Recovery Protocols for Shared Memory Database Systems

Lory D. Molesky and Krithi Ramamritham

Department of Computer Science
University of Massachusetts
Amherst MA 01003-4610

e-mail: lory@cs.umass.edu, krithi@cs.umass.edu

Abstract

Significant performance advantages can be gained by implementing a database system on a cache-coherent shared memory multiprocessor. However, problems arise when failures occur. A single node (where a *node* refers to a processor/memory pair) crash may require a reboot of the entire shared memory system. Fortunately, shared memory multiprocessors that isolate individual node failures are currently being developed. Even with these, because of the side effects of the cache coherency protocol, a transaction executing strictly on a single node may become dependent on the validity of the memory of *many* nodes thereby inducing unnecessary transaction aborts. This happens when *database objects*, such as records, and *database support structures*, such as index structures and shared lock tables, are stored in shared memory.

In this paper, we propose crash recovery protocols for shared memory database systems which avoid the unnecessary transaction aborts induced by the effects of using shared physical memory. Our recovery protocols guarantee that if one or more nodes crash, all the effects of active transactions running on the crashed nodes will be undone, and no effects of transactions running on nodes which did not crash will be undone. In order to show the practicality of our protocols, we discuss how existing features of cache-coherent multiprocessors can be utilized to implement these recovery protocols. Specifically, we demonstrate that (1) for many types of database objects and support structures, volatile (in-memory) logging is sufficient to avoid unnecessary transaction aborts, and (2) a very low overhead implementation of this strategy can be achieved with existing multiprocessor features.

1 Introduction

Shared memory systems offer significant performance advantages for applications which share data. But, when we consider how applications that have failure resilience requirements can benefit from cache-coherent shared memory (SM) systems, we find that current crash recovery mechanisms are insufficient to support these applications. With current mechanisms, a single node (where a *node* refers to a processor/memory pair) crash is likely to require a reboot of the entire shared memory system. Although the crash of a single node in an SM system should be infrequent, in very large systems if one node crash implies system failure, then the system could be down quite often. Clearly, it will be beneficial to have *independent node failures* whereby the crash of one node does not cause the failure of other nodes.

One class of applications which will benefit immensely from the provision of *independent* node failures is database applications. Primarily because of the absence of support for independent node failures, shared memory database implementations suffer from recovery problems [1]. This is unfortunate because database applications can easily exploit the performance advantages of shared memory architectures.

While the support for independent node failures in an SM system requires certain low-level mechanisms to be in place, such as mechanisms to detect and isolate hardware faults [12], these alone are not enough to guarantee that transactions are not unnecessarily aborted due to node failures. These abortions can occur as side effects of the cache coherency protocol. When a node crash occurs (whereby the contents of the physical memory on the failed node are destroyed), correctly recovering from the crash may require aborting otherwise independent transactions which execute on *other* nodes. These unnecessary transaction aborts can be avoided by appropriate design of recovery protocols. This is our goal.

Our recovery protocols guarantee the failure atomicity[1] of transactions, yet avoid unnecessary transaction aborts. Specifically, our recovery protocols guarantee that if one or more nodes crash, all effects of active transactions running on the crashed nodes will be undone, and no effects of transactions running on nodes which did not crash will be undone. The key features of these recovery protocols are (a) *logging-before-migration* (LBM) policies, which enforce specific logging policies prior to the migration of uncommitted data from one node to another, and (b) recovery mechanisms for ensuring that the appropriate undo's and redo's are performed. The *volatile* LBM policy helps achieve our recovery goals with very low runtime overheads and can be implemented with existing features of cache-coherent multiprocessors.

This research is funded in part by the National Science Foundation under grant NSF IRI-9314376, and by Sun Microsystems' Labs.

[1] Failure atomicity of a transaction, also called the *all-or-nothing* property, means that either all or none of the transaction's operations are performed even when there are failures.

This paper is structured as follows. Section 2 discusses our system and transaction model, and section 3 discusses how cache coherency can complicate recovery. In section 4, we present our recovery protocols and discuss how these protocols can be applied to database objects, such as records, and database support structures, such as index structures and lock tables. Techniques for implementing the LBM policies on a cache-coherent shared memory multiprocessor are discussed in section 5. Issues which arise when integrating our recovery protocols with other transaction processing components are discussed in section 6. A summary of the overheads associated with our recovery protocols is presented in section 7. Related work is discussed in section 8 and our conclusions are presented in section 9.

2 System and Transaction Model

In an SM database system, each node is connected to all disks in the system, and each node has access to shared memory. We consider SM database implementations where the shared memory is made coherent using a hardware-based cache coherency protocol. Hardware-based cache coherency provides low latency, high bandwidth access to shared memory, and ensures that each node reads the most recent version of the data in the shared address space. By utilizing these features of an SM system, it is possible to construct high performance multiple node database systems.

The coherency protocol, implemented in hardware, ensures that any read operation sees the most recently written value for any data item. Each node has its own cache, and before an operation is performed on a data item, the data item must first be brought into the cache. In general, operation execution time is minimal if the data item is already in the cache, more expensive if the data item is in another node's cache, and the most expensive if the data item must be fetched from disk. Typically, the hardware elements implementing the cache coherency scheme include a cache controller, a cache directory, and the cache itself. The cache contains the cached data, while the cache directory contains the addresses of all cached data. Our discussions in this paper assume a *write-invalidate* cache coherency protocol [2, 13] where, before a write to a cache line by one node occurs, all other cached copies of the line are first invalidated[2]. However, a cache line could be valid in multiple nodes after a series of read requests to that line have been issued. While the unit of I/O is a page, the unit of coherency is a cache line, and is typically smaller than a page.

Our recovery protocols rely on low-level hardware error detection and recovery mechanisms. Although we are unaware of any commercial multiprocessor which provides these mechanisms, these mechanisms are currently under development in the context of the Stanford FLASH (cache-coherent) multiprocessor research project[12]. In FLASH, node failures are detected with a diagnostic CPU in conjunction with various timeout mechanisms associated with each memory controller and network router [6]. Once a node failure is detected, all CPU's are interrupted so that they stop issuing requests, then a low-level recovery state is entered, which uses the inter-connection network to restore the cache directories to a consistent state – one which reflects the contents of the caches. Once the low-level recovery mechanism restores the cache directories to a consistent state, our (software) recovery protocols are employed to ensure the failure atomicity of transactions.

Consistent with most commercial database implementations, we assume that:

- Locking-based concurrency control is used. The basic lock modes are shared and exclusive. An exclusive lock on a record r guarantees that no other transaction will read or modify r, while a shared lock on r ensures that no other transaction will modify r. Note that several shared requests on r can be granted concurrently.
- No-force/steal buffer management policy is used [8].
- Each node maintains a log, and in-place updating is used in conjunction with the write-ahead log protocol (WAL) [3]. All update operations to this log take place in the node's cache. This in-cache log is volatile, but can be made stable by writing it to one of the shared disks. Since updates to the log are performed only by the local system, with proper memory alignment (i.e. a cache line which contains local log information stores no other sharable information), we can ensure that local logs do not migrate between systems. Figure 1 illustrates these assumptions in the context of a cache-coherent shared memory multiprocessor.

