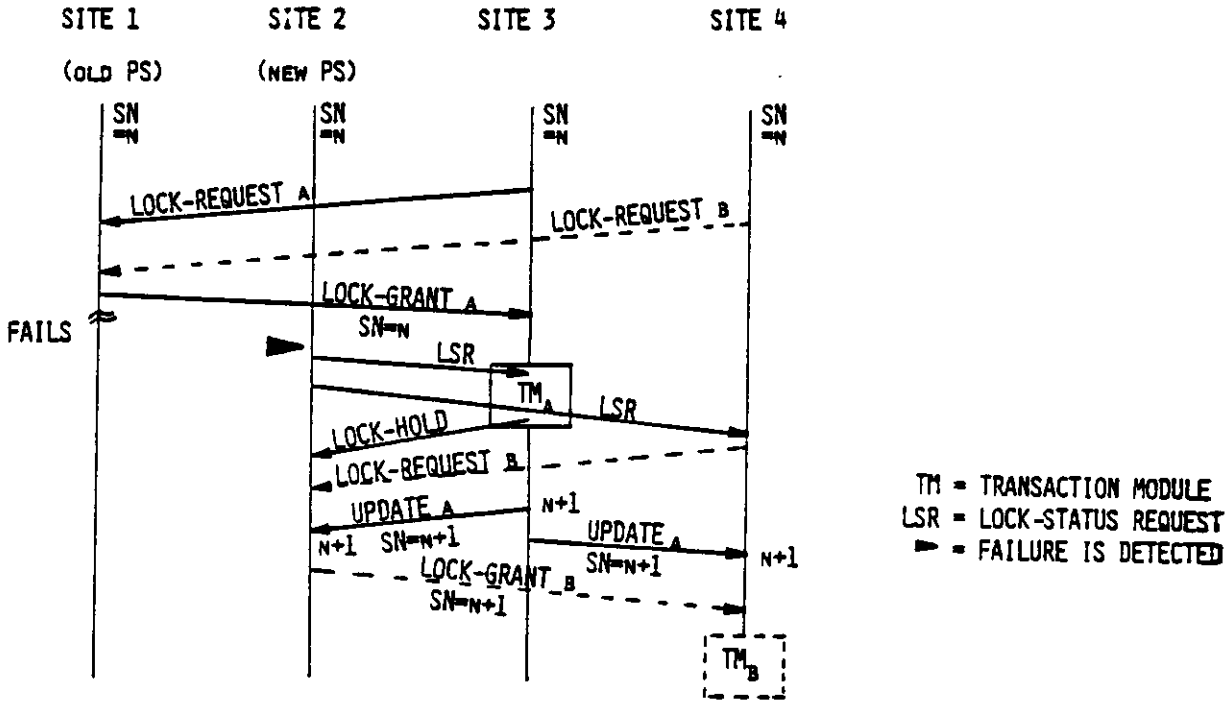


3a. No-Failure Case



3b. Recovery from the non-PS Failure

Figure 3. The Resilient PSL (continued to the next page)



3c. Recovery from the PS Failure

Figure 3. The Resilient PSL

5. Site Recovery

So far, we have assumed that failed sites are not allowed to rejoin the system. In this section, we introduce a method for having failed sites rejoin the system, assuming that all files are stored in non-volatile media.

In our commit protocol, failures are recovered by the smallest numbered surviving site. The recovering site should not be the smallest numbered surviving site until it is fully recovered. Thus the recovering site is given a number larger than any of the surviving site number. The operating sites should also buffer updates to be sent to the failed site.

When multiple failures occur while an update is being broadcast, the update may be sent only to failed sites, but not delivered to any operating site. During the site recovery, the last update must be undone even though it was completely posted in that site. Therefore, all sites should save previous values of the last posted update.

The detailed procedure for site recovery is given in the following:

1. The recovering site undoes the last update (whether or not it was completed) along with the SN, and broadcasts an 'I am up' message.

