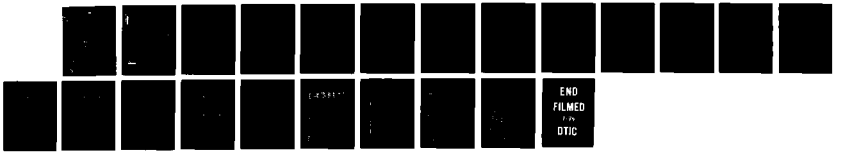
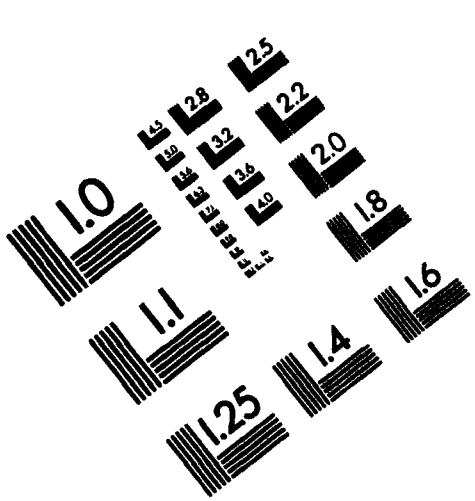
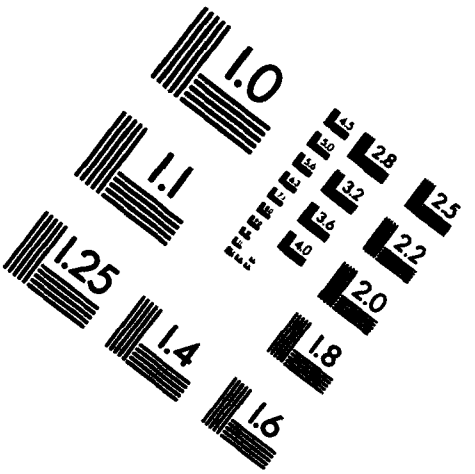
AD-A281 583 CHECKPOINTING AND ROLLBACK RECOVERY IN DISTRIBUTED SHARED MEMORY SYSTEMS(U) ILLINOIS UNIV AT URBANA CENTER FOR RELIABLE AND HIGH-PERFORMANCE COMPUTING 1/1

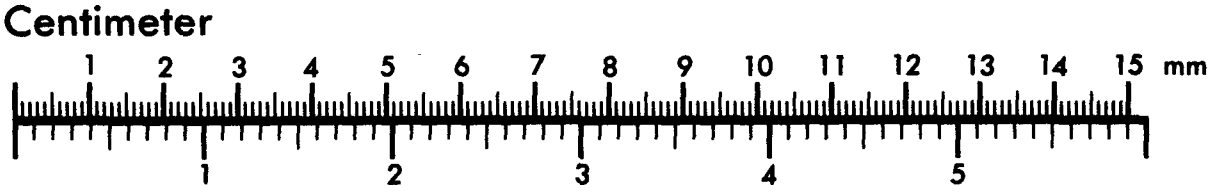
UNCLASSIFIED B JANSSENS ET AL. MAR 94 XB-ONR NL

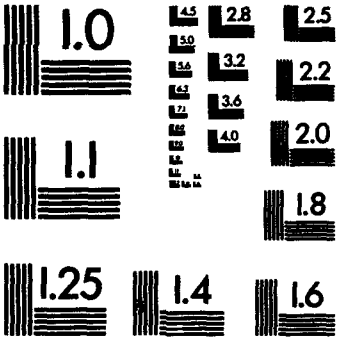


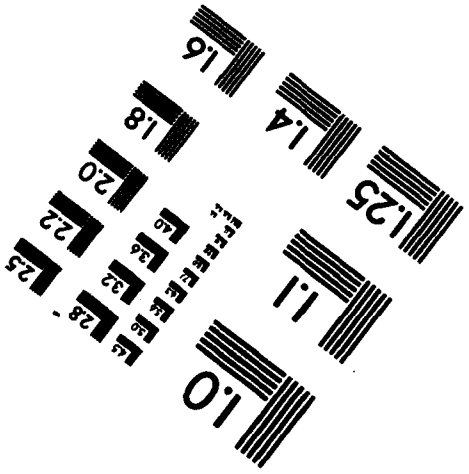


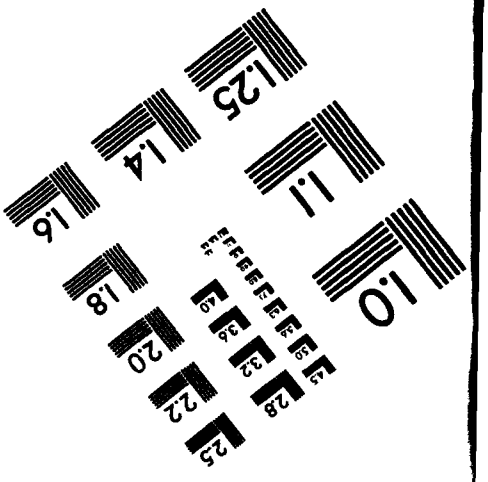
Association for Information and Image Management
1100 Wayne Avenue, Suite 1100
Silver Spring, Maryland 20910
301/587-8202











MANUFACTURED TO AIIM STANDARDS
BY APPLIED IMAGE, INC.

SUBMITTED TO: 13TH SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS

AD-A281 583

CHECKPOINTING AND ROLLBACK RECOVERY IN DISTRIBUTED SHARED MEMORY SYSTEMS

Bob Janssens and W. Kent Fuchs

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois
1308 West Main Street
Urbana, IL 61801

March 1994

Contact: Bob Janssens
Phone: 217/244-8294
email: janssens@uiuc.edu
FAX: 217/244-5686

DTIC ELECTE JUL 15 1994 S B D

Abstract

Checkpointing techniques in parallel systems use dependency tracking and/or message logging to ensure that a system rolls back to a consistent state. Traditional dependency tracking in distributed shared memory systems (DSM) is expensive because of high frequency of communication. In this paper we show that, because of information redundancy, not all message-passing dependences need to be considered to roll back to a consistent state in DSM systems, resulting in reduced dependency tracking overhead and reduced potential for rollback propagation. We develop a model of execution where client processes running an application interact atomically with a set of shared-memory server processes on every access to shared data. We show that under this model, dependences are significantly reduced over the message-passing model. We use results from simulations with multiprocessor address traces to demonstrate the reduction in dependences.

94-22067

This research was supported in part by the Office of Naval Research under contract N00014-91-J-1283, and by the National Aeronautics and Space Administration (NASA) under Grant NASA NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS).

DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE: DISTRIBUTION IS UNLIMITED

94 7 14 017

1 Introduction

Checkpointing and rollback recovery techniques are extensively used to allow a computation to make progress in spite of failures. A large body of research exists on the application of checkpointing and rollback recovery to message-passing systems with an emphasis on maintaining consistency without allowing excessive rollback propagation. The problem of maintaining consistency in recoverable shared memory systems is not as well-understood. Distributed shared memory (DSM) systems use cache coherence hardware [13] or shared virtual memory software [14] to provide a shared memory image on top of message passing. In DSM systems, communication frequency is much higher than in pure message-passing systems. Therefore, using traditional message-passing recovery techniques incurs a high overhead in tracking dependences between processes. Furthermore, the large number of dependences increase the likelyhood of the need to propagate rollbacks to maintain consistency.

In this paper, we show that, by considering information redundancies in DSM systems, the number of dependences, and therefore the overhead of maintaining consistency with rollbacks, can be significantly reduced. We model a DSM system as a set of client processes running an application program that interact atomically with a set of shared-memory server processes on every access to shared data. We show that, under this model, many of the messages transmitted during interactions do not result in dependences between processors, and therefore do not have to be considered when rolling back to a consistent state. We back our claims with results from simulations using multiprocessor address traces.

Dependences carried by messages have to be considered in any approach to checkpointing and rollback recovery for distributed systems. Even in fully *coordinated checkpointing* [6, 7, 12, 15], where all processes synchronize to take a global checkpoint, dependences may cause additional overhead by aborting tentative checkpoints. The coordination algorithm can be improved by adding dependency tracking. Then only processes that have dependences in the current checkpoint interval need to synchronize checkpointing and rollback [12]. Due to process independence, recovery efficiency, or I/O bandwidth requirements it may not be desirable to synchronize checkpoints. *Independent checkpointing* replaces synchronization by dependency tracking and/or message logging, both of which introduce overhead for every dependence-carrying

message [5, 8, 18, 20]. Every dependence-carrying message also introduces a chance of rollback propagation, which can escalate into the domino effect. Some schemes use *communication-induced* checkpointing to bound rollback propagation where messages can induce checkpoints on other processors. In these schemes, the large overhead of taking checkpoints is affected by the pattern of dependences.

Various distributed system recovery techniques have been applied to shared memory. Communication-induced checkpointing is used in the Sequoia system [4], by Wu *et al.* in both bus-based multiprocessors [22] and software DSM systems [23], and by Janssens and Fuchs in DSM systems with relaxed consistency [11]. Ahmed *et al.* presented three schemes for bus-based systems that use fully coordinated, partially coordinated and communication-induced checkpointing respectively [2]. Banâtre *et al.* have proposed a scheme that uses dependency tracking at the shared memory in a bus-based system to implement partially coordinated checkpointing [3]. Richard and Singhal have proposed using independent checkpointing and logging of all memory accesses to implement recovery in piecewise deterministic DSM systems [17].

It is obvious that the dependences of message-passing are too strict for shared-memory parallel programs. For instance, two reads by different processes to a shared variable with no intervening writes do not depend on each other even though both processes exchange messages with the shared memory element. In the literature on replay for debugging in shared memory systems, a dependence from memory access a to memory access b is generally said to exist if a accesses a shared variable that b later accesses, and at least one of the two accesses is a write [16]. Gunaseelan and LeBlanc have recently argued that a write causes a two-way dependence with a memory element, while a read only causes a dependence from the memory to the process [10]. Therefore there is no dependence from a read to a write if the read precedes the write.

A more relaxed dependency model than that for message-passing can be used for rollback recovery only if there is no possibility of deadlock due to processes waiting for messages that may never arrive. In bus-based systems, where bounded transmission delay eliminates the need for acknowledgements, deadlock is avoided. In DSM systems, however, other measures have to be taken to avoid messaging deadlock. In the bus-based recovery scheme of Banâtre *et al*, a dependence is recorded between any processor that writes a data item and another that reads it. A bi-directional dependence is recorded between two processors that

write a data item consecutively [9]. Wu *et al.*'s recovery scheme takes a checkpoint of the originating process *and* of the data item accessed on every data transfer between processing nodes [22, 23]. The effect of both schemes is to conform to the dependence relation described by Gunaseelan and LeBlanc [10]. A previous paper by us also used similar assumptions about shared-memory dependences to reduce the number of checkpoint needed to avoid rollback propagation in DSM systems with a relaxed memory consistency model [11]. In this paper we validate the dependence assumptions made in previous work, by showing how dependences can be removed in a DSM system without affecting correct execution after rollback.

2 Motivation

To motivate our approach to reducing overhead by decreasing the number of dependence-carrying messages, we present an example of a read and a write access in a typical DSM system. The system consists of a number of processing nodes which cache copies of the shared variables they reference. The shared memory is divided into blocks, each of which has a home node to manage ownership rights. As shown in Figure 1, the memory block p is managed by node M and accessed by nodes A and B. The example uses Li's fixed distributed manager algorithm for shared virtual memory to maintain coherence [14]. Since their operation is similar, the example also applies to hardware DSM systems using a distributed directory based coherence protocol [1, 13].

In our example in Figure 1, memory block p, which contains variable x, is originally cached in a read-only state by nodes A, B and C. A write to variable x by the user process on node A causes it to send the new value of x to its local server for block p and wait. If this server had write permission, it would update its state and immediately return an acknowledgement to the user process. Instead, it sends a message asking for write access to the manager for block p on node M. The manager forwards the request to the owner of p, in this case node B, and then notes that node A is now the owner of p. The server on node B sets its access permission to none for block p, and sends a copy of block p, together with a list of all nodes that have access to it, to node A. Upon receipt of the message, the server on node A knows that C still has access to

