RELIABILITY MECHANISMS FOR ADAMS

Sang H. Son
John L. Pfaltz

IPC-TR-88-002
March 20, 1988

Institute for Parallel Computation
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 22903

To appear Proceedings of third conference on Hypercube concurrent Computers and Applications Pasadena, California (January 1988)

This research was supported in part by JPL under contract #957721.

Reliability Mechanisms for ADAMS

Sang H. Son
John L. Pfaltz

Institute for Parallel Computation and Department of Computer Science
University of Virginia
Charlottesville, VA 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. This paper presents a non-interfering checkpointing mechanism being developed for ADAMS. Instead of waiting for a consistent state to occur, our checkpointing approach constructs a state that would result by completing the transactions that are in progress when the global checkpoint begins. The checkpointing algorithm is executed concurrently with transaction activity while constructing a transaction-consistent checkpoint on disk, without requiring the database quiesce. 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. Two main properties of this checkpointing algorithm are global consistency and reduced interference, both of which are crucial for achieving high availability.

1. Introduction

ADAMS (Advanced DAta Management System) is the name of the database system being developed for hypercube-based parallel computer systems by the database group of the Institute for Parallel Computation at the University of Virginia. Currently, ADAMS is being implemented on a disk farm for the NCube 10 system. In this paper, we concentrate on a presentation of an efficient and robust reliability mechanism developed for ADAMS, which is essential to a parallel database system in order to satisfy the requirement of high reliability.

The need for having recovery mechanisms in database systems is well acknowledged. 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 nodes may destroy the consistency of the database. In order to cope with those errors and failures, database systems provide recovery mechanisms, and checkpointing is a technique frequently used in such recovery mechanisms.

The goal of checkpointing in database management systems is to 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 small, too much time and resources are spent in checkpointing; if these intervals are large, too much time is spent in recovery.

Since checkpointing is performed during normal operation of the system, the interference of checkpointing with transaction processing must be kept to a minimum. It is highly desirable that users are allowed to submit transactions while the checkpointing is in progress, and the transactions are executed in the system concurrently with the checkpointing process. In parallel database systems this requirement of non-interference makes the checkpointing more complicated because we need to consider the coordination among the autonomous nodes of the system. A quick recovery from failures is also desirable to many applications of parallel databases. 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. In a parallel environment, this desirable property of global consistency increases the complexity of the checkpointing algorithm and the workload of the system.

Recently, the possibility of having a checkpointing mechanism that does not interfere with the

transaction processing, and yet achieves the global consistency of the checkpoints, has been studied [2, 5, 19]. The motivation of non-interfering checkpointing is to improve the system availability, that is, the system must be able to execute user transactions concurrently with the checkpointing process. The basic principle behind non-interfering checkpointing mechanisms is to create a diverged computation of the system such that the checkpointing process can view a consistent state that could result by running to completion all of the transactions that are in progress when the checkpoint begins, instead of viewing a consistent state that actually occurs by suspending further transaction execution.

In the rest of this paper, we briefly discuss our approach for checkpointing which efficiently generates globally consistent database state, and its associated recovery logic for ADAMS. Given our space limitations, our objective is to intuitively explain this approach and not to provide details. The details are given in separate papers [20, 21].

2. Model of Computation

2.1. Data Objects and Transactions

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 parallel database system implemented on a computing system where several autonomous computing elements (called *nodes)* are connected via a communication network. The set of data objects in a parallel database system is partitioned among its nodes. 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* [4].

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. We assume that the database system runs a correct transaction control mechanism (e.g., atomic commit algorithm[17] and concurrency control algorithm[18]), and hence assures the atomicity and the serializability of transactions.

Each transaction has a time-stamp associated with it [10]. A time-stamp is a number that is assigned to a transaction when initiated and is kept by the transaction. Two important properties of time-stamps are (1) no two transactions have the same time-stamp, and (2) only a finite number of transactions can have a time-stamp less than that of a given transaction.

