Experimental Evaluation of a Concurrent Checkpointing Algorithm

Sang H. Son and Shi-Chin Chiang

Computer Science Report No. TR-90-01
January 17, 1990

Experimental Evaluation of a Concurrent Checkpointing Algorithm

Sang H. Son
Shi-Chin Chiang

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

ABSTRACT

The goal of checkpointing in database management systems is to save database states on a separate secure device so that the database can be recovered when errors and failures occur. Recently, non-interfering checkpointing algorithms which allow user transactions to be processed concurrently with the checkpointing process have been proposed [4, 7]. This property of non-interference is highly desirable to real-time applications, where restricting transaction activity during the checkpointing operation is in many cases not feasible. The performance of those algorithms, however, has not been studied extensively. One of the reasons for the difficulty in evaluating new techniques for distributed database systems is the complexity involved due to a large number of system parameters that may change dynamically. Prototyping methods can be applied effectively to the evaluation of database techniques. In this paper, a checkpointing algorithm to improve the availability of distributed database systems is evaluated experimentally using a software prototyping environment for the development and evaluation of distributed database systems.

Index Terms - database, consistency, checkpoint, transaction, non-interference, prototyping

---

This work was supported in part by the Office of Naval Research under contract number N00014-88-K-0245, by the Department of Energy under contract number DEFG05-88-ER25063, and by the Federal Systems Division of IBM Corporation under University Agreement WG-249153.

1. Introduction

The need for a recovery mechanism in database systems is well understood. In spite of powerful database integrity checking mechanisms which detect errors and undesirable data, it is possible that some erroneous data may be included in the database. Furthermore, even with a perfect integrity checking mechanism, failures of hardware and/or software at the processing sites may destroy consistency of the database. In order to cope with those errors and failures, distributed database systems provide recovery mechanisms, and checkpointing is a technique frequently used in recovery mechanisms.

The goal of checkpointing in database systems is to read and save a consistent state of the database on a separate secure device. In case of a failure, the stored data can be used to restore the database. Checkpointing must be performed so as to minimize both the costs of performing checkpoints and the costs of recovering the database. If the checkpoint intervals are very short, too much time and resources are spent in checkpointing; if these intervals are long, too much time is spent in recovery. Since checkpointing is an effective method for maintaining consistency of database systems, it has been widely used and studied by many researchers[1, 4, 5, 7, 8, 10, 11, 13, 17, 18].

Since checkpointing is performed during normal operation of the system, the interference with transaction processing must be kept to a minimum. It is highly desirable that users are allowed to submit transactions while checkpointing is in progress, and the transactions are executed in the system concurrently with the checkpointing process. In distributed systems, this requirement of non-interference makes checkpointing complicated because we need to consider coordination among autonomous sites of the system. A quick recovery from failure is also desirable in many applications of distributed databases that require high availability. For achieving quick recovery, each checkpoint needs to be globally consistent so that a simple restoration of the latest checkpoint can bring the database to a consistent state. To make each checkpoint globally consistent, updates of a transaction must be either included completely in one checkpoint, or not included at all. In distributed database systems these desirable properties of non-interference and global consistency increase the workload of the system. It may turn out that the overhead of the

checkpointing mechanism is unacceptably high, in which case the mechanism should be abandoned in spite of its desirable properties. The practicality of non-interfering checkpointing, therefore, depends partially on the amount of extra workload incurred by the checkpointing mechanism.

One of the primary reasons for the difficulty in successfully developing and evaluating a distributed database system is that it takes a long time to develop a system, and evaluation is complicated because it involves a large number of system parameters that may change dynamically.

A prototyping technique can be applied effectively to the evaluation of control mechanisms for distributed database systems. A *database prototyping tool* is a software package that supports the investigation of the properties of a database control techniques in an environment other than that of the target database system. The advantages of an environment that provides prototyping tools are obvious. First, it is cost effective. If experiments for a twenty-node distributed database system can be executed in a software environment, it is not necessary to purchase a twenty-node distributed system, reducing the cost of evaluating design alternatives. Second, design alternatives can be evaluated in a uniform environment with the same system parameters, making a fair comparison. Finally, as technology changes, the environment need only be updated to provide researchers with the ability to perform new experiments.

A prototyping environment can reduce the time of evaluating new technologies and design alternatives. From our past experience, we assume that a relatively small portion of a typical database system's code is affected by changes in specific control mechanisms, while the majority of code deals with intrinsic problems, such as file management. Thus, by properly isolating technology-dependent portions of a database system using modular programming techniques, we can implement and evaluate design alternatives very rapidly. Although there exist tools for system development and analysis, few prototyping tools exist for distributed database experimentation. Especially if the system designer must deal with message-passing protocols and timing constraints, it is essential to have an appropriate prototyping environment for success in the design and analysis tasks.

This paper discusses checkpointing algorithms for distributed database systems, and presents a prototyping software implemented for a series of experimentation to evaluate non-interfering checkpointing algorithms. This paper is organized as follows. Section 2 introduces a model of computation used in this paper. Section 3 discusses the design issues for checkpointing algorithms and reviews previous work which has appeared in the literature. Section 4 describes the checkpointing algorithm. Section 5 introduces the structure of the prototyping environment. Sections 6 presents the results of experiments conducted for practicality analysis. Section 7 concludes the paper.

2. Model of Computation

A *database* consists of a set of data objects. Each data object has a *value* and represents the smallest unit of the database accessible to the user. All user requests for access to the database are handled by the *database system*. We consider a distributed database system implemented on a computing system where several autonomous computers (called *sites)* are connected via a communication network. The set of data objects in a distributed database system is partitioned among its sites. A database is said to be *consistent* if the values of data objects satisfy a set of assertions. The assertions that characterize the consistent states of the database are called the *consistency constraints* [6].

The basic units of user activity in database systems are *transactions*. Each transaction represents a complete and correct computation, i.e., if a transaction is executed alone on an initially consistent database, it would terminate in a finite time and produce correct results, leaving the database consistent. A transaction is the unit of consistency and hence, it must be *atomic*. By atomic, we mean that intermediate states of the database must not be visible outside the transaction, and all updates of a transaction must be executed in an all-or-nothing fashion. A transaction is said to be *committed* when it is executed to completion, and it is said to be *aborted* when it is not executed at all. When a transaction is committed, the output values are finalized and made available to all subsequent transactions.

Each transaction has a timestamp associated with it [12]. A timestamp is a number that is assigned to a transaction when initiated and is kept by the transaction. Two important properties of timestamps are

(1) no two transactions have the same timestamp, and (2) only a finite number of transactions can have a timestamp less than that of a given transaction.

The transaction managers that have been involved in the execution of a transaction are called the *participants* of the transaction. The *coordinator* is one of the participants which initiates and terminates the transaction by controlling all other participants. In our transaction processing model, we assume that the coordinator decides on the participants using suitable decision algorithms, based on the data objects in the read set and write set of the transaction.

