Recovery Options in Directory Based Software Coherency Schemes

Lory D. Molesky
Oracle Corporation
110 Spit Brook Road
Nashua, N.H. 03062

Krithi Ramamritham
Department of Computer Science
University of Massachusetts
Amherst MA 01003-4610
e-mail: lory@us.oracle.com, krithi@cs.umass.edu

September 1996

Abstract

Directory-based coherency (DBC) has proven to be a popular and effective means for making a collection of local memories appear as one large global memory. Database systems composed of multiple nodes, such as client-server and shared-disk systems, often use DBC to achieve coherency. In this paper, we present a method for achieving failure atomicity of transactions in DBC systems, which support fine-granularity locking, and that avoid unnecessary transaction aborts. We use the term *Isolated Failure Atomicity* (IFA), to denote this property because it ensures failure atomicity while isolating node failures. IFA is achieved through schemes that allow surviving nodes to recover from the effects of transactions that were executing on failed nodes and hence were aborted – without waiting for the crashed node(s) to recover first. To this end, we have developed techniques (a) to reconstruct directory information and (b) to recover the data touched by transactions that were running on crashed nodes. One of the novel ingredients of our techniques is the idea of *undo log shipping* which adds only minimal space and time overheads to normal processing. A by-product of this systematic development of recovery techniques is that it helps to demystify some of the facets of multiple node recovery.

Keywords: Shared Memory, Database Systems, Cache Coherency, Crash Recovery, Volatile Logging

---

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

Contents

1 Introduction

Hardware-based cache coherence protocols in shared memory multiprocessors (SMPs) use the convenience of a broadcast bus to achieve coherency. On the other hand, *directory-based approaches* implement coherency without relying on the broadcast properties of a shared bus. To accomplish this, *directory* information is distributed among the nodes, indicating the location and access rights of *pages*[1] which make up the distributed shared memory. To ensure coherent access, it is often necessary to send messages between nodes to obtain the most recent copy of a page. As in cache-coherent SMPs, performance advantages are achieved via replication, which reduces the average cost of read operations by allowing simultaneous reads to the same page by multiple nodes.

Even though coherent shared memory can improve performance and has programming advantages, data sharing applications such as in database and transaction processing (TP) systems, may suffer from recovery problems due to side effects of coherency. 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 the *surviving* nodes. These unnecessary transaction aborts can be avoided by appropriate design of recovery protocols. When one or more nodes crash, our *recovery objective* is to

- Abort all transactions running on crashed nodes.
- Avoid aborting transactions running on surviving nodes.

We use the term *Isolated Failure Atomicity* (IFA), to denote the above property because it ensures failure atomicity while isolating node failures.

Because of data sharing between transactions, mechanisms to detect and isolate node failures – essential to achieve the fail-stop assumption [22] – alone are not enough to guarantee that transactions are not unnecessarily aborted due to node failures. Ensuring IFA is especially difficult in systems which support *fine-granularity locking* (FGL), which is used because it allows for high degrees of transaction concurrency. For example, when a page (containing multiple data items) is the unit of data sharing (between nodes), and the data item is the unit of logical locking, multiple active data items may be shipped between nodes. In the context of FGL, it was shown in [16] that additional protocols are necessary to achieve IFA in cache-coherent SMPs, i.e., systems that use cache coherence *hardware*.

[1] While the *cache line* is the unit of data sharing in cache-coherent multiprocessors, the *page* is typically the unit of sharing in directory based systems.

In this paper, we consider ensuring IFA in the context of directory based coherency (DBC) schemes. IFA in such systems implies recovering not only the data modified by transactions that were running on crashed nodes but also reconstructing the directory information made inconsistent due to node crashes. DBC has been used in many system architectures, including distributed shared memory (DSM) architectures [11, 24], client-server database systems which employ data-shipping and local caching [5, 15, 17], and also in shared disk database systems [12, 14]. Client-server data-shipping is very similar to distributed shared memory architectures, with the following exception. Unlike in client-server systems, many distributed shared memory architectures require kernel modifications in order to determine efficiently whether a page is resident in the local memory of the requesting processor.

In DBC, coherency is typically implemented in software. This makes it feasible to change the coherency protocol to support independent recovery of surviving nodes. We achieve IFA through schemes that allow surviving nodes to recover from the effects of transactions that were executing on failed nodes and hence were aborted – *without waiting for the crashed node(s) to recover first.* The key features of our recovery protocols to achieve IFA are (a) *logging-before-migration* (LBM) protocols, 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 undos and redos are performed upon a node failure.

Loosely-coupled DSM systems are generally configured with *private disks.* Unlike SMP and shared-disk systems where many nodes may share a disk, in systems with private disks, it is not possible to access the crashed node's stable storage until the crashed node reboots. An obvious simple way to ensure the availability of information needed to *undo* the effects of transactions on a crashed node is to move the needed information to stable storage. This proposition not only increases run-time overheads (due to stable writes to disk) but also delays recovery until the disk attached to the failed node becomes accessible. Recovery mechanisms must of course be able to *recover the data* modified by transactions that were running on crashed nodes. In the case of DBC, they must also *reconstruct the directory information.* The latter is essential to produce consistent information about the location and state of the data across nodes.

Instead, we use a novel technique called *undo log shipping* whereby when data migrates from one node to another, the associated (and usually required) undo log information is also migrated. This obviates the need to access the stable storage (if any) of the crashed node and enables surviving nodes to eliminate the effects of transactions that were executing on failed nodes quickly, thus minimizing the time that a surviving node is unavailable for processing transactions. As a result, existing transactions are allowed to make progress, and new transactions can begin execution as well.

For many loosely-coupled systems supported by directory based architectures with private disks, it is important to consider support for *disconnected operation*. In the event of a node crash or disconnection, there may be no guarantee that the connection with a crashed node can be reestablished in a timely manner. Our techniques do not require such a guarantee and hence allow a surviving node to recover independently from a crashed node.

