COMA: an Opportunity for Building Fault-tolerant Scalable Shared Memory Multiprocessors*

Christine Morin, Alain Gefflaut, Michel Banâtre, Anne-Marie Kermarrec
IRISA/INRIA
Campus de Beaulieu, 35042 Rennes cedex FRANCE
email: {cmorin,banatre,akermarr}@irisa.fr

Abstract

Due to the increasing number of their components, Scalable Shared Memory Multiprocessors (SSMMs) have a very high probability of experiencing failures. Tolerating node failures therefore becomes very important for these architectures particularly if they must be used for long-running computations. In this paper, we show that the class of Cache Only Memory Architectures (COMA) are good candidates for building fault-tolerant SSMMs. A backward error recovery strategy can be implemented without significant hardware modification to previously proposed COMA by exploiting their standard replication mechanisms and extending the coherence protocol to transparently manage recovery data. Evaluation of the proposed fault-tolerant COMA is based on execution-driven simulations using some of the Splash applications. We show that, for the simulated architecture, the performance degradation caused by fault-tolerance mechanisms varies from 5 % in the best case to 35 % in the worst case. The standard memory behavior is only slightly perturbed. Moreover, results also show that the proposed scheme preserves the architecture scalability and that the memory overhead remains low for parallel applications using mostly shared data.

Key Words

Scalable Shared Memory Multiprocessors, COMA, Fault-tolerance, Backward Error Recovery, 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 efficiently. Shared memory provides a flexible and powerful computing environment. Two classes of SSMM have emerged: Cache Coherent Non Uniform Memory Access machines (CC-NUMA) [1, 20], which statically divide the main memory among the nodes of the architecture, and Cache Only Memory Architectures (COMA) [12, 9] which convert the per node memory into a large cache of the shared address space.

Unfortunately, due to the large number of their components and despite a significant increase in hardware reliability, scalable architectures still have a high probability of experiencing hardware failures. Tolerating node failures thus becomes essential if such architectures are to execute long-running applications. Backward Error Recovery (BER) is a fault tolerant strategy which attempts to restore a previous correct system state after a failure has been detected. To achieve this goal, a consistent system state, called a recovery point (made up of a set of recovery data) has to be periodically saved on a stable storage [17]. BER seems to be the most attractive solution to provide fault tolerance in SSMM since it limits the hardware development and allows the use of all the processors for a computation.

In this paper, we advocate that a COMA is a good candidate to build a fault-tolerant SSMM. We show that COMA features allow an efficient implementation of BER without a significant hardware modification to previously proposed COMA. As COMA nodes are independent from a failure point of view, recovery data of a node can be saved in the memory of another node, leading to an efficient and scalable approach. In contrast to CC-NUMA architectures, memory items in a COMA have no fixed physical location. As a result, recovery point establishment is simplified since recovery data can be created anywhere in the architecture. Reconfiguration after a failure is also considerably simplified since lost memory items can be reallocated in any node without changing their physical address. Finally, the ability of a COMA to replicate data in several memories provides an easy way to ensure recovery data stability.

To back up this idea, we present in this paper a solution based on BER in order to tolerate multiple transient and single permanent node failures in a COMA assuming *fail-silent* nodes and a fault-free interconnection network. We show that a BER strategy can be implemented as a simple extension of the existing coherence protocol of a COMA. Other aspects of fault-tolerance such as detection and error confinement are not in the scope of this paper. The performance of the extended coherence protocol has been studied through simula-

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

tions of a non-hierarchical COMA using some of the Splash applications. Simulation results show that the performance degradation due to fault-tolerance mechanisms remains low and that our approach preserves the scalability of the architecture. The memory overhead induced by the presence of recovery data in the memories is also analyzed.

The remainder of this paper is organized as follows. In Section 2, we explain BER requirements and describe the main features of COMAs. In Section 3, we present how BER can be implemented in a COMA as an extension of the existing coherence protocol. Implementation issues are discussed in Section 4 where we also report performance results. Section 5 concludes.

2 Background

2.1 Backward Error Recovery

BER is the most attractive solution to tolerate node failures in SSMM. As opposed to hardware static redundancy [2, 13] which cannot be considered for architectures with a large number of components, BER limits the hardware development. BER also allows the use of all the processors for a computation, avoiding the need of strong synchronization and the high performance degradation of active software replication schemes [4].