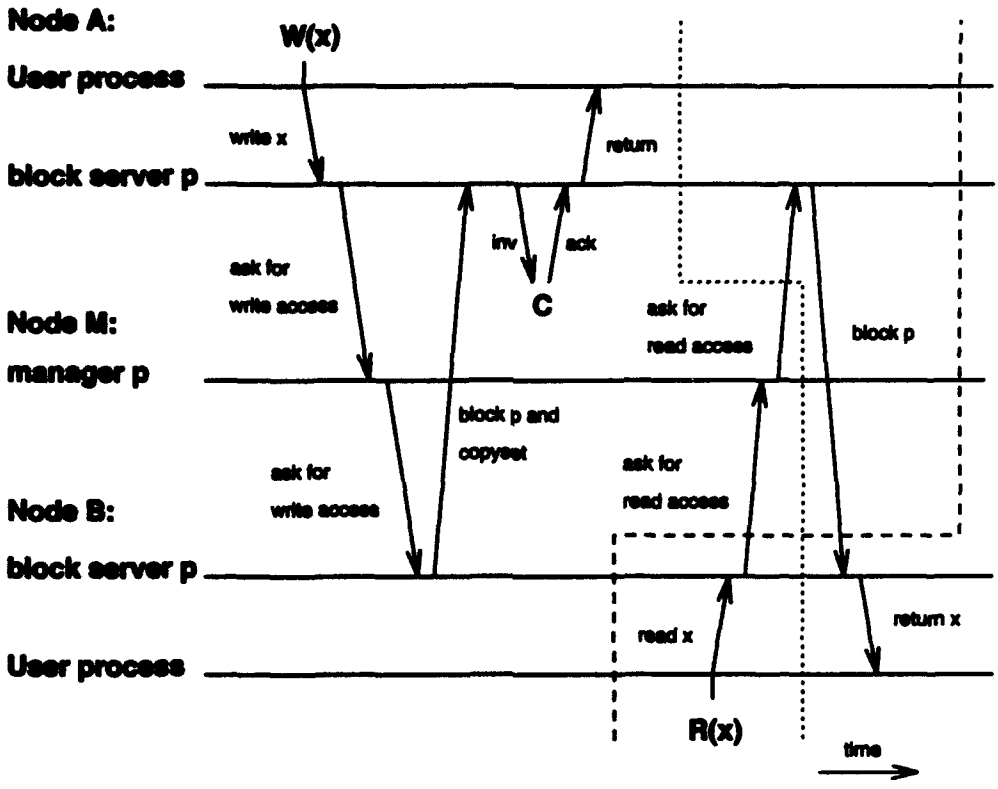


Figure 1: Client-server interactions in distributed shared memory.

the block, so it sends an invalidation message to node C. The server on node C erases its access permission to p and returns an acknowledgement. Once the server on node A receives the acknowledgement, it notifies the user process, which resumes execution.

When the user process on node B later decides to read variable x, it sends a read request to its server for block p. Since it does not have read permission, this server asks the manager of p for a readable copy. Since there is a possibly dirty copy of p on node A, the manager forwards the request to node A. The server on node A decreases its access permission to read-only, notes that node B now has access to p, and sends a copy of p to the server of node B. Upon receiving its copy of p, the server on node B returns the value of variable x to the user process.

One way to ensure that the DSM system in our example rolls back to a consistent state is to treat every message sent between nodes as a dependence-carrying message. In our example, the system would have to track 8 dependences between 4 processing nodes for the two memory accesses shown. Assume that, after

completing the read access, node B detected an error, rolled back, and restored its state from a checkpoint it took before the read access. After node B's rollback, the global state of the system would be that indicated by the dashed recovery line in Figure 1. Because the resultant global state reflects the receipt of a message by node M, but not its sending by node B, it is not consistent in terms of message passing. To maintain a consistent state, both nodes A and M would also have to roll back to a point before the read.

However, in terms of shared memory, there is no reason for the state indicated by the dashed line to be inconsistent. In fact, nodes A and M only supplied data in response to the request for read access; they did not change their internal state. Therefore, rolling them back is superfluous. The request messages from node B to node M and from node M to node A therefore do not carry any permanent dependences. The reply message supplies a block of data from node A to node B and therefore does carry a dependence. In the message-passing model, since this message crosses the recovery line, either it has to be retrievable after rollback (for instance, by logging it), or the dependence has to be considered bi-directional [6, 12, 20]. However, after rollback node B can simply send a new request to for block p when it re-executes the read access. Therefore the dependence from node A to node B can be considered unidirectional, even if the message can not be retrieved after rollback.

The only permanent dependence in our example is from node A to node B. However, there are temporary dependences that can cause incorrect execution or deadlock when nodes roll back while a request is being serviced. Consider the global state indicated by the dotted line in Figure 1. This state corresponds to node A detecting an error while servicing the read request and rolling back to a checkpoint taken prior to servicing the read request. In this case the two request messages have to be considered as dependences and nodes B and M need to roll back. Otherwise node B would wait forever to receive a reply from node A after it sent its request for read access. However, if node A had rolled back after servicing the read request, node B would have already received its reply from node A, and there would not be a consistency problem. Therefore the request messages introduce a temporary but not a permanent dependence.

From the example just discussed, it is clear that not all message-passing dependences need to be considered for correct rollback in a DSM system. Every message introduces a temporary dependence. But,

because of the request-reply nature of communication in DSM, some of the temporary dependences disappear after an interaction between processors is complete. To determine which messages cause permanent dependences, we need to a model of communication in DSM systems. Such a model is described in the next section.

3 A Passive Server Model of Distributed Shared Memory

We model program execution in DSM systems as a set of client processes which run the application program and a set of passive server processes which provide a shared-memory image to the clients. The servers are considered passive since they only change state due to interaction with a client. Processes communicate via messages sent through reliable channels. We represent the overall program execution by a pair, $P = (E, \stackrel{PD}{\longrightarrow})$, where E is a set of events and $\stackrel{PD}{\longrightarrow}$ is the *possible dependence* relation defined over E. Events within a process are ordered by the $\stackrel{XO}{\longrightarrow}$ (execution order) relation. Every event represents an atomic action which may change the state in one of the processes. A special checkpoint event can be inserted between two events to record the current state of the process.

In the clients, events can be either internal events, read events, or write events. Internal events only depend on and affect the local state of the process. Read events send a read request to a local server, wait for a reply with the value of a data item and then update their local state with the value. Write events send a write request with a value to a local server and wait for a reply. Events in servers are always triggered by the receipt of a request message, either from a client or another server. Request messages are handled by the servers in FIFO order. After the request message is received, server events may send and receive additional messages. A write or read event in a client, together with the events it causes in the servers may be collectively called an interaction. Sends and receives are ordered by the $\stackrel{M}{\longrightarrow}$ relation: $a \stackrel{M}{\longrightarrow} b$ means event a sent a message and event b received it. The $\stackrel{PD}{\longrightarrow}$ relation is the union of the other two: $\stackrel{PD}{\longrightarrow} = \stackrel{XO}{\longrightarrow} \cup \stackrel{M}{\longrightarrow}$. Figure 2 presents the two interactions in the example of Section 2 in terms of the $\stackrel{PD}{\longrightarrow}$ relation between events.

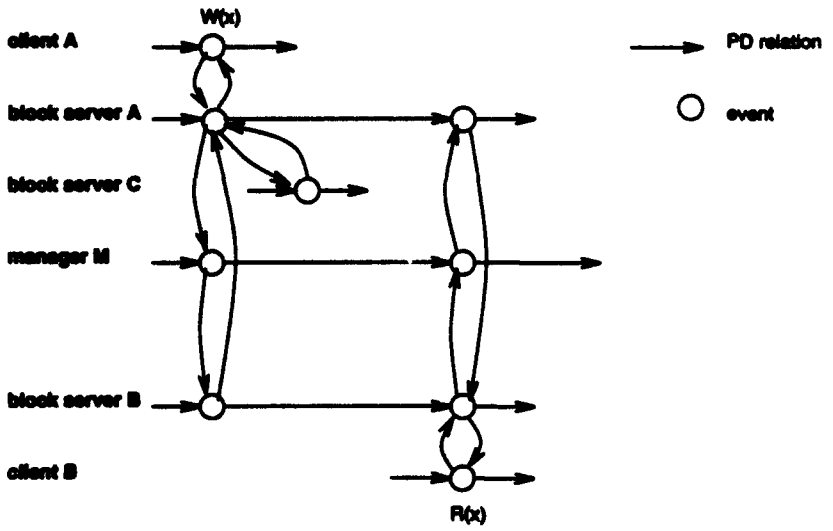


Figure 2: Write and read accesses in the passive server model.