In summary, the main contributions of this paper are:

- A systematic derivation of the requirements imposed on recovery in DBC systems.
- Development of techniques to achieve Isolated Failure Atomicity in DBC systems, in particular (a) to reconstruct directory information and (b) to recover the data modified by transactions that were running on crashed nodes.
- Development of the simple but effective technique of log shipping to minimize run-time overheads entailed by recovery.

Section 2 provides an overview of owner-based DBC protocols. Section 3 describes our transaction and failure model. To set the stage to discuss recovery issues that arise in DBC systems, we review, in Section 4 the anomalous behaviors that occur, in the form of lost and spurious updates, when data is stored in shared memory. The specific recovery issues that arise in DBC systems are discussed in Section 5. Section 6 is devoted to showing how (1) directories structures and (2) data items are recovered in DBC systems. Related work is discussed in Section 7. Section 8 summarizes the paper.

2 Overview of Owner-Based Directory Coherency Protocols

In owner-based directory schemes, each page in the global memory has a node designated as the owner. For example, in client-server DBC database system, the server would own all pages corresponding to its database. When loaner (non-owner or client) nodes require access to a page owned by some other node, the page is transferred to the requesting node by an underlying coherency protocol. We assume a *single-writer* coherency protocol, where prior to updating a page, a node must obtain exclusive access to the page. However, many nodes may simultaneously access a page for reading, and this is achieved by allowing many nodes to simultaneously cache a page. This single-write/multiple reader protocol is achieved with the following mechanisms:

Write-invalidation: A write to a page invalidates all other copies.

Read-replication: A read issued to a page which is not locally cached, but is in another cache, will cause the page to be replicated in the reading node's cache.

In the context of DBC, these mechanisms are implemented as follows: The owner maintains *location* information and the *mode* of the page. The mode of the page is either X (exclusive), S (shared), or I (invalid). For X pages, the owner maintains the node which has the writable copy (denoted the *writer*). For S pages, the owner maintains the list of nodes which have read-only copies of the page (denoted the *readers*). The owner will have a copy of the page if either the owner holds the page in X mode, or if any node holds the page in S mode. This information is maintained in the owning node's global directory, denoted *GDir*. Loaner nodes maintain local directory (*LDir*) information for pages that they possess but for which they are not the owners. The contents of an I page are invalid in that another node has modified the page but the modifications are not reflected on this node's page. Hence they must not be read.

An example of DBC follows. Suppose node x owns p. After node y requests to write p, p will be cached on node y and $LDir^y$ will have an entry indicating that p is held in X mode. A node that needs access to page p first checks either its *GDir* or *LDir* to determine if p is cached locally in a compatible mode (X for write, X or S for read). Note that we assume that ownership can be determined statically, e.g., via a simple function on the page identifier (such as a hash function). If p is cached locally in a compatible mode, then it can be accessed without any inter-node communication. Otherwise, when either a *read miss* or *write miss* occurs, it triggers inter-node communication to bring the necessary pages into the local buffer.

In summary, a *GDir* maintains complete location information for all pages owned by that node, and an *LDir* maintains location information for pages that it does not own. This *LDir* information is redundant, since each *LDir* entry mirrors location information contained in one of the *GDir*s. Because location information is distributed, with local and global directories storing potentially redundant information, directory inconsistencies may arise due to node crashes. These inconsistencies must be resolved during recovery. How this can be done is discussed in Section 6.1. In the rest of this section, we discuss the protocols for handling read and write misses. These protocols are similar to those described in [3, 4, 26, 14, 19].

Handling a Read Miss:

1. A read request for p is sent to the owner.
2. Upon receipt of the request, the owner does the following:
   If the current mode of p is X, then
   (a) The owner sends a downgrade message to current holder, requesting that the current mode of p be set to S.
   (b) The current holder downgrades its mode of p to S, then sends a copy of p to the owner.
   (c) Upon receipt of p, the owner assigns the mode of its local copy to S.

3. The owner adds the requesting node to the *readers* of p, then sends a copy of the page to the requester.

4. The requester caches p locally, and makes an entry it its $LDir$ indicating p is held in S mode.

Figure 1 graphically illustrates the messages transmitted and the resulting state changes for the case where a loaner (other than the requester) holds the exclusive copy. Three different nodes are shown: a *requestor*, *owner*, and a *holder*. *Send()* denotes a message send, and the direction of the arrow indicate the sender (source) and the receiver (destination). The labels on the directed arrows correspond to conditional operations based on the states of the sender's cache and the action taken by the sender. The resulting change in the mode of the page, due to these coherency messages, is shown at the bottom of each node. In the figure, the mode of the page is denoted as either I (invalid), S (shared), or X (exclusive).

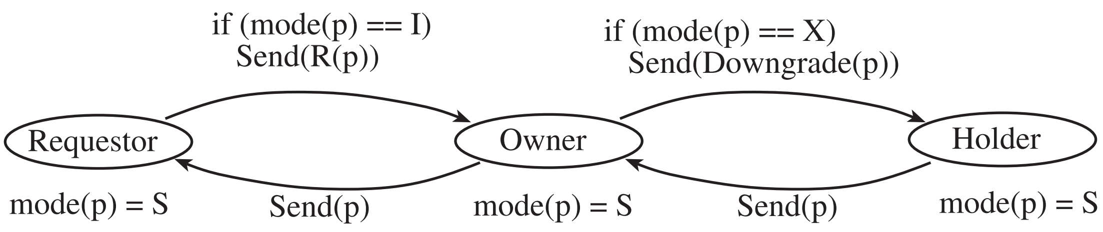

Figure 1: Resolving a Read Miss.

A similar strategy is employed on a write miss. In this case, in order to obtain an X copy of p, ownership must be transferred from the current holder to the node requesting the write operation. Next, we show the steps for resolving a write miss when there is a single holder.

