Recovery Protocols for Cache-Coherent Shared Memory Operating 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

December 9, 1994

Abstract

Significant performance advantages can be achieved by implementing certain kernel data structures in the cache-coherent memory of a shared memory multiprocessor. However, even in multiprocessor systems which are capable of detecting and isolating node failures, implementing kernel data structures in shared memory can cause recovery problems. For example, in the disk buffer, when disk blocks are buffered in cache-coherent memory, unnecessary dependencies may form between processes which access these buffers. Because of these dependencies, the crash of one node may adversely affect the integrity of processes which execute on nodes other than the one(s) which failed, violating the forward progress of these processes. Moreover, resources held by processes running on crashed nodes may fail to be released, violating system liveness properties.

In this paper, we propose low overhead recovery techniques which mask these dependencies. These techniques are based on compile time, runtime, and recovery time techniques. Low overheads are achieved by using protocols that to not need stable storage, since this may involve the more costly disk I/O. We consider the application of these recovery techniques to those kernel data structures which are suitable for implementation in cache-coherent shared memory. These kernel data structures include semaphores, maps, used to catalog disk usage, and the disk buffer.

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

1 Introduction

This paper considers the crash recovery issues which arise when kernel data structures are implemented in cache-coherent shared memory. Significant performance advantages can be gained by implementing kernel data structures in shared memory. Many kernel data structures are suitable for implementation in shared memory, including semaphores, the disk buffer, and maps used to catalog disk usage. However, when kernel data structures are implemented in coherent shared memory, failure dependencies (caused entirely by side effects of the cache coherency protocol) may form between processes and the volatile memory of other nodes[1].

With conventional methods, when a node crash occurs (whereby the contents of the physical memory on the failed node are destroyed), restoring the system to a consistent state is likely to require a reboot of the entire shared memory machine. But this method is overkill, since many processes are unnecessarily aborted (terminated prematurely). Instead, in the event of a node crash, our recovery protocols guarantee the *forward progress* of all processes executing on surviving nodes (nodes which did not crash). Furthermore, in avoiding unnecessary process aborts, our recovery protocols ensure a consistent system state by providing *liveness* guarantees. Liveness properties are supported by ensuring that when processes are prematurely terminated due to the crash of the node executing these processes, all resources held by these processes are released. Thus, our recovery protocols are designed to tolerate the crash of one or more nodes in a multiprocessor, yet allow the remaining nodes to continue execution.

This recovery strategy is motivated by proposed enhancements to multiprocessor failure models where certain low-level mechanisms are in place to detect and isolate hardware faults [8]. However, these alone are not enough to guarantee that unnecessary failure dependencies do not form between nodes. Our recovery techniques consist of low overhead software mechanisms which compensate for the unwanted and deleterious side effects of the cache coherency protocol. These mechanisms include compile time and runtime mechanisms which maintain the requisite information in case recovery is necessary, and a restart recovery mechanism which actually restores the system to a globally consistent state when a failure occurs. Low overheads are achieved by avoiding accesses to stable store, since this may involve the more costly disk I/O.

[1]The term *node* refers to a processor/memory pair.

These recovery problems surface in the two main types of cache-coherent shared memory architectures, NUMA (non-uniform memory access) and COMA (cache-only memory architectures). We identify which patterns of data sharing cause recovery problems in these architectures. These recovery problems arise in both write-invalidate and write-broadcast cache coherency protocols.

This paper is organized as follows. Section 2 presents our multiprocessor and failure model. Section 3 discusses the failure dependencies associated with cache coherency. In section 4, we present our recovery protocols, and discuss their application to kernel data structures implemented in coherent shared memory. Section 5 gives a technical summary, related work is discussed in section 6, and our concluding remarks are made in section 7.

2 System Model

The multiprocessor model used in this paper assumes that each node is connected to all disks in the system, and each node has access to shared memory. 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 [1, 9]. 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.

In this context of a cache-coherent multiprocessor, we assume the following:

- An operating system kernel executes at each node.
  Each kernel is responsible for the management of processes executing at that node. Associated with process management, each kernel is responsible for managing virtual and physical memory (including disk accesses) associated with each process.
- A node crash destroys the contents of all volatile memory on that node.
  This implies that all processes running on the crashed node fail, since their control state (registers, stack, etc.) is destroyed by the crash.

- Node crashes are independent.

  The crash of node x does not destroy the contents of volatile memory located on other nodes. This assumption is motivated by proposed enhancements to shared memory architectures whereby individual node failures are independent [8].

In the event of the crash of some node x, our recovery objective is to abort the kernel on node x, but allow all kernels executing on surviving nodes to continue execution in a consistent global system state. The abortion of a kernel x means that all processes managed by kernel x are aborted, and all resources acquired by kernel x are released. To ensure that kernels executing on surviving nodes continue to execute in a consistent global state, restart recovery ensures that *no* resources allocated by these kernels are released prematurely.

When node x crashes while holding the only copy of a cache line, we assume that this cache line is unavailable for use by other nodes until a restart recovery mechanism is applied. Node crashes which do not destroy the only copy of a cache line do not render these cache lines unavailable. The restart recovery mechanism is used to reconstruct some or all of the dirty data of the cache on node x.