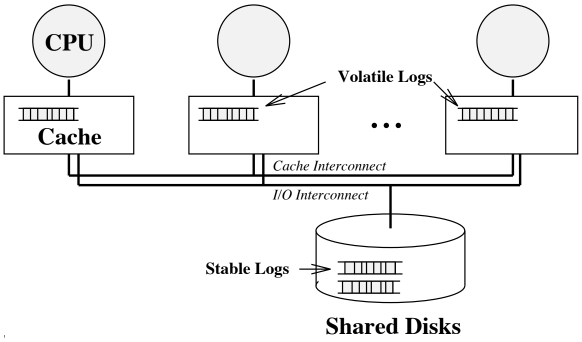

Figure 1: Multiprocessor System Model.

We focus on transaction workloads where independent transactions execute entirely on a single node. Although an SM database system is well suited for applications where a single (parallelized) transaction executes on multiple nodes, the presentation of the recovery strategies is simplified under the assumption that each transaction executes on a single node. However, our results easily generalize to address transactions which execute on multiple nodes.

3 Failure Effects Induced by Cache Coherency

Any data structure implemented in shared memory can be adversely affected by the failure effects induced by a cache-coherency protocol. In database systems, these data structures include *database objects*, such as records, and *database support structures*, such as index structures and shared tables. One case where these failure effects can arise occurs when two or more records are stored in the same cache line. Due to typical cache line sizes of current shared memory multiprocessors, it is likely (unless a lot of space is wasted) that multiple records will be stored in a cache line. For example, on the KSR-1 and KSR-2 multiprocessors [24], and on Stanford's FLASH [12] distributed shared memory machine, the cache line size is 128 bytes.

Under a write-invalidate cache-coherency protocol, when two or more records stored in the same cache line are concurrently updated

[2] Our results also apply to a *write-broadcast* cache coherency protocol. We briefly discuss the implications of write-broadcast on recovery in section 7.

by transactions runing on different nodes, the node where the last update occurred will hold the *only copy* of these records. Due to this access pattern, two types of failure effects occur: First, the crash of one node may not result in the complete annulment of transactions which execute strictly on the failed node. Second, the crash of one node may destroy updates performed by transactions which execute on other nodes.

Next, in section 3.1, we consider how typical database operations can cause these failure effects to occur. After presenting these simple examples, in section 3.2 we present the general patterns of cache line references which may cause these failure effects. Section 3.3 discusses our approach to ensuring the failure atomicity of transactions – one which does not incur unnecessary transaction aborts.

3.1 Examples of Failure Effects

The following example illustrates one case where a transaction executing on one node may become dependent on the validity of memory on some other node. Assume records $r1$ and $r2$ are stored in a single cache line, l. Transaction t_x (executing entirely on node x) locks, then updates record $r1$. Then, transaction t_y (executing on node y) locks, then updates record $r2$. As a side effect of the write-invalidate cache coherency protocol, *the only copy* of cache line l now resides on node y. If node x crashes, the control state (registers, stack, etc.) of t_x will be destroyed; but since l migrated to node y, the uncommitted update to record $r1$ will remain intact. On the other hand, if node y crashes, l will be destroyed and even though the node t_x is executing on has not failed, the uncommitted update to $r1$ performed by t_x will be destroyed. Thus, if either node x or node y crashes, the failure atomicity of transactions may be compromised.

These failure effects can also occur due to the sharing of database support structures, such as database indexes, or tables used by lock managers. Consider a shared memory implementation of a lock manager, where, in order to properly acquire and release locks, each node examines and updates information stored in shared memory. In this situation, multiple entries which describe the acquisition of database locks may be stored in a single cache line. For example, suppose two active transactions, running on different nodes, have acquired a lock on the same record (in shared mode). Further suppose that the lock control block, the shared data structure which contains the information describing the two holders of this lock, is stored in one cache line. The last node to acquire this record lock will hold the only copy of the lock control block. Without sufficient recovery provisions, a crash of this node will result in the loss of some of the information about locks granted to transactions running on other nodes. This can also lead to a violation of the failure atomicity of transactions.

3.2 Cache Line Migration and Replication

The previous examples have shown simple scenarios where recovery problems arise when two or more objects are stored a single cache line. In this section, we consider the general patterns of data sharing which may cause recovery problems. First, we consider coherency patterns due to *ww* (write-write) data sharing. In a write-invalidate cache coherency protocol, *ww* sharing occurs when writes by different nodes are issued to the same cache line. In this case, a cache line *migrates* from one node to another. The recovery scenarios discussed in 3.1 all fall into the category. Cache line *replication* may occur when one node writes, then another node reads (*wr* (write-read) data sharing) a particular cache line. Cases of *wr* data sharing are especially important because these patterns of coherency may occur due to references to cache lines which contain only a single object.

Consider *ww* patterns of sharing. We use cache line histories to consider in general how read and write operations on cache lines affect the state transitions of the cache coherency protocol.

$$\begin{aligned} H_{ww1} &= w_x[l];\ w_y[l]; \\ H_{ww2} &= w_x[l];\ (r_x[l];)^*\ r_{\bar{x}}[l];\ (r_z[l];)^*\ w_y[l]; \end{aligned}$$

In H_{ww1}, a cache line l migrates directly from node x to node y. In this case, a write by node x to line l ($w_x[l]$) occurs, causing line l to be valid only in node x's cache (recall our assumption of a write-invalidate cache coherency protocol). The next operation issued on line l is a write by node y, causing l to be invalidated in all other caches (x), and held exclusively in node y's cache.

H_{ww2} shows how intermediate read operations can cause the state of l to be held in shared mode (in potentially many caches) between the $w_x[l]$ and $w_y[l]$ operations. This shared state of l may occur after $w_x[l]$ occurs, zero or more reads on l are issued by node x, one read is issued by some node other than x ($\bar{x}$), and zero or more read operations are issued by any other node (as indicated in the above history).

wr sharing occurs when a write to a line by one node is followed by a read to that line by another node:

$$H_{wr} \quad = \quad w_x[l];\ r_y[l];$$

In H_{wr}, node x writes l, then node y reads l. At this point, line l will be replicated – valid on both nodes x and y. Some of the recovery problems which arise in *wr* data sharing also arise in *ww* data sharing. For example, in H_{wr}, the crash of node x will leave uncommitted updates on node y – requiring certain undo operations to complete the abort of transactions running on node x. For database records, H_{wr} may occur under the following conditions:

- If dirty reads are not allowed (as in *serializability* or *cursor stability* [7]), when two or more database objects are stored in the same cache line.
- If dirty reads are allowed (as in *browse* or *chaos* [7]), even when a single database object is stored in a cache line.

Thus, if dirty reads are allowed, the recovery problems due to cache coherency cannot possibly be avoided merely by storing at most one object per cache line. This applies to browsing database records, and for database support structures, where it is not necessary to ensure even browse mode.

To simplify our presentation, in section 4, our recovery examples are cast in the form of H_{ww1}. However, our recovery protocols are also designed to handle the data sharing patterns posed by histories H_{ww2} and H_{wr}. We revisit these histories when we discuss the enforcement of our logging policies in section 5.

3.3 Isolated Failure Atomicity

In the event of a node crash in a cache-coherent SM database, one way to ensure the failure atomicity of transactions would be to abort all transactions which are dependent on the memory of remote nodes. But this method is overkill, since unnecessary transaction aborts are incurred. Instead, our recovery protocols guarantee *Isolated Failure Atomicity* (IFA), by ensuring the failure atomicity of transactions, yet avoid unnecessary transaction aborts. IFA ensures that if one or more nodes crash, *all* effects of active transactions running on the crashed nodes will be undone, and *no* effects of transactions running on nodes which did not crash will be undone.

Ensuring IFA is particularly important in large multiprocessors, where the number of active transactions may be very large. Very large multiprocessors are feasible, as evidenced by the KSR-1 multiprocessor [24] which can be configured as large as 1,088 nodes. In such as system, it is conceivable that a single node failure would affect thousands of active transactions.