In a shared-memory environment, a pessimistic approach of BER, limiting the recovery data size to a single recovery point per processor, and preventing the domino effect [19], is preferable. A coordinated global checkpointing scheme in which all processors synchronize to establish a new system recovery point is the assumption we will use throughout of this paper.

Implementing BER in a shared memory architecture raises the issue of recovery data management, which is twofold: (i) storing and identifying recovery data, (ii) ensuring their stability.

Storage and Identification Two kinds of solutions are traditionally used. The first one is of storing current[1] and recovery data in two different levels of the memory hierarchy. The *Sequoia* architecture [3] uses the shared memory to keep the recovery data while current data are retained in the caches. This solution provides an easy way to identify recovery data but requires a recovery point to be established each time a data must be copied in the shared memory. Such operations, which depend on architecture characteristics and application behavior may be frequent, resulting in high performance degradation [14]. The second solution keeps current and recovery data in the same level of the memory hierarchy but on different physical supports so that recovery data can be identified by its physical location [7]. The primary drawbacks of this solution are the use of specific hardware, a non optimal use of the support used to keep recovery data, and a lack of flexibility.

Recovery Data Stability Data stability can be defined as the combination of two properties: persistence and atomic update. Persistence ensures that data cannot be altered by a failure and remains accessible despite the occurrence of failures. Atomic update guarantees that updates made to data are either successful operations or leave the data in their initial state. This property is necessary to tolerate failures during recovery point establishments. Ensuring data stability usually requires the data to be replicated on failure independent storages. The number of required copies depends on the number of failures that have to be simultaneously tolerated. Two copies are sufficient to tolerate single failures. Mirror disks, for example, replicate data on two independent disks. One of the solution to implement recovery data stability is to design specific hardware devices [3], which leads to efficient but expensive solutions. The other is to use more traditional devices such as disks which are cheaper but can be the source of an important performance degradation, especially in large-scale architectures where they could limit the recovery data throughput during recovery point establishments.

2.2 Cache Only Memory Architectures

Like other scalable shared-memory multiprocessors, COMAs are organized as a set of processing nodes connected by a high speed interconnection network. Each processing node contains one or more processors, their associated caches, and a portion of the global shared memory. In a COMA, the memory associated with each node is organized as a large cache called an Attraction Memory (AM). Because of this organization, memory lines are automatically replicated and migrated on demand in the AM of the nodes. Coherence is usually maintained on a memory line (also called an item) basis by a hardware coherence protocol, typically a directory-based write-invalidate protocol [5]. In such a scheme, there is no permanent mapping between an item identifier and a physical memory location.

The first COMA machines, such as the DDM [12] or the KSR1 [9], rely on a hierarchical network topology to locate items on misses. Directories at each level of the hierarchy maintain information about item copies located in a sub-hierarchy. A non-hierarchical organization was first proposed in [24] and recent COMA proposals [21] are based on general interconnection networks. From a fault tolerance point of view, a non hierarchical organization is preferable as the loss of an intermediate node in a hierarchy, could cause the loss of the whole underlying sub-system, resulting in multiple failures. COMA-F [24] uses a flat directory organization in which the directory entries are statically distributed among the nodes. Each directory entry maintains the state of an item (*Shared*, *Exclusive* or *Invalid*), a sharing list and the identity of the node holding the *master* copy of the item. The node holding the *master* copy is in charge of answering requests for this particular item.

Replacements of items must be handled with care in a COMA since no physical memory backs up the AMs. The replacement strategy must make sure that there is always at least one valid copy of every item in the system. In the COMA-F [24], exactly one copy of every item is marked *master* for this purpose. The *master* copy must not be purged from the system as it may be the last copy of an item. *Master* copies that are replaced in an AM generate an injection which assures that the item copy is first transferred to another node AM before being replaced.

[1] *Current* data are those used by processors for the computation.

3 Implementation of Backward Error Recovery in COMA

In this study, we consider a non-hierarchical COMA with similar characteristics to the COMA-F [24]. The solution we propose can also be applied to other kinds of COMA.

3.1 Exploitation of COMA Features for Backward Error Recovery

In this section, we show that the data replication mechanisms offered by COMA allow a simple implementation of a BER strategy in which recovery data are stored in the AMs.