2.2. Failure Assumptions

A parallel database system can fail in many different ways, and it is almost impossible to make an algorithm which can tolerate all possible failures. In general, failures in parallel database systems can be classified as failures of *omission* or *commission* depending on whether some action required by the system specification was not taken or some action not specified was taken. The simplest failures of omission are *simple crashes* in which a node simply stops running when it fails. The hardest failures are *malicious runs* in which a node continues to run, but performs incorrect actions. Most real failures lie between these two extremes.

In this paper, we do not consider failures of commission such as the "malicious runs" type of failure. When a node fails, it simply stops running (fail-stop). When the failed node recovers, the fact that it has failed is recognized, and a recovery procedure is initiated. We assume that node failures are detectable by other nodes. This can be achieved either by network protocols or by high-level time-out mechanisms in the application layer. We also assume that network partitioning never occurs. This assumption is reasonable for most local area networks and some long-haul networks.

3. Non-Interfering Checkpoint Creation

3.1. Motivation of Non-interference

The motivation of having a checkpointing scheme which does not interfere with transaction processing is well explained in [2] 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 parallel database system, each node 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 transactions 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 parallel environment. For the separation of transactions in a parallel environment, a special time-stamp which is globally agreed upon by the participating nodes 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 nodes.

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.

3.2. 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 node which is manipulated by the clock rules of Lamport[10].

(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 node.

Our basic checkpointing algorithm, called CP1, works as follows:

(1) The checkpoint coordinator broadcasts a Checkpoint Request Message with a time-stamp $LC sub CC$. The local checkpoint number of the coordinator is set to $LC sub CC$, and the coordinator sets the Boolean variable CONVERT to false:

$CONVERT sub CC$:= false
$LCPN sub CC$:= $LC sub CC$

All the transactions at the coordinator node with the time-stamps not greater than $LCPN sub CC$ are marked as BCPT.

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

$LC sub m$:= max($LC sub CC$ + 1, $LC sub m$)
$LCPN sub m$:= $LC sub m$

The checkpoint subordinate of node m replies to the coordinator with $LCPN sub m$, and sets the Boolean variable CONVERT to false:

$CONVERT sub m$:= FALSE

and marks all the transactions at the node m with the time-stamps not greater than $LCPN sub m$ as BCPT.

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

GCPN := max($LCPN sub n$) n = 1,..,N

(4) For all nodes, 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 node 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 < time-stamp(T) $<=$ 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 node. 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 participant receives a Transaction Initiation Message from the coordinator, it 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) $<=$ GCPN, then a reject message is returned, and the transaction is aborted. Therefore in order to execute step 6, each checkpointing process only needs to check active BCPT at its own node, and yet the consistency of the checkpoint can be achieved.

3.3. 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 node.

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 $T sub i$ is initiated, then $T sub i$ is never blocked by subsequent transactions that are younger than $T sub i$. The number of transactions that may block the execution of $T sub i$ is finite because only a finite number of transactions can be older than $T sub i$. Among older transactions which may block $T sub i$, 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, $T sub i$ 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 [18] can ensure the termination of transactions in a finite time.

4. Consistency of Checkpoints

In this section we give an informal proof of the correctness of the algorithm. In addition to proving the consistency of the checkpoints generated by the algorithm in the global mode, we show that the algorithm has another desirable property that each checkpoint contains all the updates of transactions with earlier time-stamps than its GCPN. This property reduces the work required in the actual recovery, which is discussed in Section 5. A longer and more thorough discussion on the correctness of the algorithm is given in [19].

The properties of the algorithm we want to show are

(1) a set of all local checkpoints with the same GCPN represents a consistent database state, and

(2) all the updates of the committed transactions with earlier time-stamps than the GCPN are reflected in the current checkpoint.

Note that only one checkpointing process can be active at a time because the checkpointing coordinator is not allowed to issue another checkpointing request before the termination of the previous one.

A database state is consistent if the set of data objects satisfies the consistency constraints[4]. Since a transaction is the unit of consistency, a database state S is consistent if the following holds:

(1) For each transaction T, S contains all subtransactions of T or it contains none of them.

