Tolerating Node Failures in Cache Only Memory Architectures*

A. Gefflaut, C. Morin, and M. Banâtre
IRISA/INRIA

Campus Universitaire de Beaulieu
35042 Rennes Cedex - France

Abstract

COMAs (Cache Only Memory Architectures) are an interesting class of large scale shared memory multiprocessors. They extend the concepts of cache memories and shared virtual memory by using the local memories of the nodes as large caches for a single shared address space. Due to their large number of components, these architectures are particularly susceptible to hardware failures and so fault tolerance mechanisms have to be introduced to ensure a high availability. In this paper, we propose an implementation of backward error recovery in a COMA which minimizes performance degradation and requires little hardware modifications. This implementation uses the features of a COMA to implement a stable storage abstraction using the standard memories of the architecture. Recovery data are replicated and mixed with current data in node memories both of which are managed in a transparent way using an extended coherence protocol.

1 Introduction

Scalable Shared Memory Multiprocessors (SSMM) are thought to be a good solution to achieving the teraflops computing power needed by grand challenge applications such as climate modeling or humane genome. These architectures consist of a set of computation nodes containing processors, caches and memories, connected by a high-bandwidth low latency interconnection network. Scalability, achieved by the scalable interconnection network and the distributed main memory, allows a large number of processors to be used with a good efficiency. Shared memory provides a flexible and powerful computing environment. Two variations of these architectures have emerged: Cache Coherent Non Uniform Memory Access machines (CC_NUMA) [1, 18], which statically divide the main memory among the nodes of the architecture, and Cache Only Memory Architectures (COMAs) [14, 10] which convert the per node memory into a large cache of the shared address space, called an Attraction Memory (AM).

Due to their increasing number of components both CC-NUMA machines and COMAs have, despite an important increase in hardware reliability, a very high probability to experience hardware failures. Tolerating node failures is therefore very important for architectures used for long running computations. Fault tolerance becomes then mandatory rather than optional for large scale shared memory multiprocessor architectures.

In this paper, we propose a new solution to cope with multiple transient and single permanent node failures in a COMA. Our approach uses a backward error recovery scheme [17] where the replication mechanisms of a COMA are used to ensure the conservation and replication of recovery data in the AMs of the architecture. To implement this, an extended coherence protocol manages transparently both current and recovery data. This solution avoids the need for specific hardware and minimizes performance degradation by using the memories and the interconnection network to handle recovery point establishment. Other aspects of fault tolerance, such as detection and error confinement are out of the scope of the paper and in the remainder we assume a fault-free network and fail-stop nodes. Detection of faulty nodes is provided through time-outs on inter-node communications.

The remainder of this paper is organized as follows. Section 2, gives an overview of COMA machines. In Section 3, after introducing the principles of our solution, we describe how the coherence protocol is extended to tolerate node failures. Performance evaluation results obtained by simulation are given in Section 4 for an implementation of the protocol in a slotted ring COMA similar to a single ring KSR1 [10]. Section 5 concludes our presentation.

2 Cache Only Memory Architectures

CC-NUMA architectures and COMAs have similar organizations. They both have a distributed main memory and a scalable interconnection network. In contrast to CC-NUMA machines, COMAs convert the per-node memory into a huge cache of the shared address space by adding tags to lines in main memory. A consequence of this is that the location of a data item in the machine is totally decoupled from its physical address, and a data item is automatically migrated and replicated in the memories following

*This work is partly supported by the DRET research contract number 93.34.124.00.470.75.01.

the memory reference pattern of the executed application. From a fault tolerance point of view, this feature constitutes a clear advantage of COMAs over CC-NUMA architectures since memory items located in a faulty node can be re-allocated on functioning nodes transparently without modification to their physical address. This is the reason why we have chosen to investigate COMAs.

COMA usually use hierarchical organization to locate memory items on a miss. Directories at each level of the hierarchy maintain information about memory item copies located in a sub-hierarchy. Such an organization is used in the DDM [14] and KSR1 [10] architectures.

Basically, all COMAs use the same coherence protocol. This protocol can be simplified to four basic item states which change according to requests received from the local processor (*Pread/Pwrite*), or from remote nodes over the network (*Nread/Nwrite*). Figure 1 depicts the standard coherence protocol used by the AMs of a COMA architecture.

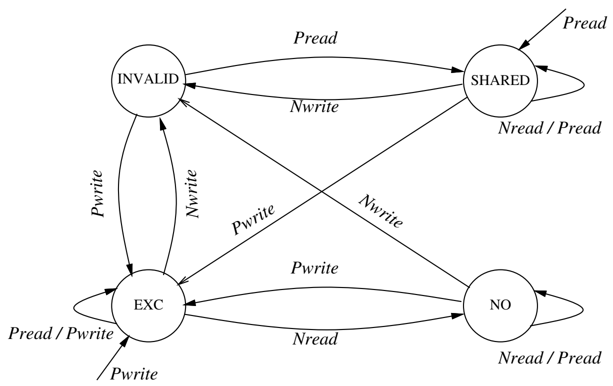

Figure 1: Standard coherence protocol for a COMA

- *Invalid*: the local AM does not contain a valid copy of the item.
- *Shared*: the local AM has a read-only copy of the item.
- *Exclusive* (EXC): the local AM owns the only valid copy of the item, no other AM has a copy of it. The item can be read or modified by the processor.
- *Non-Exclusive Owner* (NO): the local AM contains a valid copy of the item but other AMs may have a *Shared* copy. However, the local memory is the owner of the item and it must transfer its copy on another node before replacing it.

Note that a memory item has exactly one copy in state *Exclusive* or in state *Non-Exclusive Owner*. To avoid deleting the last copy of an item, such copies cannot be replaced without being first transferred to another node AM. These transactions are called *injections* in the remainder of the paper. CC-NUMA cache coherence protocols do not have to cope with this problem since a cache line can be evicted from a cache and stored in its *home* node memory.