When a cache line l is resident on node x and another node y desires to update line l, l *migrates* from node x to node y. While the unit of I/O is a page, the unit of coherency is a *cache line*, and is typically smaller than a page.

We consider recovery issues in the context the two major cache coherency protocols, *write-invalidate* and *write-broadcast* [1, 9]. In both these protocols, a cache line could be valid at multiple nodes after a series of read requests to the line have been issued. However, these protocols differ when write requests are issued: In a write-invalidate protocol, a write to a cache line *invalidates* all other copies of the line. In contrast, in a write-broadcast protocol, a write to a cache line *triggers the update* of all other copies of the line.

We consider the two main types of cache-coherent architectures, NUMA, and COMA. In the case of NUMA, volatile memory consists of primary memory and cache memory. In the case of a COMA (Cache Only Memory Architecture), volatile memory consists only of cache memory. In the context of NUMA, we consider two variants of the write-invalidate protocol – *write-through* and *write-back*. In the write-back variant, writes initially modify the cache, but changes are not reflected in primary memory until the block is removed from the cache. In a write-through variant, cache writes are immediately propagated to main memory.

3 Failure Dependencies Induced by Cache Coherency

When kernel data structures are implemented in shared memory, node crashes may leave the resulting operating system in an inconsistent state. These inconsistencies result as a side effect of the cache coherency protocol, and can stem from two basic patterns of cache line updates: *false sharing* and *concurrent updates*. False sharing occurs when two or more different data items, stored in the same cache line, are updated by different kernels. A concurrent update occurs when a *single* data item reflects updates made by different kernels.

An example of false sharing occurs when two semaphores are stored in the same cache line, and each semaphore is updated by different kernels. Two kernels, each of which updates a shared counter is an example of a concurrent update. In the event of a node crash, either of these update patterns may lead to the following inconsistencies:

- Lost Updates
- Spurious Updates

Lost semaphore updates may allow resources to be prematurely freed. A spurious semaphore update may cause resources to be blocked indefinitely. Similarly, lost allocation records, stored in a map, may cause the kernel to free disk space prematurely, and spurious allocation operations may cause the kernel to hold space indefinitely.

Consider the case where operating system kernels manage *independent* processes – i.e., processes which execute strictly on one node and do not communicate with processes on other nodes. Under this assumption, the definitions of lost and spurious updates are the following[2]: A lost update occurs when a node crash destroys an update performed by a kernel executing on a surviving node. A spurious update occurs when a node crash aborts a kernel, but one or more updates made by that kernel are not destroyed.

To illustrate, we show how lost updates can occur under certain (common) variants of write-invalidate, and how spurious updates can occur in any protocol. Examples of how false sharing can cause these anomalies are given in section 3.1. Examples of how concurrent updates can cause these anomalies are given in section 3.2.

[2]Extensions for dependent processes, and potential exceptions to the definitions of lost and spurious updates are considered in section 6.

3.1 False Sharing

In this subsection, we consider examples of how lost and spurious updates may occur when false sharing of a cache line occurs. In these examples, operations on semaphores illustrate false sharing. Two semaphores, $sem1$ and $sem2$, stored in cache line l. False sharing occurs when operations on $sem1$ and $sem2$ are issued from different nodes.

In these examples, we assume that x and y are different nodes. The crash of a particular node y is indicated with $Crash_y$. Although semaphores are typically implemented with multiprocessor synchronization primitives, such as *test-and-set* (TAS) or *compare-and-swap* (CAS) [4], for clarity of the presentation, we denote a successful semaphore acquisition with an increment (*inc*) operation. This simplification is accurate since TAS and CAS normally perform operations on one or two words stored in a single cache line.

3.1.1 Lost Updates

In history H_{fs1}, line l is falsely shared when an increment to $sem1$ is issued by the kernel executing on node x, then an increment to $sem2$ is issued by the kernel executing node y.

$$H_{fs1} \quad = \quad inc_x[sem1];\ inc_y[sem2];\ Crash_y.$$

Consider the recovery implications of these two histories in the context of write-invalidate and write-broadcast protocols. First consider variants of the write-invalidate protocol where cache line updates are *not* immediately propagated to primary memory. These include COMA write-invalidate protocols and NUMA write-back variants, where changes are only reflected in main memory when the block is removed from the cache.

After $inc_x[sem1];\ inc_y[sem2];$, the *only copy* of line l resides in node y's cache. If node y crashes at this point, as in history H_{fs1}, a lost update occurs – even though kernel x will remains active, the update made by kernel x to $sem1$ is destroyed.

Note that the write-through variant of the write-invalidate protocol does not suffer from the lost update problem. After $inc_x[sem1];\ inc_y[sem2];$, copies of l will reside in the primary memory of node x, and in the cache of y. Thus, in H_{fs1} a lost update does not occur, since the copy of l containing the effects of $inc_x[sem1]$ remains intact in the primary memory of node x.

Write-broadcast protocols do not suffer from the lost update problem. For example, in history H_{fs1}, after $inc_y[sem2]$, the write-broadcast protocol dictates that the updated contents of line l be broadcast to all nodes holding a copy. Thus, lost updates do not occur in write-broadcast (or write-through) protocols, since if all copies of an update were destroyed, the kernel which made the update would be aborted.

