Chapter Six

Error Recovery for Distributed Shared Memory Systems

The reliability requirements of large DSM multicomputers, consisting of thousands of VLSI chips, can only be met by using fault tolerance techniques [Tami84]. 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 [Rand78]. *Message logging* [Stro85, John87] 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* [Rand78, Chan85] with the rest of the system. *Coordinated checkpointing* [Bari83, Tami84, Koo87, Tami89] 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 [Fraz89, Fraz95].

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 [Wu90a, Tam90, Stum90] (see Section 6.5). Based on message logging, it has also been proposed to use independent process checkpoints together with logs of remote page accesses [Rich93]. As mentioned earlier, these schemes suffer from unacceptably high overhead during normal operation.

This chapter presents an error recovery scheme for DSM multicomputers based on coordinated checkpointing [Jana94]. 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 [Tami84, Fraz89, Fraz95].

Our error recovery 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, to terminate read and write transactions that are in progress at checkpoint or recovery, 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.

The following section is a brief summary of coordinated checkpointing for message-passing systems. Section 6.2 describes our system model and assumptions. Our checkpointing and rollback scheme is presented for the one-level managed DSM system in Section 6.3. We extend this scheme to the hierarchical DSM system in Section 6.4. Section 6.5 is an overview of prior research on DSM system recovery. Performance evaluation based on trace-driven simulations is presented in Section 5, where we compare our schemes (for the block-level DSM system and for the hierarchical DSM system) to previously proposed schemes for DSM recovery.

6.1. 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 [Bari83, Tami84, Koo87, Tami89]. Hence, the set of all the committed checkpoints in stable storage is *always* a consistent *snapshot* [Chan85] of the entire system.

One approach to coordinated checkpointing and rollback is to save and restore a consistent snapshot of the entire system state [Tami84]. This approach may be undesirable since for large systems it may take an unacceptably long time to checkpoint or roll back. Furthermore, the overhead on the system for checkpointing will be highly ‘‘bursty’’ instead of being evenly spread over time. Finally, global checkpointing and rollback does not allow tuning the overhead and the recovery latencies to the different needs of different applications which may be executing on the system simultaneously.

In order to overcome the disadvantages of global checkpointing, it is possible to keep track of which processes have communicated with each other since their last checkpoint and coordinate the checkpointing of only the *interacting set* of processes as opposed to all processes [Bari83, Koo87, Tami89]. 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 [Tami89]. The initiator of the session coordinates the consistent checkpoint of all members of the communication tree [Bari83, Koo87, Tami89] using a two-phase commit protocol [Gray78]. 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.

6.2. System Model and Assumptions

We develop the error recovery scheme for a DSM multicomputer whose design was introduced in Chapter 3. We summarize the salient features of this system below. The DSM multicomputer system consists of several nodes interconnected by point-to-point links (Figure 1.1 in Chapter 1). Each node in the system includes (Figure 3.1 in Chapter 3) an application processor, local memory, cache memory, and coprocessors to assist in the coherency management and interprocessor communication [Kusk94]. 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.

DSM is implemented on the system using a distributed directory-based coherence protocol [Chai90, Leno90]. Ownership [Katz85] 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. To identify the current owner of any block, the scheme uses the fixed distributed manager algorithm [Li89b]. Each node is statically assigned to be the default owner of a set of blocks using a hash function. The default owner keeps track of who is the current owner of the block.

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. (The meaning of ‘‘ownership’’ and the structure of the directory tables will be extended for the hierarchical DSM system in Section 6.4.)

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 [Tami83, Schn84]. 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 neighbors of a failed node detect the failure and broadcast the identity of the failed node to all the other nodes in the system.

6.3. Coordinated Checkpointing and Rollback for DSM Systems

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 [Chai90, Leno90, Li89a]. 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. In our coordinated checkpointing and rollback scheme for DSM multicomputers, we take 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.

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.

6.3.1. Tracking Interprocess Communication

A key requirement for coordinated checkpointing is the ability to identify *interacting sets* of processes. For this purpose, an ‘‘interaction’’ is any operation that may cause the state of one process to depend on the state of another process. In message passing systems any transmission of information from one process to another, i.e., any message exchange between processes *may* result in such dependency. Hence, in message passing systems a message exchange causes the participating processes to be included in the same interacting set.

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. Even accesses that result in transmission of information from one processor to another do not necessarily imply dependency between the processes running on the different processors. Assuming that all accesses to the shared address space imply dependencies 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, the load does not cause a dependency between the state of the process executing the load and the state of any other process in the system. Hence, this remote access should be ignored when determining the interacting set of processes that must be checkpointed together. 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 [Wu90a], 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 of memory blocks across the network under the control 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.

The transfer of a block whose dsc bit is set implies that the state of the process requesting the block may become dependent on the non-checkpointed state of the process that modified the block. Hence, the transfer of a block with the dsc bit set implies a possible dependency and is used for determining the interacting sets of processes.

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[Bari83, Koo87, Tami89]). 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.

6.3.2. 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 [Tami84, Tami89]. Messages in transit between the processes when rollback is initiated are discarded [Tami84, Tami89]. In a DSM multicomputer, process interaction takes place through read and write ‘‘transactions’’ which 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) abort the transaction, 2) 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 [Lisk87] and should be aborted. However, always aborting pending transactions when checkpointing is initiated results in unnecessary performance loss. In many cases it may be possible to complete transactions which are pending when checkpointing is initiated, thus avoiding the performance cost of redoing them later.

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 [Tami91]. 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.

When a failure occurs, the system must ensure that nodes do not end up waiting indefinitely for the completion of transactions that cannot be completed due to the failure. 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.

In order to facilitate the detection of transactions that must be aborted, a node failure is broadcast to all other nodes (Section 6.2). However, with many coherency protocols [Li89b, Leno90] 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 block owner 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.

6.3.3. 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 [Gray78], exactly as in message-passing systems [Bari83, Koo87, Tami89]. Since processes of different tasks do not communicate, each checkpointing session is localized to a task.