When a process detects an error, it notifies all other processes and rolls back to a checkpoint. Upon receiving notification of a rollback, a process can either roll back to a checkpoint, or continue operation. If it continues, we can treat the current volatile state as a virtual checkpoint [21]. A global recovery line is a set of real and virtual checkpoints, one per process. Consider two events a and b, where b occurs in the execution order before the global recovery line and a occurs in the execution order after the global recovery line. A global checkpoint is consistent if there are no two such events such that $a \xrightarrow{M} b$ or $b \xrightarrow{M} a$. A global checkpoint is also consistent if lost messages can be retrieved during reexecution and there are no two events such that $a \xrightarrow{M} b$.

To simplify reasoning about consistency of global checkpoints it is useful to treat the $\xrightarrow{M}$ relation as bidirectional. To do this we replace every dependence $a \xrightarrow{M} b$, by a causal dependence $a \xrightarrow{C} b$, and a logging dependence $b \xrightarrow{L} a$. Consider again two events a and b, where b occurs in the execution order before the global recovery line and a occurs in the execution order after the global recovery line. The requirements for consistency are now that there are no two such events such that $a \xrightarrow{C} b$ and there are no two such events such that $b \xrightarrow{L} a$ and the message between a and b is unlogged. As was shown by the example in the previous section, because of the structure of events and messages, some of the $\xrightarrow{C}$ and $\xrightarrow{L}$ dependences may be eliminated. By examining the various events in a specific DSM system, we can derive

new $\xrightarrow{C'}$ and $\xrightarrow{L'}$ relations that are subsets of the old.

4 Recovery to a Consistent State in Distributed Shared Memory

We now analyze the fixed distributed manager scheme for implementing DSM systems using the passive server model. The model requires that all execution by a server done in response to a request constitute one atomic event. First we show how to satisfy this requirement in a recoverable DSM system. Then we map all the interactions that occur in the fixed distributed manager scheme into the passive server model. We show that redundancies in a DSM system allow eliminating many of the dependences between processing nodes. We consider two methods of taking advantage of information redundancy which result in different patterns of dependences. In the volatile ownership method, when ownership information is lost by a block's manager, it can be reclaimed through a broadcast to all the nodes. In the volatile access rights method, when information about access rights is lost by a node server, it can be reclaimed by contacting the block manager.

4.1 Maintaining atomicity of server events

To be able to avoid issues of deadlock caused by partially completed server operations, the passive server model treats a server's whole response to a request message as an atomic event. To maintain atomicity, a process should never roll back to a recovery line with partially completed events. Therefore all checkpoints need to be constrained to occur only outside of interactions. Furthermore, if a process participating in an interaction rolls back, all other processes currently in the interaction also need to roll back.

Interactions are relatively short-lived events; most of the time clients are executing the application and servers are waiting to process the next request. A simple and low-overhead scheme to handle the case where an error does occur in an interaction is to simply roll back every process that is involved in an interaction when it receives notice that a recovery is necessary.

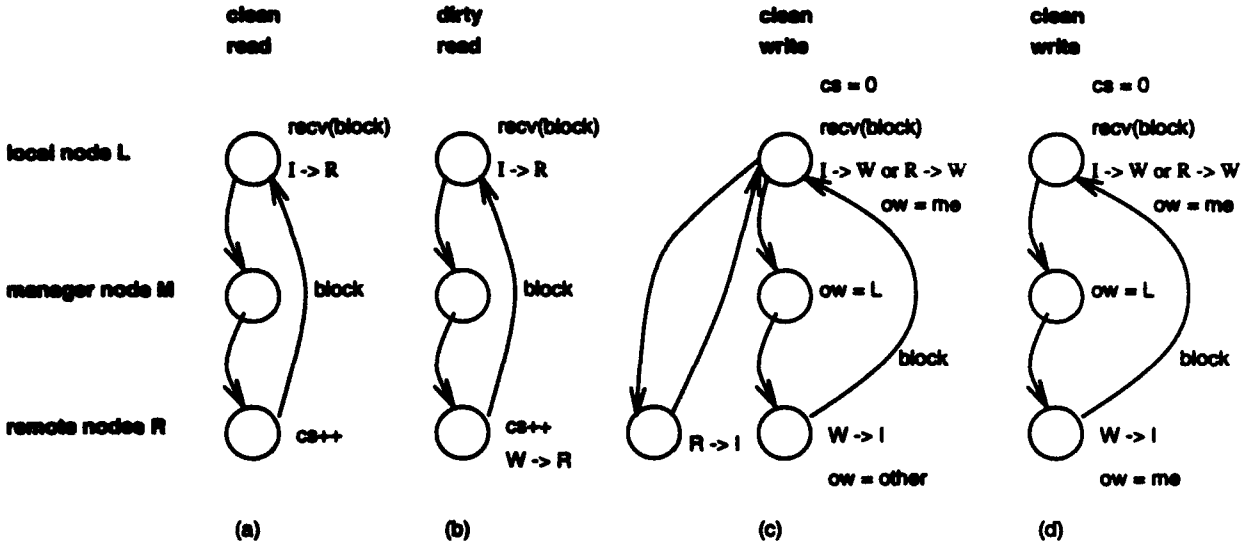


Figure 3: Possible interactions during accesses in a fixed distributed manager DSM.

4.2 Node interactions in the passive server model

Figure 3 presents all the events that can occur in the fixed distributed manager scheme during interactions between the servers on every node. Since a client process always only communicates with a server on its own node, the read or write event on the client is not shown. The dependences between the events on the different nodes are shown in terms of the $\xrightarrow{M}$ relation. Next to the graphical representation of an event, the state changes of that event are shown.

In addition to the state of the block, a block server on a node maintains three pieces of information. The access rights determine what kind of accesses a node is permitted to make. A node can either have read-write access (W), read-only access (R), or no access (I) to a block. The copyset (indicated by "cs" in the Figure) is used to eliminate the need for broadcasting when all copies of a block need to be invalidated. It keeps a record of all nodes that might possibly have read-only access to a block. The server on a node also keeps track of whether or not it is the current owner of a block. The manager for a block only maintains one piece of information, the identity of the current owner of the block.

4.3 Redundancy in DSM

Some of the possible dependences in a DSM system can be eliminated because of redundancies between nodes in the maintenance of data. There are a few specific attributes of DSM that enable the removal of dependences. We make several assumptions about the operation of DSM in our model. The first assumption is that a non-owned read-only block can be spontaneously invalidated. We also assume that write permission can be spontaneously taken away from a block to make it read-only as long as its contents are backed up on another node. A third assumption that needs to be made is that the copyset on a node may contain a superset of all the other nodes that have a read-only copy of a block. When an invalidate request is sent to a block server that does not have R access, the request is denied. It is not difficult to design a DSM system that conforms to the above assumptions. In fact, these assumptions need to by made in any DSM system where cache space on the nodes is limited. In such a system it is necessary to be able to invalidate recently unused blocks to make room for blocks of new data.