2. When the smallest numbered site receives the message from the recovering site, it determines the site number which is larger than any of the surviving site number. The new site number is then sent to all other surviving sites *in the number sequence*. The list of surviving sites and the new site number are sent to the recovering site.

3. When the recovering site receives its new site number with the list of

operating sites, an operating site is selected to recover all lost updates. The posting of newly incoming updates is postponed until all lost updates are received. The 'I am up' message is broadcast repeatedly if the new site number is not received within a certain time interval.

Since a recovering site can not send any update until all missing updates are received and posted, a failure of the site during its recovery does not affect file consistency. However, the failure of the smallest numbered site may delay the site recovery until the next smallest numbered site takes over.

6. Conclusions

We have presented a commit protocol which ensures mutual consistency among file copies in case of multiple failures. Since the proposed protocol requires additional overhead only when a failure occurs, it has much appeal for real time applications.

The commit protocol can be incorporated into concurrency control techniques, such as the EWP and the PSL protocol, with little additional overhead. Since the EWP only requires the EW site to broadcast updates and does not need locking, the recovery procedure for the EWP is much simpler and has less overhead than that for the PSL protocol.

References

[BERN80] P. A. Bernstein, D. Shipman, and J. Rothnie, Jr., "Concurrency Control in a System for Distributed Databases (SDD-1)," *ACM Transactions on Database Systems*, Vol.5, No.1, March 1980, pp.18-51.

[BERN81] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *Computing Surveys*, Vol.13, No.2, June 1981, pp.185-221.

[BHAR82] B. Bhargava, "Resiliency Features of the Optimistic Concurrency Control Approach for Distributed Database Systems," *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, PA, July,1982, pp.19-32.

[CHU82] W. W. Chu, J. Hellerstein, and M. Lan, "The Exclusive Writer Protocol: A Low Cost Approach For Updating Replicated Files In Distributed Real Time Systems," *Proceedings of the 3rd International Conference on Distributed Computing Systems*, October 1982, pp.269-277.

[DOLV82] D. Dolev and H. R. Strong, "Distributed Commit with Bounded Waiting," *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, PA, July,1982, pp.53-60.

[GARC82] H. Garcia-Molina, "Reliability Issues for Fully Replicated Distributed Databases," *IEEE Computer*, Vol.15, No.9, September 1982, pp.34-42.

[HAMM80] M. Hammer and D. Shipman, "Reliability Mechanisms for SDD-1: A System for Distributed Databases," *ACM Transactions on Database Systems*, Vol.5, No.4, December 1980, pp.431-466.

[KOHL81] W. H. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *ACM Computing Surveys*, Vol.13, No.2, June 1981, pp.149-183.

[SKEE81] D. Skeen, "Nonblocking Commit Protocols," *SIGMOD International Conference on Management of Data*, Ann Arbor, Michigan, 1981, pp.133-142.

[SKEE82] D. Skeen, "A Quorum-based Commit Protocol," *Proceedings, the 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, May 1982, pp.69-80.

[STON79] M. Stonebraker, "Concurrency Control and Consistency of Multiple Copies of Data In Distributed INGRES," *IEEE Transactions on Software Engineering*, Vol.SE-5, No.3, May 1979, pp.188-194.

[THOM79] R. Thomas, "A Majority Consensus Approach to Concurrency Control," *ACM Transactions on Database Systems*, Vol.4, No.2, June 1979, pp.180-209.

[WALK83] B. J. Walker, et al., "The LOCUS Distributed Operating System," *Proceedings of the 9th ACM Symposium on Operating System Principles (SOSP 83)*, Bretton Woods, NH, October 10-13, 1983, pp.49-70.

CHAPTER V

AN IMPLEMENTATION OF FAULT-TOLERANT LOCKING PROTOCOL FOR SHARED MEMORY SYSTEMS

An Implementation of Fault-Tolerant Locking Protocol for Shared Memory Systems

1. INTRODUCTION

The fault-tolerant locking protocol (FTL) maintains the mutual consistency of replicated file copies and the internal consistency of all shared files in the event of failures of computers, shared memories, and/or paths in the interconnection network. Section 2 provides an overview of the FTL protocol, and section 3 presents an implementation of FTL for the ARC multi-microprocessor testbed. (This implementation is based on the testbed application program dated November 4, 1982.) In section 4, four experiments are suggested to assess the performance of the FTL protocol implementation in the testbed.