(2) If T is contained in S, then each predecessor T' of T is also contained in S. (T' is a predecessor of T if it modified the data object which T accessed at some later point in time.)

For a set of local checkpoints to be globally consistent, all the local checkpoints with the same GCPN must be consistent with each other concerning the updates of transactions that are executed before and after the checkpoint. Therefore, to prove that the algorithm satisfies both properties, it is sufficient to show that the updates of a global transaction T are included in $CP sub i$ at each participating node of T, if and only if time-stamp(T)$~<= $ GCPN($CP sub i$). This is enforced by the mechanism to determine the value of the GCPN, and by the conversion of the temporary ACPT into BCPT.

A transaction is said to be *reflected* in data objects if the values of data objects represent the updates made by the transaction. We assume that the database system provides a reliable mechanism for writing into the secondary storage such that a writing operation of a transaction is atomic and always successful when the transaction commits. Because updates of a transaction are reflected in the database only after the transaction has been successfully executed and committed, partial results of transactions cannot be included in checkpoints.

The checkpointing algorithm assures that the sequence of actions are executed in some specific order. At each node, conversion of eligible transactions occurs after the GCPN is known, and local checkpointing cannot start before the Boolean variable CONVERT becomes true. CONVERT is set to false at each node after it determines the LCPN, and it becomes true only after the conversion of all the eligible transactions. Thus, it is not possible for a local checkpoint to save the state of the database in which some of the eligible transactions are not reflected because they remain unconverted.

We can show that a transaction becomes BCPT if and only if its time-stamp is not greater than the current GCPN. This implies that all the eligible BCPT will become BCPT before local checkpointing begins in step 6. Therefore, updates of all BCPT are reflected in the current checkpoint.

>From the atomic property of transactions provided by the transaction control mechanism (e.g. commit protocol in [17]), it can be assured that if a transaction is committed at a participating node then it is committed at all other participating nodes. Therefore if a transaction is committed at one node, and if it satisfies the time-stamp condition above, its updates are reflected in the database and also in the current checkpoint at all the participating nodes.

5. Discussion

The desirable properties of non-interference and global consistency not only make the checkpointing more complicated in parallel database systems, but also 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. In this section we consider practicality of non-interfering checkpointing algorithms, and discuss the robustness and recovery methods associated with the algorithm CP2.

5.1. Practicality of Non-interfering Checkpoints

There are two performance measures that can be used in discussing the practicality of non-interfering checkpointing: extra storage and extra workload required. The extra storage requirement of the algorithm is simply the committed temporary version (CTV) file size, which is a function of the expected number of ACPT of the node, the number of data objects updated by a typical transaction, and the size of the basic unit of information:

CTV file size $mark =~ N sub A times$(number of updates) $times$(size of the data object)

where $N sub A$ is the expected number of ACPT of the node.

The size of the CTV file may become unacceptably large if $N sub A$ or the number of updates becomes very large. Unfortunately, they are determined dynamically from the characteristics of transactions submitted to the database system, and hence cannot be controlled. Since $N sub A$ is proportional to the execution time of the longest BCPT at the node, it would become unacceptably large if a long-lived

transaction is being executed when a checkpoint begins at the node. The only parameter we can change in order to reduce the CTV file size is the granularity of a data object. The size of the CTV file can be minimized if we minimize the size of the data object. By doing so, however, the overhead of normal transaction processing (e.g., locking and unlocking, deadlock detection, etc) will be increased. Also, there is a trade-off between the degree of concurrency and the lock granularity[14]. Therefore the granularity of a data object should be determined carefully by considering all such trade-offs, and we cannot minimize the size of the CTV file by simply minimizing the data object granularity.

There is no extra storage requirement in intrusive checkpointing mechanisms[1, 8, 15]. However this property is balanced by the cases in which the system must block ACPT or abort half-way done global transactions because of the checkpointing process.