Furthermore, in geographically dispersed DSM (distributed shared memory) multiprocessors, the probability of a node failure is likely to be much higher than in a tightly-coupled multiprocessor. Consider a scenario where a user has the capability to "plug into" a DSM network. In this scenario, users would also be at liberty to powerdown their machines at any point, essentially simulating a node crash. If IFA were not guaranteed, it would be unlikely that these types of geographically dispersed DSM networks would be accepted by a large group of users.

4 Transaction Recovery in Shared Memory Database Systems

In this section, we propose SM crash recovery protocols which ensure IFA. Recovery protocols are generally comprised of two main mechanisms. The first takes the necessary steps to facilitate recovery at some future point in time. It ensures that sufficient information is available for the recovery procedure to re-establish a consistent database state. The second undertakes the actual recovery when needed, i.e., it restores the database to a consistent state after a node crash. We discuss these mechanisms in detail in section 4.1, in the context of providing recovery support to guarantee IFA for database objects. Section 4.2 considers additional issues which arise when IFA is ensured for database management structures.

4.1 Recovery of Database Objects

We assume that the possible operations on database objects (records) are *read* and *write*. Our recovery protocols make use of many of the mechanisms used in existing commercial database systems, including certain logging techniques, in-place updating, the WAL protocol, and the use of *strict* 2PL to ensure serializability. By utilizing mechanisms and structures which are already part of the database management system, the incremental costs associated with adopting our recovery protocols are minimized. For example, our LBM policies exploit the existence of undo and redo log records which are part of most database systems [8]. When a database record is updated, a redo log record (containing the value of the updated database record) is written to the volatile log. When a transaction first performs an update to a database record, an undo log record is written to the volatile log. This undo log record contains the before image (the last committed value) of the database record. To ensure the WAL protocol, prior to updating the *disk version* of a database record, the associated undo log record must be forced to stable store.

In the context of the WAL protocol, the assumption of *strict* 2PL allows transaction aborts to be implemented by simply replacing all the data touched by a transaction with their before images. Under strict 2PL, record locks are not released until after a transaction either commits or aborts. These protocols simplify recovery by ensuring that, at any time, only one transaction is associated with a particular uncommitted data item, and all before images exist in stable store. These assumptions also allow simple extensions to our LBM policies to implement transaction undos.

Our discussions focus on the two recovery problems for uncommitted transactions (the migration of committed data does not pose a recovery problem since the commit process ensures the durability of this data): First, the crash of one node may not result in the complete annulment of transactions which execute strictly on the failed node. Second, the crash of one node may destroy updates performed by transactions which execute on other nodes. Consider any transaction t_{active}, which was active at the time one or more nodes crash: With respect to database objects, in order to ensure IFA, our recovery protocols guarantee that if one or more nodes crash, for all t_{active},

1. If t_{active} was running on a node that crashed, then all its updates must be discarded.

   Thus, in case some of these updates migrated to other nodes, sufficient information must be available to undo them.

2. If t_{active} was running on a node that did not crash, then none of its updates must be lost.

   Thus, in case some of these updates migrated to a node that crashed, and hence are lost, sufficient information must be available to redo them.

Next we consider strategies for ensuring these guarantees. In section 4.1.1, we discuss the mechanisms which ensure the availability of specific information at recovery time, while section 4.1.2 describes the associated restart recovery schemes.

4.1.1 LBM policies

To provide sufficient information to allow restart recovery to ensure IFA in the event of a node crash, we employ LBM policies, which perform *Logging Before Migration*. Prior to the migration of uncommitted data, LBM policies log sufficient information to allow the recovery procedure to ensure the failure atomicity of transactions without unnecessarily aborting transactions. *Undo* information is logged in order to ensure that if a node crashes, *all* effects of transactions running on the crashed node will be undone. *Redo* information is logged in order to ensure that if a node crashes, *no* effects of transactions running on nodes which did not crash will be lost.

We examine two different LBM policies, one based on volatile logging, and the other based on stable logging. Volatile logging is implemented by logging information into local (volatile) memory, while stable logging is implemented by first logging into local memory, then forcing the log to stable storage. The primary advantage of Stable LBM is that the portion of the log which contains log records corresponding to uncommitted updates which have migrated is guaranteed to survive node crashes. In Volatile LBM, the absence of this guarantee requires that restart recovery has the capability to perform the appropriate undo operations without the use of the local undo log. Depending on the specific restart recovery scheme selected, supporting this undo requirement may also need a mechanism which pairs the migration of undo information with the migration of uncommitted updates (discussed in section 4.1.2).

For clarity of presentation, we examine a scenario when *one* transaction becomes dependent on the memory of *one* remote node. Consider active transaction t_x with the *only copy* of a record (r) updated by t_x having migrated to another node (y), as illustrated in figure 2. Note that this scenario may occur due to H_{ww1} or

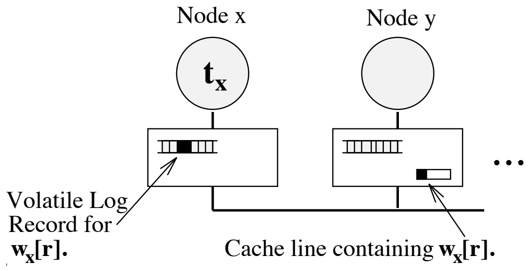
Node x
Node y
t_x
Volatile Log Record for $w_x[r]$.
Cache line containing $w_x[r]$.
...

Figure 2: Uncommitted Data Migration and Local Logging.

H_{ww2}, and a similar crash scenario may occur due to H_{wr}. When we consider how the failure atomicity of a transaction may be compromised, we must consider two basic crash scenarios:

1. x, the node executing t_x crashes.
2. y crashes.

These crash scenarios necessitate the undo and redo requirements imposed above. For case 1, the update performed by t_x on r (which now resides on node y) must be undone. For case 2, the update performed by t_x to r must be redone, since node y crashed, destroying the contents of r.

Under the Stable LBM policy, prior to the migration of record r, undo and redo log records for r are written to stable storage. For case 1, the stable undo log records are used by the restart recovery procedure to undo the update to r by t_x, ensuring the failure atomicity of t_x. For case 2, the stable redo log records are used by the restart recovery procedure to redo the update to r by t_x. Thus, under the Stable LBM policy, both undo and redo information are durably maintained prior to cache line migration to ensure the failure atomicity of transactions – without incurring unnecessary transaction aborts – when nodes fail. The obvious disadvantage of this approach is that the runtime overheads of stable logging are very high (unless non-volatile RAM is used to maintain logs). These runtime overheads can be reduced by adopting the Volatile LBM policy.

The Volatile LBM policy is sufficient to ensure the *redo* requirements of our recovery protocol, since, if some node other than the one executing the transaction in question crashes, that transaction's volatile log remains intact.

Because of the volatility of the local log, one cannot rely on using the local undo log to support undo under the Volatile LBM policy. For example, in case 1, when the node executing t_x crashes, it could easily be the case that the transaction management system *left no trace* of ever running t_x (as would be the case if neither stable logging or checkpointing had occurred after the start of t_x). This is illustrated in figure 2, where the log record corresponding $w_x[r]$ is destroyed if node x crashes. In this case, to complete the abort, the update performed by t_x (which is in node y's cache) must be undone.

Next, we outline two options for restart recovery which *do not rely on the existence of the crashed node's undo log*.