As AMs are independent, from a failure point of view, a COMA gives the opportunity to store both current and recovery data in AMs while still ensuring the recovery data persistence property necessary to implement a BER strategy. Because an item has no fixed location in memory, the current copy of an item can be stored in one AM whereas its recovery copy can be kept in any other AM of the architecture. Injection mechanisms normally used in COMA to support the replacement of *master* copies can also be used to create recovery data when a recovery point is established. The atomic update of recovery data can be ensured by a traditional two-phase commit protocol similar to the one used in [8]. The implementation of this protocol is eased by the fact that COMA replication mechanisms allow the creation of as many copies as needed. Finally, as items in AMs are managed with the states of the coherence protocol, identification of recovery data can be ensured by adding new states to the coherence protocol.

This approach has several advantages. The hardware development is limited since no specific device is required to store the recovery data. Fault-tolerance is implemented in an efficient way since the number of recovery points is not constrained by an application's access pattern and architectural characteristics and the recovery data are stored in the AMs which ensures a fast access and a large throughput when a recovery point has to be established. Another advantage is that the already existing copies of items can be used to avoid data transfers during the establishment of a new recovery point. Finally, as recovery copies are stored in the AMs, they can still be accessed by the processors as long as they are identical to the current copies and the creation of recovery copies can be delayed until the first modification of the item. This ensures an optimal use of the memory support since unaccessible recovery copies of an item are only created after the first modification of the item.

The absence of item fixed physical locations also greatly simplifies the reconfiguration step necessary after a failure. Lost items can indeed be reallocated in any valid node of the architecture without any address modification. This is a key advantage of COMA over CC-NUMA. In a CC-NUMA architecture, the reconfiguration phase would be much more complex since some of the data would have to be reallocated with different physical addresses.

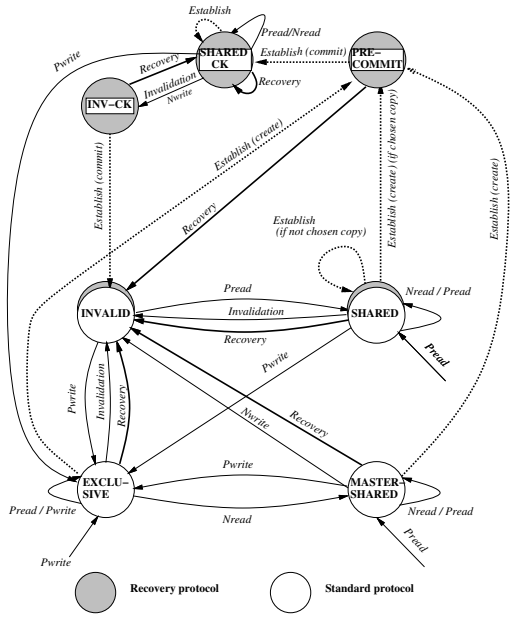
3.2 Extending the Coherence Protocol

Implementing the previous ideas in a COMA can be realized by extending the coherence protocol of the architecture to transparently combine the management of current and recovery data. The basic coherence protocol is depicted by the 4 standard states in Fig. 1. Three new states are added to identify recovery data. The *Shared-CK* (Shared Checkpointed) state identifies the two recovery copies of an item that has not been modified since the last recovery point. Such a copy can be read by the local processor and may serve read misses. The *Inv-CK* (Invalid Checkpointed) state identifies the two recovery copies of an item that has been modified since the last recovery point. Such a copy cannot be accessed by the processors and is only kept for a possible recovery. Hence, read and write hits on an *Inv-CK* copy must be treated as misses and the *Inv-CK* copy must be transferred (*injected*) to another node before the miss is performed. Similarly, an injection is also required when a write hit occurs on a *Shared-CK* copy. The *Pre-Commit* state is a transient state used during the establishment of a recovery point to represent new recovery data. Two new transitions, *Establish (create/commit)* and *Recovery*, represent the establishment and restoration of a recovery point. The resulting protocol illustrated in Fig. 1 is called the Extended Coherence Protocol (ECP) in the remainder of this paper. It ensures that at any time, every item has exactly either two *Shared-CK* copies or two *Inv-CK* copies in two distinct memories.