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. This 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 its children — all nodes containing process clusters with which it has communicated directly. When a node completes saving the state of its process cluster and receives acknowledgements from all its children, it sends an acknowledgement to its parent. Once the 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 6.1). 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 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 6.1: A checkpointing session

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. In order to avoid checkpointing multiple copies of the block, the responsibility for checkpointing the block, if needed, is assigned to the block owner. Each node participating in the checkpointing session checkpoints the private data of the local process cluster (registers and any other private memory) and all the 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 6.3.1), 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 6.1.

6.3.4. 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. As part of the recovery session, the system must ensure that the entire distributed directory accurately reflects the distribution of copies in the system. This requires modifying (repairing) the distributed directory, possibly including directory fragments on nodes that do not participate in any rollback.

6.3.4.1. State Restoration

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 [Bari83, Koo87, Tami89]. 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 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 6.3.1), 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 by restoring all the local blocks with the dsc bit set from their last

```
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;
}
```

Figure 6.2 (continued): A recovery session

checkpoint.

The failed node recovers with all the blocks in its local memory marked invalid. Hence, any modifications to the shared address space performed locally prior to rollback are automatically undone. Since the entire directory of this recovering node is invalidated, when post-recovery execution begins, every access by this node results in a miss. The required blocks will be accessed either from the last checkpoint or from the restored memory on other nodes. Since the directory is empty, references to a block that is not found locally are referred to the block’s default owner. The default owner is either the current owner or can direct the request to the current owner. The current owner either provides the block from its memory or from the last checkpoint. As described later in this section, our

```
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 6.2: A recovery session

directory repair procedures ensure proper handling of recovery sessions in which a recovering failed node is the current and/or default block owner.

Interprocessor traffic associated with restoring memory blocks can be minimized by restoring only dirty blocks that are owned locally. As discussed above, all the blocks with their dsc bit set are orphans. If they are un-owned (read-only) copies, they are invalidated and, if needed, obtained incrementally during post-recovery execution. Consistency of the directory is 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).

6.3.4.2. Directory Repair

During the recovery session, the distributed 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 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. Furthermore, the copy directories associated with these blocks are also no longer available. The repair of the coherency directory must resolve these inconsistencies.

Since the directory state possesses no application level semantics, it is not necessary to restore it to a prior valid state. It is sufficient to transform it (roll forward) to a new state that correctly represents the state of the local memories. Our scheme restores the failed node's MBT by reconstruction. All the 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

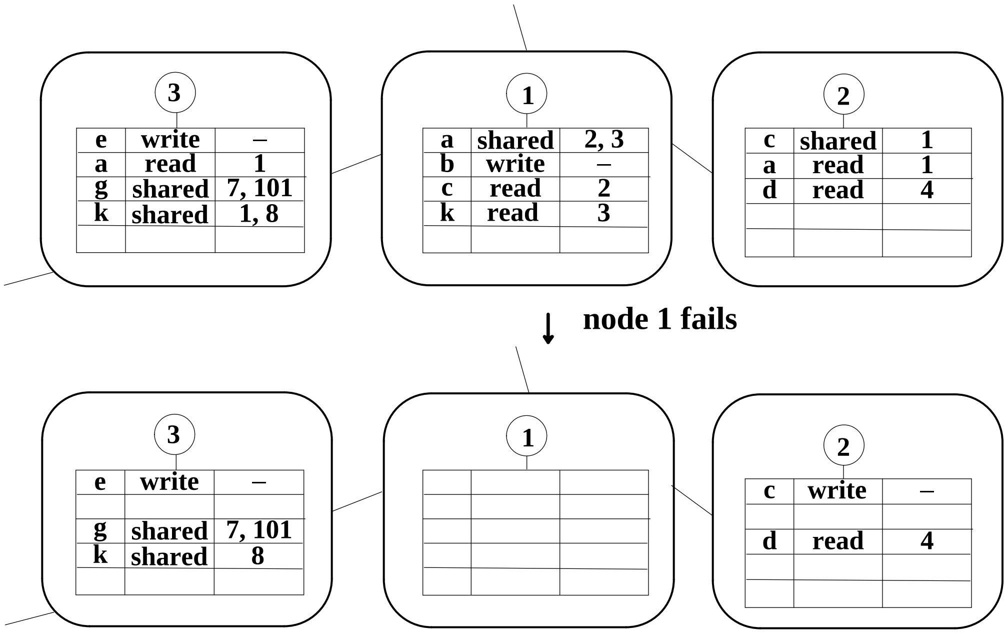


Figure 6.3: Repair of the directory state during recovery. The directory maintains the block id, the access permission, and the copy directory. A node which possesses a block in the ‘‘write’’ or ‘‘shared’’ states is the owner of the block, and maintains the location of copies. A node that possesses a ‘‘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*).

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 the 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. As discussed previously, this problem is solved by each node marking *invalid* read-only copies of all local 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 6.3. The key steps in a recovery session are summarized in Figure 6.2.