3.1.2 Spurious Updates

As the following example indicates, spurious updates can occur when false sharing occurs:

$$H_{fs2} \;=\; inc_x[sem1];\; inc_y[sem2];\; Crash_x.$$

In history H_{fs2}, a spurious update occurs under both write-invalidate (both variants) and write-broadcast. After $inc_x[sem1];\; inc_y[sem2];$, the most recent copy of the cache line containing $sem1$ and $sem2$ is stored on node y. If node x crashes after $inc_x[sem1];\; inc_y[sem2];$, even though kernel x is aborted, the update made by kernel x to $sem1$ remains intact.

In certain histories, such as H_{fs1} of section 3.1.1, spurious updates occur under write-broadcast, but not under any variant of the write-invalidate protocol.

3.2 Concurrent Updates

The same patterns of operation invocations cause recovery problems under concurrent updates. The following example indicates how lost and spurious updates can also occur when concurrent updates are done. In this example, a single integer *count*, stored in cache line l, is updated by two kernels.

$$H_{cu1} \;=\; inc_x[count];\; inc_y[count];\; Crash_y.$$

Under a write-invalidate protocol, a lost update occurs in H_{cu1} – even though kernel x remains active, the update made to *count* by kernel x is lost because (the cache containing) count is destroyed. Under a write-broadcast protocol, a spurious update occurs in H_{cu1} – even though kernel y is aborted, the update made to *count* by kernel y remains intact.

Thus, by substituting $inc_x[count];\; inc_y[count];$ for $inc_x[sem1];\; inc_y[sem2];$, the same histories which exhibited lost (or spurious) updates under false sharing also exhibit lost (or spurious) updates under concurrent updates. For example, given a particular cache coherency protocol, $H_{cu2} = inc_x[count];\; inc_y[count];\; Crash_y$ has the same recovery problems as H_{fs2}. Figure 1 summarizes these possible update anomalies, and indicates: (1) lost updates can only occur in write-invalidate protocols, (2) spurious updates can occur in any of the studied protocols, and (3) certain histories can result in spurious updates in a write-broadcast protocol, but not in a write-invalidate protocol. Because the write-invalidate protocol is the protocol favored by most multiprocessor implementations, in subsequent sections, we consider the implications of both lost and spurious updates.

	H_{fs1} or H_{cu1}	H_{fs1} or H_{cu2}
Write-invalidate (write-back) or COMA	Lost Update	Spurious Update
Write-invalidate (write-through)	—	Spurious Update
Write-broadcast	Spurious Update	Spurious Update

Figure 1: Cache Coherency Protocols and Update Anomalies.

These examples have illustrated how updates performed by different processors can become dependent on the memory of the nodes of other systems. This can occur due to side effects of the cache coherency protocol, *even when* there is no intentional or explicit interaction between kernels or between processes executing at different nodes. These dependencies are especially problematic when they affect kernels of a multiprocessor operating system. In these cases, one way to restore the system to a consistent state would be to reboot all nodes for which kernels had formed failure dependencies on the memory of remote nodes. But this method is overkill, since many processes are unnecessarily terminated prematurely. Instead, our recovery protocols avoid these unnecessary process terminations, while still ensuring a consistent overall system state. These protocols are the topic of the next section.

4 Recovery Protocols for Shared Operating Systems Data Structures

In this section, we propose low overhead recovery protocols for multiprocessor operating systems. Many multiprocessor kernel data structures can benefit through a shared memory implementation. These include *semaphores*, *maps*, and the *disk buffer* (a software structure). Similar, but subtly different recovery techniques are appropriate for each of these kernel data structures.

Our crash recovery protocols ensure global system consistency without needlessly terminating processes. These recovery protocols consist of *compile* and *runtime* steps, and a *restart recovery* procedure. Compile time steps can be used to reformat or reallocate shared data structures to avoid update anomalies, and runtime steps can be used to mask update anomalies by logging the requisite information which allows restart recovery to restore global system consistency.

In our approach to recovery, components of kernel data structures implemented in shared memory fall into two categories:

- Core Data – Core data typically reflects the state of a shared resource, in terms of *which* resources have been allocated, and *which* are free.
- Derived Data – Derived data is data that can be derived from core data. One example of derived data is a free list which is constructed from examining fields of core data.

Compile and runtime mechanisms are employed to ensure the consistency of core data. In the event of a node crash, once the consistency of core data is restored, the derived data is reconstructed from core data.

Our recovery techniques require that updates to core data are *strict*. Strict updates ensure that any particular data item reflects the updates of at most one kernel, i.e., concurrent updates to core data are not allowed. For example, this property is inherent in binary semaphores, since each semaphore can reflect the update of at most one kernel. Requiring that updates to core data be strict simplifies our recovery protocols, (1) allowing the data structure to be partitioned such that local updates do not migrate, and (2) allowing operation aborts to be implemented by simple inverse operations. These implications facilitate the recovery of core data: Implication 1 allows an implementation to avoid false sharing, and thus avoids lost updates due to cache coherency. Spurious updates can be undone at restart recovery time by applying the simple inverse operations allowed by implication 2.