2. DESCRIPTION OF THE FTL PROTOCOL

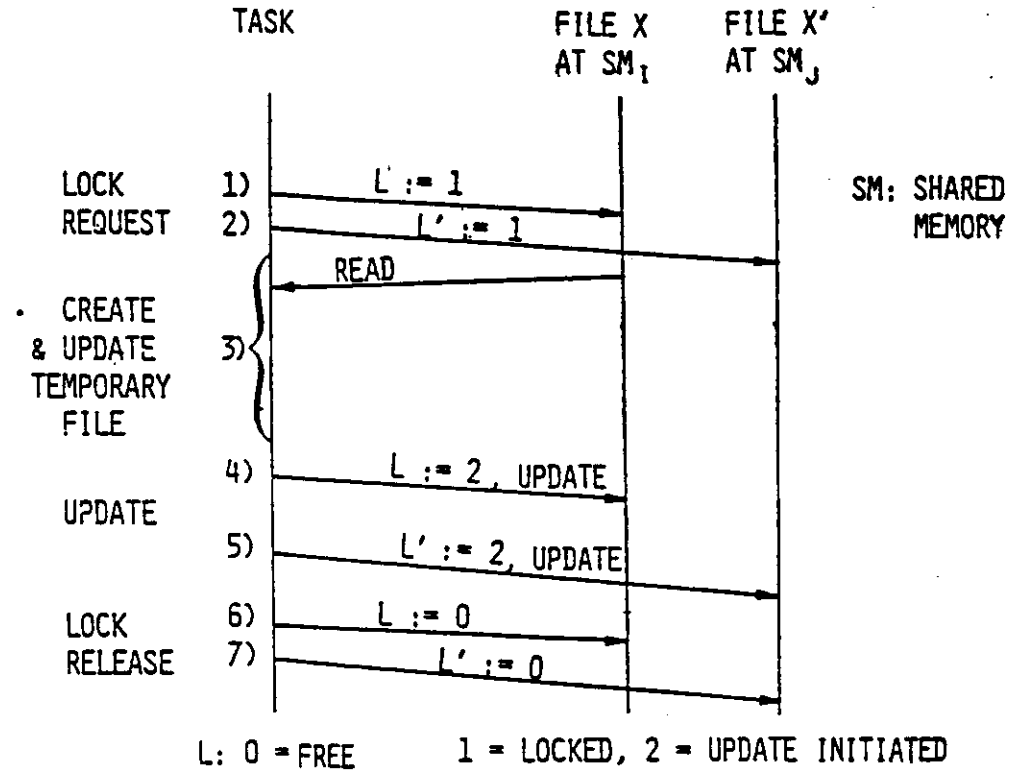
The FTL protocol requires that shared files be replicated to ensure file accessibility in the event of a failure.

2.1. Handling a Computer Failure

A computer failure during a file update may result in the loss of internal and/or mutual consistency. To avoid this, the FTL protocol requires that computers complete their modifications to one file copy before starting to modify another copy (see figure 1). Thus, at most one copy will lose internal consistency when a computer fails. A computer failure can be detected by a timeout on how long a computer holds a lock. To recover from a computer failure, lock words are used to determine which file copy is inconsistent (figure 2). The inconsistent file copy can be recovered by using the contents of a consistent copy of the same file.

2.2. Handling Memory Module and Interconnection Network Failures

Memory module and/or interconnection network failures can make file copies inaccessible. If a region of shared memory containing a file copy becomes inaccessible, the copy should be discarded.