6.4. Coordinated Checkpoint and Rollback for Hierarchical DSM Systems

The coordinated checkpoint and rollback scheme presented in the previous section was designed for DSM systems in which the node memories were maintained coherent by managing the address space at a single granularity of blocks. Ownerships and read-only copies are granted and maintained at the level of blocks. Such coherency management at one level of granularity suffers from high storage overhead when the granularity is fine (small block sizes), and from poor performance when the granularity is coarse (large block or page sizes) [Tami92]. As we discussed in Chapter 3, an hierarchical DSM system [Tami92], where coherency is managed at multiple levels of granularity, can achieve the performance advantage of fine grain block-level schemes without their high storage overhead. The organization of the coherency directory and the protocol associated with hierarchical management differ from those used with single-level management.

Adapting the coordinated checkpoint-rollback recovery scheme presented in Section 6.3 to a DSM system with hierarchical management requires key modifications in: (i) tracking dependencies between processes, (ii) the proper termination of coherency transactions, (iii) dealing with two levels of mapping and with private pages in the saving and restoring of process states, and (iv) the repair of the coherency directory. We discuss and describe these modifications below.

6.4.1. The Hierarchical DSM System

With hierarchical management (Chapter 3), ownership is maintained at *both* the page (thousands of bytes) and block (tens of bytes) levels. To limit storage overhead, pages that are dynamically private are mapped and owned only at the page level. For shared pages, ownership is maintained at both the page and block levels allowing data to be shared at the block level. Private and shared pages are distinguished *dynamically*, based on the sharing behavior of the application [Tami92].

The owner of a shared page identifies the current owner of each block of that page. A node that requires access to a block that is not available locally sends the block request to the page owner for forwarding to the block owner. The owner of the block provides read-only copies of the block and maintains the location of these read-only copies in a copy directory. To reduce storage overhead, read-only copies of blocks are *not* tagged with the identity of the block owner (unlike the one level scheme). Instead, only the identity of the page owner is maintained. When a node (other than the current block owner) needs to write to the block, the block owner invalidates all read-only copies and transfers ownership of the block to the writing node. Since the page owner identifies the owner of each block, this transfer of block ownership is committed with the page owner.

A block owner knows which blocks it owns. The page owner is the only other node that is aware of the current owner of any of the blocks of the page. The current owner of the page is tracked by the default owner of the page [Li89b]. Default ownership is assigned at the granularity of pages. A Migrated Page Table (MPT) of each node, analogous to the MBT in the block level DSM system, maintains the current owner of all default owned pages that have been migrated away. At any time, any node can determine the current owner of any page through the default owner of the page.

6.4.2. Tracking Interprocess Communication

As described earlier, in a DSM system, processes communicate by the read-write sharing of data and dependencies can be tracked by monitoring the exchange of dirty data. In the one-level DSM system, these dependencies were tracked by associating a dirty-since-checkpoint (dsc) bit with each block of data. In the hierarchical DSM system, data is mapped at both the page and block levels — private data is mapped at the page level and shared data is mapped at both page and block levels. Hence, there are choices to be made regarding the granularity at which dependencies should be tracked in the hierarchical system.

To track dependencies at the page level, a dsc bit can be associated with each page. Since the single page-level dsc bit is associated with all blocks of the page, processes that have accessed different blocks of the page will be treated as having shared data. Hence, checkpoints and rollbacks will be unnecessarily propagated to processes that do not have dependent states, forcing them to be checkpointed and rolled back together. The resulting larger interacting sets lead to higher performance overhead than necessary.