3 Extending the Coherence Protocol for Tolerating Node Failures

3.1 Principles

Among the techniques that can be used to tolerate node failures in a SSMM, Backward Error Recovery (BER) [17] seems to be the most attractive solution. This technique has several advantages over other fault tolerance approaches. Active software replication [8] requires a strong synchronization between replica leading, in particular for shared memory, to a significant increase of inter-node communications and to a high performance degradation. Hardware static replication like nMR (n Modular Redundancy) [6, 15] with voting requires a full replication of a majority of hardware components and is certainly too expensive to be applied to architectures with a large number of components. In contrast, BER limits the hardware development and allows the use of all the processors for a computation.

BER attempts to restore a correct system state after a failure detection. To achieve this, the system periodically saves a consistent image of its state, called a recovery point (a set of recovery data), such that, in the event of a failure, a known error free state exists to restart the execution. In a multiprocessor environment a pessimistic approach, limiting the recovery data size to a single recovery point per processor, and preventing the domino effect [LEE90a], is preferable. Such an approach requires a coordination of communicating processors during the establishment of a recovery point so that the set of processor recovery point always form a recovery line [17]. In this paper, for the sake of simplicity, coordination between processors is ensured by using a global checkpointing scheme (all processors are involved in a recovery point establishment).

To tolerate any single failure in a system, recovery data must satisfy two properties. First, they must not be altered and must remain accessible in the presence of single hardware failures in the architecture (persistence property). Secondly, recovery points must be atomically updated, that is, in the event of a failure during a recovery point establishment, either all recovery data remain in their initial state or they all reach their final state (all or nothing property). These two properties can be guaranteed by storing recovery points in stable storage [16] for which efficient hardware implementations have already been proposed [2, 3, 4, 7]. In SSMMs, it is however unreasonable to develop specific hardware stable storage implementations as the cost of the architecture would drastically grow. Recovery data persistence, which implies a replication of recovery data on two failure independent physical storages, can be ensured in a COMA by the standard AMs since they are allowed to store any memory item copy and hence can be used to replicate recovery data on independent nodes of the architecture. In such an approach recovery data can be located on any attraction memory which may thus contain both current and recovery data. The remaining problem is to identify recovery data from current data. In a COMA, the coherence protocol provides a good means to realize this.

The main advantage of this approach is that no dedicated hardware device is required. Moreover, it can be as efficient as hardware implementations since it uses the same technologies. Another advantage is that it allows processors to use recovery data, stored in attraction memories, as long as these data have not been modified since the preceding recovery point establishment. Finally, this approach can minimize the recovery point establishment duration by using the already present replication of memory items in the architecture to avoid some data transfers between nodes.

3.2 Coherence Protocol Extensions

To handle recovery and current data, an extended coherence protocol, used by the AMs of a COMA, is proposed. This protocol combines the management of recovery and current item copies. It ensures persistence for recovery data and permits current and recovery copies of different items to cohabit in the same memories. Though recovery data are up to date until their first modification since the preceding recovery point, usual stable storage implementations preclude their use for normal computing since they have to be located on an independent storage. The extended protocol corrects this drawback by allowing recovery item copies, not yet modified, to be read and replicated on more than two nodes.

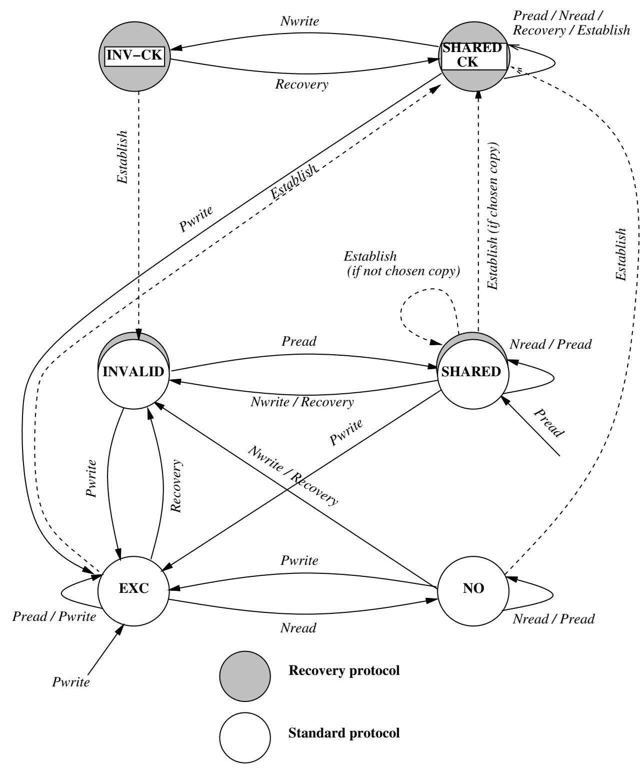

Figure 2: Extended coherence protocol

The extended coherence protocol can be viewed as the composition of two independent protocols. The current copies protocol is similar to the standard protocol used in COMAs presented in Section 2. Its purpose is to handle the coherence of current memory item copies. The purpose of the recovery copies protocol is to ensure the persistence property of recovery item copies and to allow read-only replication of recovery copies not yet modified since the preceding recovery point. The recovery copy protocol uses four states. Two of them, (*Shared-CK* and *Inv-CK*) are used to ensure a minimum replication of two recovery copies for each item. The two other states, *Invalid* and *Shared*, are similar to the states used by the standard coherence protocol. In the extended protocol, these states are combined with the two equivalent states of the standard protocol. The *Shared* state is then used to manage read-only copies of recovery or current item copies. After a successful recovery point establishment, an item has two recovery copies and possibly other read-only replicated copies in the system. Immediately after its first modification, an item has at least one current copy and two recovery copies.