As the protocol remains similar to a standard protocol, we detail here with just the cases related to the new states of the protocol. After a recovery point establishment, only two *Shared-CK* and possibly other *Shared* copies of an item exist in the architecture. When a read miss occurs, one of the two *Shared-CK* copies is used to serve the request. The requesting node receives a copy of the item and sets its local state to *Shared*. A write request on an item not modified since the last recovery point is handled in an almost identical way as a write request in the standard protocol. Invalidations messages are sent to all nodes with a copy of the item and one of the *Shared-CK* copy sends a copy of the line to the requesting node. As a result, the two Shared-CK copies change their state to *Inv-CK* and all possible *Shared* copies change their state to *Invalid*. At the end of the transaction, the requesting node owns the only valid current copy of the item, in state *Exclusive*. The architecture now contains a current copy of the item and exactly two recovery copies in state *Inv-CK*. At this point, the standard protocol is used for any request on this item and the *Inv-CK* are only kept for a possible recovery.

3.3 Establishing a Recovery point

The two-phase commit protocol shown in Fig. 2 is executed by each node to establish a new recovery point. The goal of the *create* phase is to create the new recovery point while the *commit* phase aims at discarding the previous one and confirming the new one. Before starting the *create* phase, each node first terminates all pending requests. During this phase, two copies of new recovery data are created for only those items that have been modified since the last recovery point (those having an *Exclusive* or a *Master-Shared* copy) since an incremental scheme is used. The first recovery copy is obtained by simply changing the state of *Exclusive* and *Master-Shared* copies to *Pre-Commit* and the second one by replicating the item, in state *Pre-Commit* in any other AM. For replicated *Master-Shared* items (*ie.* with existing *Shared* copies), an optimization consists in choosing



Standard coherence protocol		
States		
Invalid		
Exclusive	only valid copy of an item	owner †
Master-shared	shared copies	
Shared		
Transitions		
Pread/Pwrite	requests from the local processor	
Nread/Nwrite	requests from a remote processor	
Invalidation		

Extended coherence protocol		
Specific states		
Inv-CK	not accessible	recovery
Shared-CK	read-only	copies
Specific transitions		
Establish (create/commit)		
Recovery		

†An AM owning an item is responsible for answering the requests on this item and for transfering the item before replacing it.

Figure 1: Extended coherence protocol state transition diagram

one of the replica to become the second recovery copy (in state *Pre-Commit*), thus avoiding a data transfer and the creation of an additional copy. Once all nodes have terminated the *create* phase, the *commit* phase, local to each node, can be engaged. Each node scans its memory and simply sets all its *Inv-CK* copies to *Invalid* and all its *Pre-Commit* copies to *Shared-CK*. After a successful recovery point establishment, every item has two recovery copies in state *Shared-CK* and possibly other read-only (in *Shared* state) copies in the system.

It is important to note that any failure during the first or second phase of the algorithm can be correctly handled. During the *create* phase, the previous recovery point (the set of all *Inv-CK* and *Shared-CK* copies) is still unaltered and can be restored. During the *commit* phase, the new recovery point (the set of all *Pre-Commit* and *Shared-CK* copies) is complete and persistent since all items are already replicated. Since this phase is local, any failure during the phase can be treated at the end of the phase, as if the failure occurred during the computation after the atomic update operation.

3.4 Restoring a Recovery Point

After a node failure has been detected, the purpose of the restoration phase is to reinstall the previous recovery point made of all *Shared-CK* and *Inv-CK* item copies as the standard consistent system state. All other item copies must be 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 (in state *Shared*, *Exclusive* or *Master-Shared*) as well as *Pre-Commit* copies. Note that *Shared* copies must be invalidated because there is no way to know whether they correspond to current or recovery item copies. *Inv-CK* copies are restored to *Shared-CK* since they belong to the consistent global system state currently being restored. No action is required for *Shared-CK* copies. At the end of the error recovery phase, only two *Shared-CK* copies exist for each item of the shared memory space.

In the event of a permanent failure, a memory reconfiguration must also be performed. This reconfiguration is done in order to duplicate lost item copies which were located on the faulty node so that the persistence property is again satisfied. After the recovery phase, only *Shared-CK* item copies exist. To reconfigure the architecture, each *Shared-CK* copy 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.

4 Implementation

In this section we present the hardware modification introduced by the extended coherence protocol. We consider here an architecture similar to the COMA-F [24]. Each computing node contains a processor, a cache, an AM and a network interface. The interconnection network connecting the nodes is a 2-dimensional mesh. The coherence protocol is also similar to the COMA-F one. To locate an item on a miss, *localization pointers* are used. These pointers are statically distributed on the architecture nodes. As opposed to the COMA-F, the directory entry of an item is maintained on the node which is the current owner of the item.