Handling a Write Miss:

1. A write request for p is sent to the owner.

2. The owner sends an invalidation message to current holder.

3. Current holder supplies p to owner, and invalidates its copy.

4. Owner supplies p to requestor and makes the requestor the writer.

5. Requestor receives and caches p, and updates $LDir$ to indicate that p is held in X mode.

Figure 2 graphically illustrates the write miss protocol for the case where one loaner (other than the requestor) holds a shared copy (and the owner also has a copy in shared mode). Note that in the case where many nodes hold p in shared mode, invalidate messages must be sent to each holder (steps 2 and 3).

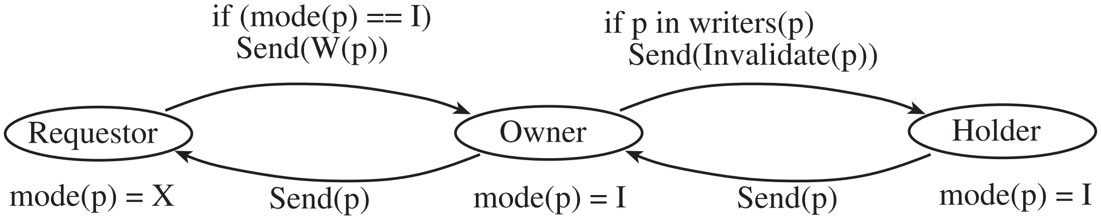

Figure 2: Resolving a Write Miss.

This section has described the basic protocols for owner-based DBC. Next we present our failure and transaction model, then discuss the recovery issues that arise in shared memory systems in general. Recovery issues in DBC systems are presented subsequently.

3 Failure and Transaction Model

In the context of DBC, we assume that node crashes are *fail-stop* [22]. The visible effects of the failure of node x are: (a) Node x stops executing, (b) the internal state and the contents of volatile storage on node x are lost, and (c) other nodes detect the failure of node x. Once a node failure is detected, transaction processing is halted, and a restart recovery procedure is executed to restore system consistency. We assume that the failed node eventually reboots successfully and that the crash of a particular node does not affect the physical contents of *volatile* memory on other (surviving) nodes. We also assume that the underlying interconnection network remains fully operational. In the context coherency, we assume that message sends and receives are atomic with respect to recovery.

In our model of transaction execution, transaction and buffer management functions are implemented symmetrically at each node. In this context, we make a number of assumptions which are consistent with most commercial database implementations. All data is updated in-place and updates follow the write-ahead log (WAL) protocol [2]. Each node maintains a log. All update operations to this log take place in the node's memory. This in-memory log is volatile, but can be made stable by writing it to the local private disk. Log records are typically written in conjunction with the data item updates – this is referred to as an *update protocol.*

We assume that locking-based concurrency control is used to enforce serializable execution histories[2]. While coherency control ensures correctness at an operation level, *concurrency control* is used to ensure correctness between groups of operations, defined as transactions. In locking-based concurrency control, the basic lock modes are shared and exclusive. An exclusive lock on a data item

[2]The recovery anomalies we address arise also in non-serializable executions, but we focus on serializable execution histories in this paper.

i guarantees that no other transaction will read or modify i, while a shared lock on i ensures that no other transaction will modify i. Note that several shared requests on i can be granted concurrently. We also assume the most flexible buffer management policy, No-Force/Steal [8]. We also assume that lock management is performed at the node which *owns* the data to be locked.

Our recovery algorithms are designed to work correctly when *fine-granularity locking* (FGL) [14] is used. For example, although pages are the grain of coherency, multiple data items may be stored in a page, and the data item is the unit of logical (database) locking. FGL allows high concurrency of transactions, but complicates recovery. These complications arise when a page containing multiple active data items are transferred between nodes.

We focus on transaction workloads where independent transactions execute entirely on a single node. Although a database system implemented on multiple nodes 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, the results can be extended to handle parallel transactions by treating each component of a parallel transaction as a separate transaction for the purposes of recovery.

4 Recovery Issues in Coherent Shared Memory Systems

Traditional database recovery only ensures that the results of *committed* transactions are unaffected by crashes. In the event of a node crash, most protocols in use today (see Section 7) will abort *all running transactions*, even those running on surviving nodes. Under our recovery objective of IFA, in addition to ensuring that *committed* transactions are unaffected by crashes, we must also ensure that *active transactions running on surviving nodes are unaffected by crashes.* Our approaches to achieving IFA are based on volatile logging, and thus incur little or no additional runtime cost to the DBMS, (which is done anyways in traditional DBMSs). In this section, we summarize the causes of recovery anomalies in shared memory systems. In subsequent sections, we show how these anomalies affect DBC systems, and discuss approaches to resolving these anomalies.

As a result of the write-invalidate and read-replication coherency mechanisms, updates performed by one node may (1) *migrate* to some other node or (2) be *replicated* on another node. If node crashes occur after migration or replication, recovery anomalies may arise [16]. In particular, spurious updates may arise due to either write-invalidation or read-replication, and lost updates may arise due to write-invalidation. Consider the following two histories, where the superscripts x and y denote *different* nodes, and data items i and j, are stored in page p.

$$H^{ww1} \quad = \quad w^x[i];\ w^y[j];\ Crash^y.$$

In the history, a write by node x is executed on data item i, then a write by node y is executed on data item j. After these two operations, due to write-invalidation, *the only copy* of line p resides on node y. Note that this history is serializable, and, when using FGL, the page migration will occur due to write-invalidation. At this point, if node y crashes, a lost update occurs.

A *lost update* occurs if a node crash destroys an update which was made by an active transaction running on a surviving node. When node y crashes (denoted $Crash^y$) in H^{ww1}, the contents of all volatile memory on node y are lost. This includes the update performed by node x ($w^x[i]$). Our recovery objective requires that transactions running on surviving nodes must remain unaffected. Thus, we must restore the lost update ($w^x[i]$).