The two new states introduced for recovery item copies are now described.

1. *Shared-CK* (Shared Checkpointed): represents an item copy which was created at the preceding recovery point and which is not yet modified. Two memories have a copy in this state. This copy is the most recent version of the item and it can be read by the local processor or used to serve read request from remote nodes. This item copy cannot be discarded without being first transferred to another node.

2. *Inv-CK* (Invalid Checkpointed): represents an item copy which has been modified since the preceding recovery point, and thus cannot be accessed by a processor. Exactly two memories have a copy in this state. *Inv-CK* copies are conserved for recovery and must be transferred before being discarded.

The extended coherence protocol combines the management of current and recovery item copies in a transparent way by integrating the two protocols in a single one. Compared to a standard coherence protocol, two transitions are added to take into account the establishment and restoration of recovery points. Figure 2 contains a state transition diagram for the protocol which works as follows.

Read hit A read hit on an item modified since the last recovery point is treated as in the standard coherence protocol. A *Shared-CK* item copy can also be read by a processor without generating any external request since this item copy is the up-to-date version of the item. An *Inv-CK* item copy corresponds to the recovery version of an item. Other current copies exist in the system. A read hit on an *Inv-CK* copy must be treated as a miss since the current copy is not accessible. Before performing the miss, the *Inv-CK* copy must be transferred to another node.

Read miss A read miss on an item modified since the preceding recovery point is treated as in the standard coherence protocol. To service a read miss on an item not

```
Establish Phase {
 For each item in the local memory {
    case (item.state) {
     Exclusive:
         Inject item in another memory in
              state Pre-commit
         item.state = Pre-commit;
     Non Exclusive Owner:
         item.state = Pre-commit;
         If (Shared copies exist)
            send Pre-commit message to one of them
         else
            Inject item in another memory in
                state Pre-commit
     Other:
         Skip;
    }
 }
} End of Establish Phase
```

```
Commit Phase {
 For each item in the local memory {
    case (item.state) {
     Pre-commit :
         item.state = Shared-CK;
     Inv-CK :
         item.state = Invalid;
     Shared :
     Shared-CK :
     Invalide :
         skip;
     Other :
         Error /* No other copies after Establish Phase */
    }
 }
} End of Commit Phase
```

Figure 3: Establish/Commit algorithm

modified since the last recovery point, one of the *Shared-CK* copies is used. The new copy is marked *Shared* in the memory of the requesting node.

Write hit A write hit on an item modified since the last recovery point is treated as in the standard coherence protocol. When the write occurs on a *Shared* item copy, a *Nwrite* transaction is generated by the node. If no modification has occurred on this item since the last recovery point, then the two *Shared-CK* copies change their state to *Inv-CK* upon reception of the *Nwrite* transaction. If a modification of the item has already occurred, the request is treated as in the standard protocol. At the end of the transaction, the requesting node sets its item copy to *Exclusive*.

A write hit on a *Shared-CK* item copy represents the first write on this item since the last recovery point. To ensure that two recovery copies exist, an injection of the item copy is first required. When the injection is performed, the requesting node re-issues its request which is now a write miss.

An *Inv-CK* copy cannot be directly accessed by a processor since there is necessarily a current copy of this item in the system. Upon a write hit on an *Inv-CK* item copy, the copy must be transferred to another node before being replaced. After the injection, the initial request is re-issued and treated as a standard miss on a current item copy.

Write miss A write miss on an item modified since the last recovery point is treated as in the standard protocol. For a write miss on an item not modified since the last recovery point, a *Nwrite* transaction is generated as in a traditional coherence protocol. The *Shared-CK* copies change their states to *Inv-CK* and possible *Shared* copies are invalidated. At the end of the transaction, the requesting node owns the only valid current copy of the item in state *Exclusive*.

3.3 New Injections

In a standard COMA, injections are used only when a copy in state *Exclusive* or *Non-Exclusive Owner* has to leave room for a more recently accessed item. The extended coherence protocol introduces five new cases of injections (see table 3.3). Two of them occur when a copy in state *Shared-CK* or *Inv-CK* must be replaced from an attraction memory. The others occur when a node wants to access an item which already resides in a recovery state in its local memory. These injections are needed in order to ensure that two recovery copies of an item always exist.

Cause	Local copy state	Action
Replacement	*Shared-CK*	Injection
Replacement	*Inv-CK*	Injection
Read access	*Inv-CK*	Injection + read miss
Write access	*Inv-CK*	Injection + write miss
Write access	*Shared-CK*	Injection + write miss

Table 1: New injections introduced by the extended coherence protocol

3.4 Recovery Point Establishment

To limit the performance degradation, the recovery point establishment algorithm must be efficient. We use an incremental technique in which only data which have been modified since the last recovery point have to be copied when a recovery point is established. Taking a global recovery point simplifies the algorithm as other mechanisms must else be used to deal with the effect of data sharing on backward error recovery [2].

The algorithm is depicted in Figure 3. To guarantee atomicity of the establishment, and hence to tolerate any possible failure during its execution, the algorithm uses a traditional two-phase commit protocol [12]. During the first phase (establish phase), the new version of the recovery point is established by replicating all modified items on two distinct nodes, in state *Pre-commit*. This new recovery point is made of all non modified recovery item copies (*Shared-CK* copies) plus all modified item copies (*Pre-commit* copies). During this first phase, the previous recovery point made of all *Shared-CK* and *Inv-CK* copies is also conserved. The second phase (commit phase) is a local phase. Its purpose is to discard all item copies belonging to the previous recovery point identified by their *Inv-CK* state, and to confirm the new recovery point.