False sharing can be avoided by dedicating an entire cache line to any data item stored in coherent shared memory. For certain data structures, i.e., those that are only referenced by a single node, dedicated cache lines will avoid all anomalies caused by cache coherency. However, for other reference patterns, dedicated cache lines are not sufficient to avoid spurious updates. For these reference patterns, undo logging can be used to avoid spurious updates.

While spurious updates are avoided by logging *Undo* information, lost updates are avoided by logging *Redo* information. As we will show in section 4.1, both of these techniques are feasible with *volatile* logging. By using both of these logging techniques, even false sharing can be tolerated.

We apply these recovery techniques to binary semaphores, maps, and the disk buffer. Section 4.1 shows how these recovery techniques can be applied to binary semaphores, and demonstrates that (1) dedicated cache lines *are* sufficient to avoid lost updates, and (2) dedicated cache lines *are not* sufficient to avoid spurious updates. This section also illustrates how volatile logging techniques can be used to mask lost and spurious updates. Unlike

binary semaphores, maps contain concurrently modified data. In section 4.2, we show how this concurrently modified data can be converted such that it can be derived from core data. Furthermore, for maps, since this core data is referenced only locally (i.e., no other node ever reads or writes this core data) dedicated cache lines are sufficient to avoid lost and spurious updates. Using the recovery techniques developed in sections 4.1 and 4.2, section 4.3 shows how the disk buffer can be recovered.

4.1 Binary Semaphores

Binary semaphores are often used in multiprocessor operating systems to ensure that resource allocation is performed mutually exclusively [2, 3]. In section 4.1.1, we consider the approach of using dedicated cache lines to solve the update anomalies caused by cache coherency. In section 4.1.2, we consider volatile logging techniques which allow false sharing, but mask update anomalies.

4.1.1 Avoiding False Sharing with Dedicated Cache Lines

One approach to solving the recovery problems caused by cache coherency is avoid concurrent updates and to dedicate an entire cache for each data item. For binary semaphores, one of these conditions is a given, since concurrent updates are not possible on binary semaphores – at any time, a binary semaphore will reflect the update of at most one kernel or process. Ensuring that each semaphore is allocated an entire cache line can easily be performed with compile time techniques. However, this approach is likely to waste cache line space, and does not avoid spurious updates.

Dedicated cache lines are appropriate if the cache line size is relatively small, and/or the number of semaphores in the system is small. The drawback of dedicated cache lines is that a lot of space will be wasted. For example, on the KSR-1 and KSR-2 multiprocessors [13], and on Stanford's FLASH [8] distributed shared memory machine, the cache line size is 128 bytes. On these architectures, dedicating an entire line for a semaphore (which typically consume 1 - 4 bytes), would waste the better portion of the cache line.

When binary semaphores are stored in dedicated cache lines, lost updates will not occur. Once acquired, a binary semaphore will not be updated by any other node until it is released. Thus, with dedicated cache lines, once acquired, the semaphore will reside on the node on which it was acquired until it is released, avoiding lost updates.

However, even with dedicated cache lines, spurious updates may occur. This can occur in both write-invalidate and write-broadcast protocols when a data item is referenced

concurrently by a single writer and one or more readers. For example, after a semaphore has been acquired by one node, reads to the semaphore (denoted by $r[sem]$) may occur due to unsuccessful TAS or CAS operations:

$$H_{spur} \quad = \quad inc_x[sem];\ r_y[sem];\ Crash_x.$$

After $inc_x[sem];\ r_y[sem];$, the cache line (l) containing sem is valid (in shared mode) on both nodes x and y. If node x crashes at this point, a spurious update (made by node x) will remain on node y.

Thus, even under strict updates, dedicated cache lines are not sufficient to ensure that spurious updates do not occur. In the next section, we present a solution based on volatile logging which ensures that lost and spurious updates do not occur. Although dedicated cache lines are not required for these techniques, the undo logging technique alone can be used with dedicated cache lines to ensure that spurious updates do not occur.

4.1.2 Tolerating False Sharing with Volatile Logging

By using volatile logging techniques, false sharing can be tolerated, yet lost and spurious updates can be masked. In these logging techniques, each update operation requires a log record to be written, indicating the steps which must be taken to redo and undo the operation. All logging operations take place in the cache of the local node (which performed the allocation or update operation).

A lost update occurs (in write-invalidate protocols) when data updated by a surviving node is destroyed due to data migration and a subsequent node crash. In this case, if sufficient redo information is logged, the lost update can be restored during restart recovery. Furthermore, since the updating system survived the crash, the volatile memory of the updating system can be used to store the redo log. Since updates to the log are performed only by the local system, with proper memory alignment (i.e., a cache line which contains local redo log information stores no other sharable information), we can ensure that local redo logs do not migrate between systems.

For semaphores, lost updates can be masked under false sharing with redo logging. Each update to the semaphore requires a redo log record to be written. In the event of a node crash, any semaphore which had been updated by a surviving node, yet was destroyed by a node crash, can be restored to its most recent state by use of the redo log.