PROTOCOL:

1) LOCK FILE X
2) LOCK FILE X'
3) CREATE AND UPDATE TEMPORARY FILE IN LOCAL MEMORY
4) MARK LOCK BIT (L := 2) AND UPDATE FILE X
5) MARK LOCK BIT (L' := 2) AND UPDATE FILE X'
6) UNLOCK FILE X
7) UNLOCK FILE X'

<Figure 1> Fault-Tolerant Locking (FTL) Protocol

LOCKS L	LOCKS L'	RECOVERY REQUIRED?	INCONSISTENT FILE COPY	UPDATE COMPLETED?
0	0	NO	N/A	N/A
1	0	NO	N/A	NO
1	1	NO	N/A	NO
2	1	YES	X	NO
2	2	YES	X'	YES
0	2	NO	N/A	YES

<Figure 2> Lock State Table

Such failures can be handled in the following way. Each computer maintains a table in its local memory that indicates which file copies are accessible. This table is consulted before retrieving a file copy. When a file copy becomes inaccessible (which will be detected by a computer trap), the detecting computer updates its local table and notifies the other computers. This notification process itself must be fault-tolerant. We recommend a message passing technique in which copies of the message are placed in two or more memory modules.

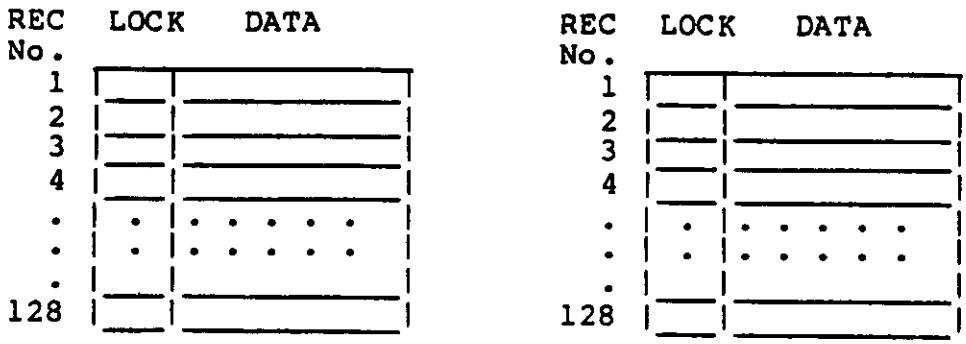
3. An Implementation of the FTL Protocol

The FTL protocol can be applied to the testbed application by duplicating the object track file. Here, the lock unit is a record instead of the entire file. The following sections describe the necessary data structures and the PDL code to implement the FTL protocol.

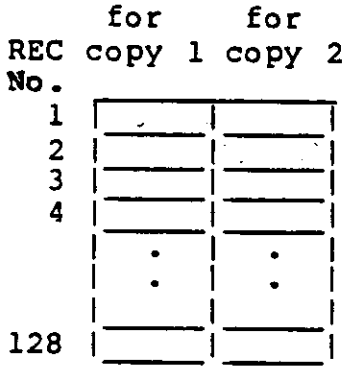
3.1. DUPLICATED RECORDS OF TRACK FILE AND A RECORD STATUS TABLE

Records of the track file are duplicated and a lock word is attached to each record copy. A record status table, which indicates the accessibility of each record copy of the track file, is maintained in the local memory of each computer (Figure 3).

Figure 4 shows the PDL TYPE declarations for the data structures. A new state 'INACC' is added to the lock word to indicate that the record copy is inaccessible due to error.



a) Duplicated Track File
in shared memory modules



b) Record Status Table
in each computer

<Figure 3> Duplicated Track File and
Record Status Table

```
TYPE

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

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

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

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

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

REC_STAT = (FAILED, GOOD);
```

<Figure 4> Type Declarations for FTL

3.2. INITIALIZATION OF RECORD STATUS TABLE AND USE OF A FILE RECORD

As shown in figure 5, the Record Status Table is initialized with all record copies accessible. Track file copies in the shared memories are addressed indirectly through local variables TRACK_COPY(1) and TRACK_COPY(2).