This algorithm is executed by each node of the architecture and is triggered by a broadcast message from an initiator node. Upon reception of a *begin-establish* message, a node terminates its possible pending transactions and starts the establish phase of the algorithm. During this phase a node may receive some injections from other nodes. These item copies are set to the *Pre-commit* state if the injection is accepted. At the end of its *establish* phase, a node sends an acknowledgment message to the initiator and waits for a *begin-commit* message before beginning the commit phase. Once the initiator has terminated its establish phase and received an acknowledgment from each node of the architecture, it broadcasts a *begin-commit* message.

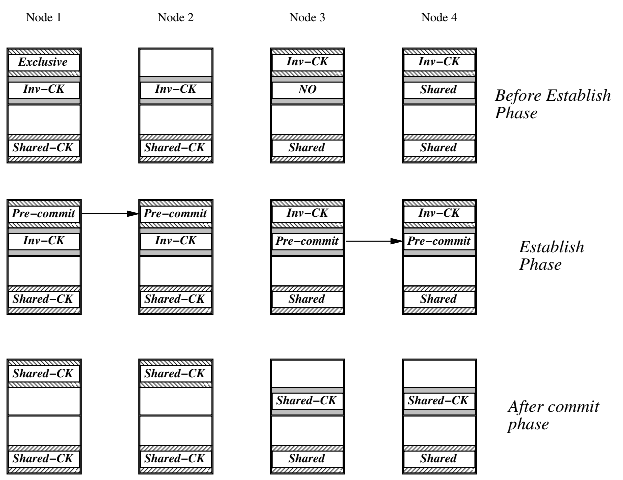

Figure 4: Simple example of recovery point establishment

The second phase of the algorithm is local to each node and so can be made very efficient. Each node scans its memory and simply sets all its *Inv-CK* item copies to *Invalid* and all its *Pre-commit* item copies to *Shared-CK*.

With the help of the new states added to the standard protocol the establish/commit algorithm is quite simple and only requires a single transfer of each modified item. Figure 4 shows the two phases of the algorithm with a simple configuration example including four nodes and three items.

Any failure during the first or second phase of the algorithm is correctly handled. During the establish phase, the previous recovery point is still unaltered and the architecture can use it to rollback. During the commit phase, the new recovery point is complete and persistent since all items are already replicated. Since this phase is local, any failure during it can be treated at the end of the phase, it is as if the failure occurred during the computation after the establishment of the recovery point.

The memory overhead induced by this protocol varies with time. The minimal number of copies for an item is two after the end of the commit phase. After the first modification of the item, the number of copies reaches three since an *Exclusive* and two *Inv-CK* copies are present. At the end of the establish phase, the minimal number of copies for a modified item is four since two recovery points are kept. These values do not represent the real memory overhead since the standard protocol already replicates shared memory items on several nodes.

3.5 Recovery Point Restoration

```
Recovery point restoration phase
{
 For each item in the local memory{
  case (item.state) {
   Inv-CK:
        item.state = Shared-CK;
   Shared-CK:
        Skip;
   Pre-commit:
   Non Exclusive Owner:
   Shared:
   Exclusive:
        item.state = Invalid;
   Invalid:
        skip;
   other : Error /* No other item copies */
  }
 }
} End of Restoration
```

Figure 5: Recovery point restoration algorithm

After a node failure has been detected, the purpose of the restoration phase is to reinstall the previous recovery point. This algorithm is depicted in Figure 5

The error recovery algorithm restores the previous recovery point made of all *Shared-CK* and *Inv-CK* item copies, as the standard consistent state. Other item copies are invalidated. As the recovery is global, a broadcast message informs the nodes when a recovery must be performed. Each node scans its local memory and invalidates all current item copies which do not belong to the recovery point. *Inv-CK* copies are restored to *Shared-CK* since they are now up to date. At the end of the error recovery phase, only two *Shared-CK* copies exist for each item of the shared memory space. *Shared* copies must be also invalidated as we cannot know whether they correspond to current or recovery item replica.

In the event of a permanent failure, a memory reconfiguration must also be performed after the recovery phase. This reconfiguration is done in order to duplicate lost item copies which were located on the faulty node so that the persistence property is satisfied again. After the recovery phase, only *Shared-CK* item copies exist. To reconfigure the architecture, each *Shared-CK* has to check whether its replica is still alive or not. If not, a new *Shared-CK* copy has to be created on a safe node. The duration of this phase is dependent on the implementation of the coherence protocol. A snooping protocol requires a broadcast message for each *Shared-CK* item. A directory based protocol can simplify the reconfiguration phase by furnishing some information on the location of the *Shared-CK* copies. Even with this, the reconfiguration phase can be quite long. This is balanced by the fact that the occurrence of a permanent failure is rare and hence such a reconfiguration will not occur frequently, limiting the performance degradation.

4 Performance Evaluation in a Slotted Ring COMA

In this section, we evaluate through simulations, the overheads introduced by the extended coherence protocol in a slotted ring COMA architecture, such as a single ring KSR1 [10], and which uses a snooping coherence protocol. For such an architecture, the protocol can be used to tolerate either hardware or software transient node failures. To take permanent node and network failures into account, some modifications to the existing hardware should be envisaged. These modifications are related to fault detection and confinement to ensure the fail-stop property, and to the reliability of the interconnection network.

4.1 Architecture

The target architecture is very similar to a single ring KSR1 architecture. It contains 32 nodes connected by a pipelined unidirectional slotted ring. Each node consists of a single 20 Mhz processor, 256 KB data and instruction first level (L1) caches, a 32 MB AM, a cache directory made of four control units and a ring interface. The L1 data cache is 2-way associative and uses 2 KB block allocation while the L1 cache/local memory transaction unit is 64-bytes sub-block. The AM is 16-way set associative and uses a 16 KB page allocation. Each page is subdivided in 128 sub-pages of 128 bytes. The data transfer unit between nodes is a sub-page and coherence is maintained on a sub-page basis. The AM directory contains an entry for each 16 KB page of the local cache. Each entry includes an address tag and the state of each sub-page of the page. The coherence protocol used by the memory is similar to the protocol presented in Section 2.