A *spurious update* occurs if a node where some active transaction was executing crashes, but an update made by this transaction remains in the cache of some other node. Spurious updates may arise due to either write-invalidation or read-replication. These cases are exemplified in the following histories:

$$\begin{aligned} H^{ww2} &= w^x[i];\ w^y[j];\ Crash^x. \\ H^{wr} &= w^x[i];\ r^y[j];\ Crash^x. \end{aligned}$$

When node x crashes in H^{ww2}, the contents of volatile memory on node x are lost and thus, all transactions executing on node x must be aborted. However, the update made by node x ($w^x[i]$) remains in the cache of node y, since p had migrated to node y prior to the crash of node x.

In H^{wr}, as in H^{ww2}, the crash of node x only partially aborts t^x, failing to eliminate $w^x[i]$. Note that read replication does not lead to lost updates, since invalidation occurs only on a write. For example, after $w^x[i];\ r^y[j]$, since p remains on node x, if node y were to crash, all copies of p are not destroyed.

5 Recovery Issues in Directory Based Coherency Protocols

Directories can become inconsistent when either (i) p's global directory ($GDir(p)$) is destroyed due to a node crash, but one or more local directory entries ($LDir(p)$) of p survive, or (ii) when one or more $LDir(p)$'s are destroyed but $GDir(p)$ survives. With respect to a particular page, we classify these cases as *owner crashes* and *loaner crashes* respectively [3]. Owner crashes may lead to lost $GDir(p)$ entries, while loaner crashes may lead to spurious $GDir(p)$ entries:

[3] Note that if more than one node crashes, causing the loss of both $GDir(p)$ and $LDir(p)$, then the only possible directory inconsistency for p arises if one or more $LDir(p)$ survive. This still belongs to the category of owner crashes.

- Lost $GDir(p)$

  A Lost $GDir(p)$ occurs when the owner's directory entry for p ($GDir(p)$) is destroyed, but the loaner's directory entry for p ($LDir(p)$) remains unaffected by the crash.

- Spurious $GDir(p)$

  A Spurious $GDir(p)$ occurs when the loaner's directory entry for p ($LDir(p)$) is destroyed, but the owner's directory entry for p ($GDir(p)$) remains unaffected by the crash.

Next, we consider the details of owner and loaner crashes.

5.1 Owner Crashes

Table 1 illustrates the possible directory inconsistencies and update anomalies which may arise due to *owner* crashes. For completeness, the table also includes the associated lost and spurious updates to data. In all entries of the table, given page p, $owner(p) = x$, data items $i, j \in p$. In all histories, node x crashes, destroying x's global directory $GDir^x$. Node y, and its associated $LDir^y$ exemplify local directories which survive the crash of node x. The four histories O1, O2, O3 and O4 illustrate the possible directory inconsistencies which may arise due to different patterns of read (r) and write(w):

Case/ Type	History	Destroyed $GDir^x$	Destroyed $LDir^y$	Update Anomaly	Directory Inconsistency
O1	$w^x[i];\ w^y[j];\ Crash^x$	yes	no	Spurious w^x	Lost $GDir^x(p)$
O2	$w^x[i];\ r^y[j];\ Crash^x$	yes	no	Spurious w^x	Lost $GDir^x(p)$
O3	$w^y[j];\ w^x[i];\ Crash^x$	yes	no	Lost w^y	—
O4	$w^y[j];\ r^x[i];\ Crash^x$	yes	no	—	Lost $GDir^x(p)$

Table 1: Directory Inconsistencies and Update Anomalies when x, the Owner of p, Crashes.

These four cases exemplify all the recovery problems which may arise *due to the crash of the owning node.* Cases O1 and O2 exemplify uncommitted data migration and replication from the owner to a loaner respectively, while cases O3 and O4 exemplify uncommitted data migration and replication from a loaner node to the owner. Since these cases exemplify general patterns of update anomalies and directory inconsistencies, they are also referred to as *types* of inconsistencies.

5.2 Loaner Crashes

Table 2 illustrates the possible directory inconsistencies and update anomalies which may arise due to *loaner* crashes: The histories shown are abbreviated L1, L2, L3 and L4. These four cases exemplify all recovery problems which may arise *due to the crash of a loaner node.* Cases L1 and L2 exemplify

Case	History	Destroyed $GDir^x$	Destroyed $LDir^y$	Update Anomaly	Directory Inconsistency
L1	$w^x[i];\ w^y[j];\ Crash^y$	no	yes	Lost w^x[4]	Spurious $GDir^x(p)$
L2	$w^x[i];\ r^y[j];\ Crash^y$	no	yes	—	Spurious $GDir^x(p)$
L3	$w^y[j];\ w^x[i];\ Crash^y$	no	yes	Spurious w^y	—
L4	$w^y[j];\ r^x[i];\ Crash^y$	no	yes	Spurious w^y	—

Table 2: Directory Inconsistencies and Update Anomalies when y, the Loaner of p, Crashes.

uncommitted data migration and replication from the owner to a loaner respectively, while cases L3 and L4 exemplify uncommitted data migration and replication from a loaner node to the owner.

6 Recovery in Directory Based Coherency

Since a node crash destroys the volatile memory on the crashed node, recovery in DBC requires that *directories* and *data* be recovered.

- Recovery on Survivor Nodes.
  1. Directories must be cleaned up so as to eliminate spurious references to pages on the crashed node(s).
  2. Lost updates are restored.
  3. Spurious updates are eliminated.
- Recovery of Crashed Nodes.
  1. The $GDir$ must be reconstructed to include entries for pages held on surviving nodes.
  2. Traditional recovery follows to ensure FA.

After a survivor node recovers but during recovery of the crashed node, transactions executing on survivors can continue to make progress until they need to lock data items held in owned by the crashed node. Figures 3 and 4 summarize the steps for recovering directories on surviving and crashed nodes.