A spurious update occurs when the updating node crashes, but one or more updates performed by this node survive the crash (because these updates migrated to another node).

In this case, using the local (crashed) node's to store undo log records would be useless. Instead, we wish to guarantee that, if the active allocation (an allocation remains active until the corresponding deallocation is issued) migrates, so does the undo log record corresponding to this active allocation.

One technique for implementing this undo logging technique is to use *in-line* logging [12]. With in-line logging, undo log records are stored in the same cache line where the allocation has been performed. In most cases, an in-line log record requires only a small amount of information to be written, such as a *node identifier* and an *undo operation code*. The node identifier enables restart recovery to identify which updates should be undone, while the undo operation code specifies one of a set of undo operations to execute.

For semaphores, spurious updates can be masked by writing an in-line log record for each semaphore acquisition. This log record consists of the node ID (identifier) which acquired the semaphore, and an undo operation code, which, for binary semaphores, corresponds to any operation which will return the semaphore to the initial state.

It is essential to ensure that the undo log record is written prior to the migration of the associated updated cache line. If this did not hold, a spurious update could occur. Note that for redo logging, it is not necessary to guarantee logging before migration. For semaphores, this guarantee of undo logging before migration can be implemented with existing multiprocessor synchronization techniques, such as and *cache line locks* [13], and *compare-and-swap* (CAS) [4].

The line lock is a primitive which locks a line into the cache until it is released. Ensuring that the log record is written prior to migration can be accomplished as follows. The cache line containing the semaphore is locked, the semaphore is updated, the log record written, then the line is released.

For the CAS method, the comparison condition would be that the semaphore was at its initial value, and the update condition would set the semaphore, install the node ID and the undo op-code in the cache line containing the semaphore.

Figure 2 summarizes these recovery techniques for binary semaphores. It indicates that (1) lost updates can be avoided with dedicated cache lines, (2) lost updates can be masked (yet allow false sharing) with redo logging, and (3) spurious updates can be masked with inline undo logging.

	Lost Updates	Spurious Updates
Avoid	*Dedicated Cache Lines*	—
Mask	*Redo Logging*	*Inline Undo Logging*

Figure 2: Recovery Techniques for Binary Semaphores.

4.2 Maps

A *Map* is a kernel data structure used to catalog the number of units of an allocatable resource. Specifically, a map is an array where each entry consists of an address of an allocatable resource and the number of resource units available there. In the Unix operating system, maps are used to control the allocation of resources on disk partitions designated as swap devices [2]. Swap devices are used both for *swapping*, where an entire process is transferred between primary memory and the swap device, and for *demand paging*, where individual pages are transferred between primary memory and the swap device.

When multiple kernels share access to the same disk, the swap space for separate kernels can be allocated very efficiently by implementing the swap map in coherent shared memory. However, when a map is implemented in coherent shared memory, the side-effects of a cache-coherency protocol may cause failure dependencies to form between a kernel executing on one node and the memory of another node. Next, we show how a conventional implementation of a map may exhibit these recovery problems. Then, we show how these recovery problems can be avoided with an alternate data structure.

4.2.1 Conventional Implementation

Figure 3 shows the initial state of a map, followed by two allocation operations issued by different kernels. Initially, the map has 5000 units available, and the first unit lies at address is 1. Kernel x first allocates 100 units ($malloc_x(100)$), resulting in 4900 remaining units starting at address 101. Next, kernel y allocates 100 units ($malloc_y(100)$), resulting in 4800 remaining units starting at address 201.

The main problem with this representation is that core data reflects concurrent updates. In this case, after the two *malloc* operations, the map contents contain a single entry, $(201, 4800)$ indicating that 4800 units are free starting at address 201. Thus, a single map entry reflects the updates of more than one kernel. If a node crash occurs after this sequence of *malloc* instructions, lost or spurious updates may occur. For example, consider H_{map}, where the crash of node y has the effect of terminating kernel y and all processes executing

Operation	Map Contents	
	Address	Units
Initially	1	5000
$malloc_x(100)$	101	4900
$malloc_y(200)$	301	4700

Figure 3: Operations and Resulting States of a *Map*.

on node y:

$$H_{map} = malloc_x(100);\ malloc_y(200);\ Crash_y.$$

Consider H_{map} in the context of a write-invalidate cache coherency protocol. After $malloc_x(100);\ malloc_y(200);$, the only copy of the cache line which stores the map is located on node y, and is thus destroyed due to $Crash_y$. Thus, in H_{map}, the entire contents of the map are lost. The naive method of reinitializing the map to its initial state would leave have the anomalous effect of freeing the space allocated by the kernel on node x.

Consider H_{map} in the context of a write-broadcast cache coherency protocol. After $malloc_x(100);\ malloc_y(200);$, the cache line which stores the map will be located on nodes x and y. The crash of y will destroy kernel y, but will leave the map contents in their most recent state, causing a spurious update by kernel y. Thus, unless some additional provisions are made, the 200 units of the map allocated by kernel y will never be returned to the map.

4.2.2 Eliminating Concurrent Core Data

The inconsistencies which arise in this map implementation stem from the maintenance of *concurrently modified* core data. An alternative map representation can be designed which ensures that all updates to core data are strict, and all concurrently modified data is derived. Figure 4 illustrates an alternate representation of the map data structure for which all updates to core data are strict. We refer to this representation as a *strict-core* representation.

