Coordinated Checkpointing-Rollback Error Recovery for Distributed Shared Memory Multicomputers

G. Janakiraman and Yuval Tamir
Computer Science Department
UCLA
Los Angeles, California 90024

Abstract

Most recovery schemes that have been proposed for Distributed Shared Memory (DSM) systems require unnecessarily high checkpointing frequency and checkpoint traffic, which are sensitive to the frequency of interprocess communication in the applications. For message-passing systems, low overhead error recovery based on coordinated checkpointing allows the frequency of checkpointing to be determined only by the reliability requirements of the application. Efficient adaptation of this approach to DSM multicomputers is complicated by the absence of explicit messages in DSM systems, the presence of a shared and partially replicated address space, and the presence of a distributed coherency directory. We present solutions to these issues, and propose an error recovery scheme based on coordinated checkpointing and rollback for DSM multicomputers. Our performance evaluation based on trace-driven simulations indicates that this scheme incurs less checkpoint traffic than recovery schemes previously proposed for DSM systems.

I. Introduction

DSM multicomputers [2, 3, 11, 12] provide a single shared address space on hardware where memory is physically distributed, using cache coherency protocols [3, 11, 13]. The reliability requirements of large DSM multicomputers, consisting of thousands of VLSI chips, can only be met by using fault tolerance techniques.

Application-transparent error recovery in message-passing systems can be achieved using *message logging* or based on *coordinated checkpointing*. These schemes do not require checkpointing at every interprocess communication and yet are not susceptible to the domino effect [15]. *Message logging* [18, 8] involves the logging of interprocess messages as well as periodic process checkpoints. The message logs are used to restore a recovering process to a state *consistent* [15, 4] with the rest of the system. *Coordinated checkpointing* [1, 20, 10, 21] is based on checkpointing ‘‘together’’ consistent states of all the processes that have interacted since their last checkpoint. Recovery requires rolling back the entire interacting set of processes containing the failed process. In multicomputers, message logging requires significantly higher overhead than coordinated checkpointing during normal operation due to the need to log all communication as well as to generate and process for each message information for tracking dependencies [5].

Recovery schemes for DSM multicomputers must address the same issues as messages-passing multicomputers. Some proposed schemes use the simple but costly approach of checkpointing prior to every ‘‘communication’’ with another process [24, 23, 19] (see Section V). Based on message logging, it has also been proposed to use independent process checkpoints together with logs of remote page accesses [16]. As mentioned earlier, these schemes suffer from unacceptably high overhead during normal operation.

This paper presents an error recovery scheme for DSM multicomputers based on coordinated checkpointing. Coordinated checkpointing schemes minimize the overhead during normal operation since they require little or no special recovery-related actions for interprocess communication. There is no need to log messages or process complex dependency-tracking information. Recovery may be more expensive than with message logging. However, the overall cost (overhead) of checkpointing will usually be lower. Far fewer checkpoints may be taken since the frequency of checkpointing can be tuned to the *needs* of the application instead of being determined by the frequency of interprocess communication. Thus, for communication-intensive applications, recovery schemes based on coordinated checkpointing are preferable over other schemes [20, 5].

In DSM multicomputers, memory accesses that result in a miss on the local node can require multiple message exchanges between the nodes to transmit the data and update the directories [3, 11, 12]. It is possible to implement coordinated checkpointing and rollback on DSM multicomputers by treating each one of these

messages as an ‘‘interaction’’ between processes. With this approach, since the directories modified by these ‘‘messages’’ are accessed by all tasks, processes of different tasks (which do not communicate) become part of the same *interacting set* and must be checkpointed and rolled back together. Hence, this naive approach to implementing coordinated checkpointing would result in all the processes in the system and all the cache directories quickly becoming part of one interacting set. Thus, virtually all checkpoints and rollbacks would have to be of the entire system.

Our checkpointing and rollback scheme takes into account the fact that not all messages exchanged on behalf of a process need to be considered in determining the interacting set of processes that are checkpointed and rolled back together. The identification of ‘‘real’’ process interactions is integrated with the coherency protocol. The concept of unique ‘‘ownership’’ [9] in the coherency mechanism is exploited to checkpoint and recover the replicated address space. The state of the distributed coherency directory is not checkpointed. During recovery, the coherency directory is transformed to accurately reflect the modified state of the physical memories. We have evaluated the performance of our coordinated checkpointing scheme using trace-driven simulations. Our evaluation shows that coordinated checkpointing presents a low overhead alternative to previously proposed DSM error recovery schemes.

The following section is a brief summary of coordinated checkpointing for message-passing systems. Section III describes our system model and assumptions. Our checkpointing and rollback scheme is presented in Section IV. Section V is an overview of prior research on DSM system recovery. Performance evaluation based on trace-driven simulations is presented in Section VI, where we compare our scheme to previously proposed schemes for DSM recovery.

II. Coordinated Checkpointing and Rollback

In a system where processes interact with each other, the states of two processes are consistent if they agree on what messages have been exchanged between them. Coordinated checkpointing and rollback schemes require that the most recent checkpointed state of every process is consistent with the most recent checkpointed state of every other process [1, 20, 10, 21]. Hence, the set of all the committed checkpoints in stable storage is *always* a consistent *snapshot* [4] of the entire system.