There is further redundancy between the ownership information maintained by the block manager and the access right information maintained by the block servers. The ownership information maintained by the block managers need not always be correct for correct operation of the protocol. If the manager routes a request for access to a block server that does not have ownership, the server denies the request, and the manager finds the owner through a broadcast to all nodes. Therefore it is possible to allow *volatile ownership* of blocks after rollback. Ownership information is not checkpointed, and if a manager needs to roll back, the ownership field is set to unknown. The first access to the block after the rollback will then cause the correct owner to be found through broadcast.

Alternatively, one can always maintain correct ownership information in the manager, but allow *volatile access rights*. In this case, access rights and ownership information in the block servers need not be restored after rollback [17, 23]. If a node block server needs access to a block after rollback, it contacts the block manager. The block manager then forwards the request to the node that it considers the owner of the block. This will always be the node that had ownership last before the recovery line.

One final redundancy in the DSM system is the copyset. It is used only to determine which nodes have valid copies when a block needs to be invalidated on all nodes. If the copyset is not available, an invalidate request can simply be broadcast to every node. Therefore, in both the volatile ownership and the volatile access rights cases, the copyset maintained by the block servers can also be considered volatile.

4.4 Eliminating dependences with volatile ownership

Through analysis of the specific interactions in the volatile ownership case, where ownership and copyset information is lost after rollback, it is possible to eliminate some of the message-passing dependences.

Consider a remote read access to a clean block in the fixed distributed manager protocol (Figure 3a). The message-passing dependences between local (L), manager (M), and remote (R) nodes are: $L \xrightarrow{C} M$, $M \xrightarrow{L} L$, $M \xrightarrow{C} R$, $R \xrightarrow{L} M$, $R \xrightarrow{C} L$, and $L \xrightarrow{L} R$. The state of node R is not affected by the interaction, except for the addition of a member to the copyset. If node R node rolls back, only the copyset information is lost and can be recovered through broadcast. So $R \xrightarrow{L} M$ can be eliminated. If nodes L and M roll back, the state of the recovery line is the same as if all three nodes rolled back, except for the extra member of the copyset. Since one of our assumptions allows extra members of the copyset, the resulting state is consistent. So $M \xrightarrow{C} R$ can be eliminated. Since node M does not change state at all during the interaction, $L \xrightarrow{C} M$ and $M \xrightarrow{L} L$ can also be eliminated. The only remaining dependences are between node R and node L. Assume node L rolls back. The recovery line is equivalent to the earlier treated case where both nodes L and M roll back, so it is consistent. Thus $L \xrightarrow{L} R$ is eliminated. The final causal dependence, $R \xrightarrow{C} L$, can not be eliminated since a block of data is transmitted from node R to node L.

Now consider a remote read access to a dirty block (Figure 3b). Node M merely forwards the request and does not change state. Therefore the only possible dependences are between nodes L and R. If node L rolls back, node R has read-only access to a modified block. However, node R is still the owner of the block, so any further requests for the block will always be routed to and serviced by node R. Therefore $R \xrightarrow{L} L$ can be eliminated. Again, the causal dependence $L \xrightarrow{C} R$ has to be maintained.

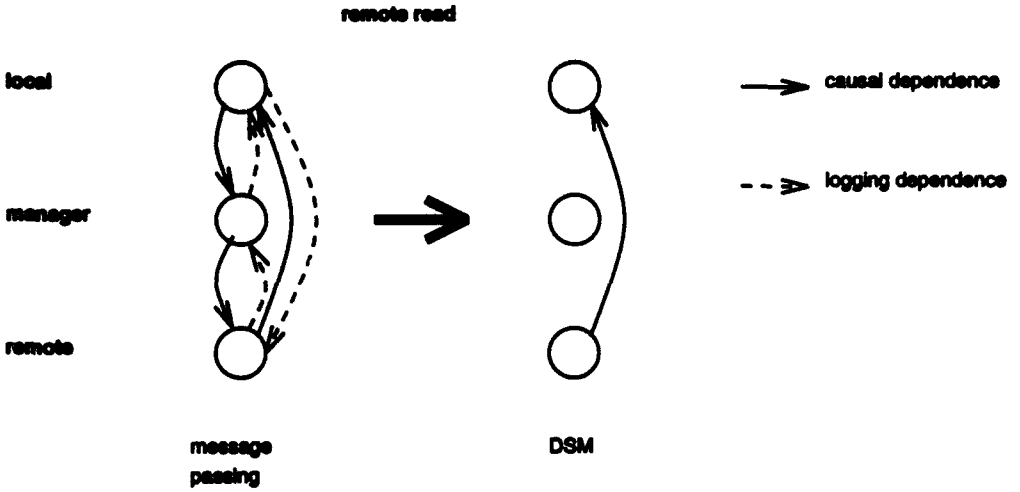


Figure 4: Reducing dependences on remote reads with volatile ownership.

Figure 4 shows the dependences on read access in the message-passing model, and with our model. They have decreased from three causal and three logging dependences between three nodes to just one causal dependence between two nodes.

Next, we consider a remote write access (Figure 3c, 3d). Ignoring invalidations, the message-passing dependences are the same as in the read accesses. In a remote write interaction, node M changes state; it records the new owner of the block. However, since ownership information in the manager is volatile, we can again ignore the dependences to node M. If node L rolls back, the recovery line may have no node claiming ownership of the block, so even with a broadcast no owner will be found. So the dependence $R \xrightarrow{L} L$ remains. The causal dependence $L \xrightarrow{C} R$ also remains since it transmits a block of data.

If the block is readable by more than one remote node when the local node asks for write access, all the copies in the remote nodes will be invalidated. One of the assumptions we made earlier is that the system can handle the spontaneous invalidation of a read-only block when it is not owned by the node on which it resides. Therefore node L can roll back past an interaction in which it invalidated a remote node R'. At the recovery line, it will simply appear as if node R' has been invalidated spontaneously. Therefore there is no dependence $L \longrightarrow R'$. If node R' rolls back, however, there is a possibility that both read-only and writable copies exist in the system. This is not allowable in the coherence protocol. If the invalidation message is logged, however, it can be replayed after the remote node rolls back, guaranteeing there exists