A record in the Track File should be updated in the following way:

1) Call RECORD_LOCK. RECORD_LOCK returns status indicating the accessibility of record copies (see figure 6).
2) Make a copy of the accessible record in local memory.
3) Update the local copy.
4) Call RECORD_UPDATE to update the record copies in shared memory (see figure 7).
5) Call RECORD_UNLOCK (see figure 8).

```
VAR

REC_STAT_TABLE : ARRAY [1..2,1..128] OF REC_STAT;
TRACK_COPY : ARRAY [1..2] OF ^TRACK_FILE;
TRACK_REC  : TRACK_DATA;
REC_ID     : INTEGER;
RSTAT      : INTEGER;
I,J        : INTEGER;

BEGIN

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

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

     . . . . . .
     . . . . . .


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

RECORD_LOCK (REC_ID,RSTAT);
       {lock the record}
IF RSTAT=-1 THEN ERR_ROUTINE;
       {both copies are inaccessible}

TRACK_REC := TRACK_COPY(RSTAT)^(REC_ID).REC_DATA;
      {get a local copy of the record}

WITH TRACK_REC DO
  <<UPDATE THE RECORD LOCALLY>>;
ENDWITH;

RECORD_UPDATE(REC_ID,TRACK_REC);
       {update copies in the shared memory}
RECORD_UNLOCK(REC_ID);
       {unlock the record}
END
```

<Figure 5> Initialization of the Record Status Table and Use of a Record of the Track File

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

  {RSTAT : return status
    1,2 ) successful lock and use REC #1 or REC #2
    -1  ) unsuccessful return}

CONST
  MAX_TRY = 100;

VAR
  I    : INTEGER;
  STAT : INTEGER;

BEGIN
  START:
  RSTAT:=-1;
  LOOP FOR I:=1 TO 2:
    IF REC_STAT_TABLE(I,REC_ID)=GOOD THEN
      CAPTUR$(TRACK_COPY(I)^(REC_ID).REC_LOCK,
              MAX_TRY, STAT);
      IF STAT=0 THEN        {locked successfully}
        RSTAT:=I;
      ELSEIF STAT=-1 THEN  {the copy is inaccessible}
        REC_STAT_TABLE(I,REC_ID):=FAILED;
      ELSEIF STAT=-2  THEN {the copy has been locked for too long
                            a time period by another computer}
        RECONF$(REC_ID, STAT);
        IF STAT=-1 THEN ERR_ROUTINE; ENDIF;
                           {both copies are inaccessible}
        GO TO START;
      ENDIF;
    ENDIF;
  ENDLOOP;
END;
```

<Figure 6> Subroutine RECORD_LOCK

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

VAR
  I: INTEGER;

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

<Figure 7> Subroutine RECORD_UPDATE

```
PROCEDURE RECORD_UNLOCK (REC_ID:INTEGER);

VAR
  I: INTEGER;

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

<Figure 8> Subroutine RECORD_UNLOCK

3.3. SUBROUTINE CAPTUR$

CAPTUR$ reads the lock word of a given record copy and locks that copy if it is free. CPATUR$ returns status indicating that: (1) the record was locked successfully, (2) the record was inaccessible, or (3) the computer that holds the lock has failed.

A simple timeout mechanism is used for failure detection as follows:

1) An integer COUNT is attached to each lock word and is increased by one each time the copy is locked.
2) CAPTUR$ repeatedly tries to lock the record copy.
3) If the COUNT remains unchanged for too many tries, the current lock holder is considered to have failed.