3. Related Work

3.1. Checkpointing

In order to achieve the goal of efficient database system recoverability, it is necessary to consider the following issues when a checkpointing mechanism is designed for a distributed database system.

(1) it should generate globally consistent checkpoints,

(2) it should be non-interfering in that it does not affect the ongoing processing of transactions,

(3) the storage and the communication overhead should be small.

The need and the desirability of these properties is self evident. For example, even though an inconsistent checkpoint may be quick and inexpensive to obtain, it may require a lot of additional work to recover a consistent state of the database. Some of the schemes appearing in the literature (e.g. [1, 5]) do not meet this criteria.

Checkpointing can be classified into three categories according to the coordination necessary among the autonomous sites. These are (1) fully synchronized[10], (2) loosely synchronized[17], and (3) nonsynchronized[5]. Fully synchronized checkpointing is done only when there is no active transaction in the database system. In this scheme, before writing a local checkpoint, all sites must have reached a state of inactivity. In a loosely synchronized system, each site is not compelled to write its local checkpoint in the same global interval of time. Instead, each site can choose the point of time to stop processing and take

the checkpoint. A distinguished site locally manages a checkpoint sequence number and broadcasts it for the creation of a checkpoint. Each site takes local checkpoint as soon as it is possible, and then resumes normal transaction processing. It is then the responsibility of the local transaction managers to guarantee that all global transactions run in the local checkpoint intervals bounded by checkpoints with the same sequence numbers. In nonsynchronized checkpointing, global coordination with respect to the recording of checkpoints does not take place at all. Each site is independent from all others with respect to the frequency of checkpointing and the time instants when local checkpoints are recorded. A logically consistent database state is not constructed until a global reconstruction of the database is required.

One of the drawbacks common to the checkpointing schemes above is that the processing of transactions must be stopped for checkpointing. Maintaining transaction inactivity for the duration of the checkpointing operation is undesirable, or even not feasible, depending on the availability constraints imposed by the system.

In [13], checkpointing is always performed exclusively as part of the commitment of transactions. This scheme has the advantage of not having a separate checkpointing mechanism, but may have problems if the number of transactions allowed is too many or if it is necessary to keep checkpoints for a long time. A similar checkpointing mechanism is suggested in [11]. The synchronization of checkpointing in [11] is achieved through the time-stamp ordering, making the global reconstruction easier than in the scheme of [13]. The storage requirements of these transaction-based checkpointing mechanisms depend upon the amount of information saved for each transaction, and are difficult to compare with the checkpointing mechanisms which save only the values of data objects.

In [1], a backup database is created by pretending that the backup database is a new site being added to the system. An initialization algorithm is executed to bring the new site up-to-date. One drawback of this scheme is that the backup generation does interfere with update transactions.

In [7], a different approach based on a formal model of asynchronous parallel processes and an abstract distributed transaction system is proposed. It is called *non-intrusive* in the sense that no opera-

tions of the underlying system need be halted while the global checkpoint is being executed. The non-intrusive checkpointing approach as suggested in [7] describes the behavior of an abstract system and does not provide a practical procedure for obtaining a checkpoint.

3.2. Consistency and Concurrency Control

To maintain the consistency of a distributed database system, atomicity and serializability of transactions must be assured by a correct commit algorithm (e.g., [19]) and concurrency control algorithms. Concurrency control in distributed database systems is based on three basic methods: locking, timestamp ordering, and validation[24]. These basic methods can be applied to either single-version or multiversion database systems.

Since the checkpointing algorithm presented in this paper uses timestamps to select a database state, it fits naturally with concurrency control algorithms based on timestamp ordering. Consider basic timestamp ordering algorithm (BTO) for example. In order to ensure transaction atomicity, BTO is integrated with the two-phase commit procedure[25]. Instead of performing write operations, prewrite operations are issued by transactions and they are not applied to the local database. Only when all prewrite operations of a transaction are accepted, the transaction is committed and proceeds to perform its write operations on the database. When a checkpoint begins, performing write operations on the database depends on the timestamp of the transactions. If timestamp(T_i) $\leq$ the timestamp of the current checkpoint, write operations are performed not on the database, but on a separate file, called the *committed temporary versions* (CTV) file.[1]

The checkpointing algorithm also works nicely with multiversion timestamp ordering methods[24]. Since each committed transaction creates new versions instead of overwriting old versions, CTV file management would not be necessary. In this case, a checkpointing process can mark appropriate versions for a consistent state of the database without actually storing them. If old versions are garbage collected, however, this may not work, and the system should maintain the CTV file for checkpointing.

[1]Committed temporary versions are created by committed transactions whose updates must not be included in the current checkpoint.

Although the checkpointing algorithm does not fit very naturally with locking-based concurrency control algorithms, it is still possible to integrate locking with the checkpointing algorithm. It is the ordering of transaction execution that is required to generate a consistent database state. In timestamp-based concurrency control, such ordering information can be inferred to by timestamps. If we can use similar information when determining transactions whose updates must be included in the current checkpoint,[2] our checkpointing algorithm can return a consistent state of the database. However, there would be more overhead involved in doing this than using timestamps, because precedency relationship among transactions must be maintained at each data object for checkpointing.

4. An Algorithm for Non-Interfering Checkpoints

4.1. Motivation of Non-interference

The motivation of having a checkpointing scheme which does not interfere with transaction processing is well explained in [4] by using the analogy of migrating birds and a group of photographers. Suppose a group of photographers observe a sky filled with migrating birds. Because the scene is so vast that it cannot be captured by a single photograph, the photographers must take several snapshots and piece the snapshots together to form a picture of the overall scene. Furthermore, it is desirable that the photographers do not disturb the process that is being photographed. The snapshots cannot all be taken at precisely the same instance because of synchronization problems, and yet they should generate a ''meaningful'' composite picture.

In a distributed database system, each site saves the state of the data objects stored at it to generate a local checkpoint. We cannot ensure that the local checkpoints are saved at the same instance, unless a global clock can be accessed by all the checkpointing processes. Moreover, we cannot guarantee that the global checkpoint, consisting of local checkpoints saved, is consistent. Non-interfering checkpointing algorithms are very useful for the situations in which a quick recovery as well as no blocking of transac-

[2]Such transactions are called *before-checkpoint-transactions* (BCPT). The meaning of those terms will be explained in the next section.

tions is desirable. Instead of waiting for a consistent state to occur, the non-interfering checkpointing approach constructs a state that would result by completing the transactions that are in progress when the global checkpoint begins.

In order to make each checkpoint globally consistent, updates of a transaction must be either included in the checkpoint completely or not at all. To achieve this, transactions are divided into two groups according to their relations to the current checkpoint: *after-checkpoint-transactions* (ACPT) and *before-checkpoint-transactions* (BCPT). Updates belonging to BCPT are included in the current checkpoint while those belonging to ACPT are not included. In a centralized database system, it is an easy task to separate transactions for this purpose. However, it is not easy in a distributed environment. For the separation of transactions in a distributed environment, a special time-stamp which is globally agreed upon by the participating sites is used. This special time-stamp is called the *Global Checkpoint Number* (GCPN), and it is determined as the maximum of the *Local Checkpoint Numbers* (LCPN) through the coordination of all the participating sites.