Coordinated checkpointing and rollback do *not* require all system processes to be checkpointed and rolled back together. Instead, communication between processes is tracked, and the state of *interacting sets* of processes are checkpointed or rolled back in coordination [1, 10, 21]. For each process, the system maintains the set of processes with which it has interacted directly since its last checkpoint. When a process initiates a checkpointing session, this communication information is used to build a *communication tree* consisting of all the processes with which the first process has communicated directly or indirectly since the last checkpoint [21]. The initiator of the session coordinates the consistent checkpoint of all members of the communication tree [1, 10, 21] using a two-phase commit protocol [6]. All participants in the checkpointing session suspend their execution and all messages in transit between them are flushed to their destinations and become part of the destinations’ state. Recovery is similar to the checkpoint session, except that the flushed messages are discarded and the process states are restored from stable storage.

III. System Model and Assumptions

The DSM multicomputer system consists of several nodes interconnected by point-to-point links (Figure 1). Each node in the system includes an application processor, local memory, cache memory, and coprocessors to assist in the coherency management and interprocessor communication. Each node may be time-shared between multiple processes. A set of processes that share their address space form a *task*. The system can execute multiple tasks simultaneously. Unique task identifiers partition the total virtual system space into disjoint task virtual spaces. Processes belonging to different tasks execute in disjoint address spaces and cannot share data.

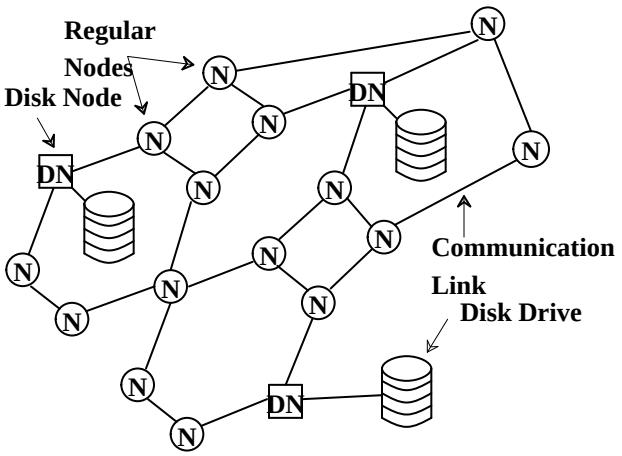

Figure 1: A DSM multicomputer system

DSM is implemented on the system using a distributed directory-based coherence protocol [3, 11]. Ownership [9] is managed at a granularity of *blocks* (tens of bytes). Multiple read-only copies can be distributed

through the system. A node must obtain exclusive ownership of a block prior to modifying it, by invalidating all other copies. Each block of the address space is initially managed by a *default owner*. The default owner keeps track of who is the current owner of the block. A hash function is used to map the default ownership of blocks to the nodes in the system [13].

Each node maintains part of the *distributed directory* which indicates, for each block present locally, the access permission and the identity of the block owner. The block owner maintains, in addition, the location of read-only copies. Each node also maintains a Migrated Block Table (MBT), in which it identifies the current owner of all *its* default owned blocks that have been migrated away from the node. A node that does not know the identity of the block owner can request access through the default owner of the block.

We assume that the underlying network guarantees reliable FIFO delivery of messages between any pair of nodes. Each node on the system is assumed to be self-checking and fail-stop [17]. A subset of the nodes in the system are connected to disks, and checkpoints are saved onto these disks. These disks are assumed to be stable storage; a disk failure will lead to a crash of the system. Upon node failure, all the contents of the node memory and the node directory are considered lost. The identity of the failed node is broadcast to all other nodes in the system.

IV. Coordinated Checkpointing and Rollback for DSM Systems

In order to adapt coordinated checkpointing and rollback to DSM multicomputers, we must examine the unique characteristics of DSM systems: 1) processes communicate by reading from and writing to a shared address space, 2) a single communication *event* at the application level (load or store) may trigger a complex *transaction*, involving multiple internode messages, 3) the address space of each process is distributed among multiple nodes, 4) multiple copies of memory blocks are allowed, and 5) there is a distributed directory which is common to all tasks but is not part of the state of any process.

A. Tracking Interprocess Communication

A key requirement for coordinated checkpointing is the ability to identify *interacting sets* of processes. This is trivial when explicit messages are used for interprocess communication. In a DSM system, processes communicate by reading from and writing to a shared address space. However, not all accesses to the shared address space imply dependency. Including all these accesses would unnecessarily enlarge the interacting set. For example, consider a *load* that is satisfied by accessing a block on a remote node. If the block has not been modified since the last checkpoints of all the processes in the system, this remote access should be ignored when determining the interacting set of processes that must be checkpointed together. Hence, an efficient checkpointing scheme must identify which accesses to the shared address space cause dependencies that must be taken into account in determining the interacting set.

Processes of a task that run on the same node communicate by direct access to locations in local memory. These intra-node interactions can be identified with some architectural support in the local memory access mechanism [24], but the cost of providing this support is likely to be too high for the benefits gained. Hence, we view all processes of a task that run on the same node as having interdependent states. We label the set of processes that belong to the same task and run on the same node a *process-cluster*. Processes of a process-cluster are always checkpointed and rolled back together.

Processes of a task that run on different nodes communicate by read-write sharing memory blocks across the network with the cooperation of the coherency protocol. To identify these interactions, we extend the functionality of the coherency protocol. With each block in local memory there is an associated *dirty-since-checkpoint* (*dsc*) bit in the local directory. The dsc bit indicates whether the block has been modified since the last time it was checkpointed. Every block transfer is tagged with the block's dsc bit. Interprocess communication that imply dependency can be identified by the transfer of a block with the dsc bit set.