Since problems can arise if more than one computer concurrently detect the failure of another computer, exactly one computer is allowed to detect the failure.

```
PROCEDURE CAPTUR$ (VAR LOCK: LOCK_WORD, TIMES: INTEGER,
                   VAR RSTAT: INTEGER);

   {RSTAT: return status
       0) successful return
      -1) not done due to inaccessible record
      -2) not done within the given no. of try}

VAR
   NO_LOOP : INTEGER;
   CUR_COUNT : INTEGER;

BEGIN
   RSTAT := -2;

   WITH LOCK DO
   START:
   CUR_COUNT := LOCK_COUNT;
   LOOPS:
     LOOP FOR NO_LOOP :=1 TO TIMES:
       << EXCULSIVE ACCESS TO LOCK_WORD >>;
       CASE LOCK_FLAG OF
         INACC: BEGIN
                  << RELEASE EXCLUSIVE ACCESS >>;
                  RSTAT:=-1;
                  ESCAPE LOOPS;
                END;
         FREE:  BEGIN
                  LOCK_FLAG:=LOCKED; {lock it}
                  << RELEASE EXCLUSIVE ACCESS >>;
                  LOCK_COUNT:=CUR_COUNT+1;
                        {increase LOCK_COUNT by 1}
                  RSTAT:=0;
                  ESCAPE LOOPS;
                END;
         OTHERWISE {LOCKED or UPD_INITED}:
                  << RELEASE EXCLUSIVE ACCESS >>;
       ENDCASE;
       << DELAY >>;
       {if LOCK_COUNT has been changed, start from the
        beginning}
       IF CUR_COUNT <> LOCK_COUNT THEN GO TO START; ENDIF;
     ENDLOOP;
```

<Figure 9> Subroutine CAPTUR$ (continued...)

```
      {Let only one computer detect the failure and
       let others keep requesting of lock}
      IF RSTAT=-2 THEN
        << EXCLUSIVE ACCESS TO LOCK WORD >>;
        IF CUR COUNT<>LOCK COUNT THEN
          << RELEASE EXCLUSIVE ACCESS >>;
          GO TO START;
        ELSE
          LOCK COUNT:=CUR COUNT+1;
          << RELEASE EXCLUSIVE ACCESS >>;
        ENDIF;
      ENDIF;
    ENDWITH;
END;
```

<Figure 9> Subroutine CAPTUR$

3.4. SUBROUTINE RECONF$

When a computer failure is detected, the record copies locked by that computer must be: (1) made internally and mutually consistent and (2) released so that other computers can use them. RECONF$ executes a recovery procedure according to the states of the lock words of the record copies (figure 10).

```
PROCEDURE RECONF$ (REC_ID: INTEGER, VAR RSTAT: INTEGER);

   {RSTAT: return status
      0) successful reconf.
     -1) unsuccessful reconf.}

VAR
  STAT: ARRAY [1..2] OF LOCK_STAT;
  I   : INTEGER;

BEGIN

  {get the states of the lock words of the given record copy}
  LOOP FOR I:=1 TO 2:
    STAT(I) := INACC;
    IF REC_STAT_TABLE(I,REC_ID)=GOOD THEN
      STAT(I):= TRACK_COPY(I)^(REC_ID).REC_LOCK.LOCK_FLAG;
      IF STAT(I)=INACC THEN
        REC_STAT_TABLE(I,REC_ID):=FAILED;
      ENDIF;
    ENDIF;
  ENDLOOP;


  WITH PRI_LOCK_FLAG =
          TRACK_COPY(1)^(REC_ID).REC_LOCK.LOCK_FLAG,
       SEC_LOCK_FLAG =
          TRACK_COPY(2)^(REC_ID).REC_LOCK.LOCK_FLAG,
       PRI_REC_DATA  =
          TRACK_COPY(1)^(REC_ID).REC_DATA,
       SEC_REC_DATA  =
          TRACK_COPY(2)^(REC_ID).REC_DATA DO

    RSTAT:=-1;
    IF STAT(1)=INACC THEN
      IF STAT(2)=FREE OR STAT(2)=LOCKED THEN
        {copy #1 inaccessible, copy #2  free or just locked}
        SEC_LOCK_FLAG:=FREE;
        RSTAT:=0
      ELSEIF STAT(2)=UPD_INITED
        {copy #1 inaccessible, copy #2 being updated}
        SEC_LOCK_FLAG:=INACC;
        REC_STAT_TABLE(2,REC_ID):=FAILED;
      ENDIF;
```