Let us compare the conventional implementation of a map in figure 3 with the strict-core representation in figure 4. In addition to storing an entry corresponding to the location and number of free units available, the alternative representation also stores an entry for each successful allocation operation. For example, After $malloc_x(100);\ malloc_y(200)$, additional, separate entries, $(1, 100)$ and $(101, 200)$ are stored in the map table corresponding to the space reserved by $malloc_x(100)$ and $malloc_y(200)$.

False sharing of core data can be avoided if updates to core data are performed only

Operation	Type	Map Contents	
		Address	Units
Initially	—	1	5000
$malloc_x(100)$	core	1	100
	derived	101	4900
$malloc_y(200)$	core	1	100
	core	101	200
	derived	301	4700

Figure 4: Strict-Core Representation of a *Map*.

locally. This can be implemented as follows. The core portion of the map is partitioned into many segments, one per node. Each segment contains one or more cache lines, which are updated only by the local node. For example, the core data updated by $malloc_x(100)$; (the entry $(1, 100)$) is stored in a cache line which is never updated by any other kernel other than kernel x.

For the strict-core map representation, we can also guarantee that updates performed on one node are never referenced by another node. Recall we could not make this assertion for binary semaphores. Given this, it follows that spurious map updates will not occur in either the write-invalidate or write-broadcast protocols. Thus, dedicated cache lines are sufficient to ensure the consistency of core data.

In the strict-core representation, updates to derived data are done exactly as in the conventional representation. Immediately after the derived data is updated, the core data is updated. Note that sufficient synchronization primitives should be employed to ensure that the derived data is not updated by another node prior to the corresponding update of the core data. In the event of a node crash, two problems may occur: first, the derived data may be destroyed, and second, the derived data may not be destroyed, but reflect spurious updates. In either case, restart recovery can reconstruct a consistent version of the derived data from the surviving core data.

This subsection has illustrated two main points. First, avoiding concurrent updates to core data can be accomplished by reformatting and reallocating the map data structure. Second, since core data for maps is referenced only locally, lost and spurious updates to core data can be avoided. Finally, by maintaining the contiguous block list of the conventional map data structure as derived data, the advantages of the original map interface can be retained.

4.3 Disk Buffer

The disk buffer[3] is a pool of internal data buffers used to minimize the frequency of disk accesses. When a disk block is first accessed, it is stored in a reserved portion of memory called the disk buffer. Subsequent accesses to recently used cached disk blocks do not require disk access. This in-memory buffering of disk blocks is temporary since (a) the kernel may issue a flush command to force the block to disk, and (b) the buffer size is fixed at kernel initialization time, so only the most recently used disk blocks are buffered [2].

By maintaining the disk buffer in coherent shared memory, file system integrity can be maintained by serializing access by multiple processes to the same disk block. This strategy was adopted in the multiprocessor operating system of the AT&T 32B multiprocessor [3]. However, when implemented in coherent shared memory, we must consider the recovery problems which may be caused by cache coherency. Before discussing these recovery issues, we first discuss the structure of the disk buffer.

The data structures used to structure the disk buffer support two basic types of requests. The kernel may request access to a specific disk block, or may request the allocation of any free buffer. Hash tables facilitate a fast search for a specific disk block, while a free list is maintained to facilitate fast access to any free block.

A buffer consists of a buffer header, and a data portion, containing the contents of a disk block of a file system. The buffer header contains the pointer fields which form the free list and hash tables, plus other status information. When the kernel is about to read or write data from a particular disk block, it firsts checks whether the block is in the buffer pool, and if not, allocates a free buffer for that block. When a buffer is allocated, the kernel marks it used. When the kernel is finished reading or writing the block, the used flag is cleared and it is returned to the free list.

We next consider the recovery issues for the shared memory implementation of the disk buffer. In [3], a semaphore is associated with each buffer in order to serialize access to a particular buffer. Since we have already discussed the recovery issues for semaphores, we focus on the free list and the hash table.

When implemented in coherent shared memory, updates to the free list are likely to be done by any kernel, and these updates may occur at any position in the free list. For example, modifications to the free list may occur at the head, when any free block of the

[3]Another name for the disk buffer is the *buffer cache*[2].

disk buffer is required, or to the middle, when a specific free block is required [4].

Given these observations, we conclude that the simplest and most effective method for ensuring that the free list is consistent in the event of node crashes is to treat the free list as derived data. The used flag of every buffer header indicates which buffers should and which should not be on the free list. Thus, in the event of crashes, if a consistent set of used flags can be guaranteed, then a consistent free list can be reconstructed.

Avoiding lost and spurious updates for the used flag can be done with the same techniques used for binary semaphores. In-line undo logging can be used to provide sufficient information for restart recovery to eliminate spurious updates. Like binary semaphores, updates to the used flag are strict, so dedicated cache lines can be used to avoid lost updates. Alternatively, redo logging can be used to provide sufficient information for restart recovery to ensure the absence of lost updates.