An ACPT can be reclassified as a BCPT if it turns out that the transaction must be executed before the current checkpoint. This is called the *conversion* of transactions. The updates of a converted transaction are included in the current checkpoint.

4.2. The Algorithm

There are two types of processes involved in the execution of the algorithm: *checkpoint coordinator* (CC) and *checkpoint subordinate* (CS). The checkpoint coordinator starts and terminates the global checkpointing process. Once a checkpoint has started, the coordinator does not issue the next checkpoint request until the first one has terminated.

The variables used in the algorithm are as follows:

(1) *Local Clock* (LC): a clock maintained at each site which is manipulated by the clock rules of Lamport[12].

(2) *Local Checkpoint Number* (LCPN): a number determined locally for the current checkpoint.

(3) *Global Checkpoint Number* (GCPN): a globally unique number for the current checkpoint.

(4) CONVERT: a Boolean variable showing the completion of the conversion of all the eligible transactions at the site.

Our checkpointing algorithm works as follows:

(1) The checkpoint coordinator broadcasts a Checkpoint Request Message with a time-stamp LC_{CC}. The local checkpoint number of the coordinator is set to LC_{CC}. The coordinator sets the Boolean variable CONVERT to false:

$$CONVERT_{CC} := false$$

and marks all the transactions at the coordinator site with the time-stamps not greater than $LCPN_{CC}$ as BCPT.

(2) On receiving a Checkpoint Request Message, the local clock of site m is updated and $LCPN_m$ is determined by the checkpoint subordinate as follows:

$$LC_m := \max(LC_{CC} + 1, LC_m)$$
$$LCPN_m := LC_m$$

The checkpoint subordinate of site m replies to the coordinator with $LCPN_m$, and sets the Boolean variable CONVERT to false:

$$CONVERT_m := FALSE$$

and marks all the transactions at the site m with the time-stamps not greater than $LCPN_m$ as BCPT.

(3) The coordinator broadcasts the GCPN which is decided by:

$$GCPN := \max(LCPN_n) \qquad n = 1,..,N$$

(4) For all sites, after LCPN is fixed, all the transactions with the time-stamps greater than LCPN are marked as temporary ACPT. If a temporary ACPT wants to update any data objects, those data objects are copied from the database to the buffer space of the transaction. When a temporary ACPT commits, updated data objects are not stored in the database as usual, but are maintained as *committed temporary versions* (CTV) of data objects. The data manager of each site maintains the permanent and temporary versions of data objects. When a read request is made for a data object which has committed temporary versions, the value of the latest committed temporary version is returned. When a write request is made for a data object which has committed temporary versions, another committed temporary version is created for it rather than overwriting the previous committed temporary version.

(5) When the GCPN is known, each checkpointing process compares the time-stamps of the temporary ACPT with the GCPN. Transactions that satisfy the following condition become BCPT; their updates are reflected into the database, and are included in the current checkpoint.

$$LCPN < \text{time-stamp}(T) \leq GCPN$$

The remaining temporary ACPT are treated as actual ACPT; their updates are not included in the current checkpoint. These updates are included in the database after the current checkpointing has been completed. After the conversion of all the eligible BCPT, the checkpointing process sets the Boolean variable CONVERT to true:

$$CONVERT := true$$

(6) Local checkpointing is executed by saving the state of data objects when there is no active BCPT and the variable CONVERT is true.

(7) After the execution of local checkpointing, the values of the latest committed temporary versions are used to replace the values of data objects in the actual database. Then, all committed temporary versions are deleted.

The above checkpointing algorithm essentially consists of two phases. The function of the first phase (steps 1 through 3) is the assignment of GCPN that is determined from the local clocks of the system. The second phase begins by fixing the LCPN at each site. This is necessary because each LCPN sent to the checkpoint coordinator is a candidate of the GCPN of the current checkpoint, and the committed temporary versions must be created for the data objects updated by ACPT. The notions of committed temporary versions and conversion from ACPT to BCPT are introduced to assure that each checkpoint contains all the updates made by transactions with earlier time-stamps than the GCPN of the checkpoint.

When a site receives a Transaction Initiation Message, the transaction manager checks whether or not the transaction can be executed at this time. If the checkpointing process has already executed step 5 and time-stamp(T) $\leq$ GCPN, then a TIM-NACK message is returned. Therefore in order to execute step 6, each checkpointing process only needs to check active BCPT at its own site, and yet the consistency of the checkpoint can be achieved.

4.3. Termination of the Algorithm

The algorithm described so far has no restriction on the method of arranging the execution order of transactions. With no restriction, however, it is possible that the algorithm may never terminate. In order to ensure that the algorithm terminates in a finite time, we must ensure that all BCPT terminate in a finite time, because local checkpointing in step 6 can occur only when there is no active BCPT at the site.

Termination of transactions in a finite time is ensured if the concurrency control mechanism gives priority to older transactions over younger transactions. With such a time-based priority, it is guaranteed that once a transaction T_i is initiated by sending Start Transaction Messages, then T_i is never blocked by subsequent transactions that are younger than T_i. The number of transactions that may block the execution of T_i is finite because only a finite number of transactions can be older than T_i. Among older

transactions which may block T_i, there must be the oldest transaction which will terminate in a finite time, since no other transaction can block it. When it terminates, the second oldest transaction can be executed, and then the third, and so on. Therefore, T_i will be executed in a finite time. Since we have a finite number of BCPT when the checkpointing is initiated, all of them will terminate in a finite time, and hence the checkpointing itself will terminate in a finite time. Concurrency control mechanisms based on time-stamp ordering as in [2, 20] can ensure the termination of transactions in a finite time.

5. Structure of the Prototyping Environment

For a prototyping tool for distributed database systems to be effective, appropriate operating system support is mandatory. Database control mechanisms need to be integrated with the operating system, because the correct functioning of control algorithms depends on the services of the underlying operating system; therefore, an integrated design reduces the significant overhead of a layered approach during execution.

Although an integrated approach is desirable, the system needs to support flexibility which may not be possible in an integrated approach. In this regard, the concept of developing a library of modules with different performance and reliability characteristics for an operating system as well as database control functions seems promising. Our prototyping environment follows this approach [21, 23]. It is designed as a modular, message-passing system to support easy extensions and modifications. Server processes can be created, relocated, and new implementations of server processes can be dynamically substituted. It efficiently supports a spectrum of distributed database functions at the operating system level, and facilitates the construction of multiple "views" with different characteristics. For experimentation, system functionality can be adjusted according to application-dependent requirements without much overhead for new system setup.