Section 6.1 shows how directories can be recovered, Section 6.2 shows how runtime extensions can be employed to eliminate some types of update anomalies, and Section 6.3 shows how data can be recovered.

[4]We show later in this section how this update anomaly can be avoided by extending the DBC protocol.

```
RecoverSurvivor()
{
        /* Cleanup Directories */
        CrashPageList = SurvivorCleanupLDir();
        SurvivorCleanupGDir();

        /* Restore Lost Updates of type O3 (type L1 are fixed with OLV) */
        if (CrashPageList != 0)
               for (i=StartLSN, i<LastLSN, i++)
                   if ((Log[i].page ∈ CrashPageList) AND RedoCondition(Log[i]))
                       RedoUpdate(i);

        EliminateSpuriousUpdates();
}
```

Figure 3: Recovery on Surviving Nodes

6.1 Recovering Directory Structures

In this section we show how lost *GDir* entries can be recovered (section 6.1.1) and how spurious *GDir* entries can be eliminated (6.1.2) after the failure of a node.

6.1.1 Recovering Lost *GDir* Entries

Consider the lost *GDir* entry problem, which arises in cases O1, O2, and O4. Our basic approach is to reconstruct, when x recovers, the *GDir* of x based on any *LDir* entries which mirror the entries that were on x prior to the crash. This is performed as follows:

1. Each surviving node y examines its *LDir*, then sends (to node x) a list of pages owned by x.

```
RecoverCrashed()
{
        ∀ s ∈ Survivors
            Receive(s,Pagelist);

        Reconstruct_GDir();

        Follow Traditional Recovery.
}
```

Figure 4: Recovery on Crashed Nodes

2. Upon the reception of these lists from all survivors, x initializes its *GDir* to contain entries for all the pages in the receive list.

The pseudo-code for step 1 of the above strategy is illustrated in Figure 5, denoted *SurvivorCleanupLDir()*. The figure also includes the definition of the *GDir* and *LDir* data structures. To simplify the presentation, *GDir* and *LDir* are represented as arrays (of structures) indexed by a page identifier – in practice, for performance reasons, hash tables are preferable to arrays.

After each survivor completes SurvivorCleanupLDir(), the recovering (crashed) node performs step 2. That is, cases where $GDir(p)$ was destroyed but $LDir(p)$ survived (as in O1, O2, and O4), at recovery time, $GDir(p)$ is reconstructed to reflect all $LDir(p)$s. Recall that O3 causes no directory inconsistencies. Thus, this strategy resolves all lost *GDir* entries. Note that this algorithm can easily handle multiple node crashes, simply by invoking SurvivorCleanupLDir() once for each crashed node.

```
typedef struct GDir {
      int pid;
      int mode;
      int readers[];
      int writer[];
      int *p;
} GDir[MAX_GLOBAL_PAGES];

typedef struct LDir {
      int pid;
      int mode;
      int owner;
      int *p;
} LDir[MAX_LOCAL_PAGES];

/*
   The surviving node constructs a list of all pages in the
   local directory (LDir) which were owned by the crashed node,
   then sends this list to the crashed node.
*/
SurvivorCleanupLDir()
{
      CrashPageList = ∅;
      ∀ page ∈ LDir
         if (LDir[page].owner == CrashedNode)
            Append(CrashPageList,LDir[page].pid);
      Send(CrashedNode,CrashPageList);
      return(CrashPageList);
}
```

Figure 5: Resolving Local Directory Inconsistencies on Survivors

6.1.2 Eliminating Spurious *GDir* Entries

Our approach is to eliminate *GDir* references that indicate that a page is held by a node that crashed.

To this end, for all surviving nodes x,

- Examine $GDir^x$, and eliminate references to pages held by crashed nodes. These references fall into two categories:

  1. Those of type L1.

     Resolving these and the related lost update of type L1 is potentially expensive. Even though the redo log for the update done by w^x is likely to be in x, it is possible that no node has page p in its memory to apply this redo log. It is possible that y may have written p to its stable storage, but this can be resolved only after y recovers and in that case, it will require access to the disk of the failed node to obtain the most recent version of p. As we show in the next section, lost updates of type L1 and the related Spurious GDir entry of type L1 can be avoided with a simple extension to DBC.

  2. Those of type L2.

     If the crashed node is one of many holders of p (i.e., in shared mode), then the crashed node is merely deleted from the *readers* of $GDir^x(p)$.

6.2 Runtime Recovery Extensions for DBC

Masking lost updates of type L1 can be potentially expensive if performed at restart recovery time. This is because of the need to obtain the previous version of p – which must either be reconstructed from the log (this is feasible if entire pages are logged), or otherwise the page must be fetched from disk. In [15], it was shown that a software coherency protocol can be extended (in the case of a write miss) in order to avoid going to disk at restart recovery time.

> COHERENCY PROTOCOL EXTENSION: **Owner Last Version (OLV)**
> Whenever the owner grants X access to another node, the owner maintains a copy of the previous version of the page. This version is not used unless the current holder of the page crashes.

Given this extension, the resolution of spurious *GDir* entries of type L1 can be performed very quickly as follows.

> OLV RECOVERY:
> If the only (loaner) holder of p crashes, *writers* is reinitialized to be the owner node (and *readers* is assigned to null). At this point, the old version of p stored at the owner becomes the current version of the page.

Using OLV, the pseudo-code for resolving global directory inconsistencies on surviving nodes is given in Figure 6. Note that SurviviorCleanupGDir() can easily handle the case where multiple nodes crash by invoking it once for each crashed node.

To see how OLV avoids update anomalies of type L1, consider the history corresponding to type L1, ($w^x[i]$; $w^y[j]$; $Crash^y$). During normal runtime, after $w^x[i]$; $w^y[j]$, node x retains the version of p produced by $w^x[i]$. After $Crash^y$, since the only holder (node y) crashed, node x (the owner node of p), installs the previous version of p ($v^{-1}(p)$). Thus, no lost updates arise (i.e., w^x is not lost).