4.1.2 Restart Recovery

In this subsection, we consider two schemes for restart recovery, *Redo All*, and *Selective Redo*. Since neither of these schemes rely on the existence of the crashed node's log, both these schemes will work with in conjunction with either the Stable or Volatile LBM policies. These restart recovery schemes are named based on the degree of redo required for active transactions. In general, the Redo All scheme requires more redo operations to be performed at recovery time than does Selective Redo. However, Selective Redo requires slightly more runtime support.

First we consider *Redo All*:

- **Redo All**
  If a node crashes,
  1. On each surviving node, all cached database records are discarded from volatile memory.
  2. On each surviving node, the cache of database objects is reconstructed based on the local redo log. More specifically, redo must be performed for all log records for which the updates are not reflected in the stable database.

By requiring each node to flush its cache in step 1 of the Redo All scheme, any uncommitted updates which may have migrated to surviving nodes are effectively undone. However, in the process, many updates made by both active and committed transactions running on surviving nodes may also be undone. To ensure IFA, these updates made by surviving nodes must be redone. Note that under a no-force buffer management policy, updates of a committed transaction are not necessarily propagated to the stable database, so redo may be required for committed transactions. Also, under a steal buffer management policy, updates of uncommitted transactions may have already been propagated to the stable database, and if so, these propagated updates would not require redo.

Next we consider the *Selective Redo* scheme:

- **Selective Redo**
  If a node crashes,
  1. Each surviving node performs redo for only those record updates which were made by the local node, but which were exclusively resident on the crashed node.
  2. Each surviving node undoes the updates of aborted transactions using the undo tags (described below) stored in each record.

In the Selective Redo scheme, just those updates which were resident *only* on the crashed node are redone. This set can most easily be determined by observing the conditions for which no redo is necessary (we assume that the cache directory does not survive the node failure):

- If the update has been propagated to the stable database.
- If the cache line containing the update is either resident in the local cache, or resident in some other surviving node's cache.

The first condition can be easily checked based on the last checkpoint – this determines which committed updates have been propagated to the stable database. The second condition can be tested by temporarily (i.e., just during recovery) disabling the cache miss requests which incur I/O – if a memory reference cannot be satisfied with a cache line in a surviving node, an invalid flag is returned. Thus, in Selective Redo, the redo log is processed forward from the last checkpoint, and each log record is tested to see if the corresponding update remains cached on a surviving node. In this case, no redo is necessary. Otherwise, redo is performed.

As a result of the first step of the Selective Redo protocol, a node's cache contains the updated records that exist only on this node. But the cache may also contain updates of transactions that are aborted on other nodes. Thus, we need an additional mechanism to identify the active records, i.e., records updated by active transactions, which require undo. For these records, which had been updated by a transaction on a failed node and which subsequently migrated to another node, the undo of the aborted transaction's updates can be effected by installing each record's last committed value. Given our assumption of the WAL protocol, the last committed value of these records will necessarily be in stable store – either in the stable log, or in the stable database.

A simple way to identify these active lines is to associate a node identifier with each data object. The node ID is stored in the *same cache line* as the active data object:

> **Tagging Rule**: If multiple database objects are stored in a cache line, each active object is tagged with a node identifier. The node identifier indicates which node was executing the transaction that had updated the object. In the event of a crash of node x, all objects which are in the cache and tagged with node x are candidates for undo.

Note that because of the strict 2PL assumption, only one node will be associated with each active record. Once the data is no longer active, the node ID is assigned a null value. Thus, the node ID enables active cache lines (those cache lines which contain active data) to be identified for undo purposes. Of course, this approach requires additional space in order to maintain the per object node ID.

For Volatile LBM with Selective Redo, we envision the following implementation of undo at recovery time. Any active updates which have been written to the stable database (steal) will necessarily have their corresponding undo log records written to stable store (by the WAL protocol). Thus, these undo's can be performed based on the stable log records. Any additional undo's will correspond to in-cache records. To perform these undo's, each surviving node will perform a sequential search of all cache lines, performing the appropriate undo operation on an ID match. For example, for an active record, this undo corresponds to installing the most recent committed version of the record.

We have shown how the Stable LBM or the Volatile LBM policies can ensure IFA for database objects. These policies ensure that under any crash scenario, when a transaction becomes dependent on the memory of any other node, sufficient information will be available for the restart recovery procedure to mask this dependency. In order to guarantee IFA for (independent) transactions, we must also consider the effects of sharing database management structures, such as index structures and those related to database locking. This is the topic of the next subsection.

4.2 Recovery of Database Management Structures

By implementing database management structures in shared memory, the performance advantages of a shared memory multiprocessor can be fully exploited. Examples of these database management structures include hash tables, index structures such as B-trees, and tables used for lock management. In this section we examine the last two in detail.

For many transaction management issues, it is important to distinguish between structural and non-structural changes to database management structures. Examples of structural changes include B-tree page splits and the dynamic allocation of space used to store lock management information. In a multi-programmed uniprocessor database implementation, the subsequent use of this newly altered or created space by other transactions can cause transaction abort dependencies to form. To avoid these dependencies, when one transaction performs a structural change, it is customary to allow these changes to commit regardless of the future commit or abort of the transaction that caused the change (typically implemented as nested top-level actions) [14, 15, 16, 18].

However, in an SM multiprocessor database implementation which ensures IFA, volatile structural changes can also cause one transaction to become dependent on the memory of another node. For example, suppose space is allocated (in volatile memory) on behalf of t_x, and no corresponding redo log record for this allocation is on stable store. If t_y subsequently uses this space, the crash of node x will require the abort of t_y.

In an SM multiprocessor database implementation, one way to avoid this dependency is to immediately commit any operations which may result in such dependencies. In our previous example, the space allocated by t_x will be committed before any another transaction is allowed to use this space. Given this assumption, we can be sure that structural changes will not result in dependencies between active transactions and the memory of some other node. Thus, for structural changes, no additional recovery provisions are necessary in a SM implementation. However, for non-structural changes, uncommitted data migration can occur, potentially violating IFA. Next, we consider applying our recovery protocols to database management structures for which non-structural changes are likely.

4.2.1 A Shared Memory Implementation of B-trees

In this section, we consider the recovery issues which arise in a shared memory B+-tree (where records are stored only in leaf nodes) implementation of an index. We focus on non-structural updates, such as insert and delete. For the most part, the recovery issues for insert and delete operations are the same as for records. For example, in a SM implementation of a B+-tree, insert and delete operations can trigger the migration of uncommitted data between nodes. Consider the case where one transaction performs an insert operation. If other records may also be stored in the same cache line l where the newly inserted record is stored, it is possible for the newly inserted (uncommitted) record to migrate to some other node. In this case, just as for record updates, dependencies may form between an independent transaction executing entirely on one node and the memory of some other node.

For such updates to B+-trees, these dependencies can be avoided by applying the record oriented recovery techniques. For example, under Volatile LBM with Selective Redo, just as for record updates, to enable the undo of active insert and delete operations, a node identifier can be tagged to each active record. If a node crash does not result in the complete annulment of a transaction, the restart recovery procedure can identify all cache lines which need undo based on the node identifier.

However, issues related to space management allow a subtle implementation of record delete operations be employed. To ensure that the space freed by a delete is not used until the transaction which performed the delete commits, it is customary to perform record deletes logically, by marking the record as deleted [16]. Once the transaction which performed the delete commits, the space freed by the deleted record can be used by other transactions. This strategy also enables an efficient implementation of the undo requirement for Volatile LBM with Selective Redo – since any migrating cache line which contains an uncommitted delete will also contain the original record, the undo of a delete is effected by merely "unmarking" this record.