The prototyping environment provides support for transaction processing, including transparency to concurrent access, data distribution, and atomicity. An instance of the prototyping environment can manage any number of virtual sites specified by the user. Modules that implement transaction processing

are decomposed into several server processes, and they communicate among themselves through ports. The clean interface between server processes simplifies incorporating new algorithms and facilities into the prototyping environment, or testing alternate implementations of algorithms. To permit concurrent transactions on a single site, there is a separate process for each transaction that coordinates with other server processes.

Figure 1 illustrates the structure of the prototyping environment. The prototyping environment is based on a concurrent programming kernel, called the StarLite kernel, written in Modula-2. The StarLite kernel supports process control to create, ready, block, and terminate processes. Scheduler in the kernel maintains the simulation clock and provides the **hold** primitive to simulate the passage of time.

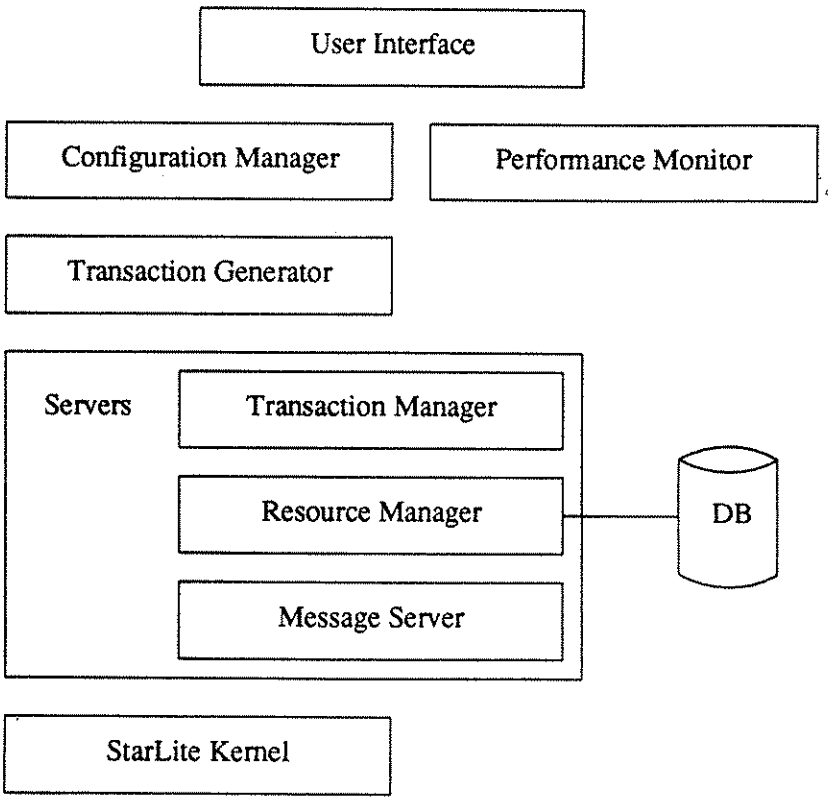

Fig. 1. Structure of the prototyping environment

User Interface (UI) is a front-end invoked when the prototyping environment begins. UI is menu-driven, and designed to be flexible in allowing users to experiment various configurations with different system parameters. A user can specify the following:

- system configuration: number of sites and the number of server processes at each site.
- database configuration: database at each site with user defined structure, size, granularity, and levels of replication.
- load characteristics: number of transactions to be executed, size of their read-sets and write-sets, transaction types (read-only or update) and their priorities, and the mean interarrival time of transactions.
- concurrency control: locking, timestamp ordering, and priority-based.

UI initiates the Configuration Manager (CM) which initializes necessary data structures for transaction processing based on user specification. CM invokes the Transaction Generator at an appropriate time interval to generate the next transaction to form a Poisson process of transaction arrival. When a transaction is generated, it is assigned an identifier that is unique among all transactions in the system.

Transaction execution consists of read and write operations. Each read or write operation is preceded by an access request sent to the Resource Manager, which maintains the local database at each site. Each transaction is assigned to the Transaction Manager (TM). TM issues service requests on behalf of the transaction and reacts appropriately to the request replies. For instance, if a transaction requests access to a file and that file is locked, TM executes either blocking operation to wait until the data object can be accessed, or aborting procedure, depending on the situation. If granting access to a resource will produce deadlock, TM receives an abort response and aborts the transaction. Transactions commit in two phases. The first commit phase consists of at least one round of messages to determine if the transaction can be globally committed. Additional rounds may be used to handle potential failures. The second commit phase causes the data objects to be written to the database for successful transactions. TM executes the two commit phases to ensure that a transaction commits or aborts globally.

The Message Server (MS) is a process listening on a well-known port for messages from remote sites. When a message is sent to a remote site, it is placed on the message queue of the destination site and the sender blocks itself on a private semaphore until the message is retrieved by MS. If the receiving site is not operational, a time-out mechanism will unblock the sender process. When MS retrieves a message, it wakes the sender process and forwards the message to the proper servers or TM. The prototyping environment implements Ada-style rendezvous (synchronous) as well as asynchronous message passing. Inter-process communication within a site does not go through the Message Server; processes send and receive messages directly through their associated ports.

The Performance Monitor interacts with the transaction managers to record, priority/timestamp and read/write data set for each transaction, time when each event occurred, statistics for each transaction and cpu hold interval in each node. The statistics for a transaction includes arrival time, start time, total processing time, blocked interval, whether deadline was missed or not, and number of aborts.

Since each TM is a separate process, each has its own data area in which to keep track of the time when a service request is sent out and the time the response arrives, as well as the time when a transaction begins blocking, waiting for a resource, and the time the resource is granted. When a transaction commits, it calls a procedure that records the above measures; when the simulation clock has expired, these measures are printed out for all transactions.

6. Implementation and Experiments

The proposed non-interfering checkpointing algorithm improves the system availability, but also introduces overhead to the system. Using the prototyping environment, we have implemented the checkpointing algorithm and conducted a series of experiments to investigate its practicality and limitations. In this section we first describe the implementation of the checkpointing algorithm briefly, and then present the results of the experiments performed to evaluate the overhead of the algorithm.

6.1. Implementation of The Algorithm

Two types of processes are implemented for the execution of the algorithm: checkpoint coordinator (CC) and checkpoint subordinate (CS). The checkpoint coordinator starts and terminates the global checkpointing process. Each checkpointing process participates in the actions of

- deciding the global checkpoint number (GCPN),
- keeping tracks of ACPTs and BCPTs,
- waiting until BCPTs are finished,
- performing a local checkpoint, i.e. saving the local database to a safe storage,
- reflecting updates of ACPT back to the database.

In the rest of the paper, we use CP period or CP frame for referring to the entire checkpointing procedure, while local CP refers to the action of saving the local database to a safe storage.

In implementing the checkpointing algorithm presented in Section 4, we have fixed the coordinator site for simplicity. The checkpoint process at the coordinator site initiates the checkpointing by broadcasting a "CP request" message. User transactions are generated and executed while the checkpointing is in progress.