Each page of the global shared address space is either entirely represented in the system or not at all. Due to the absence of physical location for a page in a COMA, the page replacement algorithm must ensure that at least one copy of a page exists in the system. To simplify this algorithm, the KSR1 designers assigned for each page a "home" node where an irreplaceable home page is allocated. The home page provides space for every sub-page even if none of them is valid and ensures that any non-home page evicted from an AM will find space for its *Exclusive* or *Non exclusive Owner* sub-pages. A significant fraction of a node AM is set aside by the operating system to receive home pages. This fraction is called the memory *home associativity* in the remainder of the paper.

4.2 Protocol Implementation

In such an architecture, the implementation of a snooping coherence protocol can be quite complex since it has to handle possible conflicts between nodes simultaneously trying to access the same memory item. The conflict resolution adopted here for the standard part of the protocol is quite similar to that presented in [5]. It introduces some new transient states which cannot be described here due to lack of space.

Conceptually, the extended coherence protocol can be implemented exactly as it is described in Figure 2. The standard snooping coherence protocol has to be modified so that the new item states and transactions can be directly managed by the protocol. The hardware overhead is then limited to new bits added to manage the additional states and a modification of the coherence controller to accept the new requests. However, to prevent the creation of two simultaneous *Exclusive* item copies, we use two distinct states for the two *Shared-CK* copies, with only one of them being able to deliver the exclusive access right.

The extended coherence protocol introduces new cases of item copy injection presented in Section 3. The snooping coherence protocol simplifies the injections since their treatment requires a broadcast and so an injection potentially visits all the nodes of the architecture. Ideally, an injected item copy should always replace an *Invalid* item copy. Injections introduce however a trade-off between minimizing the time to complete the injection by using copies other than only *Invalid* copies, and minimizing the number of injections. In this evaluation, injections of *Shared-CK* item copies are accepted by nodes with an *Invalid* or a *Shared* copy. Since *Shared-CK* copies represent the current value of an item, such injections are also accepted by nodes waiting for a read-only copy of the corresponding sub-page. Injections of *Inv-CK* copies are accepted by nodes with an *Invalid* copy and also by nodes with a *Shared* copy to avoid an injection being unserviced in a single network trip.

The overhead of injections required before a miss on the same memory sub-page (see Section 3) can be reduced by a request combining technique. Instead of using two different requests, combining can be applied so that the injection and the corresponding miss constitute a single network request. This combining scheme limits the overhead of an injection since only one request trip may be required.

4.3 Recovery Point Establishment and Restoration

The snooping nature of the protocol penalizes the implementation of the extended protocol by increasing the recovery point establishment duration. A snooping protocol does not maintain any information about the number and location of item copies. Hence during the establish phase,

Application	Data Refs	Read Refs	Write Refs	Sh. reads Refs	Sh. Writes Refs
Mp3d	21.2	13.3 (62.8%)	7.9 (37.2%)	10.7 (50.4%)	6.8 (30.9%)
Cholesky	22.3	17.6 (79%)	4.7 (21%)	14.2 (63.5%)	2.5 (11.3%)
Water	24.1	18.6 (77.2%)	5.5 (22.8%)	3.4 (14.2%)	0.4 (1.72%)
Barnes	46.5	29.4 (63.5%)	17 (36.5%)	6.8 (14.6%)	1.9 (0.4%)
Solve	4.2	3.1 (73.8%)	1.1 (26.2%)	1.1 (26.2%)	0.034 (0.81%)

Table 2: Trace characteristics (references in millions)

a transfer for each *Non Exclusive Owner* copy is necessary even if other *Shared* copies exist and already provide the needed replication.

The implementation of the establish/commit algorithm is very similar to the one presented in the general case. The network ring is used as a simple way to broadcast the different messages required by the protocol. The replication of *Exclusive* or *Non Exclusive Owner* sub-page copies is performed through inject transactions made during the establish phase. In the considered architecture, a node failure can be detected when a request message returns to the requesting node unanswered after a bounded number of retransmissions or returns with a special answer indicating an internal fault of a node. The recovery algorithm is then realized by the four Cache Coherent Units located on each node, which scan the directory information to discard useless sub-page copies. The L1 data cache has also to be invalidated and, in the event of a permanent failure, a reconfiguration is required.

4.4 Memory Management

With the extended coherence protocol, a maximum of four memory sub-pages is necessary during the recovery point establishment. Two strategies can be used to allocate memory pages: (1) dynamic memory page allocation, (2) static four page allocation. Dynamic page allocation minimizes memory occupancy but requires more complex and costly algorithms when a new recovery point is established.

In this study, a static page allocation scheme is used. The home associativity is increased by 3 and each modifiable page is allocated on at least four distinct nodes where they are marked as home pages. This static allocation strategy ensures that there is always enough memory space for establishing a new recovery point. Since shared memory pages are already allocated on several nodes with the standard protocol, this increase of the number of home pages does not represent the real memory overhead. The memory overhead is studied in the evaluation section.

4.5 Evaluation

In this section, we present an evaluation of the proposed scheme and compare it to a standard non fault tolerant architecture. The overheads introduced by the extended coherence protocol and the saving of recovery points are clearly identified through the simulation of five parallel applications representing a variety of shared memory access patterns. Table 2 describes their characteristics. Four of the applications (Mp3d, Water, Cholesky, Pthor and Barnes) come from the SPLASH benchmark suite [22]. The last one (*solve*), is a very simple parallel application resolving a system of N equations by iterations [20].

The simulator is implemented with a discrete event simulation library providing management, scheduling and synchronization of lightweight processes [21]. Each node is simulated by a process which interacts with the other components of the architecture. To collect address traces, the simulator uses the SPAM execution-driven simulation kernel [11].