No such special provisions need be made for space management for the undo of an insert, since allocated space can always be freed.

4.2.2 A Shared Memory Implementation of Locking

Next, we consider how a shared memory implementation of database locking may benefit from our recovery protocols. For a lock table implemented in shared memory, almost all operations on a lock table are non-structural (space allocation operations are the exception). Because of the likelihood of many non-structural operations to a lock table during transaction execution, to guarantee IFA for transactions, it is important to apply our recovery protocols to this database management structure.

One strategy for implementing a lock manager in a multi-node system is to designate some node as being responsible for managing each database object, and allow remote nodes to access locks by using message passing. This is the approach of many shared-disk (SD) systems [19, 21, 25]. The presence of shared memory in an SM database system allows a more efficient approach to be taken for the implementation of database locking. In this strategy, which we call *SM locking*, LCB's (lock control blocks) are stored in shared memory, and transactions acquire and release locks via operations on these LCB's. The performance gains of SM locking stem from the elimination of all inter-process communication [20].

Consider acquiring a record lock using SM locking. A lock request consists of a lock *name* and a lock *mode*. Using a hash function, the name is translated to an LCB address specific to one lock. An LCB stores the current mode of the lock, plus two transaction lists, one containing the current holder(s) of the lock, the other containing any transaction(s) waiting for the lock. All updates to the LCB are performed inside a critical section. If the requested mode is compatible with the mode stored in the LCB, and there are no conflicting waiters, an entry containing the requesting transaction and requested mode is added to the holder list, and the lock is granted. This entry is called the *lock acquisition record*. Otherwise an entry is added to the wait list, and a not-granted flag is returned to the requestor. The strategy for releasing a lock is similar. After finding the appropriate LCB, the tuple identified by the transaction is deleted from the holder list, and any lock requests in the wait list which become compatible due to the release are granted.

When lock information pertaining to two or more transactions is stored in a single cache line, recovery issues similar to those for record updates arise. For example, after two transactions running on different nodes have acquired a compatible lock, the LCB will be valid at the node which last acquired the lock. In this case, a node crash may lose some but not all of a transaction's lock information. Note that this scenario is only applicable to uncommitted transactions, since committed transactions have no effect on the lock space (once a transaction has committed, all its locks are released)[3]. In contrast, each lock acquired by an uncommitted transaction will have a corresponding entry in the lock space.

Next, we consider recovery issues for acquired locks[4]. Consider any transaction t_{active}, which was active at the time that one or more nodes crash. To ensure IFA for SM locking, we guarantee that if one or more nodes crash, for all t_{active},

1. All locks acquired by transaction t_{active} running on a node which crashes will be released.
2. No locks acquired by transaction t_{active} running on a node which did not crash will be released.

Because of (1), any lock acquired by t_{active} running on a node which *had crashed* and stored in LCB's which *survived* the crash must be released by the restart recovery procedure. Because of (2), any lock acquired by t_{active} running on a node which *did not crash* and stored in LCB's for which *no copy survived* the crash must be restored by the restart recovery procedure. As with database objects, guaranteeing condition 1 requires undo information to be maintained, while guaranteeing condition 2 requires redo information to be maintained. Depending on the specifics of the LCB data structure, different strategies are necessary for ensuring these conditions.

Before we address the recovery options which are dependent on the specifics of the LCB data structures, we first point out a few of the salient aspects of SM locking.

- Prior to acquiring (or releasing) a lock on node x, a logical log record [7] is written to the log on node x. Note that in order to ensure that redo can be performed in the event that a node crash destroys LCB's of transactions running on surviving nodes, both write and read locks must be logged. Also, any lock requests which are queued (due to conflicts with lock holders) must also be logged.
- In most lock manager implementations, the transaction ID is stored along with the queued lock request or lock grant. If the transaction ID also encodes the node ID, this information is already available for use by the Volatile LBM policy.

For LCB's, crash recovery is further complicated when pointer based data structures must be supported. For example, in order to efficiently implement aborts, typically all locks held by one transaction are chained together with a linked list. If a node crash destroys an internal segment of a linked list, the restart recovery procedure must restore the reachability of all entries in the list. Detecting and remapping pointers to lost entries is inherently difficult. This suggests that for volatile pointer based data structures, the best method to ensure their consistency is to first restore the data that the pointers are derived from, then reconstruct the pointers which organize this data. For example, the transaction chain of LCB's is derived from the transaction ID. At restart recovery, once the transaction ID (and lock acquisition or request mode) of each LCB is restored, the transaction chain can be reconstructed based on this information.

The data structure used to represent individual LCB's also impacts the implementation of the restart recovery procedure. For example, it may be feasible to ensure that an LCB spans at most one cache line. Consider the queue of lock grants under this assumption. Whether this queue is organized as a table or a linked list, a node crash will either destroy all or none of a specific LCB. In this case, only those LCB's which were destroyed need be reconstructed.

A more difficult recovery scenario can occur if LCB queues are pointer based data structures which may span multiple cache lines. Under this assumption, a node crash could destroy arbitrary segments of the lock grant queue of a particular LCB. Rather than

[3] An exception to this rule occurs in some SD systems [19] where since lock acquisition is expensive in SD systems, locks are sometimes *retained* on local nodes after the transaction commit has occurred.

[4] The recovery issues for transactions which are waiting for locks are similar, but the discussion has been omitted due to space limitations.

attempting to repair only the missing portion of this LCB, it would be much easier reconstruct the entire LCB based on the log records on all surviving nodes.

Some issues related to ensuring the failure atomicity of database management structures are covered in [23, 25, 11], where crash recovery issues for an SD lock manager implementation on a VAXcluster are discussed. When a node crash is detected, all locking activity in the database system is stopped. Then, any updates performed by failed transactions are undone. This is accomplished by the installation of the before images of the associated records. After this restart recovery procedure is complete, user activity may proceed.

In contrast, in a cache-coherent SM database system, if certain mechanisms, discussed presently, are available, locking activity does not need to be stopped when a node crash is detected. In an implementation of SM locking, problems of ensuring a consistent lock space may arise if a node holding the only copy of a LCB crashes, but other nodes, not detecting the existence of this LCB, create a new LCB and incorrectly release an acquired lock. This will not be a problem if the underlying hardware support of the SM multiprocessor ensures that (just the) references made to cache lines residing on crashed nodes are stalled.

This section showed how our recovery protocols can be applied both to database objects and database management structures to achieve independent failure atomicity (IFA) for transactions. In the next section, we discuss how a low overhead implementation of our recovery protocols can be achieved in a cache-coherent SM multiprocessor.

5 Implementing LBM Policies in Shared Memory Systems

The LBM policies require that prior to the migration of a cache line, either stable logging or volatile logging is performed. Here, we discuss the implementation of these Logging Before Migration policies on a shared memory multiprocessor. We will show that sufficient primitives are already available on existing multiprocessor hardware to efficiently enforce the Volatile LBM policy, but not the Stable LBM policy.

To enforce the Volatile and Stable LBM policies, it is sufficient to construct the appropriate log record at any point after line l is updated and before l migrates. For a number of reasons, it is best to perform volatile logging immediately after an update is performed. This is both a logical and efficient point to perform volatile logging, since at the time of an update, most of the relevant information is already cached locally, and performing a few additional local memory references to write the log record minimizes the additional overheads. However, to reduce the overheads of stable logging, it is wise to minimize the frequency of log forces. Thus, while it is best to enforce Volatile LBM immediately after an update, it is best to delay enforcing Stable LBM as long as possible. We discuss the enforcement of the Volatile LBM policy in section 5.1, and the Stable LBM policy in section 5.2.