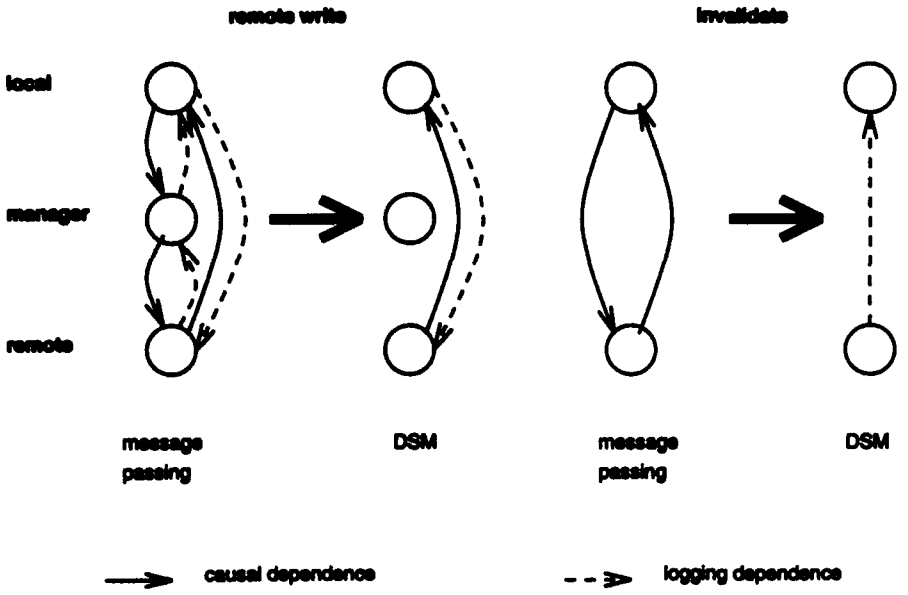


Figure 5: Reducing dependences on remote writes and invalidates with volatile ownership.

only a writable copy of the block in the system. So the causal dependence $R' \xrightarrow{C} L$ can be eliminated, but there remains a logging dependence $R' \xrightarrow{L} L$.

Figure 5 shows the dependences on a remote write access in the message-passing model, and with our model. The dependences that occur on each invalidation message are shown separately. Our model decreases the dependences on a remote write access from three causal and three logging dependences between three nodes to one causal and one logging dependence between two nodes. In addition, for any invalidation that needs to be performed, it decreases the dependences from two causal dependences to one logging dependence.

4.5 Eliminating dependences with volatile access rights

With different assumptions about which information is lost upon rollbacks, different dependences exist between nodes. We now analyze the volatile access rights case, where the block servers lose access rights, ownership, and copyset information upon rollback, but the block managers do retain ownership information. Upon access to a node that has lost access rights due to rollback, the manager is consulted for information

about the ownership of a node. The manager then requests the page from the node it considers the owner. This node then supplies the page to the requesting node.

Again we have to consider all the cases of remote accesses. On a remote read access to a clean block (Figure 3a), the state of manager node M is not affected. Therefore, as in previous cases, the dependences with node M can be eliminated. When node L rolls back, the state of the recovery line is the same as if node R also rolled back, except for the extra member of the copyset. Since the copyset is allowed to be a superset of all the nodes that have readable copies, the recovery line is consistent. So the dependence $L \xrightarrow{L} R$ can be eliminated. When the remote read access is to a dirty block (Figure 3b), a rollback of node L will cause a situation where node R has lost write permission without guaranteeing that a copy of the dirty page has been saved on another node. However, the manager still considers node R the owner, so any further requests will be supplied from its copy of the block. Therefore the dependence $L \xrightarrow{L} R$ can again be eliminated. So, on a remote read, there remains only the causal dependence, $R \xrightarrow{C} L$, from the remote node to the local node.

On a remote write access (Figure 3c, 3d), without the volatile ownership assumption, it is now necessary to consider the dependences with the manager. If node L rolls back past the interaction, but node M does not, node M maintains that ownership belongs to node L, even though node L may have an out-of-date copy of the block. Therefore the dependence, $L \xrightarrow{C} M$, between the local node and the manager can not be eliminated. If node M rolls back paste the interaction, but node L does not, node M maintains that ownership does not belong to node L, even though node L has the latest copy of the block. So the dependence $M \xrightarrow{L} L$ also remains. The dependences between node M and node R are removable. Since there is already a two-way dependence between events on nodes L and R, and between events on nodes L and M, the two-way dependence between events on nodes M and R is superfluous. If either node R or node L rolls back, node M also rolls back because of causal dependences $R \xrightarrow{C} L$ and $L \xrightarrow{C} M$. If node M rolls back, the logging dependences $M \xrightarrow{L} L$ and $L \xrightarrow{L} R$ assure that the state of node M is restored correctly.

The only dependences left to consider are those due to invalidations of remote read copies by the