For a single-site database system, the time to start local CP is obvious; local CP is initiated when there is no active BCPT, i.e, when all BCPTs are either committed or aborted. However, there can be two timing choices for distributed database systems to start local checkpointing. One approach is "WaitAll" and the other is "NoWait". The WaitAll approach synchronizes local checkpoints among all the participating sites. In this approach, the coordinator will wait until all of its participants have no active BCPTs. The coordinator then issues "perform local checkpoint" message to all participants. Each checkpointing process performs local checkpointing only after receiving this message.

The WaitAll choice follows the spirit of non-interference. In a distributed environment, it is possible that an update request of a transaction from a remote site arrives after the site starts local CP. If the

remote transaction is BCPT, it must be rejected because its update cannot be included consistently in the current checkpoint. In the WaitAll approach, no BCPT will be rejected since each site waits until the last BCPT in the system to terminate before starting its local CP. However, if there exist long-lived BCPTs in the system during a checkpoint period, the waiting time for the completion of all BCPTs can be very long.

The NoWait approach allows asynchronous local checkpointing in the system. In this approach, each site starts local checkpointing after BCPTs of its own site have been finished, without waiting for the "perform local checkpoint" message from the coordinator. Once a site has started local checkpointing, it will reject any update requests from BCPTs of other sites to maintain the consistency of the current checkpoint. The merit of following the NoWait approach is avoiding potentially long waiting time for those long-lived BCPTs.

The local checkpointing at each site is implemented as the following:

```
PROCEDURE DoLocalCheckpoint
BEGIN
  save local database into a separate storage device.
  reflect data objects updated by committed ACPTs into the database.
END.
```

Local checkpoint refers to the action of saving the local database. Reflecting ACPT updates is done after local CP. During the reflecting period, transactions can not access database to avoid accessing inconsistent data object. The database system will return to normal (i.e., non-CP) mode after the reflect operation is completed. Transactions will then resume normal execution. Due to this reason, we have implemented the reflect operation in the DoLocalCheckpoint procedure instead of separating two operations.

6.2. System Model and Preformance Measures

There are two performance measures that can be used in discussing the practicality of non-interfering checkpointing: extra storage and extra workload. We conduct experiments only to measure

extra workload, since extra storage required to maintain CTV is not much and critical in most realistic systems [22]. The extra workload imposed by the algorithm mainly consists of the workload for (1) determining the GCPN, (2) committing ACPT,(3) reflecting the committed temporary versions from the CTV file to the database, and (4) making the CTV file clear when the reflect operation is finished. Among these, workload for (2) and (3) dominates others. They are determined by the number of ACPTs and the number of updates.

ACPTs are transactions submitted to the system after global checkpoint number has been decided and before local CP is finished. Therefore, the number of ACPTs in one CP frame will be affected by (1) the length of one CP frame which is determined dynamically by the speed of the CP process, and (2) the number of transactions submitted to the system during that period. Parameters that affect the first factor include transaction types, degree of distribution, CP timing choice, and frequency and delay for disk I/O. Parameters that affect the second factor is the mean interarrival time of transactions. We have examined the impacts of those parameters through our experiments.

The experiments are conducted based on the following system model and implementation on the prototyping environment:

- We use strict timestamp ordering as the underlying concurrency control mechanism. Since the checkpointing algorithm uses timestamp to determine the class of transactions, this is a natural choice for system implementation.
- The workload imposed by executing local CP and reflecting updates of ACPTs from CTV file is measured by first taking the CPU time to process the request, followed by taking the I/O time to transfer data.
- We increase the system workload by increasing the number of transactions submitted to the system. Since we wanted to investigate the practicality of the proposed checkpointing algorithm, experiments were conducted using a set of parameters which can represent stable states of a distributed database system without checkpointing (a non-CP system).

- Each experiment concentrates on one major parameter. Average response time of transactions for a non-CP system is compared with that of the CP system. Extra workload or overhead is evaluated as the increase in average response time, and is computed by subtracting the average response time of the non-CP system from that of the CP system.

6.3. Experimental Results

6.3.1. Transaction Type

Extra workload in the CP system is mainly introduced by processing ACPTs and maintaining updates. Therefore, we can expect that the workload consisting mostly of read-only transactions will cause little overhead, whereas the workload consisting mostly of update transactions will cause much higher overhead. To emphasize the difference, we conducted experiments on two extreme cases: workload with read-only transactions only and one with update transactions only. Each transaction requires five data objects for read or write access. The results confirm our prediction as shown in Fig. 1. Read-only transactions cause negligible overhead regardless of the workload presented in the system. The overhead is only from administration of CP process. Update transactions cause linearly increasing overhead as the number of transactions increases. Obviously, it is due to increased data contention and increased number of ACPTs and their updates. In order to evaluate the maximum impact of the proposed CP algorithm, we conducted the rest of the experiments for the workload consisting of update transactions only.

6.3.2. Effects of Data Distribution

The purpose of this experiment is to examine the overhead of the checkpointing algorithm in distributed environments. In a distributed database system, each site is equipped with its own processor. In addition, each site has its own stable storage which contains a portion of the database. Data objects are not replicated; each data object is stored at only one site. Transactions make data access requests that are either local or remote. Local access requests are processed by accessing data objects stored at the site. Remote requests are handled by sending a request message to the data manager of the site that stores the

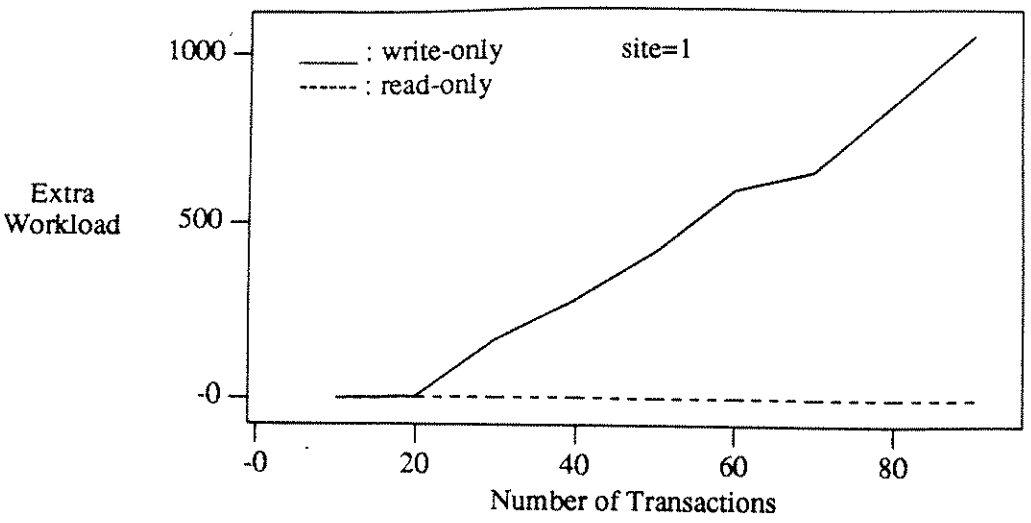

Fig. 1. Overhead of update transactions.

data object.