5.1 Enforcing the Volatile LBM Policy

A (cache) *line lock* [24], commercially available on the KSR-1 multiprocessor, is an example of a mechanism useful for efficiently implementing critical sections in a cache-coherent multiprocessor. Thus, they can be utilized to achieve a very low overhead implementation of the Volatile LBM policy. Once a line lock is acquired on cache line l by a process running on node x, the underlying hardware ensures the following properties:

- Line l is held *exclusively* in cache x.
- No other process, whether it be on the same or a different node, can read or write to l until l is explicitly released by the holding process.

The `getline(`l`)` instruction is used to obtain and hold a cache line l in a mutually exclusive (ME) state. The semantics of the `getline` primitive are such that, if the cache line is not already in ME state in any cache, the local cache acquires it in ME state. The `releaseline(`l`)` primitive releases the cache line from ME state[5]. The advantages of the line lock are that (a) locking and unlocking the line each requires only a single instruction, and (b) in the process of locking the line, the line also becomes resident in the local cache.

The Volatile LBM policy can be efficiently enforced with the use of the line lock as follows. Whenever an update to a database object or database support structure is performed, a log record is written describing this update. To ensure that a cache line l does not migrate between the time it is updated and the time the log record is written, a line lock can be used. Thus, prior to performing an update to data stored in cache line l, `getline(`l`)` is issued to lock the line in cache. The update is performed, and the log record is written prior to releasing the line with `releaseline(`l`)`.

Our experience in the implementation and empirical performance evaluation of database mechanisms on the KSR-1 confirms the expected performance gains provided by the line lock primitive. In [20], we implemented a prototype lock manager, using the line lock to ensure mutually exclusive updates to the shared memory implementation of the lock space. Our empirical performance studies have shown that under low contention, the mean execution time to obtain a line lock is less than 10 μs, and under high contention (32 processors simultaneously attempting to acquire the *same* line), the mean execution time to obtain a line lock is less than 40 μs.

5.2 Enforcing the Stable LBM Policy

One approach to enforcing the Stable LBM rule would be to force the log as part of the update protocol. In this solution, line locks would be retained on the updated cache lines until the update is performed, the log record is written, and the log force is completed. Although this solution would guarantee the stable logged rule, it is also very inefficient – a log force is performed on each update, regardless of whether the cache line ever migrates. In order to guarantee the Stable LBM rule and minimize the frequency of log forces, we must address the following:

- What is the latest point in a cache line use history where the log must be forced?
- What are the appropriate enforcement mechanisms?

To answer these questions, we must consider how read and write operations on cache lines affect the state transitions of the cache coherency protocol. The following discussion references the cache line histories given in section 3.2, H_{ww1}, H_{ww2} and H_{wr}.

In order to minimize the frequency of log forces, we would like to determine the latest point at which it is necessary to force the undo and redo logs. In H_{ww1}, this point would be immediately

[5] On the KSR-1, these primitives are called `gsp` and `rsp`, get subpage and release subpage. We have renamed these primitives to be consistent with the literature on cache coherency.

prior to $w_y[l]$, since this is the operation which causes the transition from exclusive in cache x to exclusive in cache y.

However, in H_{ww2} and in H_{wr}, this latest point occurs as soon as the next read by some node other than x occurs. For example, in H_{wr}, after $w_x[l]$; $r_y[l]$, ($r_y[l]$) will downgrade l from exclusive to shared mode on node x, allowing node y to also hold l in shared mode. If node x crashes after $w_x[l]; r_y[l]$, the crash recovery procedure would require undo information to complete the abort of active transactions that were running on node x. Clearly, to permit this, at least the undo portion of the log must be forced prior to $r_y[l]$.

Thus, for a line l which has been updated by some node x, the latest point at which the Stable LBM polices must be enforced corresponds to the downgrade or invalidation of l (for undo) and the invalidation of l (for redo). By triggering log forces based on these cache line state changes, the number of log forces can be minimized. This log force would only be done if the cache line contains database related information for which the corresponding log records had not been forced to stable store.

Unfortunately, triggers associated with the change of a cache line state are not a feature of any commercial multiprocessor that we know of. This extension to the cache coherency protocol can be implemented by dedicating one bit per cache line to indicate whether the line contains active data. Updates to the cache line would set this bit, and log forces would clear the bits of all associated cache lines.

Summarizing this section, it is clear that even though the Stable LBM policies are simpler to explain, they are more expensive to realize and in fact are not implementable with the features in today's multiprocessors. Volatile LBM policies, on the other hand, lend themselves to fairly efficient implementations with very little additional demands being placed on multiprocessors.

6 Integrating the Recovery Protocol with other Transaction Processing Mechanisms

In this section we consider the important issues involved when our recovery protocols are integrated with other salient components of a database system. Although many of the implementation issues of interest to us have been addressed in the context of SD systems [17, 19, 21, 22, 23, 25], significant differences between SD and SM systems require that, for the most efficient implementation, different mechanisms be developed for SM. These differences stem from the different approaches used to achieving coherency. In SM, cache coherency is achieved transparently by the underlying hardware cache coherency protocol. In contrast, in an SD system, coherency is achieved entirely in software and is closely coupled with the lock and buffer managers [19, 21]. Two significant implications of these factors are (1) an SM database need not include the SD mechanisms used for ensuring coherency, and (2) an SM database can utilize the low latency access to shared memory to yield very efficient adaptations of other SD mechanisms.

In the rest of this section, we discuss how these implications affect the transaction processing components needed for SM systems in the context of our recovery mechanisms. We focus on the protocols and implementation mechanisms used in a recent SD system [19]. This system supports record level locking and uses in-place updating in conjunction with the WAL protocol. It uses the repeating history paradigm followed by undos to recover from failures. We discuss how WAL and the techniques for ensuring the repeating history paradigm can be efficiently implemented in SM, in light of the mechanisms for achieving LBM.

Since this system has also addressed some of the issues related to the migration of uncommitted data, we discuss this aspect of [19] first. In order to ensure inter-node coherency, [19] defines four inter-node page transfer schemes, two of which allow the migration of uncommitted data, called *fast*, and *super-fast* schemes. In the fast scheme, all updates to page p must be stable logged prior to p's migration. In the super-fast scheme, all updates to page p must be volatile logged prior to p's migration. In the super-fast scheme, because page updates are not necessarily stable logged prior to uncommitted data migration, enforcing the WAL protocol may require referencing merged log. To implement the WAL rule for SD, each updating node remembers an LSN (log sequence number) greater than or equal to its last update to page p [19]. Page p can be written to the StableDB only after all systems which have updated p have forced their logs up to this LSN.

Closely related to the page transfer schemes is the *ordered update logging* rule [17, 19], which is important for supporting the repeating history paradigm (even for a multiprogrammed uniprocessor database system). This rule guarantees that the order of logging of updates to a page is the same as the order with which those updates are performed on the page. In [17], the *ordered update logging* rule is satisfied as part of the update protocol. This guarantee is provided by acquiring and holding a semaphore on the page to be updated for the duration of the update *and* the log write.

On the surface, there are many similarities between the SD protocols described in [19] and our recovery protocols. However, there are important differences. Many of the protocols of [19], such as the page transfer protocols, are designed for the purpose of enforcing inter-system page coherency. In contrast, for cache-coherent SM, our LBM policies are designed specifically to isolate the crash of one node from affecting transactions which execute on other nodes. Thus, whereas the protocols of [19] are aimed at achieving page coherency, we are motivated by the need to eliminate the ill-effects of system-ensured cache coherency! Furthermore, as was discussed in section 5, efficiently enforcing the LBM policies on a cache-coherent SM multiprocessor requires a careful analysis of the coherency protocol, and a novel application of multiprocessor features.