Dependencies can be tracked at the block level by maintaining a dsc bit for every block in memory. This can eliminate the unnecessary checkpoint and rollback propagation of page-level dependency tracking. However, such a scheme incurs significant storage overhead over maintaining a dsc bit per page, and may be wasteful. Specifically, a large fraction of the data resident in memory is dynamically private and thus mapped only at the page level. Associating a dsc bit with every block of these dynamically private pages may not provide much benefit in reducing checkpoint and rollback propagation since these pages are not currently being accessed by multiple nodes.

In order to keep both the performance and the storage overheads low, block-level dsc bits are maintained for blocks in pages that are dynamically shared while only page-level dsc bits are maintained for dynamically private pages. Since shared pages are also mapped at the block level, integrating the dsc bits with the block level data structure maintained for shared pages by the memory management mechanism is simple. Since pages dynamically change between private and shared status (depending on the pattern of accesses to the pages at execution), the system dynamically transforms between the per-page dsc bits of private pages and the per-block dsc bits of shared pages when a page changes status. When a private page with its dsc bit set becomes shared, the dsc bit is set for each of its blocks. The dsc bit of a page that changes from shared to private status is set if the dsc bit was previously set for any of the blocks of the page. The use of these dsc bits to track dependencies is identical to the one-level scheme. The dsc bit is transferred along with the data (block or page). When data is transferred with its dsc bit set, the node updates its list of process clusters it has directly communicated with.

6.4.3. Handling of Pending Transactions

As discussed in Section 6.3.2, the handling of coherency transactions that are in progress at checkpoint or recovery is a key issue. As with the one level scheme, we adopt the approach of ensuring that all such transactions are completed (successfully or unsuccessfully) before proceeding with the checkpoint or recovery. The hierarchical DSM system performs additional, more complex, coherency transactions which must also meet the requirement of guaranteed completion. For example, in the hierarchical system, a node requesting a copy of a block may send the request to the default owner of the <u>page</u> to which the block belongs. This request may be forwarded by the default owner to the current owner of the page, which may, in turn, forward the request to the current owner of the <u>block,</u> who will service the read request directly. Similar to the scenario discussed in Section 6.3.2, the node that originated the request is unaware of the identity of the current page owner, or the current block owner, and, hence, will be unable to associate the failure of either of these nodes with the failure of the transaction.

Coherency transactions in the hierarchical DSM system can be restructured exactly as in a one-level system to guarantee their completion (successfully or unsuccessfully) across failures. The coherency protocol is modified so that the reply is returned through the page owner and the default owner, so that the transaction can be aborted by the requesting node upon failure of any of these participant nodes. The resulting transactions are similar to the transactions on the one-level coherent system except for the presence of a page owner in addition to a block owner, and can be guaranteed to complete before checkpoint and recovery.

6.4.4. Checkpointing and Recovery of Process States

The checkpointing and recovery mechanisms respectively checkpoint and recover consistent states of an interacting set of process-clusters using a two-phase protocol, exactly as with the one-level system. The only difference that must be addressed is that in the hierarchical system some of the memory blocks are in dynamically private pages. Fortunately, the mechanism we developed for the one-level system extends naturally for the hierarchical system.

The checkpointing mechanism saves data that has been modified to stable storage. In the one-level scheme, to ensure that only a single committed copy of each memory block is maintained in stable storage, a memory block is saved only by its owner. As we discussed earlier, the hierarchical system maintains dsc bits for every block of shared pages and a single dsc bit for each private page. During checkpointing, nodes checkpoint their modified private pages and modified locally-owned blocks in shared pages. The dsc bit of private pages (subsection 6.4.2) is used to identify private pages that need to be checkpointed. As in the one-level scheme, during recovery the faulty computation is undone in the recovering nodes by locally invalidating all read-only copies of blocks that have their dsc bit set and by restoring all modified locally-owned blocks from stable storage. In addition, all modified private pages are restored from stable storage.