In the event of a node crash, once restart recovery restores a consistent set of used flags, restart recovery can then guarantee that all buffers allocated by crashed nodes are entered on the free list, and all buffers allocated by surviving nodes are not entered on the free list as follows: First, the free list is initialized to contain no buffers. Next, all buffers which have their corresponding used flags set (allocated) are entered on the free list.

Similar issues arise when we consider ensuring a consistent hash table in the context of node crashes. Since a hash table may be updated by many kernels, it is best to treat this as derived data, and in the event of a crash, reconstructing the hash table during restart recovery. However, ensuring consistency is much easier for the hash table, since all buffers (not just those which are free) belong to it.

This subsection has illustrated that, for a fairly complex data structure like the disk buffer, simple and efficient recovery techniques can be applied to ensure global consistency in the event of node crashes. Note that not all kernel data structures are suitable for implementation in shared memory. For example, although maps implemented in coherent shared memory are useful for cataloging disk space, the trend toward a large virtual address space in distributed and multiprocessor systems suggests that this approach may not be appropriate for maintaining virtual memory page maps. A large virtual address space enables the partitioning of this space, allowing the allocation of virtual addresses to be performed strictly locally.

[4] A free buffer may be have been returned to the free list, but the cached disk block may still be valid.

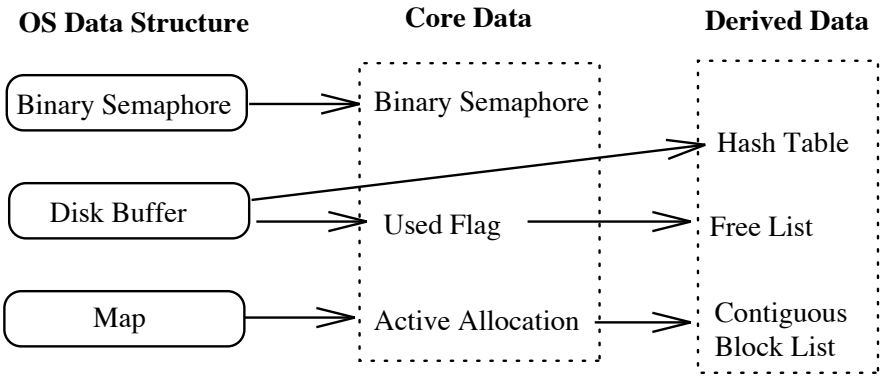

Figure 5: Recovery Support for Operating System Data Structures.

In the next section, we summarize our approach to recovery for cache-coherent shared memory operating systems.

5 Summary

In this paper, we have proposed solutions to crash recovery problems which arise when operating system data structures are implemented in cache-coherent shared memory. These recovery problems may arise in a wide range of memory architectures and cache coherency protocols, including COMA and NUMA architectures, and with both write-invalidate and write-broadcast protocols. In the event of node crashes, spurious updates can be caused by either of these protocols, and lost updates can be caused by write-invalidate protocols.

By avoiding or masking lost and spurious updates, our crash recovery methods ensure a consistent system state. For all processes running on surviving nodes, forward progress is ensured by guaranteeing that lost updates do not occur. Liveness properties are supported by ensuring that when processes are prematurely terminated due to the crash of the node where these processes execute, all resources held by these processes are released.

We have applied our recovery protocols to three kernel data structures suitable for implementation in coherent shared memory, namely, binary semaphores, maps, and the disk buffer. Our recovery approach decomposes a kernel data structure into two types of data, core and derived data. In the event of a node crash, derived data is reconstructed from consistent core data. Figure 5 summarizes how these techniques are applied to binary semaphores, maps, and the disk buffer.

The derived data associated with the disk buffer includes a free list buffers and a hash table which facilitates searches for specific buffers. In the event of a crash, hash tables can easily be reconstructed from any unique attribute of a buffer. A consistent free list can be

reconstructed given a consistent set of buffer used flags. The derived data associated with a map consists of a contiguous block list. This list can be reconstructed given a consistent set of active allocations.

We have offered a number of low overhead techniques to ensure the consistency of core data. The simplest method, used for maps, ensures that core data is referenced by only a single node, is thus not subject to the update anomalies caused by cache coherency. In a map, when a node makes an allocation request, it reads derived data (the contiguous block list), then, after determining which blocks to allocate, updates derived data, and also stores the allocation record (core data) in local memory.

Other core data, such as binary semaphores and the used flag associated with the disk buffer, store at most one active update, but may be read by multiple nodes. Both lost and spurious updates can occur for these types of structures. The use of (volatile) in-line undo logging provides sufficient information for restart recovery to discard spurious updates. We have proposed two techniques for eliminating lost updates. Lost updates can be avoided by using dedicated cache lines. Alternatively, the use of (volatile) redo logging provides sufficient information for restart recovery to restore lost updates.

6 Related Work

While our work addresses recovery issues in the context of *cache-coherent shared memory multiprocessors*, recovery issues in the context of *distributed* computations in *message-passing* systems have been studied in [5, 14]. The *process checkpointing* schemes of [5, 14] 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 the node it is running on fails. In this case, a checkpoint consists of the necessary process state for restarting execution, such as the program counter, process identifier, and register contents. In [5, 14], to ensure the forward progress of a distributed computation, 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.