Finally, the availability of shared memory in SM systems allows for more efficient implementations of many transaction processing mechanisms. To illustrate this, next we show how shared memory can be utilized in the adaptation of two mechanisms used in the SD system of [19]: the ordered update rule and enforcing the WAL protocol under the Volatile LBM policy.

In section 4, we discussed how line locks could be used to enforce the Volatile LBM policy as part of the update protocol. The line lock mechanism can also be employed in SM to efficiently enforce the ordered update logging rule. Consider updating record r stored in page p, when it is also necessary to update the page's Page-LSN field[6]. Once a record lock is obtained on r, a line lock is acquired on (a) the cache line (by convention, the first cache line of page p) containing the Page-LSN of p, and on (b) the cache line containing the record (assuming these cache lines are different). Once these line locks are acquired, r and Page-LSN(p) are updated. Finally,

[6] Each database page has a Page-LSN field which contains the LSN of the log record that describes the latest update to that page. The Page-LSN is used during restart and media-recovery to determine which logged updates have been applied to the page.

the log record for the update is written and the two line locks are released. By using line locks instead of semaphores to enforce this protocol, runtime overheads, as measured in terms of the number of instructions executed, are substantially reduced.

To enforce WAL under Volatile LBM, we can adopt the same bookkeeping technique as done in SD, but exploit the available shared memory to minimize the runtime overheads. Each updating node remembers an LSN equal to its last update to page p. Page p can be written to the StableDB only after all nodes which have updated p have forced their logs up to this LSN. The determination of whether any other node is required to force its log can be computed very fast by maintaining this table of (page,LSN) pairs in shared memory. Recovery problems for this table can be avoided since this information is written only by the local node, and, in the event of a node crash, will be reinitialized on the crashed node.

7 Summary of Overheads of Ensuring IFA

Ensuring IFA contributes to the availability of multiprocessor database systems, but also entails certain overheads. In this section, we consider which additional, incremental overheads are incurred in order to ensure IFA as compared to ensuring just failure atomicity (FA). Recall that the basic difference between these assurances is that while IFA guarantees that all active transactions running on non-failed nodes will survive failures, FA does not.

The overheads associated with IFA occur during the normal (failure-free) operation of the database system, and during restart recovery. Assuming low failure rates, of primary concern are the overheads associated with the normal operation of the database system (compared to those associated with restart recovery). Thus, we dedicate this section to summarizing overheads associated with normal operation.

Many of the mechanisms used to ensure IFA are necessary to ensure FA, so we do not include these overheads in our assessment. These common mechanisms include volatile (redo and undo) logging, forcing the log at transaction commit, and taking checkpoints. For our recovery protocols,table 1 lists the incremental overheads associated with normal operation:

Overhead	*Protocol*		
	Stable LBM	Volatile LBM w/Selective Redo	Volatile LBM w/Redo All
Early Commit of Structural Changes	√	√	√
Logging of Read Locks	√	√	√
Undo Tagging		√	
Higher Frequency of Log Forces	√		

Table 1: Overheads of Protocols which Ensure IFA

All the recovery protocols presented incur overheads due to the early commit of structural changes and the logging of read locks.

- Early Commit of Structural Changes.
  Structural changes include operations such as the allocation of space that can be used by potentially many transactions. To avoid inter-node transaction dependencies, we require that structural changes be committed early – before any transaction on a remote node is allowed access the data and thereby form a dependency.
- Logging of Read Locks.
  Typically, transaction management systems log only write locks. When lock tables are stored in shared memory, in order to support IFA, our protocols require that read locks are also logged. This requirement enables uncommitted transactions (running on surviving nodes) to redo the acquisition of any locks destroyed due to node crashes. However, this requirement could be obviated with a different (but less efficient) lock manager architecture – one that did not exploit shared memory. This alternative lock manager architecture is essentially the same as used in certain SD systems [19, 21, 25], where locks are acquired with message passing and lock tables are replicated in order to survive failures. Some aspects of this tradeoff have been studied in [20].

These are the only two overheads associated with Volatile LBM with Redo All. In addition to these overheads, Volatile LBM with Selective Redo has additional overheads associated with Undo Tagging:

- Undo Tagging.
  Undo Tagging requires that, for each update, an undo tag is also written. This enables the recovery procedure to eliminate updates made by uncommitted transactions running on crashed nodes, without requiring the force of the local undo log. The overheads associated with Undo Tagging include a small amount of additional space (to store the tag) per updatable object, and a small amount of computation time to perform the (volatile) write of the tag.

Finally, in addition to the first two overheads mentioned, associated only with Stable LBM is a higher frequency of log forces.

- Higher Frequency of Log Forces.
  By itself, the Stable LBM policy ensures that sufficient information will be available during recovery to ensure IFA. Under Stable LBM, the frequency of log forces can be reduced with an extension to the cache coherency protocol which performs a log force triggered by the migration of a cache line containing uncommitted data. The undo log is forced by the invalidation of a cache line, while the redo log is forced by the invalidation or downgrade of the cache line.

One of the advantages of Stable LBM is that one does not need to keep track of other nodes' uncommitted updates to a page in order to enforce WAL (mechanisms for maintaining this table in the context of Volatile LBM were discussed in section 6). However, since this mechanism is needed to ensure FA as well as IFA, this overhead was not included in table 1.

If the only available stable storage is disk, then the increased latency associated with the higher frequency of log forces may be substantial, especially compared to the low latencies associated with an SM multiprocessor. However, we chose to present this policy for a number of reasons. First, the Stable LBM policy imposes the least programming complexity, and is also the easiest to explain. Second, advances in technology, such as the proliferation of non-volatile RAM, may make it feasible to store large portions of the log in low latency stable store. In this case, a Stable LBM policy may incur reasonably low overheads and hence may be of practical interest.

By avoiding costly disk I/O's, the Volatile LBM policies offer a low latency implementation of IFA. In the Volatile LBM policy, we have offered a choice between Selective Redo and Redo All schemes. The Redo All scheme does not require Undo Tagging, but requires potentially more time to perform restart recovery. However, there are additional reasons why a Selective Redo scheme would be appropriate. Under a write-broadcast cache coherency protocol, data sharing patterns such as H_{ww1} and H_{ww2} would not imply that the last node to update a cache line has an exclusive copy – both nodes would end up with a copy. In general, a write-broadcast protocol does not require redo – only undo would be required at restart recovery. Thus, to support the undo-only requirements of a write-broadcast cache coherency protocol, the Selective Redo scheme would be the best choice.

8 Related Work

Many studies have demonstrated the performance advantages of using SM implementation platforms for database systems. Based on a TP1 benchmark performed on a Sequent Symmetry shared memory multiprocessor, [27] conclude that an SM database system can deliver very high performance. In [28], an analytical and simulation study compares SN (shared nothing), SD, and SIM (shared intermediate memory[7]). This comparison concludes that the data sharing architectures, especially SIM, are more resilient to transaction load surges. In [4, 5], simulation studies compare SN, SD, and SM (called SE (shared everything) in this reference), and concludes that SM outperforms SN and SD by a fairly wide margin. Our work exploits the performance advantages of SM systems, yet guarantees good failure properties for transactions.