6.4.5. Directory Repair

The modifications to the directory repair mechanism address two key differences between the organization of the directory in the hierarchical DSM system and its organization in the one-level (block-level) system. First, in addition to ownership at the block level, memory is also owned at the page level. The page owner maintains an additional copy directory, which identifies the current owner of each of the blocks of the page. The repair mechanism should address the possible loss of this data structure maintained by the page owner. Second, nodes that possess read-only copies of a block do not store the identity of the block owner. In the one-level scheme, when the block owner fails, nodes that possess read-only copies recognize the loss of the block owner's copy directory (which maintains the location of read-only copies) and can repair it by invalidating their read-only copies themselves. In the hierarchical DSM system, nodes that possess read-only copies cannot unilaterally identify the failure of the block owner. Rather, since only the page owner can identify the owner of each block of the page, the cooperation of this page owner is necessary to repair the loss of the copy directories.

The distributed coherency directory in the hierarchical DSM system can incur the following losses due to node failure: (i) loss of the MPT maintained in the failed node, (ii) loss of read-only copies held by the failed node, (iii) loss of the directory which identifies the owners of blocks belonging to pages owned by the failed node, and (iv) loss of block copies owned by the failed node and their associated copy directories. The loss of the MPT and the loss of read-only copies present scenarios that are identical to the equivalent cases addressed by the one-level DSM recovery scheme and require no new solutions. As with the restoration of the MBT in the one-level system, the Migrated Page Table (MPT) is restored in the failed node using update messages. As in the one-level system, the block owner is aware of the location of read-only block copies. Hence, a block owner can repair its directory by eliminating references to read-only copies which were on the failed node.

The loss of the list of block owners on a failed page owner is equivalent to the loss of the MBT in the one-level scheme. The list of block owners serves the same function in the hierarchical DSM system as the MBT in the one-level system. It tracks the current owners of blocks, so that requests for a block can be forwarded to the current owner. In both cases block requests cannot be forwarded to the block owner if there is a loss. The page owner's directory of block owners is rebuilt using update messages just as the MBT is rebuilt in the one-level system. Any block whose ownership is not claimed by an update message is owned by the page owner.

When a node fails, it loses locally owned blocks along with their copy directories which identified the locations of read-only copies of these blocks. To repair this loss in the one-level scheme, each node in the system invalidated *its* local read-only copies of such blocks (blocks owned by the failed node). This allowed the failed node to recover with null copy directories for its owned blocks.

As mentioned earlier, this identical mechanism is not feasible in the hierarchical DSM system because nodes that possess read-only copies are not aware of the identity of the block owner. These nodes cannot *unilaterally* identify that the node owning the block has failed and thus cannot unilaterally invalidate their copy.

Since the page owner is the only node that can identify the owners of that page's blocks, the page owner can initiate the repair of directory inconsistencies caused by the failure of block owners. Specifically, if the page owner has not failed, it can determine that the block owner has failed and initiate the invalidation of all read-only copies of the block. Since the page owner cannot identify the location of these read-only copies, this invalidation can only be achieved by a broadcast. The failed node can subsequently recover with a null copy directory. (An alternative solution is to have the page owner assume exclusive ownership of these blocks. This solution will defer the broadcast invalidate traffic to the time the block is next written).

Solutions that rely on the directory of the page owner to recover from the failure of the block owner are not useful if the page owner has also failed. Hence, if the page owner fails, the copy directories of *all* the blocks of the page are invalidated (set to null). In order to permit the block owners to clear their copy directories, all nodes in the system locally invalidate their read-only copies of all the blocks of a page whose page owner has failed.