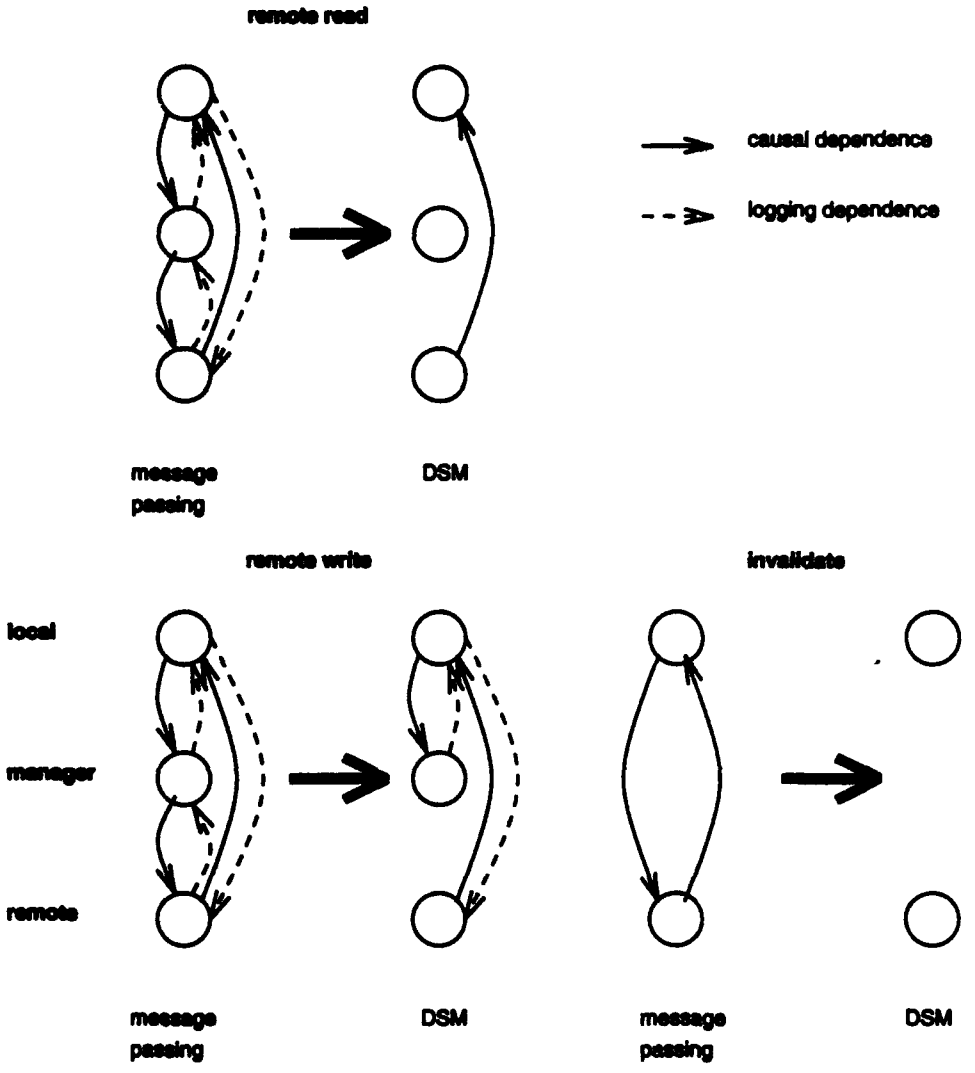


Figure 6: Reducing dependences with volatile access rights

local node L. Node L can roll back past an interaction in which it invalidated a remote node R' since one of our assumptions allows non-owned read-only nodes to be invalidated spontaneously. Without volatile ownership, it was not allowable to roll back node R' past an invalidation without replaying the invalidation message from a log. However, if ownership is not volatile upon rollback, node R' always gets a new copy from the owner on the first access to a block. In that case there is no possibility of inconsistency. Therefore the remaining $R' \xrightarrow{L} L$ dependence is eliminated, resulting in a dependence-free invalidation interaction.

Figure 6 shows the dependences eliminated when using the volatile access rights assumption As before,

on a remote read access, the three causal and three logging dependences are reduced to just one causal dependence. On a remote write access, it is not possible to remove all dependences with the manager. But all dependences on invalidations are eliminated.

4.6 Implementation Issues

A DSM system consists of a number of processing nodes connected by a high-speed network. In a typical shared virtual memory (software DSM) system [14], all of a node's block servers and block managers are implemented in a single entity, inside or outside the operating system kernel. Access right and ownership information is maintained transparently to the user, in page tables. In some hardware DSM systems, block server access rights information is kept in the directories of the processor caches, while centralized directories at the main memory elements are used to store block manager ownership information [1]. In an alterative hardware approach, directories at the node/network interface server both as block servers and block managers [13]. The detection of an error on a processing node forces the rollback of the node, including its directories and page tables. Similarly, the forced rollback, due to rollback propagation, of any block server or block manager on a node forces the rollback of the complete node.

When using the volatile ownership checkpointing and rollback method, it is not necessary to checkpoint ownership information maintained by the block managers. Upon rollback, all ownership table entries are set to unknown, and necessary ownership information is recovered through broadcast. This may simplify implementation in hardware DSM, because ownership information in the central directories need not be checkpointed. A more important advantage of the volatile ownership method is that the block managers are not involved in any dependences. Therefore ownership tables do not need to participate in dependency tracking. Rollback propagation is reduced, because there is no possibility of introducing a dependence with a node through interaction with its block manager.

The volatile access rights method has the advantage that access right information never needs to be saved in a checkpoint. Besides the actual computational state of the node processes, only ownership tables need

Table 1: Address trace characteristics.

program	description	tot. num. of references	data reads		data writes	
			total	shared	total	shared
gravsim	N-body simulator	92,178,814	33,266,880	12,484,455	6,392,078	251,694
faim	fault simulator	149,918,375	50,950,933	39,326,911	3,958,919	999,127
tgen	test generator	101,264,382	32,613,809	16,550,450	4,461,889	642,796
pace	circuit extractor	87,861,165	23,266,576	1,286,787	7,842,338	348,524
phigure	global router	132,998,231	38,244,233	4,281,207	11,530,981	1,876,400

to be checkpointed. In a shared virtual memory system, where page tables may reside in the system address space, this simplifies the task of user-level checkpointing of a node. In cache-based systems the cache directory is usually inaccessible for checkpointing except through special hardware. The volatile access rights method avoids the need for such special hardware. Invalidations do not cause any dependences under the volatile access rights method. Since invalidations usually occur on multiple nodes, this reduces the chance of excessive rollback propagation to many nodes.

5 Dependence Frequency Measurements

To evaluate the effectiveness in reducing dependences of our schemes, we performed trace-driven simulations with multiprocessor address traces from five parallel scientific programs running on an Encore Multimax. The traces are from execution on seven processors, and each contain at least 80 million memory references [19]. Table 1 describes the characteristics of the traces used.

Figure 7 presents the frequency of dependences in the message-passing model, the volatile ownership model, and the volatile access rights model. Both DSM models reduce the number of dependences significantly. The volatile ownership model slightly outperforms the volatile access rights model. An implementation using the relaxed models therefore incurs less dependency tracking overhead. In addition, the probability of rollback propagation is reduced. Both causal dependences and total dependences are plotted in Figure 8. In both DSM models the causal dependences are in the majority, but logging dependences take up a large proportion. An implementation that uses logging will not have to consider logging dependences