In the previous sections, we discussed how our work is related to work in shared disk systems [19, 21, 23, 25]. Architectural differences between SD and cache-coherent SM, such as the unit of inter-system data sharing, how coherency is achieved, and whether shared memory is available, have a significant influence on the design and implementation of SM crash recovery protocols. Furthermore, our goal was to achieve IFA with minimal extra overheads while capitalizing on features available on or proposed for SM multiprocessor systems.

In [22], augmenting an SD system with a non-volatile global extended memory (GEM) is considered. System performance can be improved by adding GEM to a system that otherwise can only communicate by message passing. Failure atomicity for data structures can be ensured by propagating their updates to the GEM. However, non-volatile memory is much more expensive than volatile memory, and is a significant departure from a database implementation based on off-the-shelf shared memory multiprocessors wherein the cache – where (parts of) data structures may reside – is volatile.

Transactional Memory [9] is another approach to supporting transactions on a cache-coherent shared-memory architecture. Transactional memory allows programmers to define customized read-modify-write operations that apply to multiple, independently-chosen words of memory. However, this approach is intended to replace short critical sections, i.e., it works well for short transactions with relatively small data sets. Our recovery protocols do not have these restrictions.

Volatile logging has been used in the context of *process checkpointing* schemes [10, 26] to ensure a consistent state of a distributed computation without necessitating the rollback of any processes other than ones that failed. In process checkpointing, information is periodically checkpointed to disk in order to ensure forward progress of a computation in the event that a processor and its associated memory fail. In this case, a checkpoint consists of the necessary process state for restarting execution, such as the program counter, process identifier, and register contents. Here, messages sent between processes trigger log records to be written to volatile memory. Recovery of a failed process is achieved by restarting the failed process from its checkpoint and replaying the message from the sender's logs.

9 Summary and Conclusions

When database objects and support structures are implemented in the shared memory of a cache-coherent shared memory multiprocessor, dependencies, caused entirely by the cache coherency protocol, may form between a transaction running on one node and the memory of another node. Moreover, these dependencies can arise due to typical patterns of cache line sharing. Unless steps are taken to address this problem, it is very likely that the crash of a single node requires the abort of *all* transactions in the entire shared memory multiprocessor. This is a very undesirable situation in large shared memory multiprocessors, where the number of active transactions is likely to be large, and in geographically dispersed shared memory machines, where untimely node disconnections may be common.

In this paper, we have presented crash recovery protocols which avoid unnecessary transaction aborts in cache-coherent shared memory database systems. For independent transactions, our recovery protocols guarantee IFA – that is, if one or more nodes crash in a system that isolates individual failures, all effects of active transactions running on crashed nodes will be undone, and no effects of active transactions running on nodes which did not crash will be undone. By applying our recovery protocols to database objects and database support structures, IFA is ensured for transactions under any crash scenario.

We have also demonstrated how these protocols can be integrated with well established database design and recovery principles, such as the use of in-place updating in conjunction with the WAL protocol, the flexible no-force/steal buffer management policies, fine-granularity locking, and the repeating history paradigm. By exploiting mechanisms and structures which are already part of many databases, the incremental overheads associated with adopting our recovery protocols are minimized.

For a parallel transaction (one which executes on multiple nodes), the recovery measures are similar to those for independent transactions. However, if one of the nodes executing this transaction were to crash, the entire transaction must be aborted.

The recovery protocols developed in this paper assume that only read/write operations are performed on database objects. We are currently working on the extensions required to accommodate arbitrary operations on (abstract data type) objects.

In addition to providing support for SM database systems, our recovery protocols can also be applied to provide operating system support for handling independent node failures. For example, many operating system data structures, including *semaphores*, *maps* used to catalog disk usage, and the *disk buffer*, used to cache recently used disk blocks, lend themselves to a shared memory

[7] The SIM model is slightly different than the shared memory model. In SIM, a shared intermediate memory serves as a global shared buffer for all nodes.

implementation. Recovery techniques similar to ours can be applied to these operating system data structures in order to ensure that the crash of one node does not necessarily affect the integrity of the process management information on other nodes.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments.

References

[1] Panel Discussion on Shared Nothing, Shared Disk, and Shared Memory Database Systems. *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, 23, May 1994.

[2] J. Archibald and J. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

[4] A. Bhide. An Analysis of Three Transaction Processing Architectures. *Proceedings of the 14th International Conference on Very Large Data Bases*, 14:339–350, September 1988.

[5] A. Bhide and M. Stonebraker. A Performance Comparison of Two Architectures for Fast Transaction Processing. *IEEE Proc. 4th Intl. Conference on Data Engineering*, pages 536–545, February 1988.

[6] J. Chapin. Personal Communication. 1995.

[7] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[8] T. Haerder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[9] M. Herlihy and E. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993.

[10] D. Johnson and W. Zwaenepoel. Sender-Based Message Logging. *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, pages 14–19, 1987.

[11] N. Kronenberg, H. Levy, and W. Streker. Vaxclusters: A Closely-Coupled Distributed System. *ACM Transactions on Computer Systems*, 4(2):130–146, May 1986.

[12] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. *Proceedings of the 21th International Symposium on Computer Architecture*, pages 302–313, April 1994.

[13] D. Lilja. Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons. *ACM Computing Surveys*, 25(3):303–338, September 1993.

[14] B. Liskov and R. Scheifler. Gardians and Actions: Linguistic Support for Robust, Distributed Programming Languages. *Ninth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languates*, pages 7–19, February 1982.

[15] D. Lomet and B. Salzberg. Access Method Concurrency with Recovery. *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, 21:351–360, June 1992.

[16] C. Mohan and D. Haderle. Algorithms for Flexible Space Management in Transaction Systems Supporting Fine-Granularity Locking. *Proc. International Conference on Extending Data Base Technology*, March 1994.

[17] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17:94–162, March 1992.

[18] C. Mohan and F. Levine. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, 21:371–380, June 1992.

[19] C. Mohan and I. Narang. Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared-Disk Transaction Environment. *Proceedings of the 17th International Conference on Very Large Data Bases*, 17:193–207, 1991.

[20] L. D. Molesky and K. Ramamritham. Efficient Locking for Shared-Memory Database Systems. Technical Report 94–10, University of Massachusetts Dept. of Computer Science, February 1994.

[21] E. Rahm. Concurrency and Coherency Control in Database Sharing Systems. *Technical Report, University of Kaiserslautern, Germany*, December 1991.

[22] E. Rahm. Use of Global Extended Memory for Distributed Transaction Processing. *Proceedings of the 4th Int. Workshop on High Performance Transaction Systems, Asilomar, CA.*, September 1991.

[23] T. Rengarajan, P. Spiro, and W. Wright. High Availability Mechanisms of VAX DBMS Software. *Digital Technical Journal*, (8):88–98, February 1989.

[24] Kendall Square Research. *KSR1 Principles of Operation*. KSR Research, Waltham, Mass., 1992.

[25] W. Snaman and D. Thiel. The VAX/VMS Distributed Lock Manager. *Digital Technical Journal*, (5):29–44, September 1987.

[26] R. Strom, D. Bacon, and S. Yemini. Volatile Logging in n-Fault-Tolerant Distributed Systems. *Proceedings of the 18th International Symposium on Fault-Tolerant Computing*, pages 44–49, 1988.

[27] S. Thakkar and M. Sweiger. Performance of an OLTP Application on Symmetry Multiprocessor System. *Proceedings of the 17th International Symposium on Computer Architecture*, pages 228–238, May 1990.

[28] P. Yu and A. Dan. Performance Evaluation of Transaction Processing Coupling Architecutres for Handling System Dynamics. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):139–153, June 1994.