Using the dsc bit mechanism, each node maintains for each process-cluster on the node a list of all process-clusters with which there has been *direct* communication since the last checkpoint (similar information is maintained for each process in message-passing systems[1, 10, 21]). The dependency between process clusters on two nodes, P and Q, is established when a block of a the task to which both process clusters belong is transmitted from node Q to node P, with the block's dsc bit set. This dependency is recorded independently in both nodes.

B. Transactions in Progress when Checkpointing or Rollback are Initiated

In order to save a consistent snapshot of an interacting set of processes in a message-passing system, messages in transit between the processes when

checkpointing is initiated must also be saved. This can be done by ‘‘flushing’’ messages to their destination and saving them with the destination process checkpoint [20, 21]. Messages in transit between the processes when rollback is initiated are discarded [20, 21]. In a DSM multicomputer, process interaction takes place through read and write ‘‘transactions’’ (which, in general, are not necessarily atomic). Each transaction involves the exchange of several network messages. When checkpointing or rollback are initiated, transactions may be *pending*, with the processor waiting for a response from one of the other processors participating in the transaction.

The handling of transactions which are pending when checkpointing or rollback are initiated is a critical issue. There are four alternatives to be considered: 1) always abort the transaction, 2) always suspend the transaction, 3) continue the transactions while the checkpointing or recovery session proceeds, or 4) attempt to complete the transaction before proceeding with the checkpointing or recovery session. This is more complicated than flushing messages since it involves multiple nodes and interactions with the coherency management protocols.

When rollback is initiated, pending transactions associated with processes being rolled back are orphans [14] and should be aborted. However, always aborting pending transactions when checkpointing is initiated results in unnecessary performance loss.

It may be possible to always suspend pending transactions when checkpointing is initiated. However, this would add significantly to the complexity of checkpointing since it would require saving the state of partially executed transactions. Depending on how this is done, some of this ‘‘transaction state’’ may involve nodes that would otherwise not participate in the checkpointing session.

For performance reasons, it might be desirable to allow pending transactions to proceed in parallel with a checkpointing session (so that the transactions and checkpointing are not delayed). This results in a situation similar to the handling of dynamic interacting sets in a message-passing system with asynchronous coordinated checkpointing [22]. Specifically, the size of the interacting set may increase while the checkpointing session is already in progress. Whether or not a block transmitted as part of a pending transaction should be part of the checkpoint depends on the status of all the participants in the transaction. The complexity of special handling needed to manage this situation presents sufficient reason to look at alternative approaches.

The approach we adopt is to wait for the completion (successfully or unsuccessfully) of pending transactions before proceeding with checkpointing or recovery. It is necessary to ensure that checkpointing or recovery sessions do not wait endlessly for the transactions to complete. Hence, the coherency protocols must be extended to provide a negative acknowledgment terminating (unsuccessfully) the transaction when service cannot be provided.

We assume that a node failure is broadcast to all other nodes (Section III). Every participant in a transaction must be capable of identifying the situation when the transaction cannot progress due to the failure of another participant. In this case, the node identifying the situation must terminate the transaction. With many coherency protocols [13, 11] the node initiating the transaction is not always aware of the identity of the node servicing the transaction. With these protocols, there is no simple way for the initiator to determine that the transaction cannot progress due to the failure of the node servicing the transaction. For example, a node requesting a read copy of a block may send the request to the block’s default owner which, in turn, forwards the request to the current block owner. In this situation, the initiator of the transaction does not know the identity of the block owner, and, hence, cannot associate the failure of the latter node with the failure of the transaction. At the same time, the default owner is not expecting a reply from the current owner (the reply is sent directly to the requester) and thus will not react to the failure of the block owner. The coherency protocols can be modified to solve this problem. For example, the response from the block owner could be always propagated through the default owner. The default owner will thus be able to alert the requester if it determines that the owner failed before responding to the request.

To summarize, with the above modifications to the coherency protocol, checkpointing and recovery session wait for the termination of all pending transactions. Any inconsistency in the directory state due to the failure of a transaction participant is repaired by the mechanism that recovers the failed participant.

C. Checkpointing

A checkpointing session is initiated by a checkpoint ‘‘timer’’ associated with each process-cluster, or when a process-cluster needs to write a dirty block to stable storage. All process-clusters that have communicated (directly or indirectly) with the initiating process-cluster checkpoint together using a two-phase protocol [6], exactly as in message-passing systems [1, 10, 21]. Since processes of different tasks do not communicate, each checkpointing session is localized to a task.

```
checkpoint → {
    suspend processes in process-cluster;
    await termination of pending transactions;
    reject all transaction requests;
    begin saving local process states and dirty private blocks;
    begin saving shared locally-owned blocks with dsc = 1;
    propagate checkpoint messages
    await acknowledgments for checkpoint messages;
    await completion of locally-initiated transmission and
    storage of checkpoint data
    acknowledge parent;
    await checkpoint commit message;
    propagate checkpoint commit message;
    commit checkpoint;
    clear all dsc bits;     }
```