In a distributed environment, one of the key factors that affects performance is the communication delay. This delay reflects the cost in terms of response time for the communication that must take place between two sites. In our model the system is fully connected and the inter-site communication delay is 5 times longer than the delay involved in inter-process communication within the same site.

The experiments investigate the overhead of the CP algorithm in distributed environments with 2 and 3 sites. In each case transactions access 5 data objects. The transaction mix consists of 100% update transactions. Figure 2 shows the extra workload of the system in a distributed environments with 1, 2 and 3 sites respectively.

Our first observation is that the extra workload in distributed database systems is generally much lower than that in the centralized case. This is due to the fact that in distributed systems we increase the computational power by providing additional processors. The communication delay introduced as a result of distributing data objects across the network is offset by the increased computational power. With only one site, as shown in the slope of the curve, there is a direct relationship between the number of transactions and the extra workload. The slope for two sites is not as sharp, indicating that the effect of

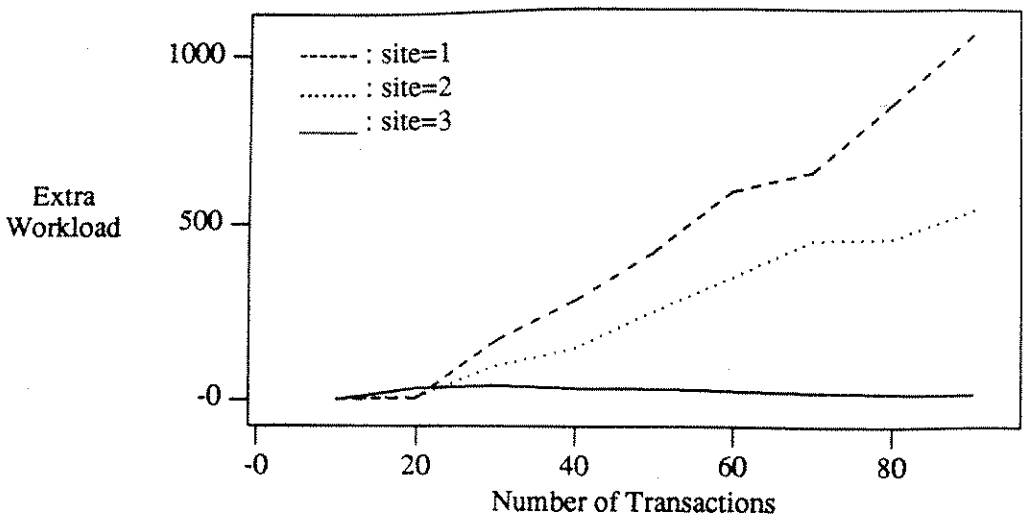

Fig. 2. Overhead in distributed environments.

increasing the number of transactions on the extra workload is more subtle. This is more clear for the system of three sites in which the number of transactions does not seem to affect the extra workload.

In our experiments data objects accessed by transactions are uniformly distributed across the entire database. This strategy causes the workload to be divided and shared equally by processors at different sites. Therefore, a significant increase in the workload was not experienced by each processor as we increased the number of transactions. If, however, the access pattern is biased toward a certain subset of the database, increasing the number of transactions does not uniformly increase the workload of all processors. In such cases, a small number of processors which are located at the sites where the "hot" areas of the database reside perform the majority of the work. Therefore, a large subset of the processors wait idlely, while a selected few process the remote requests of idle transactions. Consequently, increasing the number of transactions would increase the workload of only a small set of the processors, and therefore we can expect to see a more direct relationship between the number of transactions and the average response time.

6.3.3. CP Timing Choice

The choice of the time to perform local checkpointing is another factor that affects the extra workload of the checkpointing algorithm. A single-site database system does not need CP timing choice; it needs to be considered only in distributed environments. We conducted experiments for two different CP timing choices on the system consisting of two sites and that of three sites. The results are shown in Figures 3 and 4.

Our first observation is that the NoWait choice requires less extra workload than the WaitAll choice in the two-site system. As the number of sites increases, WaitAll choice has very similar performance as the NoWait choice. This result can be explained by the nature of these two choices.

The NoWait choice is that local CP is performed asynchronously: after completion of its BCPTs, each site can initiate its local CP without waiting for others. Write requests of BCPTs from other sites will be rejected to maintain database consistency. That leads to the abort of remote BCPTs. The NoWait choice prevents one site from waiting indefinitely for other sites if there are long-lived BCPTs at other sites. This choice, however, may lead to increased cost for transaction processing since half-way done BCPTs at other sites may be forced to abort. Such an action wastes resources and work already done and

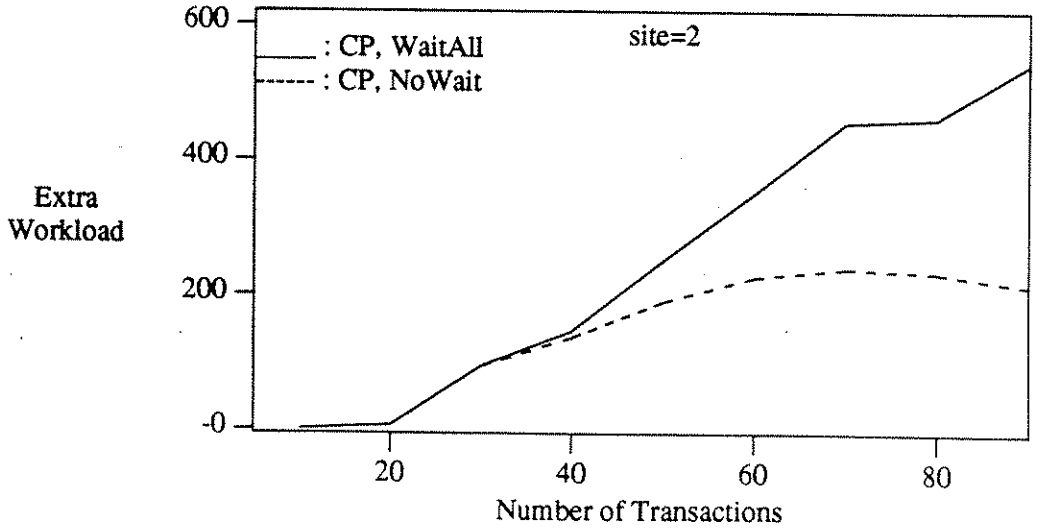

Fig. 3. Overhead with different timing choice: two site case.