The simulated architecture is not exactly similar to a real KSR1 ring. It retains however, all of the most important features of this architecture. The network accepts 16 simultaneous requests though the real architecture uses 13 slots. The processor has a single pipeline (two in the real processor of the KSR) and hence can issue one instruction per cycle (20 Mips). As the instruction L1 cache is large, we assume a 100% instruction hit ratio. This assumption does not interfere on the simulation results since the actual hit rate in the instruction cache is extremely high. The L1 cache access time is fixed to 1 processor cycle and the memory access time to 17 cycles such that a miss serviced from the local AM takes 18 cycles. A miss request serviced by a remote node takes at least 120 cycles if there is no contention on the network. The simulated architecture uses the combining technique and the recovery point establishment algorithm described previously. The injection of a modified item during the establish phase as well as other messages required by the establish algorithm, require a whole network trip (96 cycles). For the commit phase we assume that the four Cache Coherent Units work in parallel. Each tested page or sub-page requires 1 processor cycle. All pages are tested but sub-pages of unallocated pages are not. Finally, only the parallel phase of the computation is considered in the evaluation.

As the recovery point establishment frequency is mainly influenced by the number of operations coming from or going to the outside world, different frequencies are used for the simulations. All the simulations are sufficiently long so that several recovery point establishments occur. The frequencies range from 400 to 0 recovery points per second. These frequencies may seem quite high in front of

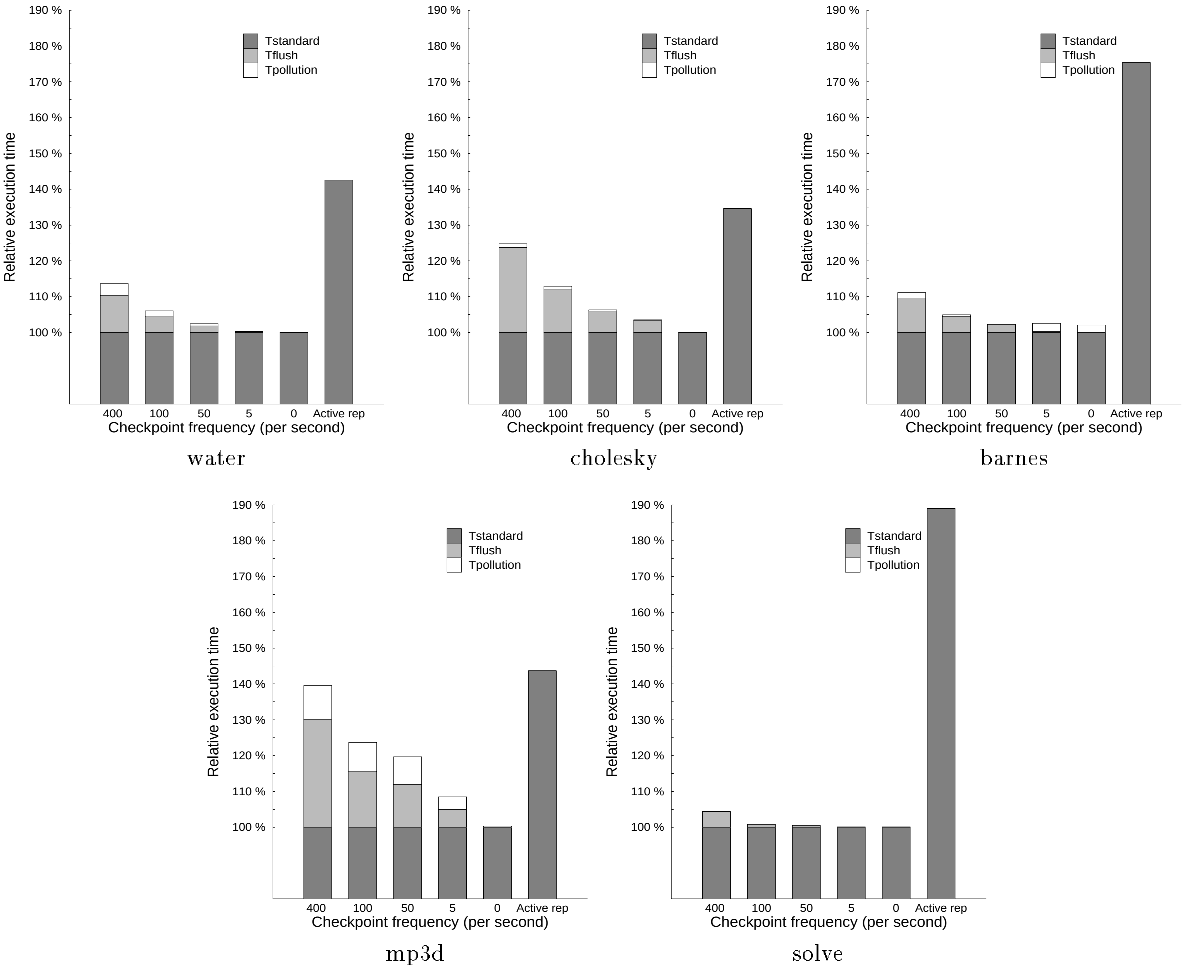

Figure 6: Time overhead

other evaluations like in [9]. In the absence of real recovery point frequencies, they give, however, the performance degradation for different computing environments.

Two types of overhead are introduced by the extended coherence protocol: a time overhead resulting in longer execution times than the standard architecture, and a memory overhead due to the increase of the memory size necessary for running an application.

4.5.1 Time Overhead

The time overhead can be divided in two separate effects: (1) the time required to establish/commit new recovery points, (2) a memory pollution effect caused by the increase in the number of sub-page misses and injections. The execution time with the extended coherence protocol can be expressed as the sum of three components, $T_{Ft} = T_{Standard} + T_{Flush} + T_{Pollution}$, where $T_{Standard}$ is the execution time on the standard architecture, T_{Flush} the time spent in the establish and commit phases and $T_{Pollution}$, the overhead due to the increase of the number of misses and injections.