The directory repair mechanisms are illustrated in Figures 6.4 and 6.5. Figure 6.4 shows the repair of two types of losses — the loss of read-only copies and the loss of the block owner's copy directory when the block owner is distinct from the page owner. The repair of the loss of the page owner's directory of block

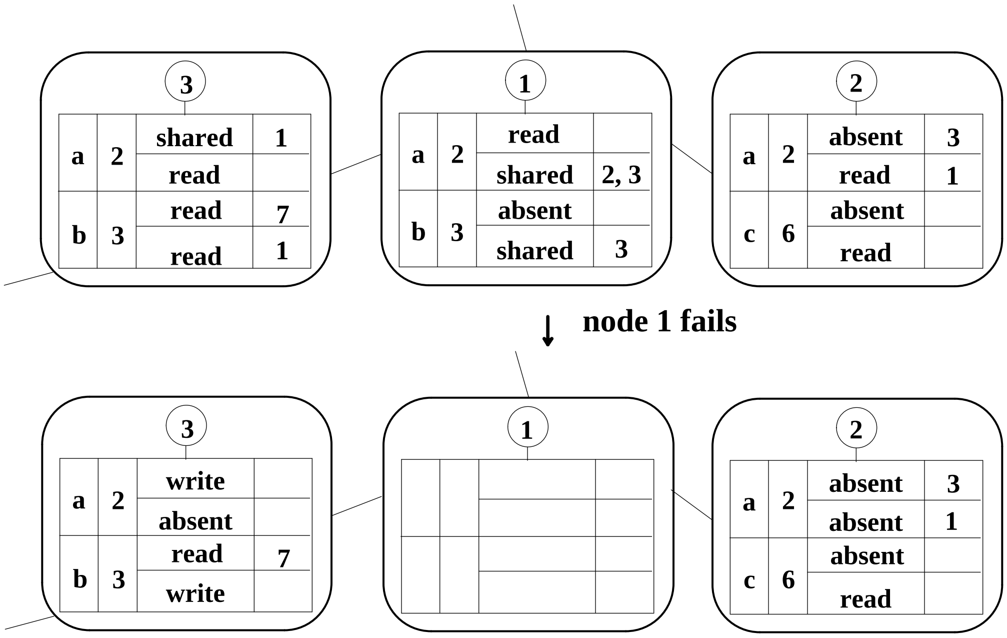


Figure 6.4: Repair of the coherency directory in a hierarchical DSM system — non-page-owner failure. The directory maintains the page id, the page owner, block access permissions, and the copy directory for each block. A page is comprised of two blocks (block *0* and block *1*). A node which possesses a block in the ‘‘write’’ or ‘‘shared’’ state is the block owner. If a node is a page owner, for every block that it does not own it maintains the identity of the block's owner (e.g., node 2, page *a*, block *1*). Pages *a* and *b* are present but not owned in node 1 prior to failure. Node 1 fails and recovers with an empty memory state. Node 3 removes node 1 from the copy directory of block *0* of page *a* to reflect the loss of node 1's copy. Node 2 owns page *a*, and determines that node 1 was the owner of block *1* of page *a*. Hence, node 2 broadcast invalidates all copies of block *1* of the page, and the block is restored in the exclusively owned state in node 1. The case of block *1* of page *b* is similar but here the page owner node 3 chooses to assume exclusive ownership of the block.

owner identities and the loss of the block owner's copy directory when it is also the page owner is shown in Figure 6.5.

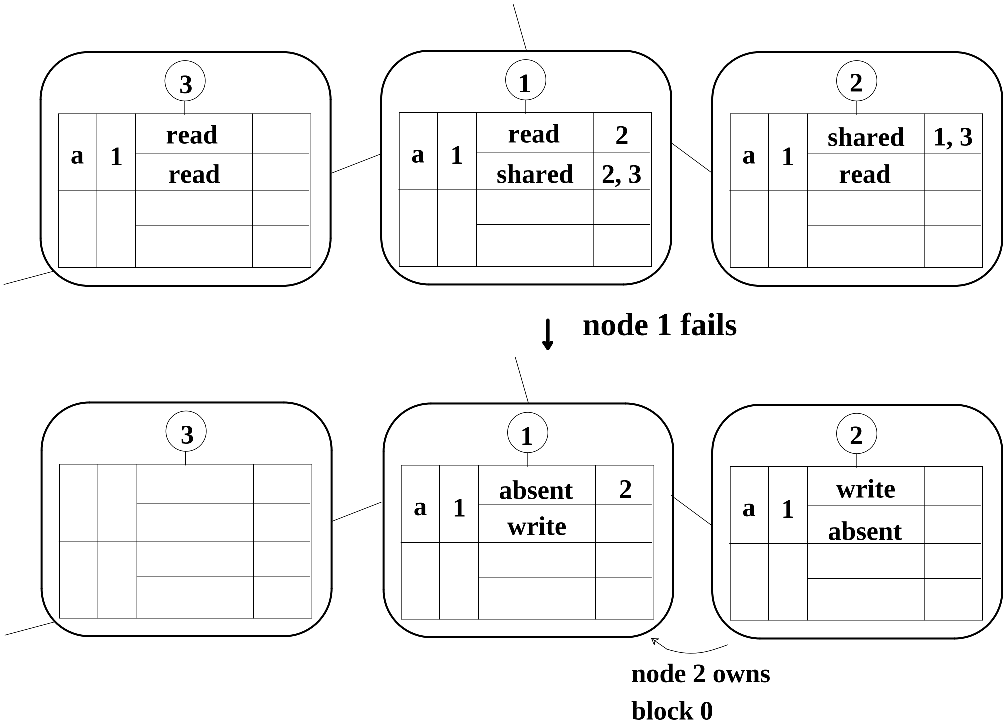


Figure 6.5: Repair of the coherency directory in a hierarchical DSM system — page owner failure. Node 1 owns page *a* prior to failure. Hence, nodes 2 and 3 invalidate their local read-only copies of all blocks of page *a* during recovery. Node 2 transfers block *0* of page *a* to the exclusively owned state, and informs recovering node 1 that it owns the block. The ownership of block *1* is unclaimed, and node 1 assumes exclusive ownership of the block.

6.5. Previous DSM Recovery Schemes

Many recovery schemes for DSM systems focus on avoiding rollback propagation, at the cost of frequent checkpointing [Wu90a, Tam90, Stum90, Jans93] or high overhead for maintaining logs [Rich93]. 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 [Wu90a] 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 [Jans93], 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 [Tam90] propose a scheme where a modified page is permitted to migrate only after all modified pages are logged to stable storage (as in [Wu90a]). 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 <u>without</u> consulting all the directory fragments on the system (as required by our scheme). Stumm and Zhou [Stum90] 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 [Rich93]. 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.

6.6. 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 [Wu90a].

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 [Chai90].

The simulator faithfully implements the checkpointing phase of both recovery schemes. We have implemented the simulator both for the block-level DSM system and the hierarchical DSM system. 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 size of the data being transmitted, and that the size of a process' register state is 128 bytes. Simulations were performed for a range of block sizes to evaluate the block-level schemes. For the hierarchical DSM system, simulations were performed for a page size of 1 KB and various block sizes. For coordinated checkpointing the impact of varying the checkpointing frequency was measured. With the block-level scheme, the system allows a maximum of 512 blocks per node to be in a shared state, using a 512 entry Shared Blocks Table (SBT). Similarly, with the hierarchical system, a maximum of 64 pages per node may contain a block in a shared state, using a 64 entry Shared Pages Table (SPT) [Tami92].

6.6.1. Results for the Block-Level DSM system

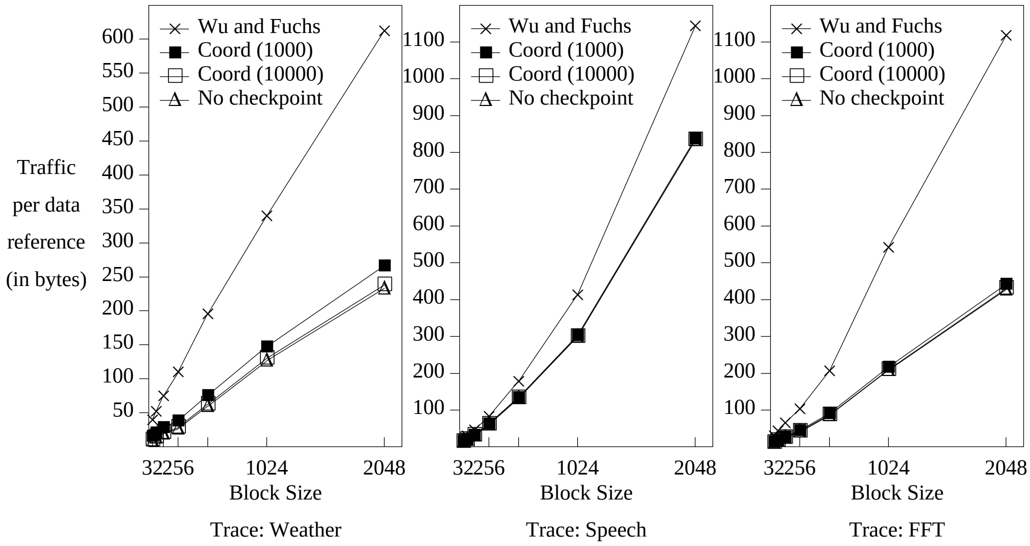
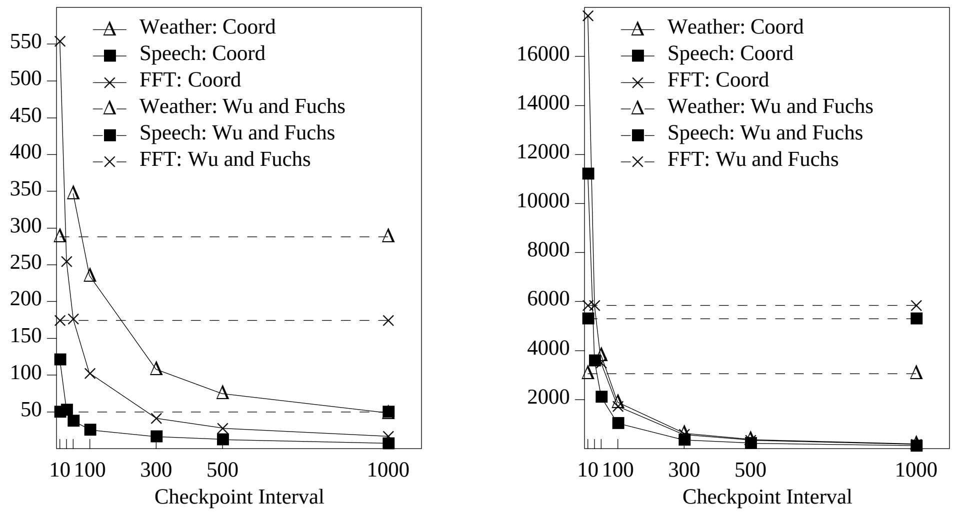


Figure 6.6: Network traffic in a block-level system as a function of block size. Results for a system without checkpointing, with the Wu and Fuchs [Wu90a] scheme, and with coordinated checkpointing. With coordinated checkpointing, results for two checkpointing intervals: every 1000 and every 10000 references by any one processor.