Figure 2: A checkpointing session

In the first phase, the system constructs a tree of all process-clusters that have communicated (directly or indirectly) with the process-cluster that initiates checkpointing. The communication tree is built recursively, with each node sending checkpoint messages to all nodes containing process clusters with which it has communicated directly. Upon receiving a checkpoint message for a task, the node suspends execution of all local processes of that task. No new coherency transactions are initiated or serviced for the task until the checkpointing session is terminated. When all the coherency transactions pending for the task have terminated, the node begins sending the checkpoints of the local members of the task to stable storage. In parallel, the node propagates the checkpoint message to all nodes containing process clusters with which it has communicated directly. Once successful acknowledgments are received from all these nodes, a successful acknowledgment is propagated back up the tree. When all process-clusters complete saving their checkpoints, this informations is propagated up the tree. Once checkpointing coordinator at the root of the tree is informed that all the checkpoints have been saved to stable storage, it commits its checkpoint and initiates the second phase by sending ‘‘commit’’ messages down the tree. Upon receipt of the commit message, each node commits its checkpoint.

With coordinated checkpointing in a message passing system, at any point in time there is only a *single, committed* checkpoint of every process in stable storage (Section II). In a DSM system it is desirable to maintain this property and store in stable storage only a *single* copy of each block in a location that is independent of the current block owner or the existence of multiple copies of the block in system nodes. This simplifies the management of checkpoints in stable storage and reduces the size of checkpoints.

Shared blocks may be replicated on multiple nodes, and may thus be saved by any one of these nodes. Since each block has a *unique* owner node, that node is assigned the responsibility for checkpointing the block, if needed. Each node participating in the checkpointing session checkpoints private data of the local process cluster (registers and any other private memory) and all shared blocks which are owned locally and for which the dsc bit is set. Since interprocess dependency used to construct the interacting set is determined by the transfer of a block with the dsc bit set (Section IV.A), it is guaranteed that if a block of the task has its dsc bit set in any of the participants in the checkpointing session, the current owner of the block will also be one of the participants. Hence, all the blocks of this task that have been modified on the checkpointing nodes since their last checkpoint (for this task) will be part of the checkpointed state. The dsc bit of all these blocks are cleared at the end of the checkpointing session.

Since the checkpoint mechanism does not change the distribution of data in the local memories of the nodes, the coherency directory continues to be valid, and is, hence, not altered. The key steps in a checkpointing session are summarized in Figure 2.

D. Recovery

The failure of a node causes the loss of all state in that node — register state of processes that were active on that node, sections of the address space of several tasks that were resident in the node’s local memory, and the fragment of the distributed coherency directory resident in the node. The most recent committed checkpoint contains the private process states and corresponding shared memory state from which correct execution may be restarted. However, the state of the coherency directory is not a part of the state of any task, and is thus not implicitly checkpointed or recovered with memory and process states. Following recovery, the entire distributed directory, including directory fragments on nodes that do not participate in any rollback, must accurately reflect the distribution of copies in the system.

With coordinated recovery, the rollback of processes that were active on the failed node requires the rollback of processes on other nodes that communicated (directly or indirectly) with the failed processes. All these processes are rolled back to their previous checkpoints in coordination, employing the same two-phase protocol used for checkpointing. Recovery is initiated for each process-cluster that was active on the

```
recover → { /* in a recovering failed node */
    send rollback messages down the ''rollback tree''
    reject all transaction requests;
    invalidate MBT entries;
    send repair_directory (selfid) messages to all nodes;
    await ownership messages;
    await acknowledgments to rollback messages;
    restore processor state from checkpoint;
    send rollback commit messages to
    children in the rollback tree;
}

repair_directory (nodeid) → {
      /* repair coherency directory */
    invalidate local copies of all
    blocks owned by nodeid;
    eliminate nodeid from the copy directories of
    all locally-owned blocks;
    send to nodeid ownership messages for owned
    blocks that are default owned by nodeid;
}

rollback → {
    suspend processes in process-cluster;
    await for termination of pending transactions;
    reject all transaction requests;
    propagate rollback messages down the rollback tree;
    await acknowledgments to rollback messages;
    acknowledge parent;
    await rollback commit message;
      /* roll back task state */
    restore state of process-cluster from checkpoint;
    invalidate all dirty private blocks;
    forall (shared blocks with block.dsc = 1)
        if (readonly) invalidate block;
        else {
            restore block from last checkpoint;
            invalidate copy directory;
            block.dsc = 0;
        }
    propagate rollback commit messages down
    the recovery tree;
}
```

Figure 3: A recovery session

failed node. Since processes belonging to different tasks do not communicate with each other, different tasks can recover independently. Hence, the two-phase protocol is executed independently for each task that was active on the failed node. A task can resume execution once its state is restored and the associated coherency directory repair is completed.

The first step in recovery is to roll back to the most recent checkpoint the private states (e.g., registers) of all the processes belonging to interacting sets that include processes on the failed node. Next, all modifications to the task's shared address space performed by all these processes since their most recent checkpoints are undone. On nodes that have *not* failed, all memory blocks modified since their last checkpoint have their dsc bit set. Since interprocess dependency used to construct the interacting set is determined by the transfer of a block with the dsc bit set (Section IV.A), it is guaranteed that if a node that has modified a block since the node's last checkpoint is a member of the interacting set, any other node holding a copy of the block is also a member. Hence, the ''unwanted'' modifications of the shared address space can be undone by each member of the interacting set restoring all blocks with the dsc bit set from their last checkpoint.