The extra workload imposed by the algorithm mainly consists of the workload for (1) determining the GCPN, (2) committing ACPT (move data objects to the CTV file), (3) reflecting the CTV file (move 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. As in extra storage estimation, they are determined by the number of ACPT and the number of updates. Therefore, as far as the values of these variables can be maintained within a certain threshold level, non-interfering checkpointing would not severely degrade the performance of the system. A detailed discussion on the practicality of non-interfering checkpointing is given in [19].

5.2. Node Failures

So far, we assumed that no failure occurs during checkpointing. This assumption can be justified if the probability of failures during a single checkpoint is extremely small. However, it is not always the case, and we now consider the method to make the algorithm resilient to failures.

During the global mode of operation, the algorithm CP2 is insensitive to failures of subordinates. If a subordinate fails before the broadcast of a Checkpoint Request Message, it is excluded from the next checkpoint. If a subordinate does not send its LCPN to the coordinator, it is excluded from the current checkpoint. When the node recovers from the failure, the recovery manager of the node must find out the GCPN of the latest checkpoint. After receiving information of transactions which must be executed for recovery, the recovery manager brings the database up to date by executing all the transactions whose time-stamps are not greater than the latest GCPN. Other transactions are executed after the state of the data objects at the node is saved by the checkpointing process.

An atomic commit protocol guarantees that a transaction is aborted if any participant fails before it sends a Precommit message to the coordinator. Therefore, node failures during the execution of the algorithm cannot affect the consistency of checkpoints because each checkpoint reflects only the updates of committed BCPT.

In the local mode of operation, a failure of a participant prevents the coordinator from receiving OK from all the participants, or prevents the participants from receiving the decision message from the coordinator. However, because a transaction is aborted by an atomic commit protocol, it is not necessary to make checkpointing robust to failures of participants.

The algorithm is, however, sensitive to failures of the coordinator. In particular, if the coordinator crashes during the first phase of the global mode of operation (i.e., before the GCPN message is sent to subordinates), every transaction becomes ACPT, requiring too much storage for committed temporary versions.

One possible solution to this involves the use of a number of *backup* processes; these are processes that can assume responsibility for completing the coordinator's activity in the event of its failure. These backup processes are in fact checkpointing subordinates. If the coordinator fails before it broadcasts the GCPN message, one of the backups takes the control. A similar mechanism is used in SDD-1 [7] for reliable commitment of transactions. Proper coordination among the backup processes is crucial here. In the event of the failure of the coordinator, one, and only one backup process has to assume the control. The algorithm for accomplishing this assumes an ordering among the backup processes, designated in order as $p sub 1$, $p sub 2$, ..., $p sub n$. Process $p sub {k-1}$ is referred to as the *predecessor* of process $p sub k$ (for k > 0), and the coordinator is taken as the predecessor of process $p sub 1$.

We assume that the communication service enables processes to be informed when a given node achieves a specified status (simply UP or DOWN in this case). Initially, each of the backup processes checks the failure of its predecessor. Then the following rules are used.

(1) If the predecessor is found to be down, then the process begins to check the predecessor of the failed process.

(2) If the coordinator is found to be down, the first backup process assumes the control of checkpointing.

(3) If a backup process recovers, it ceases to be a part of the current checkpointing.

(4) After each checkpoint, the list of backup processes is adjusted by including all the UP nodes.

These rules guarantee that at most one process, either the coordinator or one of the backup processes, will be in control at any given time. Thus a checkpointing will terminate in a finite time once it begins.

5.3. Recovery

The recovery from node crashes is called the *node recovery*. The complexity of the node recovery varies in parallel database systems according to the failure situation[15]. If the crashed node has no replicated data objects and if all the recovery information is available at the crashed node, local recovery is enough. Global recovery is necessary because of failures which require the global database to be restored to some earlier consistent state. For instance, if the transaction log is partially destroyed at the crashed node, local recovery cannot be executed to completion.

When a global recovery is required, the database system has two alternatives: a *fast* recovery and a *complete* recovery. A fast recovery is a simple restoration of the latest global checkpoint. Since each checkpoint generated by the algorithm is globally consistent, the restored state of the database is assured to be consistent. However, all the transactions committed during the time interval from the latest checkpoint until the time of crash would be lost. A complete recovery is performed to restore as many transactions that can be redone as possible. The trade-offs between the two recovery methods are the recovery time and the number of transactions saved by the recovery.