Note that this mechanism may introduce additional cases where spurious updates may arise, but these cases are handled by the restart recovery protocols of Section 6.3. This may occur when two loaner nodes alternately update a page, then one of these nodes crashes, such as in the following history, where $i, j, k \in p$, and $owner(p) = x$:

$$w^z[i];\ w^y[j];\ w^z[k];\ Crash^z.$$

In this case, the restoration of $v^{-1}(p)$ by the owner results in $w^z[i]$ and $w^y[j]$ being reflected in the post recovery version of p. Of these two writes, $w^z[i]$ is spurious. However, this does not present any additional problems, since spurious undos by transactions executing on failed nodes are eliminated with the logging and restart recovery protocols described in Section 4.

6.3 Recovering Data in Directory Based Coherency

In the previous section, we have described how software-based coherency protocols can be extended to deal with lost update in L1. Here we give general solutions to deal with lost and spurious updates to data in DBC systems. An important issue is that in DBC systems, the use of private (non-shared) disks affects the design of recovery strategies (in contrast with SMPs). To this end, we consider extensions to the *Selective Redo* restart recovery scheme [16], originally proposed for handling recovery in shared-memory database systems, since it enables independent recovery on survivors. In Selective Redo, the restart recovery strategy for surviving nodes consists of two basic steps.

1. Each surviving node performs redo for only those data item 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 information (described below) stored along with each data item.

Selective Redo minimizes the number of redos – only updates made by surviving nodes which were present *exclusively* on the crashed node may need redo.

As a result of the first step of the Selective Redo protocol, all updates made by surviving nodes will be restored. However, the caches of surviving nodes may also contain spurious updates – those made by transactions that were running on crashed nodes. Thus, we need an additional mechanism to identify the *active* data items, i.e., data items updated by active transactions, which require undo. This is described in Section 6.3.2 which discusses how spurious updates on crashed nodes can be masked without requiring access to the crashed node's stable storage. Section 6.3.3 discusses additional issues related to recovery on the crashed node. First we deal with the restoration of lost updates.

6.3.1 Handling Lost Updates in DBC

As tables 1 and 2 show, lost updates of type O3 or L1 may arise due to node crashes. We have already shown how OLV can be used to mask lost updates of type L1, thus, in this section we focus on the resolution of lost updates of type O3. To accomplish this, the directory structure of DBC is used to efficiently support selective redo for surviving nodes. This is illustrated in the RecoverySurvivor() procedure of Figure 3. The SurvivorCleanupLDir() routine returns a list of pages which were owned by the crashed node and were locally cached. Since we only need to mask lost updates of type O3, for a specific surviving node, these are the only pages that could possibly need redo. If the CrashPageList is empty, then no redo need be performed.

Note that Figure 3 refers to a *RedoCondition*, which is necessary to avoid the redo of pages that already reflect the logged update. For an update U, *Redo-Condition* evaluates to true if either of the following conditions is true: (a) U was performed by a surviving active transaction, and U has not been propagated to the stable database (as may arise due to steal), or (b) U was performed by a committed transaction, and U has not been propagated to the stable database. These conditions can be efficiently evaluated by using techniques and data structures which are already maintained for use in traditional restart recovery [7, 13]. These data structures are used to (a) determine the earliest point in the log to begin forward processing the log, and (b) for a given page, determine whether a particular update needs to be redone.

6.3.2 Handling Spurious Updates with Page Logging

We seek recovery strategies for surviving nodes that are *independent* of crashed node recovery. If the crashed node manages a private disk, surviving nodes must wait for the crashed node to reboot before fetching the undo log records from the crashed node's disk. Moreover, the crashed node may require substantial disk I/O for its own local recovery, causing a bottleneck for access to the crashed node's disk. In loosely coupled systems, one extreme example of recovery delay on the crashed node arises when a node disconnection occurs and surviving nodes lose contact with the crashed node –

possibly for hours or days.

Dealing with fine-granularity sharing is another significant challenge, since it allows a single page to contain multiple active updates.

To address these problems, we propose techniques for eliminating spurious updates based on an *undo log shipping* mechanism. Each data page has an associated *undo log page*. *Whenever a data page containing active data is transferred to another node, the associated undo log page is also transferred*[5].

For each active data item in the data page, the before image (BI) is stored in the undo log page, as show in Figure 7[6]. The figure graphically illustrates this data structure, and also provides the C language definition of the data structure. Associated with each BI in the log page is the ID (the *tag*) of the node that performed the active update. This undo log page is initialized as follows:

- When a page is first loaded in to the database buffer from disk, the undo log page is initialized. That is, the BI's are assigned to the current values of the data items in the data page, and all Node IDs in the undo log page are assigned to the local node.
- When a page is forced to disk, the undo log page is discarded from volatile memory.
- At commit time, all after images are copied to the corresponding undo log page for subsequent use.

Note that BI/AI logging technique, and its adaptation to the undo log page, is appropriate when strict two-phase locking is used for writes – where any active data item reflects the updates of at most one transaction, and locks on these data items are not released until transaction commit. This is typically the assumption of commercial database systems. If strict two-phase locking were not used, an active data item could reflect the updates of more than one transaction, and correct recovery would be more complex.

Note that to simplify the presentation, the pointer to the undo log page is shown as a field of the data page. To avoid maintaining this pointer in the data page, an alternative representation, which stores the pointer to the undo log page in the directory structure, is also feasible.

Figure 8 shows the pseudo-code for accomplishing the elimination of spurious updates. Pages in either the *LDir* or the *GDir* could contain spurious updates. These pages are examined, and the

[5]Log shipping has also been proposed for other purposes, such as optimizing two-phase commit [1].