With a block-level DSM system, the simulation results for coordinated checkpointing and for the Wu and Fuchs scheme [Wu90a] are shown in Figures 6.6 and 6.7. Figure 6.6 shows the network traffic per data reference as a function of the block size. In all three applications, the additional traffic due to coordinated checkpointing is low relative to the traffic without checkpointing for a wide range of block sizes. The Wu and Fuchs scheme [Wu90a] generates significantly higher network traffic than the coordinated checkpointing scheme, especially with larger blocks. Larger blocks increase false-sharing, thus increasing the frequency of read-write sharing. With the Wu and Fuchs scheme, this forces an increase in the frequency of checkpointing and thus increased overhead. Hence, the Wu and

Fuchs scheme will not be a good choice for DSM systems that use large blocks (e.g., 1 KB blocks are used in [Li89a] and [Wu90a]).



a) Relative traffic overhead as percentage of normal traffic

b) Node checkpoints per 10^5 system references

Figure 6.7: The impact of the checkpointing interval in a block-level system with 128 byte blocks. The relative traffic overhead is the additional traffic due to checkpointing relative to the traffic without checkpointing. 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 Wu and Fuchs scheme, the checkpointing interval is determined by the behavior of the application and cannot be set by the system.

Figure 6.7 shows the impact of changing the checkpointing interval for a system with 128 byte blocks. For coordinated checkpointing, as the checkpointing frequency is decreased, the overhead decreases approximately linearly. Hence, if the requirements of the application (e.g., real-time response specifications) allow, the overhead for coordinated checkpointing can be reduced by simply lowering the checkpointing frequency. With the Wu and Fuchs scheme, the checkpointing interval is a characteristic of the application and cannot be set by the system. Hence, the checkpointing overhead is fixed and high regardless of the needs of the application. Thus, these results demonstrate the benefits of being able to tune the frequency of checkpoints in an *application-independent* manner.

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 [Tami91]. In this case, volatile checkpoints are taken locally on each node and are spooled gradually to stable storage once normal operation resumes.

6.6.2. Results for the Hierarchical DSM System

As shown in Figure 6.8, in terms of network traffic overhead, the coordinated checkpointing scheme retains its advantage over the Wu and Fuchs scheme even on the hierarchical DSM system.

Even without checkpointing, for the same block size, the hierarchical DSM system often requires higher network bandwidth than a block-level system. As shown in Figure 6.8, this disadvantage of the hierarchical system is often exacerbated in a system with checkpointing. Both coordinated checkpointing and the Wu and Fuchs scheme incur higher checkpointing traffic on the hierarchical system. The additional traffic on the hierarchical DSM system is due to its support for page-level mapping and transfers. Since private data is mapped in pages, they must be checkpointed in pages rather than blocks. Also, when a private page that

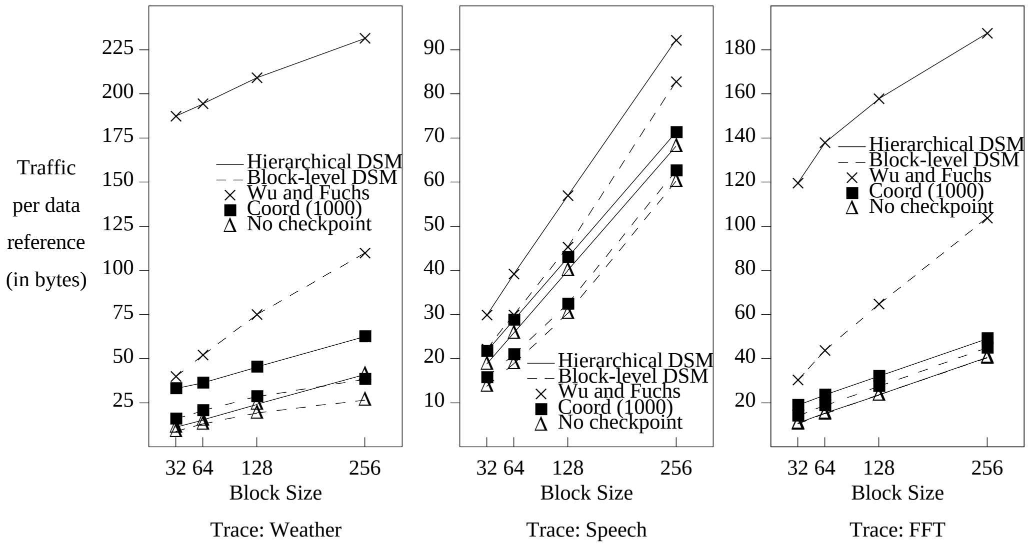


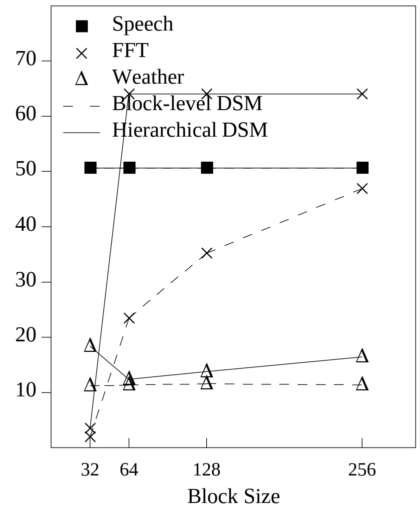
Figure 6.8: The impact of block size on network traffic for the hierarchical and the block level systems with coordinated checkpointing and with the Wu and Fuchs scheme. The hierarchical system uses 1 K byte pages. A checkpointing interval of 1000 references is used with coordinated checkpointing. For a given block size, network traffic, with and without checkpointing, is higher in the hierarchical system than the block-level system.

has its dsc bit set moves to the shared state, the dsc bits of all blocks are set, even if only a few blocks were modified. In both cases, a page of blocks may be checkpointed when only a few of the blocks actually require checkpointing.

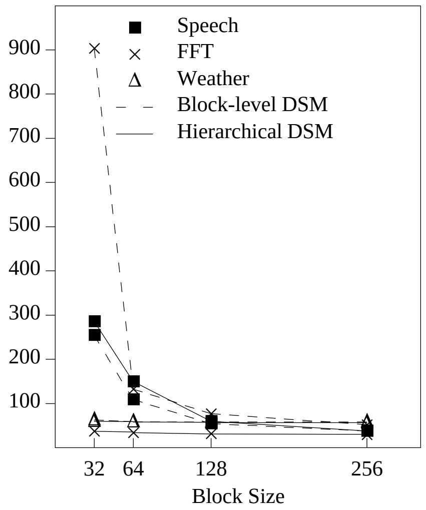
The absence of block-level dsc bits for private pages in the hierarchical system can lead to larger interacting sets with coordinated checkpointing and more frequent checkpointing with the Wu and Fuchs scheme. The dsc bit of a private page is set when it is modified. If the page is subsequently read by another process, the page is transferred to a shared state and the dsc bits of all the blocks of the page are set. With the Wu and Fuchs scheme, the process that modified the page must checkpoint. With coordinated checkpointing, the process reading the page must be included in the interacting set. In the block-level DSM system, a read issued from another process would necessitate these actions only if the block being read was modified.