Quick recovery from failures is critical for some applications of parallel database systems which require high availability (e.g., ballistic missile defense or air traffic control). For those applications, the fate of the mission, or even the lives of human beings, may depend on the correct values of the data and the accessibility to it. Availability of a consistent state is of primary concern for them, not the most up-to-date consistent state. If a simple restoration of the latest checkpoint could bring the database to a consistent state, it may not be worthwhile to spend time in recovery by executing a complete recovery to recover some of the transactions.

For the applications in which each committed transaction is so important that the most up-to-date consistent state of the database is highly desirable, or if the checkpoint intervals are large such that a lot of transactions may be lost by the fast recovery, a complete recovery is appropriate to use. The cost of a complete recovery is the increased recovery time which reduces the availability of the database. Searching through the transaction log is necessary for a complete recovery. The second property of the algorithm (i.e., each checkpoint reflects all the updates of transactions with earlier time-stamps than its GCPN) is useful in reducing the amount of searching because the set of transactions whose updates must be redone can be determined by the simple comparison of the time-stamps of transactions with the GCPN of the checkpoint. Complete recovery mechanisms based on the special time-stamp of checkpoints (e.g., GCPN) have been proposed in [9, 20].

6. Concluding Remarks

During normal operation of the database system, checkpointing is performed to save information necessary for recovery from a failure. For better recoverability and availability of parallel database systems, checkpointing must be able to generate a globally consistent database state, without interfering with transaction processing. Autonomy of processing nodes in parallel database systems makes the checkpointing more complicated than in single processor database systems. In this paper, a new checkpointing algorithm for parallel database systems is presented and discussed. The correctness of the algorithm is shown, and the practicality of the algorithm and recovery procedures associated with it are discussed.

A non-interfering checkpointing is desirable in many applications, and it has been shown that it can be a viable solution if the extra workload and storage requirement remain in an acceptable range. Two important parameters in making a non-interfering checkpointing practical are the mean number of transaction arrivals and the average number of updates of a transaction. As far as they remain below certain threshold values, the overhead of non-interfering checkpointing can be justified. The properties of global consistency and non-interference of checkpointing result in some overhead on the one hand, and increase the system availability on the other hand. For the applications where the ability of continuous processing of transactions is so critical that the blocking of transaction processing for checkpointing is not feasible, we believe that the checkpointing algorithm presented in this paper provides a practical solution to the problem of constructing globally consistent states in parallel database systems.

ACKNOWLEDGEMENTS

This work was supported in part by the Office of Naval Research under contract no. N00014-86-K-0245,

and by the Jet Propulsion Laboratory of the California Institute of Technology under contract no. 957721 to the University of Virginia.

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

[3] 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.

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

[5] 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.

[6] Gelenbe, E., On the Optimum Checkpoint Interval, *Journal of ACM*, April 1979, pp 259-270.

[7] 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.

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

[9] Kuss, H., On Totally Ordering Checkpoints in Distributed Databases, *ACM SIGMOD International Conference on Management of Data*, 1982, pp 293-302.

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

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

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] Son, S. H., On Multiversion Replication Control in Distributed Systems, *Computer Systems Science and Engineering*, Vol. 2, No. 2, April 1987, pp 76-84.

[19] Son, S. H. and Agrawala, A., Practicality of Non-Interfering Checkpoints in Distributed Database Systems, *IEEE Real-Time Systems Symposium*, New Orleans, Louisiana, December 1986, pp 234-241.

[20] Son, S. H. and Agrawala, A., An Algorithm for Database Reconstruction in Distributed Environments, *6th International Conference on Distributed Computing Systems*, Cambridge, Massachusetts, May 1986, pp 532-539.

[21] Son, S. H., "An Adaptive Checkpointing Scheme for Distributed Databases with Mixed Types of Transactions," *Fourth International Conference on Data Engineering*, Los Angeles, February 1988, pp 528-535.