[6]Note that undo logs also need to be maintained in traditional DBMS that allow buffer steals.

updating node (the *Node ID*) was the crashed node, undo is accomplished by assigning the before image (stored in the undo log page) to the data page's data item.

Clearly, the use of the undo log page requires that twice the amount of data is sent between nodes during coherency operations for database pages. However, in many cases, this may be an acceptable tradeoff because of the independent recovery of survivor nodes facilitated by log shipping.

6.3.3 Additional Considerations for Recovery on the Crashed Node

In this section, we consider some additional issues related to recovery on the crashed node. These additional issues are related to the maintenance of dirty page tables (DPTs) on each node. For example, because of the way that the dirty page table is managed on client and server nodes, pages owned by the crashed node that were not present in the cache of any surviving node may be missed with the basic ARIES restart recovery algorithm. For example, consider page p at server s. Suppose p was updated by a client transaction, and then was transferred back to the server's cache after the server's last logged checkpoint, then the server crashes. This results in a dirty page which (a) was present only in the cache of the server, (b) is not in the crashed server's last checkpointed dirty page table, (c) was not updated by the crashed node after the last checkpoint was written.[7]

To recover these pages in the event of a server crash, *coordinated recovery* [17][8] can be used, where information from the `DPTs` (dirty page tables) of all clients are merged at recovery time, in order to determine which pages need redo. Specifically, the protocol involves each remote node sending (to the crashed node) (a) its list of cached pages owned by the crashed node, and (b) all entries in the `DPT` owned by the crashed node. After this is complete, the crashed node merges these and its own `DPT` to determine which pages are dirty. In order to support an ARIES style recovery method, these methods for determining which pages are dirty and for determine where to start the redo log scan can be adopted to support IFA.

7 Related Work

Directory-based coherency has been implemented in basically two types of database systems, SD (shared-disk), and Client-server systems. We cover this work in chronological order, starting with SD systems.

Some of the earliest published work on recovery in SD systems includes Digital Equipment Corporation's recovery in VAXcluster systems [10, 21, 23]. This recovery architecture has shaped

[7]These are essentially the same conditions for which *missing redos* may result in ESM-CS [5] and ARIES/CSA [15], which we discuss in Chapter 7.

[8]Note that in this design, both locking and sharing are done at the page level, which restricts concurrency.

the basic strategies for handling issues that arise in many other multiple node recovery systems. For example, in the VAXcluster, each node can manage a lock table for a partition of the database. In the event of a node failure, the strategy for recovering lock tables is as follows: All cluster nodes must complete the recovery of their lock managers before cluster operation can continue. First, a lock manager deallocates all locks acquired on behalf of other nodes. Next, a lock manager reacquires each of the locks it had when the transition began. The net result is to release locks held by a node if that node has left the cluster. This can be viewed as one instance of our more general notion of undo's of changes to the state of data. However, there are at least three significant differences between recovery in a VAXcluster and our recovery approach. The VAXcluster architecture employs a single log (as opposed to multiple local logs), a undo/no-redo recovery strategy (that is, at commit, force is employed), and prior to migration, a page is forced to the stable database. Even though these polices simplify recovery, they degrade performance because of the disk writes caused by the force policy.

Issues of interest to us have also been examined in [13, 14, 19, 20] where coherency is implemented in software by transferring *pages* between nodes. But there are some significant differences:

- Most of these systems do not support fine-granularity (data item) locking [19, 20]. Instead, page granularity locking is used in conjunction with strict two-phase locking. The main drawback of this approach is the loss of concurrency resulting from page granularity locking.
- The SD proposals that support fine granularity (data item) locking [12, 14] ensure FA. They can ensure IFA only by performing stable logging to disk prior to the migration of uncommitted data. Our protocols ensure IFA without this negative performance impact.

Given these differences, in this section we review the proposals of [12, 14], and consider how the FA ensuring mechanisms defined for these systems can be adapted to achieve IFA. These SD recovery proposals are based on ARIES [13], which uses the repeating history paradigm followed by undos to recover from failures.

The SD system in [14] addresses some of the issues related to the migration of uncommitted data. In order to ensure inter-node coherency, [14] 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. This, in conjunction with traditional recovery approaches, satisfies IFA. In the super-fast scheme, all updates to page p must be volatile logged prior to p's migration. The super-fast scheme does not support IFA, since no provision is made for handling the undo of active updates, performed by transactions running on crashed nodes - necessary in case these active updates had migrated to another node. Similarly, the approach of [12] does not define an undo approach for these cases.

Finally, some client-server systems [18] allow a page to be updated simultaneously by many client nodes (multiple-writers). Allowing multiple-writers requires that the server eventually merges the updates. The main disadvantage of this approach is that additional book-keeping must be done in order to track the portions of a page which have been updated by a client and to recover properly from either client or server crashes. For example, to handle recovery properly, servers must remember the current PSN (page sequence number) of p each time p is sent to a client. To handle server crashes, prior to forcing a page to the server's disk, a list of the PSN values that the server remembered is first logged. Thus, allowing clients to simultaneously update a page imposes additional overheads on the runtime and the logging system.

For a client-server system allowing multiple simultaneous writers, under some conditions, some of the recovery problems discussed in this paper can be avoided. For example, if clients only send committed updates back to the server, and the server does not perform updates, uncommitted data will not migrate. However, in the absence of these conditions, uncommitted data will migrate, and additional measures must be employed to ensure IFA. This is necessary if the additional performance that can be derived from relaxing these conditions can be exploited by the application.

8 Summary

In this paper, we have presented protocols for ensuring IFA recovery in DBC systems which allow fine-granularity locking. In contrast with our previous work in the area of recovery in shared-memory database systems [16], in DBC systems, directory recovery must also be addressed, and the fact that software coherency protocols allow for runtime modifications must be exploited, to to facilitate recovery. In this paper, we have analyzed the possible directory inconsistencies that may arise in DBC systems due to node crashes, and presented protocols that resolve these inconsistencies. These protocols require no additional runtime overheads – directory resolution is performed entirely at recovery time, and no additional logging (beyond what is normally done to ensure FA) is needed.