The failed node recovers with all blocks in its local memory marked invalid. Hence, any modifications to the shared address space performed locally prior to rollback are automatically undone. During post-recovery execution, valid data will be accessed either from the last checkpoint or from the restored memory on other nodes. The correct choice will be made since, with an empty directory, references to a block that is not found locally are referred to the block's default owner. If the default owner is the node that has failed, it can recover the block from the last checkpoint.

Interprocessor traffic associated with restoring memory blocks can be minimized by restoring only dirty blocks that are owned locally. Stale read-only copies can be invalidated and incrementally obtained during post-recovery execution. Consistency of the directory can be maintained by restoring the owner's copy of the block from the last checkpoint in the exclusively owned state (to indicate the absence of read-only copies).

During the recovery session, the coherency directory must be restored to a valid state. Node failure causes three inconsistencies in the directory state. First, the Migrated Block Table (MBT) entries that were in the local memory of the failed node are lost. The absence of these entries in the default owner implies that the default owner retains ownership of the blocks, leading to a conflict of ownership with the actual current owner of

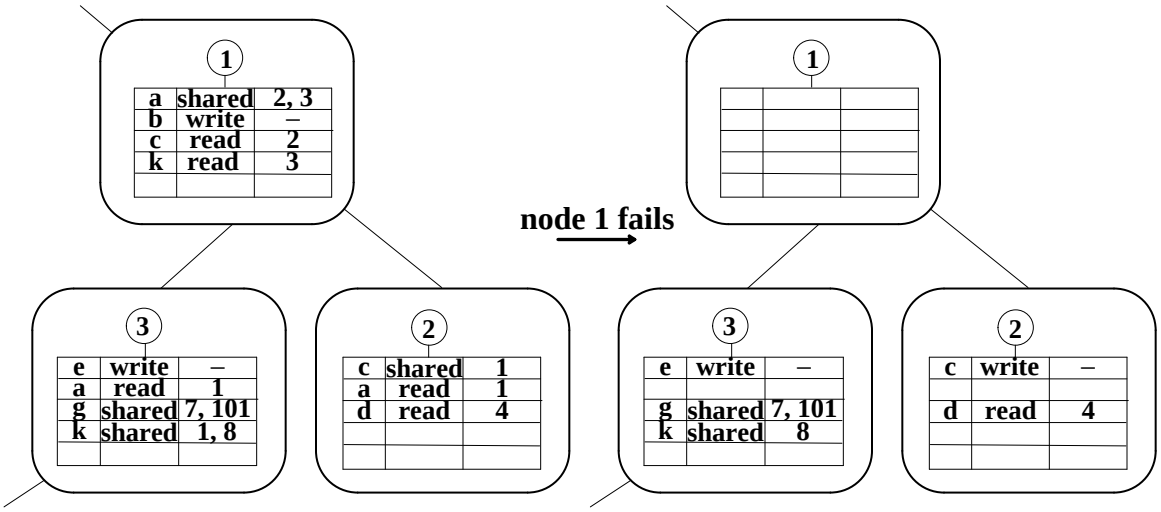

Figure 4: Repair of the directory state during recovery. In the figure, the directory maintains the block id, the access permission, and the copy directory. A node which possesses a block in the ''write'' or ''shared'' states is the owner of the block, and maintains the location of copies. A node that possesses a ''read'' copy of a block maintains the identity of the block owner. Node 1 owns blocks *a* and *b*, and has copies of blocks *c* and *k*. Node 1 fails and later recovers with an empty memory state. Directory fragments on other nodes are repaired by marking the local copy of blocks owned by node 1 (*a* and *b*) invalid, and by eliminating references to read-only copies in node 1 (blocks *c* and *k*).

these blocks. Second, read-only copies that the coherency directory (on other nodes) indicates as present in the failed node, are no longer present there. Third, blocks that are owned by the failed node are no longer available in the failed node, and neither is the associated copy directory which maintains the distribution of the copies. The repair of the coherency directory must resolve these inconsistencies.

Since the directory state possesses no application level semantics, it is unnecessary to restore it to any prior state. It is sufficient to transform it to a new state that correctly represents the state of the local memories. Our scheme restores the failed node's MBT by reconstruction. All nodes in the system update the restarting node with a list of their owned blocks that are default-owned by the failed node. The recovering node uses this information to reconstruct its MBT.

Directory inconsistencies associated with the loss of read-only blocks and of copy directories in the failed node are repaired by transforming the directory and memory state in other nodes. While several transformations are possible, we employ one in which each node requires only local information, thus minimizing network traffic. To repair the inconsistency associated with the loss of read-only copies in the failed node, each node eliminates all references to such copies from its copy directories (recall that all nodes know the identity of the failed node). When a node loses its copy directories, it loses information about the location of read-only copies of its owned blocks. Hence, each node marks *invalid* all read-only blocks owned by the failed node. This allows the recovering failed node to be restored with null copy directories for all its owned blocks.

Blocks that were owned by the failed node prior to failure are absent from its local memory following recovery. However, these blocks continue to be owned by the recovering node. Accesses to any of these owned blocks are satisfied, on demand, by restoring them with exclusive ownership from their last checkpoint.

It is important to note that the directory repair requires the participation of all nodes that hold any of the blocks that were managed or were resident in the failed node. This includes nodes that were not members of any of the recovery trees used to coordinate rollback. The repair of directory inconsistencies is illustrated in Figure 4. The key steps in a recovery session are summarized in Figure 3.