Figure 6.9 shows that the average size of the interacting set with coordinated checkpointing and the checkpointing frequency with the Wu and Fuchs recovery scheme are generally higher on the hierarchical system than on the block-level system using the same block size. With coordinated checkpointing at an interval of 10000 processor references (Figure 6.9-a), both FFT and Weather have larger interacting sets on the hierarchical system, especially at larger block sizes. With the Weather application, there is the somewhat surprising phenomenon of the interacting set size initially *decreasing* when the block size of the hierarchical system is increased. Such behavior is possible when false sharing forces the checkpoint of the sub-blocks that comprise the large block. This effectively clears the dsc bit that would have been associated with each of the sub-blocks if the system used smaller blocks. With smaller blocks, the dsc bits of these sub-blocks would remain set, causing the interacting set of subsequent checkpoint sessions to be larger than when the large blocks are used. With Speech, within the range of block sizes studied, larger blocks do not cause significant additional false sharing. Hence, with coordinated checkpointing, the interacting set size is not affected by the block size or by the use of the hierarchical scheme versus the block-level scheme. With the Wu and Fuchs scheme, the expected decrease in checkpointing interval (increase in checkpointing frequency) on the hierarchical system is observed only for FFT (Figure 6.9-b).

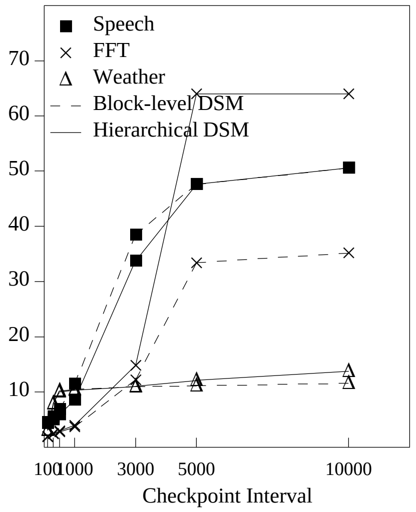
Without tracking interprocess dependencies, every checkpoint or rollback



a) Interacting set size, coordinated checkpointing, checkpoint interval of 10000 references



b) References per processor between checkpoints, Wu and Fuchs scheme



c) Interacting set size, coordinated checkpointing, 128 byte blocks

Figure 6.9: The impact of the hierarchical scheme on the interacting set size with coordinated checkpointing and on checkpointing frequency with the Wu and Fuchs scheme. The hierarchical system uses 1 KB pages.

must be global — involving all the processes of the task. Figures 6.9-b and 6.9-c demonstrate that with real applications tracking dependencies can reduce the size of the set of processes that need to be checkpointed or rolled back together. As shown in these figures, the precise beneficial impact (if any) of tracking dependencies depends on the application, the checkpointing interval, the block size, and whether a hierarchical or block-level scheme is used.

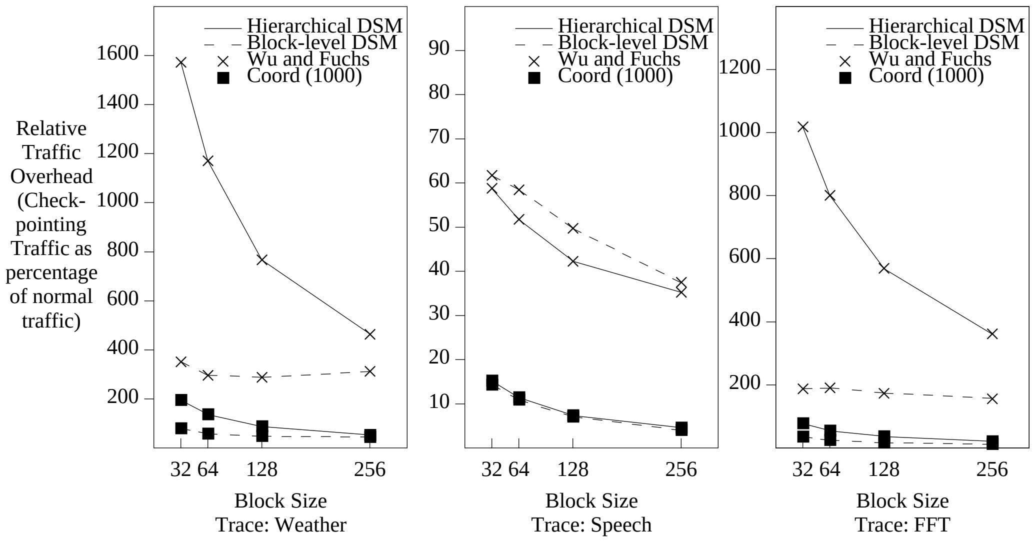


Figure 6.10: Comparison of relative traffic overhead in the hierarchical and block-level systems at various block sizes. The relative traffic overhead is the additional traffic due to checkpointing expressed as a percentage of the traffic without checkpointing. Results for the Wu and Fuchs scheme and the the coordinated checkpointing scheme. Coordinated checkpointing scheme results are shown at a checkpoint interval of 1000 per-processor references. The hierarchical system uses 1K byte pages.

Figure 6.10 compares the relative traffic overhead due to checkpointing in the hierarchical and block-level DSM systems. The relative traffic overhead is the additional traffic due to checkpointing expressed as a percentage of traffic without checkpointing. For Weather and FFT, the relative traffic overhead with both checkpointing schemes is higher in the hierarchical system. As previously discussed, one of the key reasons for the increase in checkpointing traffic in the hierarchical system is that often whole pages instead of blocks are checkpointed. Specifically, with these applications, 80-90% of the data that is checkpointed is checkpointed as pages. With the Speech application, traffic due to checkpointing is small in proportion to the traffic on a system without checkpointing. The heavy read-sharing of a dictionary in Speech causes the traffic without checkpointing to be higher on the hierarchical system than on the block-level system. However, since this data is not modified during most of the application, it does not need to be checkpointed so there is little change in checkpointing traffic between the hierarchical and block-level systems.

Figure 6.7-b, shows that checkpoints occur far less frequently with coordinated checkpointing using an interval of 1000 references than with the Wu and Fuchs scheme. As a result, as shown in Figure 6.10, the increase in overhead in the hierarchical system is significantly smaller with coordinated checkpointing than with the Wu and Fuchs scheme.

The difference between the relative traffic overhead of checkpointing on the hierarchical system and on the block-level system decreases with increasing block size (Figure 6.10). This is due to the fact that one of the key reasons for the increase in checkpointing traffic in the hierarchical system is that often whole pages instead of blocks are checkpointed. As the block size decreases, this effect becomes more significant. Thus, for small blocks the negative impact of page-level management in the hierarchical system is more severe on the checkpointing performance than on the performance of the non-checkpointing system.