<Figure 10> Subroutine RECONF$ (continued...)

```
    ELSEIF STAT(2)=INACC THEN
      IF STAT(1)=FREE OR STAT(1)=LOCKED THEN
        {copy #2 inaccessible, copy #1  free or just locked}
        PRI_LOCK_FLAG:=FREE;
        RSTAT:=0
      ELSEIF STAT(1)=UPD_INITED THEN
      {copy #2 inaccessible, copy #1 being updated}
      PRI_LOCK_FLAG:=INACC;
      REC_STAT_TABLE(1,REC_ID):=FAILED;
    ENDIF;

    ELSEIF STAT(1)=UPD_INITED AND STAT(2)=LOCKED THEN
      {copy #1 being updated, copy #2 just locked}
      {make copy #1 identical to copy #2}
      PRI_REC_DATA := SEC_REC_DATA;
      PRI_LOCK_FLAG:=FREE;
      SEC_LOCK_FLAG:=FREE;
      RSTAT:=0;

    ELSEIF STAT(1)=UPD_INITED AND STAT(2)=UPD_INITED THEN
      {copy #1 completely updated, copy #2 being updated}
      {make copy #2 identical to copy #1}
      SEC_REC_DATA := PRI_REC_DATA;
      PRI_LOCK_FLAG:=FREE;
      SEC_LOCK_FLAG:=FREE;
      RSTAT:=0;

    ELSE
      {two copies are already consistent}
      PRI_LOCK_FLAG:=FREE;
      SEC_LOCK_FLAG:=FREE;
      RSTAT:=0;
    ENDIF;
  ENDWITH;

END;
```

<Figure 10> Subroutine RECONF$

4. EXPERIMENTS

Here, we suggest experiments to assess the performance of the FTL protocol implementation in the testbed. The first three experiments deal with the FTL protocol in the absence of failures: 1) overhead of the FTL protocol, 2) choice of lock-request retry period, and 3) choice of time-out period for computer failure detection. Experiment 4 studies the FTL protocol in the presence of computer failures.

4.1. Experiment 1: Overhead of the FTL Protocol

The FTL protocol requires more processing than when fault-tolerance is not considered (i.e. the baseline system). The purpose of this experiment is to compare port-to-port times and computer utilizations of the FTL protocol implementation with those of the baseline to determine the magnitude of these additional processing requirements.

4.2. Experiment 2: Choice of Lock-Request Retry Period

When a computer can not lock a TRACK record, it retries repeatedly until: (a) it succeeds or (b) a failure is detected. If the period between retries (retry period) is too short, shared memory conflicts increase; if the retry period is too long, computers may wait for a lock even though the record is free. A set of experiments should be conducted to see the effect on system performance of

different values for lock-request retry period. Measurements of port-to-port and computer utilization should be taken.

4.3. Experiment 3: Choice of Time-out Period for Computer Failure Detection

The time-out period for detecting a computer failure during database update should be longer than the maximum time tasks may hold locks. Thus, experiments should be conducted to determine how long tasks hold locks. Since lock holding times may vary with tactical time, the data should be collected in appropriate time intervals.

4.4. Experiment 4: Performance of the FTL Protocol with Computer Failures

This set of experiments studies the behavior of the FTL protocol when computers fail. Some techniques are needed to simulate computer failures during database update (e.g. forcing a computer to get into an infinite loop while it holds a lock), and the times at which failures occur should be logged. In addition to port-to-port times and computer utilizations, the following data should be collected:

1) time to detect a computer failure, and
2) time to recover from a computer failure

ACKNOWLEDGEMENTS

The authors would like to thank Joseph Bannister of UCLA for his discussions and carefully reading a draft of this report, and Aeri Lee for her secretarial and administrative support in preparing the report.

DISTRIBUTION LIST

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

2. HQDA (DACS-BMT)
Alexandria, VA 22333

3. Ballistic Missile Defense Program Office
ATTN: DACS-BMT
AMC Building, 7th Floor
5001 Eisenhower Avenue
Alexandria, VA 22333

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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