Figure 6 depicts these different values for each application. All these times are normalized to $T_{Standard}$ such that the graphs directly give the time overhead represented by each component. In each graph, the last bar represents the relative execution time of the same application with 16 processors and using the same slotted ring network (same bandwidth and latency). It actually gives the best performance that could be obtained with an active replication strategy using two replicated processes.

Globally, the time overhead ranges from 40% to less than 5% and decreases with lower recovery point frequencies. For all applications studied here, the solution proposed in this paper performs better than an active replication solution using only half of the processors for the computation.

Since the recovery point frequency directly influences the number of recovery sub-pages transferred during the recovery point establishments, T_{Flush} decreases with lower recovery point frequencies. With Mp3d, the number of sub-pages transferred at 400 recovery points per second is 4 times the number of sub-pages transferred at 5. This represents a flush overhead decrease of more than 6 times.

With low recovery point frequencies, T_{Flush} becomes very low for all applications (less than 5%). Applications with a large working set (Mp3d, Cholesky) report also a larger flush time overhead since they incur longer commit phases due to more sub-pages to test. Solutions using a node recovery point counter, incremented each time a new recovery point is confirmed, and recovery point counters associated to sub-pages could be used to avoid scanning the AMs during the commit phase. Such a solution could limit the T_{Flush} overhead to the time required by the establish phases.

The second time overhead, $T_{Pollution}$, is induced by the extended coherence protocol which keeps the recovery item copies in the AMs. Whatever the recovery point establishment rate, this overhead is quite limited for all applications and ranges from approximately 10% in the worst case to less than 1%. This limited pollution effect is mainly due to the low increase in the number of misses and injections. Most of the new injections are caused by write misses on recovery sub-pages. The influence of the protocol on read misses is negligible.

Applications with important data modification rates report the most significant pollution overhead. For these applications, this overhead increases with the recovery point frequency since the number of write miss/injections is also more important (*Exclusive* copies are more often changed to *Shared-CK*). The network occupancy also interacts on $T_{Pollution}$. Cholesky and Water, for approximately the same number of injections, show different pollution overhead since the cost per injection is around 155 cycles for Water although it is only 135 for Cholesky.

An important observation is that with lower recovery point frequencies, $T_{Pollution}$ decreases for most of the applications. This proves the efficiency of the protocol. For most applications, the number of memory misses with 0 recovery points per second, is similar to the number of misses for the standard architecture. This shows that the protocol is particularly good in using unused sub-pages allocated by false sharing to store recovery item copies which then do not cause new injections. The ring network facilitates this behavior since a sub-page can visit all nodes. A combination of misses and injections is also a reason for this limited pollution overhead. The only exception is Barnes where $T_{Pollution}$ increases with higher recovery point frequencies. The reason is that this application uses mostly read objects [13] replicated on many nodes. The presence of *Inv-CK* copies generates new readmiss/injections. Even for this application the pollution overhead remain low (less than 5%).

For most of the applications, the extended coherence protocol limits the pollution effect in particular because it allows recovery data to be used as long as they are not modified. The time overhead is largely influenced by T_{Flush} and hence by the recovery point frequency. Larger frequencies increase T_{Flush} and $T_{Pollution}$ especially for applications incurring a large amount of modifications. Recovery point establishments are usually triggered by external irrecoverable operations like I/O. For scientific computation, such operations are infrequent, thus resulting in a low performance degradation. For other applications, methods preventing the need to establish a new recovery point for each I/O should be used to limit the performance degradation. This is a general problem with BER strategies which have to deal with irrecoverable operations. However, the proposed solution efficiently supports high recovery point establishment frequencies.

4.5.2 Memory Overhead

The second overhead introduced by the extended protocol is the additional memory space used by the recovery sub-page copies. Figure 7 presents this overhead. For each application, the number of pages allocated by the standard architecture as well as the number of pages allocated by the fault tolerant one, are presented.

The memory overhead ranges from 1.14 to 3.68. Shared memory pages do not produce any memory overhead. For all applications, the number of shared pages allocated with the standard or fault tolerant version of the architecture, is similar even if four copies are statically allocated for each data page. This result is mainly due to the large page size (16KB) used in the architecture, which favors false sharing and hence page replication. Private memory pages are normally allocated on a single node. With the static 4 page allocation strategy, the overhead induced by private pages is 4 times the number of private pages allocated in the standard architecture.

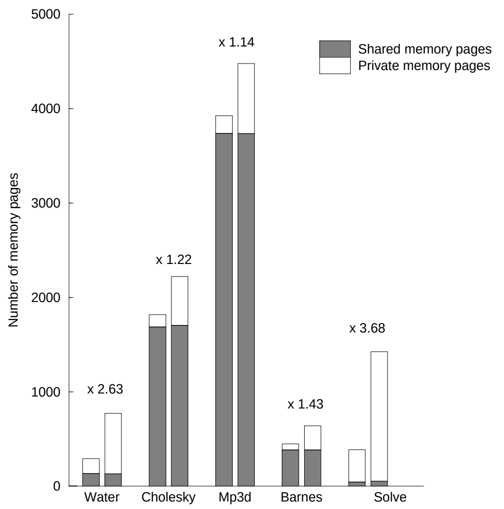

Figure 7: Memory overhead

Globally, for applications with a majority of shared pages, the memory overhead remains very low. Mp3d, Cholesky and Barnes have a memory overhead inferior to 1.5 times the number of pages allocated in the standard architecture. This proves that the protocol uses the already present replication to store shared recovery data without requiring a large memory overhead.