V. Previous DSM Recovery Schemes

Many recovery schemes proposed for DSM systems focus on avoiding rollback propagation, at the cost of frequent checkpointing [24, 23, 19, 7] or high overhead for maintaining logs [16]. Due to the absence of rollback propagation, these schemes have the advantage of fast recovery, at the cost of high overhead

during normal operation.

Consider a process Q that reads data modified by a process P, so that the state of Q depends on the state of P. Rollback propagation occurs if P fails and recovers by rolling back, forcing Q to also roll back since P may compute a different value for the data during re-execution.

Wu and Fuchs [24] have proposed a scheme where rollback propagation is avoided by forcing processes to checkpoint before providing modified data to any other process. Checkpointing a process involves the flushing of pages modified by the process since the last checkpoint and saving the process state onto disk. At recovery, the processes that were running on the failed node are restarted from their checkpointed states, and re-acquire memory pages on demand. If the state of the centralized directory is lost, the directory information is reconstructed on demand. When a page is requested by a node, a special message is broadcast requesting the owner and the copy holders of the page to identify themselves to the new centralized manager. If no node claims ownership of the page, the new centralized manager assumes ownership of the page. The requested page is provided by the owner of the page, unless the owner is recovering, in which case the page is fetched from stable storage. In [7], this scheme is extended to systems supporting weak consistency by checkpointing a process only when a synchronization variable is to be read by another process.

Tam and Hsu [23] propose a scheme where a modified page is permitted to migrate only after all modified pages are logged to stable storage (as in [24]). At recovery, the process' state is restored from the checkpoint. The key contribution of this scheme is the use of a ''token database'' that allows the coherency directory to be repaired without consulting all the directory fragments on the system (as required by our scheme). Stumm and Zhou [19] propose a variation that tolerates only single faults. When a node is about to migrate modified data to another node, all modified data from the first node is checkpointed in the local memory of the second node.

Richard and Singhal present a recovery scheme for an environment where checkpoints are costly [16]. All pages read by a process are logged to volatile storage. When a modified page is to be read by another process, the log is committed to stable storage. Upon failure of a process, the process is restored from the checkpoint, and its up-to-date state is recomputed using the logged pages. With this scheme there is no rollback propagation and no need for frequent checkpointing. However, there is high overhead for collecting, committing, and storing the logs.

VI. Evaluation

A key measure of error recovery schemes based on checkpointing and rollback is the overhead they incur during normal operation. This overhead includes the cost of checkpointing as well as any other special processing, such as for keeping track of dependencies, logging messages, etc. Our evaluation is focused on the overhead for checkpointing, specifically on the extra interconnection network traffic due to checkpointing. We compare our coordinated checkpointing scheme to the scheme proposed by Wu and Fuchs [24].

Our evaluation is based on trace-driven simulation, using address traces from a possible execution on a 64 node multiprocessor of three parallel applications: Weather, Speech, and FFT. Weather uses finite difference methods to solve a set of partial differential equations describing the state of the atmosphere. Speech uses a modified Viterbi search algorithm to find the best match between paths through a directed graph representing a dictionary and another through a directed graph representing the input. FFT is a radix-2 fast Fourier transform. Details about the applications and the traces can be found in [3].

The simulator faithfully implements the checkpointing phase of both recovery schemes on top of the block level coherency scheme. Simulations were performed for a range of block sizes and, in the case of coordinated checkpointing, for a range of checkpointing frequencies. The interconnection network traffic includes interprocessor traffic as well as traffic to stable storage. This traffic is calculated assuming that the size of a control message is 16 bytes, the size of data messages is 16 bytes plus the data being transmitted, and that the size of a process' register state is 128 bytes.

Figure 5 shows the number of bytes of network traffic per data reference as a function of the block size. In all three applications, the additional traffic due to coordinated checkpointing is quite low for a wide range of block sizes. The checkpointing scheme proposed in [24] has higher network traffic than the coordinated checkpointing scheme. Furthermore, this overhead becomes significantly higher for larger block sizes. Larger blocks increase false-sharing, thus increasing the frequency of read-write sharing and hence, the frequency of checkpointing with the Wu and Fuchs scheme. This is important since some DSM system designs use large block sizes (e.g., 1 KB blocks are used in [12] and [24]).

Figure 6 shows the interconnection network traffic per data reference and the number of checkpoints as a function of the checkpointing interval, with a block size