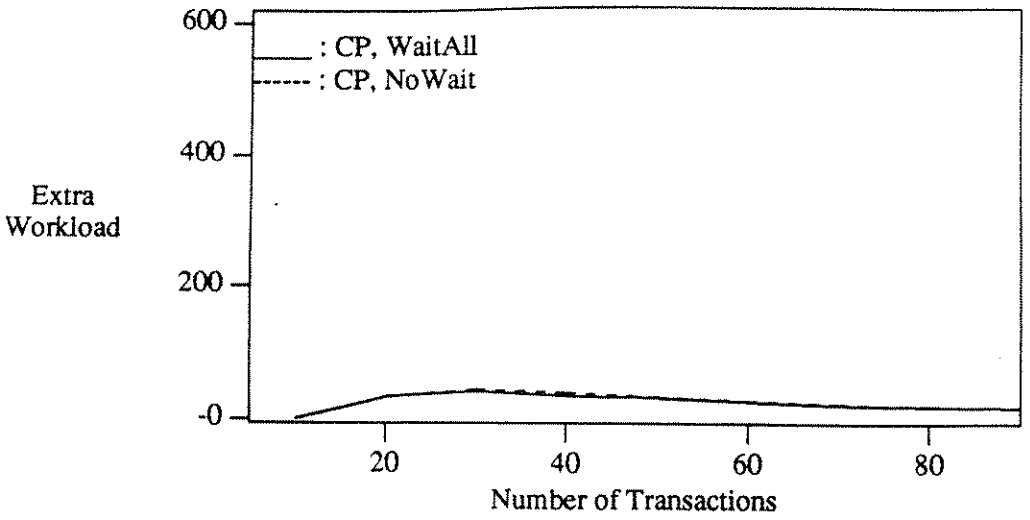

Fig. 4. Overhead with different timing choice: three site case.

results in an increased response time due to abort.

On the other hand, since the WaitAll choice performs local CP synchronously, each site will wait for active BCPTs in the system until they are either committed or aborted. Because none of the sites will start local CP until there is no active BCPT, there cannot be a situation where BCPT is forced to abort due to a write request to a site which is already in its local CP mode as in the NoWait choice. This saves the work performed for the BCPT and reduces the probability of abort. The trade-off here is the risk of long wait before a local CP can be started. Transactions that are submitted during this waiting period will be ACPTs and consequently will increase average response time. Therefore the WaitAll choice would work better than NoWait if the BCPTs of each site would finish approximately at the same time.

There are two types of long-lived transactions. The first type is a transaction that updates much more data objects than others, and hence takes a long time to complete. The second type is a forced long-lived transaction due to high workload in the system and consequently increased resource competition and data object conflicts. In our experiment, transaction size is set to five, so there are no natural long-lived BCPTs, only forced long-lived ones.

A two-site system provides less processing power and I/O capability than a three-site system. Therefore, when workload gets high in a two-site system, BCPTs take long time to complete. Such BCPTs can be considered as forced long-lived BCPTs. Since the NoWait choice allows long-lived BCPTs to be aborted easily, the benefit of avoiding long wait surpasses the overhead of aborts. On the other hand, WaitAll choice may cause serious long-wait situations. With a two-site system, the NoWait choice performs better than its WaitAll choice counterpart. With three sites, the system can provide adequate processing power and hence, the probability of producing forced long-lived BCPTs are smaller. Without long-lived BCPTs in the system, NoWait and WaitAll choices perform similarly with WaitAll choice slightly better. This is because WaitAll choice does not suffer the cost for BCPT aborts; whereas NoWait choice may have the overhead for BCPT aborts if each site does not complete its BCPTs at the same time.

6.3.4. Disk I/O

The cost of I/O operation is a dominating factor in the extra workload of the checkpointing algorithm. Both local checkpointing and reflecting updates take I/O time, and hence, I/O speed is critical for system performance. In this experiment, we investigated the sensitivity of extra workload on different I/O speed. Extra workload with two different I/O time (10 msec and 35 msec) has been measured. I/O time of 10 msec was the default value that has been used in all other experiments. The results of the experiment are shown in Fig. 5.

As shown in the graph, the system with I/O time of 35 msec results in higher extra workload as the number of transactions increases. The reason for this higher extra workload is basically due to longer CP period. Although the system provides the same amount of processing power to both cases, the longer disk I/O time implies the longer CP period which will include more ACPTs and block more transactions when ACPT updates are to be reflected. The effects of such delays are accumulated, and as a result, average transaction response time as well as extra workload will increase.

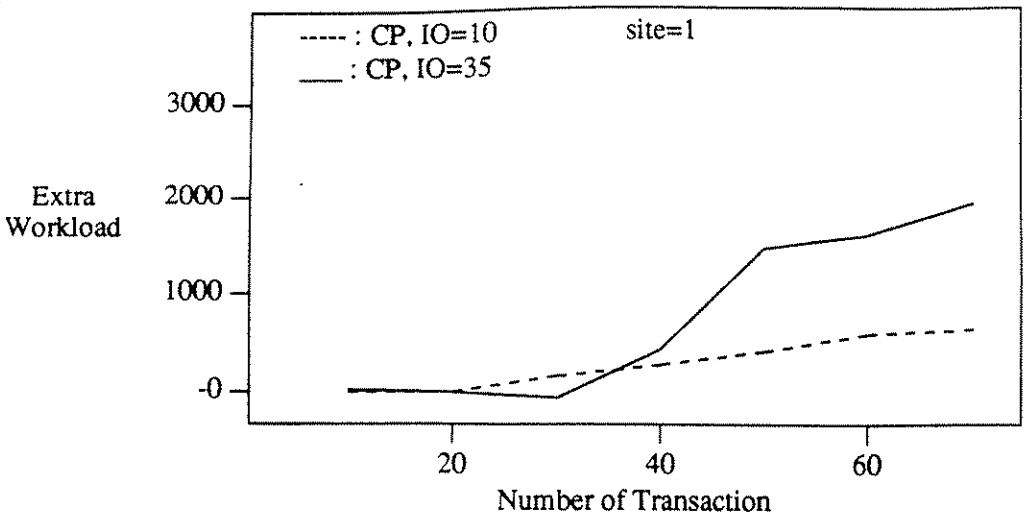

Fig. 5. Overhead sensitivity for I/O time.

6.4. Analysis of Results

Throughout our experiments, we focused on the impact of execution of a single checkpoint. The results of the experiments indicate that extra workload introduced by executing the checkpointing algorithm is proportional to the workload of the system. As the workload of the system increases, extra workload also increases due to the increased resource competition and data object conflicts, and increased number of transaction aborts. In general, any factor which increases system workload will also increase extra workload. Factors in our experiments such as workload of only update transactions, increased number of transactions submitted to the system, and slow disk I/O, increase the workload and hence increase the extra workload. On the other hand, factors which may increase processing power of the system result in reduced extra work load. Degree of distribution is one such factor. We also observed that NoWait choice for starting local checkpoint works better for heavily loaded systems, whereas WaitAll choice works better for lightly loaded systems.