We have also investigated extensions to coherency that enable independent recovery of surviving nodes. During recovery of the crashed node, transactions executing on survivors can continue to make progress until they need to lock data items owned by the crashed node. We showed how OLV avoids lost updates of type L1. In DBC, OLV is an inexpensive optimization, requiring only that the owner node cache a copy of any page that contains an uncommitted update. We also proposed a novel mechanism for eliminating spurious updates, *undo log shipping*, which has the advantage that communication with the crashed node is not needed during recovery of surviving nodes. By eliminating spurious updates from a surviving node independently of the crashed node, IFA can be ensured, while allowing new transactions to begin on surviving nodes with minimal delay. A by-product of our systematic development of recovery techniques is that it helps to demystify some of the facets of multiple node recovery.

Some distributed shared memory architectures enforce sequential consistency [25], while others support more relaxed memory consistency models, such as *release consistency* [6] or *lazy release consistency* [9]. Here we have assumed that sequential consistency, which is equivalent to serializability is the desired correctness. Achieving IFA under a relaxed memory consistency model is worthy of examination.

References

[1] Y. Al-Houmaily and P. Chrysanthis. A Two Phase Commit Optimization for Gigabit-Networked Databases. *Proceedings of the 8th International Conference on Parallel and Distributed Computing Systems (PDCS)*, pages 554–560, September 1995.

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

[3] I. Censier and P. Feautrier. A New Solution to Coherence Problems in Multi-Cache Systems. *IEEE Transactions on Computers*, 27(12):1112–1118, 1978.

[4] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-Based Cache Coherence in Large-Scale Multiprocessors. *IEEE Computer*, pages 49–58, June 1990.

[5] M. Franklin, M. Zwilling, C. Tan, M. Carey, and D. DeWitt. Crash Recovery in Client-Server EXODUS. *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, 21:165–174, June 1992.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.

[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] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. *Proceedings of the 19th International Symposium on Computer Architecture*, pages 13–21, May 1992.

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

[11] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Database Systems*, 7(4):321–359, November 1989.

[12] D. Lomet. Recovery for Shared Disk Systems Using Multiple Redo Logs. *Digital Equipment Cambridge Research Laboratory Technical Report*, (4), October 1990.

[13] 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.

[14] 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.

[15] C. Mohan and I. Narang. ARIES/CSA: A Method for Database Recovery in Client-Server Architectures. *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, 23:55–66, 1994.

[16] L. Molesky and K. Ramamritham. Recovery Protocols for Shared Memory Database Systems. *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 11–22, May 1995.

[17] E. Panagos, A. Biliris, H. Jagadish, and R. Rastogi. Client-Based Logging for High Performance Distributed Architectures. *Proceedings of the 13th IEEE Conference on Data Engineering*, 1996.

[18] E. Panagos, A. Biliris, H. Jagadish, and R. Rastogi. Fine-granularity Locking and Client-Based Logging for Distributed Architectures. *Proceedings of the Seventh Conference on Extended Database Technology*, 1996.

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

[20] 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.

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

[22] R. Schlichting and F. Schnieder. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computing Systems*, 1(3):222–238, August 1983.

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

[24] M. Stumm and S. Zhou. Algorithms Implementing Distributed Shared Memory. *IEEE Computer*, pages 54–64, May 1990.

[25] M. Tam. CapNet – Using Gigabit Networks as a High Speed Backplane. *PhD. Thesis, Department of Computer and Information Science, University of Pennsylvania*, 1994.

[26] C. K. Tang. Cache Design in the Tightly Coupled Multiprocessor System. *AFIPS National Computer Conference*, pages 749–753, 1976.

```
/*
    Remove from readers and writers of the survivor (s) any pages owned by s
    that were resident on the crashed node. If this results in empty writers
    or readers, then restore the previous version of p at the owner node.
    This is accomplished by setting the mode of GDir to be exclusive.
*/
SurvivorCleanupGDir()
{
        ∀ p ∈ GDir
           if (GDir[p].mode == shared)
                 if (CrashedNode ∈ GDir[p].readers {
                       Delete(GDir[p].readers,CrashedNode);
                       if (GDir[p].readers == ∅)
                             GDir[p].mode = exclusive;
                 }
           else if (GDir[p].mode == exclusive)
                 if (CrashedNode ∈ GDir[p].writer {
                       Delete(GDir[p].writer,CrashedNode);
                       GDir[p].mode = exclusive;
                 }
}
```

Figure 6: Resolving Global Directory Inconsistencies on Survivors

```
struct p {                          typedef struct UndoLog {
     int PSN;                            int nodeID[NumItems];
     UndoLog *ulog;                      string data[NumItems];
     string data[NumItems];              } UndoLog;
}
```

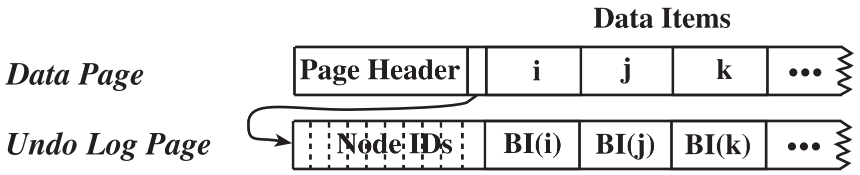

Figure 7: Masking Spurious Updates with Page Logging.

```
/*
   Eliminate Spurious Updates.
   Any Active Updates made by the Crashed
   Node are assigned the before image.
   */
EliminateSpuriousUpdates()
{
      ∀ p ∈ (LDir ∪ GDir) {
           ∀ item ∈ p
             if (p->ulog.nodeID[item] == CrashedNode)
                  p.data[item] = p->ulog.data[item];
      }
}
```

Figure 8: Eliminating Spurious Updates.