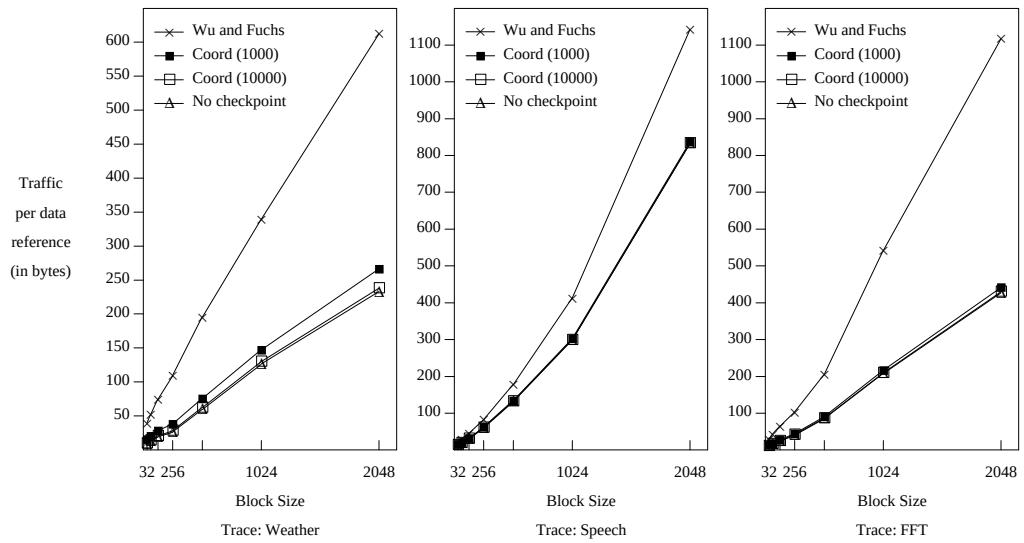

Figure 5: Network traffic as a function of block size with checkpointing as proposed in [24] (Wu and Fuchs), with our proposed coordinated checkpointing, and without checkpoints. The coordinated checkpointing scheme results are shown at two checkpoint intervals: after every 1000 references by any processor, and after every 10000 references by any processor.

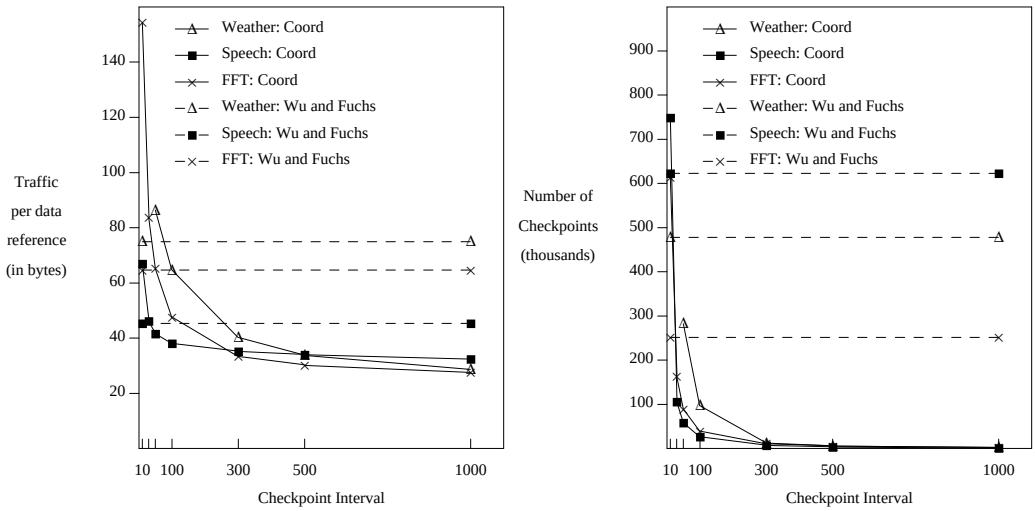

Figure 6: Network traffic and number of checkpoints as a function of the checkpointing interval. The block size is 128 bytes. The checkpointing interval is the maximum number of references made by any processor between checkpoints — it is an approximate measure of the time between checkpoints. For the scheme proposed in [24] (Wu and Fuchs), the checkpointing interval is determined by the behavior of the application and cannot be set by the system.

of 128 bytes. With the Wu and Fuchs scheme, the checkpointing interval is a characteristic of the particular application and cannot be varied. These curves demonstrate the benefits of being able to tune the frequency of checkpoints in an *application-independent* manner instead of being forced to checkpoint by the sharing behavior of the application. A lower checkpointing frequency can imply significantly lower overhead. Hence, if the requirements of the application (e.g., real-time response specifications) allow it, the overhead for coordinated checkpointing can be reduced by simply lowering the checkpointing frequency. This is not possible with the Wu and Fuchs scheme — whether or nor there is a need, checkpointing will be frequent and costly.

The Wu and Fuchs scheme has the advantage of

checkpointing traffic which is evenly distributed over time. With coordinated checkpointing, the checkpointing traffic is likely to be burstier. This disadvantage of coordinated checkpointing can be eliminated through the use of asynchronous coordinated checkpointing with a memory system that supports copy-on-write pages [22]. In this case, volatile checkpoints are taken locally on each node and are spooled gradually to stable storage once normal operation resumes.

VII. Summary and Conclusions

Many error recovery schemes for multiprocessors and multicomputers are based on checkpointing a process whenever it communicates with another process or on the use of message logging techniques. For scalable high performance systems, used for communication-intensive applications, all of these schemes incur unacceptably high overhead.

We have presented an error recovery scheme for DSM multicomputers based on coordinated checkpointing and rollback. Our scheme is an adaptation of coordinated checkpointing and rollback for message-passing systems. This adaptation efficiently solves several potential difficulties with DSM systems which are due to the lack of explicit messages, the presence of a shared replicated address space, and the distributed coherency directory. We have developed mechanisms to identify and track read-write communication in the DSM system and to checkpoint and recover the shared replicated address space. Our scheme does not checkpoint and rollback the coherency directory. Instead, at recovery, the distributed coherency directory is ‘‘repaired’’ to restore it to a correct state. Our checkpointing scheme allows the frequency of checkpointing to be adjusted independently of the sharing behavior of the application. Trace-driven simulations have demonstrated the significant performance advantages of our recovery scheme. By allowing lower checkpointing frequency, leading to lower overall checkpointing traffic, our approach can result in dramatically lower overhead than alternate schemes, leading to minimal impact on the normal operation of the system.