5 Conclusion

In this paper a proposal for a backward error recovery strategy in Cache Only Memory Architectures has been presented. This proposal allows transient as well as single permanent node failures to be tolerated through the use of an extended coherence protocol which implements backward error recovery with a set of standard AMs. This solution is cheap since the hardware and the memory overheads are low. It is also efficient since the performance degradation remains low even for relatively high recovery point frequencies. This low performance degradation is mainly due to the use of memories to store recovery data and to a low pollution overhead introduced by the management of recovery item copies.

To limit the recovery point establishment overhead, other techniques could be envisaged. Dependency tracking between communicating processors could limit the number of processors included in a recovery point establishment operation [2]. Recovery point establishment performed in parallel with the execution of the application [19, 9] could hide the time required by this operation.

The extended protocol can be implemented in any COMA. For a real implementation, other aspects of fault tolerance have however to be investigated. In particular error detection and confinement should be included in the nodes to ensure the fail-stop property. Reliability of the network is another property that should be considered. The operating system should also be studied for a real implementation. A single symmetric OS facilitates the recovery since the kernel data structures are shared by all nodes and thus can automatically be recovered after a failure. Finally, this protocol could also be used in a distributed system to implement a recoverable distributed shared memory.

Acknowledgments

We would like to thanks A.M. Kermarrec for useful comments and suggestions on this work. We would like to thank C. Bryce for carefully reading and correcting preliminary versions of this paper. We would also like to thank H. Nilsson for providing us the code of the solve application.

References

[1] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Ma, and D. Nussbaum. – The Mit Alewife Machine : A Large-Scale Distributed Memory Multiprocessor. – Technical Report MIT/LCS/TM-454, MIT Laboratory for Computer Science, June 1991.

[2] M. Banâtre, A. Gefflaut, P. Joubert, P.A. Lee, and C. Morin. – An Architecture for Tolerating Processor Failures in Shared-Memory Multiprocessors. – Technical Report 1965, INRIA, March 1993.

[3] M. Banâtre and P. Joubert. – Cache Management in a Tightly Coupled Fault Tolerant Multiprocessor. – In *Proc. of 20th International Symposium on Fault-Tolerant Computing Systems*, pages 89–96, Newcastle, June 1990.

[4] M. Banâtre, G. Muller, B. Rochat, and P. Sanchez. – Design Decisions for the FTM : A General Purpose Fault Tolerant Machine. – In *Proc. of 21st International Symposium on Fault-Tolerant Computing Systems*, pages 71–78, Montréal, Canada, June 1991.

[5] L. A. Barroso and M. Dubois. – Cache Coherence on a Slotted Ring. – In *Proc. of 1991 International Conference on Parallel Processing*, volume 1, pages 230–237, August 1991.

[6] J. Bartlett, J. Gray, and B. Horst. – Fault Tolerance in Tandem Computer Systems. – In A. Avizienis, H. Kopetz, and J.C. Laprie, editors, *The Evolution of Fault-Tolerant Computing*, volume 1, pages 55–76. Springer Verlag, 1987.

[7] Ph. A. Bernstein. – Sequoia: A Fault-tolerant Tightly Coupled Multiprocessor for Transaction Processing. – *IEEE Computer*, 21(2):37–45, February 1988.

[8] K.P. Birman. – Replication and Fault-tolerance in the ISIS System. – In *Proc. of 10th ACM Symposium on Operating Systems Principles*, pages 79–86, Washington, December 1985.

[9] D. B. Johnson E. L. Elnozahy and W. Zwaenepeol. – The Performance of Consistent Checkpoint. – In *Proc. of 11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[10] S. Frank, H. Burkhardt, and J. Rothnie. – The KSR1 : Bridging the Gap Between Shared Memory and MPPs. – In IEEE Computer Society, editor, *Proc. of spring COMPCON'93*, pages 285–294, February 1993.

[11] A. Gefflaut and P. Joubert. – SPAM : A Multiprocessor Execution Driven Simulation Kernel. – *International Journal in Computer Simulation*, To appear, 1994.

[12] J. Gray. – *Notes on Database Operating Systems.*, volume 60 of *Lecture Notes in Computer Science*. – Springer Verlag, 1978.

[13] A. Gupta and W.D. Weber. – Cache Invalidation Patterns in Shared-Memory Multiprocessors. – *IEEE Transactions on Computers*, 41(7):794–810, July 1992.

[14] E. Hagersten, A. Landin, and S. Haridi. – DDM - A Cache-Only Memory Architecture. – *IEEE Computer*, 25(9):44–54, September 1992.

[15] E. S. Harrison and E. Schmitt. – The Structure of System/88, a Fault-Tolerant Computer. – *IBM Systems Journal*, 26(3):293–318, 1987.

[16] B. Lampson. – Atomic Transactions. – In *Distributed Systems and Architecture and Implementation : an Advanced Course*, volume 105 of *Lecture Notes in Computer Science*, pages 246–265. Springer Verlag, 1981.

[17] P.A. Lee and T. Anderson. – *Fault Tolerance: Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*. – Springer Verlag, second revised edition, 1990.

[18] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. – The Stanford Dash Multiprocessor. – *IEEE Computer*, 25(3):63–79, March 1992.

[19] K. Li, J.F. Naughton, and J.S. Plank. – Real-Time Concurrent Checkpoint for Parallel Programs. – In *Second ACM SIGPLAN Symposium on Principles and Practice Parallel Programming (PPOPP), SIGPLAN notices*, volume 25, pages 79–88, 1990.

[20] H. Nilsson and P. Stenström. – Performance Evaluation of Link-Based Cache Coherence Schemes. – *Proc. of the 26th Annual Hawaï International Conference on System Sciences*, pages 486–495, 1993.

[21] H. Schwetman. – Csim user's guide, rev. 2. – Technical Report ACT-126-90, Rev. 2, MCC, July 1992.

[22] J.P. Singh, W.D. Weber, and A. Gupta. – SPLASH : Stanford Parallel Applications for Shared-Memory. – Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, April 1991.