An important observation is that although extra workload increases as the system workload increases, it increases linearly, instead of exponentially, even with the worst situation in our experiments. This observation implies that the proposed checkpointing algorithm is very practical under the parameter

setting of our experiments. Note that we have conducted the checkpointing experiments on the environment when average response time of its non-CP counterpart does not go unacceptably high, i.e. when the non-CP system operates in its practical working situation. We believe that this is a reasonable limitation, because we are looking for a practical limit for a distributed database system with the proposed checkpointing algorithm.

7. Concluding Remarks

The goal of checkpointing in database management systems is to save database states on a separate secure device so that the database can be recovered when errors and failures occur. Since checkpointing is performed during normal operation of the system, it is highly desirable that user transactions can be executed in the system concurrently with the checkpointing process. The property of non-interference is even more desirable in distributed environments. However, if the overhead of a checkpointing algorithm is unacceptably high, the algorithm should be abandoned in spite of its desirable properties. The practicality of non-interfering checkpointing, therefore, depends partially on the amount of extra workload incurred by the checkpointing algorithm.

In this paper, we have presented a checkpointing algorithm which is non-interfering and which efficiently generates globally consistent checkpoints. We have also presented results of a series of experiments conducted using the database prototyping environment to study the practicality of our algorithm. As shown in the results of the experiments, the proposed checkpointing algorithm is practical in realistic distributed environments. Even though extra workload increases as workload of the system increases, its increasing rate is not exponential, when its underneath non-CP counterpart is within its boundary for reasonable operation. In our experiments, we concentrated on the maximum impact of a single checkpoint, and increased workload by increasing the number of transactions submitted to the system. This method is sufficient for the goal of examining whether the checkpointing algorithm is practical under a reasonable operational environment.

There are other performance issues associated with non-interfering checkpointing algorithm implementation. For example, the analytic study of non-interfering checkpointing algorithms provides an interesting observation that the use of a non-interfering checkpointing algorithm could reduce the average response time of transactions over intrusive checkpointing algorithms [22]. The extra workload of a non-interfering checkpointing algorithm incurs mainly due to the processing of ACPTs and maintaining their updates. However, the number of ACPTs may also be used as the metrics for the improved performance comparing with other intrusive checkpointing algorithms. Since intrusive checkpointing algorithms will block all the ACPTs to wait for BCPTs to complete for achieving a transaction-consistent state, the average response time for those blocked ACPTs should be high. This will result in the increased average response time of the system. Since the number of ACPTs running in a non-interfering checkpointing system indicates the number of ACPTs blocked in an intrusive checkpointing system, that number may be proportional to the improved performance over intrusive intrusive checkpointing algorithms. Is it necessarily true? We are investigating this and other related issues using the prototyping environment.

REFERENCES

[1] Attar, R., Bernstein, P. A. and Goodman, N., Site Initialization, Recovery, and Backup in a Distributed Database System, IEEE Trans. on Software Engineering, November 1984, pp 645-650.

[2] Bernstein, P., Goodman N., Concurrency Control in Distributed Database Systems, ACM Computing Surveys, June 1981, pp 185-222.

[3] Bernstein, P., Goodman, N., An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases, ACM Trans. on Database Systems, Dec. 1984, pp 596-615.

[4] Chandy, K. M., Lamport, L., Distributed Snapshots: Determining Global States of Distributed Systems, ACM Trans. on Computer Systems, February 1985, pp 63-75.

[5] Dadam, P. and Schlageter, G., Recovery in Distributed Databases Based on Non-synchronized Local Checkpoints, Information Processing 80, North-Holland Publishing Company, Amsterdam, 1980, pp 457-462.

[6] Eswaran, K. P. et al, The Notion of Consistency and Predicate Locks in a Database System, Commun. of ACM, Nov. 1976, pp 624-633.

[7] Fischer, M. J., Griffeth, N. D. and Lynch, N. A., Global States of a Distributed System, IEEE Trans. on Software Engineering, May 1982, pp 198-202.

[8] Gelenbe, E., On the Optimum Checkpoint Interval, JACM, April 1979, pp 259-270.

[9] Hammer, M. and Shipman, D., Reliability Mechanisms for SDD-1: A System for Distributed Databases, ACM Trans. on Database Systems, December 1980, pp 431-466.

[10] Jouve, M., Reliability Aspects in a Distributed Database Management System, Proc. of AICA, 1977, pp 199-209.

[11] Kuss, H., On Totally Ordering Checkpoints in Distributed Databases, Proc. ACM SIGMOD, 1982, pp 293-302.

[12] Lamport, L., Time, Clocks and Ordering of Events in Distributed Systems, Commun. ACM, July 1978, pp 558-565.

[13] McDermid, J., Checkpointing and Error Recovery in Distributed Systems, Proc. 2nd International Conference on Distributed Computing Systems, April 1981, pp 271-282.

[14] Mohan, C., Strong, R., and Finkelstein, S., Method for Distributed Transaction Commit and Recovery Using Byzantine Agreement Within Clusters of Processors, Proc. 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing, August 1983.

[15] Ricart, G. and Agrawala, A. K., An Optimal Algorithm for Mutual Exclusion in Computer Networks, Commun. of ACM, Jan. 1981, pp 9-17.

[16] Ries, D., The Effect of Concurrency Control on The Performance of A Distributed Data Management System, 4th Berkeley Conference on Distributed Data Management and Computer Networks, Aug. 1979, pp 221-234.

[17] Schlageter, G. and Dadam, P., Reconstruction of Consistent Global States in Distributed Databases, International Symposium on Distributed Databases, North-Holland Publishing Company, INRIA, 1980, pp 191-200.

[18] Shin, K. G., Lin, T.-H., Lee, Y.-H., Optimal Checkpointing of Real-Time Tasks, 5th Symposium on Reliability in Distributed Software and Database Systems, January 1986, pp 151-158.

[19] Skeen, D., Nonblocking Commit Protocols, Proc. ACM SIGMOD International Conference on Management of Data, 1981, pp 133-142.

[20] Son, S. H., A Resilient Replication Method in Distributed Database Systems, *IEEE INFOCOM '89*, Ottawa, Canada, April 1989, pp 363-372.

[21] Son, S. H. and Y. Kim, A Software Prototyping Environment and Its Use in Developing a Multiversion Distributed Database System, *18th International Conference on Parallel Processing*, St. Charles, Illinois, August 1989, Vol. 2. pp 81-88.

[22] Son, S. H., An Algorithm for Non-Interfering Checkpoints and its Practicality in Distributed Database Systems, *Information Systems*, Vol. 14, No. 4, December 1989.

[23] Son, S. H., A Message-Based Approach to Distributed Database Prototyping, *Fifth IEEE Workshop on Real-Time Software and Operating Systems*, Washington, DC, May 1988, pp 71-74.

[24] Cellary, W., E. Gelenbe, and T. Morzy, Concurrency Control in Distributed database Systems, North-Holland, 1988.

[25] Ceri, S. and G. Pellagatti, Distributed Databases: Principles and Systems, McGraw-Hill, 1984.