References

1. G. Barigazzi and L. Strigini, ‘‘Application-Transparent Setting of Recovery Points,’’ *13th Fault-Tolerant Computing Symposium*, Milano, Italy, pp. 48-55 (June 1983).
2. H. Burkhardt III, ‘‘Overview of the KSR1 Computer System,’’ Technical Report KSR-TR-9202001, Kendall Square Research, Boston, MA (February 1992).
3. D. Chaiken et al., ‘‘Directory-Based Cache Coherence in Large-Scale Multiprocessors,’’ *Computer* **23**(6), pp. 49-58 (June 1990).
4. K. M. Chandy and L. Lamport, ‘‘Distributed Snapshots: Determining Global States of Distributed Systems,’’ *ACM Transactions on Computer Systems* **3**(1), pp. 63-75 (February 1985).
5. T. M. Frazier and Y. Tamir, ‘‘Application-Transparent Error-Recovery Techniques for Multicomputers,’’ *The Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, Monterey, CA **1**, pp. 103-108 (March 1989).
6. J. N. Gray, ‘‘Notes on Data Base Operating Systems,’’ pp. 393-481 in *Operating Systems: An Advanced Course*, ed. G. Goos and J. Hartmanis, Springer-Verlag, Berlin (1978). Lecture Notes in Computer Science 60.
7. B. Janssens and W. K. Fuchs, ‘‘Relaxing Consistency in Recoverable Distributed Shared Memory,’’ *23rd International Symposium on Fault-Tolerant Computing*, Toulouse, France, pp. 155-163 (June 1993).
8. D. B. Johnson and W. Zwaenepoel, ‘‘Sender-Based Message Logging,’’ *17th Fault-Tolerant Computing Symposium*, Pittsburgh, PA, pp. 14-19 (July 1987).
9. R. H. Katz et al., ‘‘Implementing a Cache Consistency Protocol,’’ *12th Annual Symposium on Computer Architecture*, Boston, MA, pp. 276-283 (June 1985).
10. R. Koo and S. Toueg, ‘‘Checkpointing and Rollback-Recovery for Distributed Systems,’’ *IEEE Transactions on Software Engineering* **SE-13**(1), pp. 23-31 (January 1987).
11. D. Lenoski et al., ‘‘The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor,’’ *17th Annual International Symposium on Computer Architecture*, Seattle, WA, pp. 148-159 (May 1990).
12. K. Li and R. Schaefer, ‘‘A Hypercube Shared Virtual Memory System,’’ *International Conference on Parallel Processing*, St. Charles, IL, pp. I/125-I/132 (August 1989).
13. K. Li and P. Hudak, ‘‘Memory Coherence in Shared Virtual Memory Systems,’’ *ACM Transactions on Computer Systems* **7**(4), pp. 321-359 (November 1989).
14. B. Liskov et al., ‘‘Orphan Detection,’’ *17th Fault-Tolerant Computing Symposium*, Pittsburgh, PA, pp. 2-7 (July 1987).
15. B. Randell et al., ‘‘Reliability Issues in Computing System Design,’’ *Computing Surveys* **10**(2), pp. 123-165 (June 1978).
16. G. G. Richard III and M. Singhal, ‘‘Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory,’’ *12th Symposium on Reliable Distributed Systems*, Princeton, NJ, pp. 58-67 (October 1993).
17. F. B. Schneider, ‘‘Byzantine Generals in Action: Implementing Fail-Stop Processors,’’ *ACM Transactions on Computer Systems* **2**(2), pp. 145-154 (May 1984).
18. R. E. Strom and S. Yemini, ‘‘Optimistic Recovery in Distributed Systems,’’ *ACM Transactions on Computer Systems* **3**(3), pp. 204-226 (August 1985).
19. M. Stumm and S. Zhou, ‘‘Fault Tolerant Distributed Shared Memory Algorithms,’’ *Proceedings of the Second IEEE Symposium on Parallel and Distributed Computing*, Dallas, Texas, pp. 719-724 (December 1990).
20. Y. Tamir and C. H. Séquin, ‘‘Error Recovery in Multicomputers Using Global Checkpoints,’’ *13th International Conference on Parallel Processing*, Bellaire, MI, pp. 32-41 (August 1984).
21. Y. Tamir and T. M. Frazier, ‘‘Application-Transparent Process-Level Error Recovery for Multicomputers,’’ *Hawaii International Conference on System Sciences-22*, Kailua-Kona, Hawaii, pp. 296-305, Vol I (January 1989).
22. Y. Tamir and T. M. Frazier, ‘‘Error-Recovery in Multicomputers Using Asynchronous Coordinated Checkpointing,’’ Computer Science Department Technical Report CSD-910066, University of California, Los Angeles, CA (September 1991).
23. V.-O. Tam and M. Hsu, ‘‘Fast Recovery in Distributed Shared Virtual Memory Systems,’’ *The 10th International Conference on Distributed Computing Systems*, Paris, France, pp. 38-45 (May 1990).
24. K.-L. Wu and W. K. Fuchs, ‘‘Recoverable Distributed Shared Virtual Memory,’’ *IEEE Transactions on Computers* **39**(4), pp. 460-469 (April 1990).