Other related work on distributed computations require that checkpoints are taken *to disk* to avoid rollback propagation. In [16], a process checkpointing scheme for distributed shared virtual memory is presented. A centralized page manager implements coherency in

a loosely coupled multicomputer system. In this system, checkpointing is used to prevent rollback propagation between nodes – before a requested page is sent so some process q, if it has been modified by p since p's last checkpoint, then a checkpoint of p is initiated. In [10], a process checkpointing scheme for avoiding rollback dependencies in distributed object based systems is discussed. In this work, dependencies between processors form when threads are invoked in the address space of an object located on a remote node. Based on the type of operation involved (lookup or modify), checkpoints are taken to disk to avoid rollback propagation.

The recovery problems we have addressed in this paper for cache-coherent shared memory systems are significantly different than those addressed in distributed systems. In a cache-coherent shared memory system, cache coherency can adversely affect independent processes (those which execute strictly on one node and do not communicate with other processes). However, in the message passing paradigm used in distributed systems, independent processes encounter no such recovery problem, since all inter-node interaction is done explicitly with message passing.

Our recovery protocols can also coexist with process checkpointing schemes, and thus can also be used to support dependent computations. Recall that our recovery protocols ensure forward progress for processes running on surviving nodes. By combining process checkpointing with our protocols, we can also ensure the forward progress of computations running on crashed nodes. In this case, our definitions of lost and spurious updates need to be refined to reflect the processes checkpointed state. For example, any update to a shared data structure included in a checkpoint should not be considered spurious. Furthermore, the checkpointed state of a process will not be subject to lost updates, since this portion of the process state will exist on stable storage. However, the definitions of lost and spurious updates remain unchanged for the portion of the process which has not been checkpointed. Similar issues arise in database systems, when transaction checkpoints are issued, or active transactions commit updates to internal system structures regardless of the commit or abort of the updating transaction [11].

The recovery methods for cache-coherent shared memory operating systems presented in this paper are suitable for providing low-level recovery support for applications such as database systems. However, when applications exploit shared memory, additional recovery problems arise. For example, lost or spurious record updates, inserts, or deletes may afflict transactions. In a related paper [12], we discuss methods for avoiding unnecessary *transaction*

aborts in a cache-coherent shared memory database system.

7 Concluding Remarks

Shared memory systems offer significant performance advantages for applications which share data. These systems have been successful in supporting both parallel and timesharing applications. At present, most shared memory systems are constructed as tightly-coupled multiprocessors. However, current technological trends, such as advances in processor and network technology [7], are enabling the emergence of loosely-coupled shared memory systems connected by local or wide area networks [8, 6, 15]. With current mechanisms, a single node failure is likely to require a reboot of the entire shared memory system. While this requirement may have been bearable when a tightly-coupled shared memory system executed a single application program, it certainly will not be acceptable when multiple applications are executed, especially in a geographically dispersed setting.

To this end, our recovery protocols are designed to allow the graceful degradation of a cache-coherent shared memory system, avoiding global system reboots. Forward progress is ensured for all processes executing on surviving nodes. Our protocols also guarantee that all aborted processes release their resources, enhancing system liveness. These desirable system properties are supported by ensuring the absence of lost and spurious updates.

For data structures implemented in shared memory, lost updates can occur as a side effect of a write-invalidate protocol, and spurious updates can occur as a side effect of either a write-invalidate or write-broadcast protocol. Lost updates are avoided by using dedicated cache lines, or are masked by using volatile redo logging. Spurious updates are masked with (volatile) inline logging. The runtime overheads of supporting these properties are small – amounting to one or two additional memory references per update operation to a shared data structure. The low overheads associated with our protocols should make them attractive for improving system reliability of tightly-coupled and loosely-coupled cache-coherent shared memory systems.

References

[1] 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.

[2] M. Bach. *The Design of the Unix Operating System.* Prentice-Hall, 1991.

[3] M. Bach and S. Buroff. Multiprocessor Unix Systems. *AT&T Bell Laboratories Technical Journal*, 68(8):1733–1750, October 1984.

[4] Motorola Inc. *MC68020 32-Bit Microprocessor User's Manual.* Prentice-hall, Englewood Cliffs, N.J., 1985.

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

[6] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. *1994 Winter Usenix Conference*, 1994.

[7] H. T. Kung. Gigabit Local Area Networks: A Systems Perspective. *IEEE Communications Magazine*, pages 79–89, April 1992.

[8] 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.

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

[10] L. Lin and M. Ahamad. Checkpointing and Rollback-Recovery in Distributed Object Based Systems. *Proceedings of the 20th International Symposium on Fault-Tolerant Computing*, pages 97–104, 1990.

[11] 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.

[12] L. Molesky and K. Ramamritham. Recovery Protocols for Shared Memory Database Systems. *Submitted to the 1995 ACM SIGMOD International Conference on Management of Data.*

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

[14] 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.

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

[16] K. Wu, W. Fuchs, and J. Patel. Recoverable Distributed Shared Virtual Memory. *IEEE Transactions on Computers*, 39(4):460–469, April 1990.