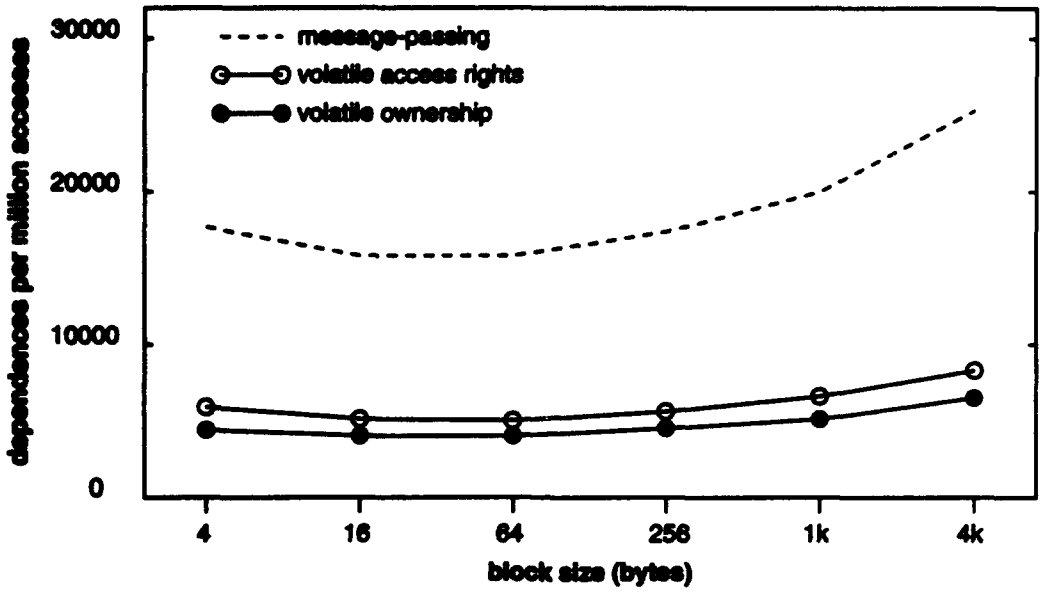


Figure 7: Frequency of dependence comparison

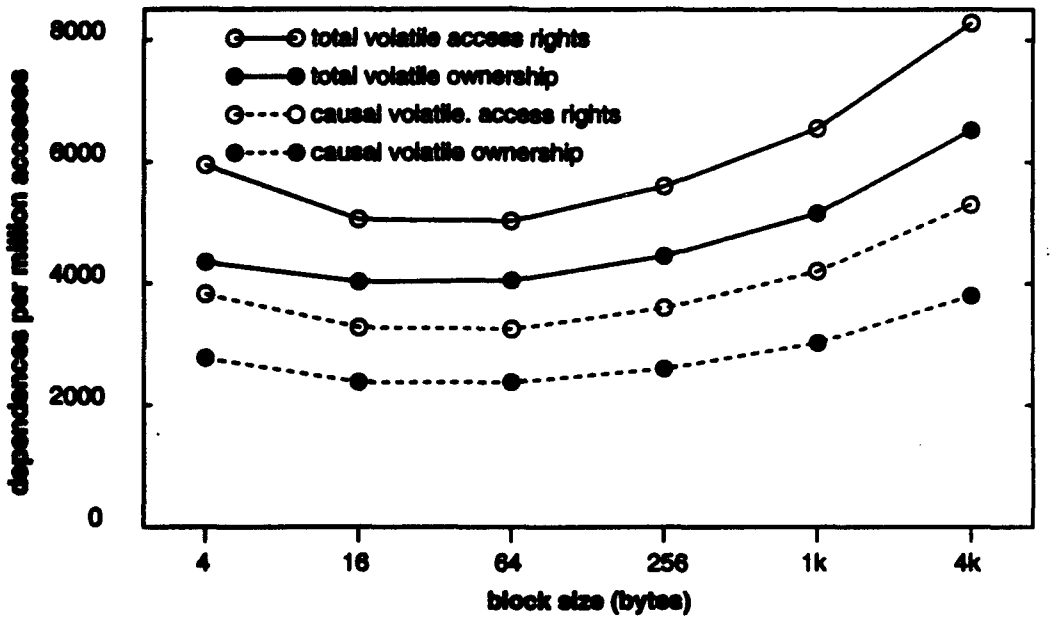


Figure 8: Frequency of causal dependences and all dependences

during rollback, further decreasing the probability of rollback propagation.

6 Summary

Since a DSM system is a specialized implementation of a message-passing system, it is possible to use message-passing dependency tracking in implementing checkpointing and rollback error recovery. However, we have shown that, by analyzing specific properties of every interaction between processors in a DSM system, it is possible to eliminate some of the message-passing dependencies. Reducing the number of dependences reduces the overhead of checkpointing methods, and the chance of needing to undo an excessive amount of work after an error due to rollback propagation. Results from trace-driven simulations illustrate the effectiveness of our methods.

References

[1] A. Agarwal, R. Simoni, J. Hennessy, M. Horowitz, "An evaluation of directory schemes for cache coherence," *Proc. 15th Int. Symp on Computer Architecture*, 1988, pp. 280–289.

[2] R. E. Ahmed, R. C. Frazier, and P. N. Marinos, "Cache-aided rollback error recovery (CARER) algorithms for shared-memory multiprocessor systems," *Proc. 20th Int. Symp. on Fault-Tolerant Computing*, 1990, pp. 82–88.

[3] M. Banâtre, A. Gefflaut, P. Joubert, P. Lee, and C. Morin, "An architecture for tolerating processor failures in shared-memory multiprocessors," Tech. Report 707, IRISA, Rennes, France, Mar. 1993.

[4] P. A. Bernstein, "Sequoia: a fault-tolerant tightly coupled multiprocessor for transaction processing," *Computer*, Vol. 21, No. 2, Feb. 1988, pp. 37–45.

[5] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, "Fault tolerance under UNIX," *ACM Trans. on Computer Systems*, Vol. 7, No. 1, Feb. 1989, pp. 1–24.

[6] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Trans. on Computer Systems*, Vol. 3, No. 1, Feb. 1985, pp. 63–75.

[7] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing," *Proc. 11th Symp. on Reliable Distributed Systems*, 1992, pp. 39–47.

[8] E. N. Elnozahy and W. Zwaenepoel, "Manetho: transparent rollback recovery with low overhead, limited rollback, and fast output commit," *IEEE Trans. on Computers*, Vol. 41, No. 5, May 1992, pp. 526–531.

[9] A. Gefflaut, personal communication, Feb. 1994.

[10] L. Gunaseelan and R. J. LeBlanc, "Event ordering in a shared memory distributed system," *Proc. 13th Int. Conf. on Distributed Computing Systems*, 1993.

[11] B. Janssens and W. K. Fuchs, "Relaxing consistency in recoverable distributed shared memory," *Proc. 23rd Int. Symp. on Fault-Tolerant Computing*, 1993, pp. 155-163.

[12] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. on Software Engineering*, Vol. SE-13, No. 1, Jan. 1987, pp. 23–31.

[13] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy, "The directory-based cache coherence protocol for the DASH multiprocessor," *Proc. 17th Int. Symp on Computer Architecture*, 1990, pp. 148–159.

[14] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. on Computer Systems*, Vol. 7, No. 4, Nov. 1989, pp. 321–359.

[15] K. Li, J. F. Naughton, and J. S. Plank, "Checkpointing multicomputer applications," *Proc. 10th Symp. on Reliable Distributed Systems*, 1991, pp. 1–10.

[16] R. H. Netzer, "Optimal tracing and replay for debugging shared-memory parallel programs," *Proc. 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, 1993.

[17] G. G. Richard III and M. Singhal, "Using logging and asynchronous checkpointing to implement recoverable distributed shared memory," *Proc. 12th Symp. on Reliable Distributed Systems*, 1993.

[18] R. E. Strom and S. Yemeni, "Optimistic recovery in distributed systems," ACM Trans. on Computer Systems, Vol. 3, No. 3, Aug. 1985, pp. 204–226.

[19] C. B. Stunkel, B. Janssens, and W. K. Fuchs, "Address tracing of parallel systems via TRAPEDS," *Microprocessors and Microsystems*, Vol. 16, No. 5, 1992, pp. 249–261.

[20] Y-M. Wang and W. K. Fuchs, "Optimistic message logging for independent checkpointing in message-passing systems," *Proc. 11th Symp. on Reliable Distributed Systems*, 1992, pp. 147–154.

[21] Y-M. Wang and W. K. Fuchs, "Lazy checkpoint coordination for bounding rollback propagation," *Proc. 12th Symp. on Reliable Distributed Systems*, 1993.

[22] K.-L. Wu, W. K. Fuchs, and J. H. Patel, "Error recovery in shared memory multiprocessors using private caches," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 1, No. 2, April 1990, pp. 231–240.

[23] K.-L. Wu and W. K. Fuchs, "Recoverable distributed shared virtual memory," *IEEE Trans. on Computers*, Vol. 39, No. 4, Apr. 1990, pp. 460–469.

END

FILMED

DATE: 7-94

DTIC