4.1 ECP Implementation Issues

On a node, all the modifications introduced by the ECP are related to the implementation of the AM and its associated controllers implementing the coherence protocol. The processor and its cache do not need to be modified. To simplify the presentation, we assume that the implementation of the coherence protocol uses two distinct protocols. The *below protocol* implements the part of the coherence protocol dealing with accesses from the local processor. The *above protocol* deals with remote accesses.

```
Create 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;
    Master-Shared:
        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 Create Phase


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

Figure 2: Create/commit algorithm

The first modification required by the ECP is in the AM. New states must be added to take into account the new states introduced in the coherence protocol. To avoid any coherence violation (multiple owners), two different *Shared-CK* states (*Shared-CK1*, *Shared-CK2*) must be distinguished so that only one of them (the *Shared-CK1* copy) can deliver exclusive access rights to a given item. As *Inv-CK* copies are likely to be restored as *Shared-CK* copies, the *Inv-CK* state must also be split into two distinct states. With two distinct *Pre-Commit* states, the number of new stable states introduced by the ECP reaches six. Encoding these new states require three additional bits per item.

Modifications of the *below protocol* are very simple and do not introduce new functionalities. The *below protocol* must allow read accesses to *Shared-CK* copies and trigger an injection on a write access to a *Shared-CK* copy or on any access to an *Invalid-CK* copy. The replacements triggered by the *below protocol* introduce new transient states maintained with other transient states in a separate, small transient state memory.

The modifications introduced in the *above protocol* are related to read or write requests on a *Shared-CK1* copy, new injections of recovery copies and new algorithms necessary to implement the establishment and restoration of a recovery point. Read and write requests on a *Shared-CK1* copy are handled in a similar way as read and write requests on a *Master-Shared* copy. The only difference is that in the write request, the *Shared-CK1* copy changes its state to *Invalid-CK1* and an invalidation is also sent to the *Shared-CK2* copy.

Injections of recovery data (see Table 1) must be handled with care since they could result in the loss of a recovery item copy. In order to easily find a place for an injected line, a logical ring is mapped onto the physical interconnection network. This logical ring must be reconfigured in the event of a failure. To accept an injection, an AM can only replace one of its *Invalid* or *Shared* lines[2]. If the injection cannot be accepted, the node forwards the injection to the next node on the logical ring. Injections are accomplished in two steps. In a first step, an injection message is sent to find a victim line on a remote node. When the victim node replies, the data is sent. As long as the injection is pending, the source node of the injection cannot replace the injected line. Handling injections represents the most complex modification in the standard coherence protocol.

As in traditional COMAs, an architecture using the ECP must guarantee that an injected copy of a line will always find a place in the set of AMs. In the KSR1, this problem is solved by allocating an irreplaceable page for each page allocated in the architecture [9]. A similar problem is raised by recovery lines at recovery point establishment time. Four copies are necessary during the *create* phase. In our study, four pages are statically allocated as irreplaceable pages instead of one, to ensure that there is always enough memory space for establishing a new recovery point.

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 ECP

The implementation of the *create/commit* and *recovery* algorithms is quite simple. Memory item replication needed for a recovery point establishment does not require any new functionality since these requests are similar to item injections, the only difference being that the injected item copy is not replaced in the memory of the node performing the injection.

One of the problems of the algorithm presented in Fig. 2, is that it is based on a sequential scanning of the state memory. Such a scanning can be very long if the AM is large. To limit the time necessary to identify modified items that have to be replicated during the *create* phase, we assume that some supplementary information allows a node to identify a modified line during the injection time of a previous line. Such information can be organized as a tree where each level of the tree indicates whether there are modified lines in a sub-part of the memory. As a consequence, a line is ready to be injected as soon as the previous injection is done. The implementation of the *commit* phase uses a simple optimization which consists in testing only the allocated pages in the AM. Another simple optimization could be to test only modified pages, if such information is available.

We do not address in this paper the software issues raised by the implementation of a BER strategy. There is however evidence that some modifications of the operating systems of the architecture would be necessary to integrate the establishment and restoration of recovery points.

[2] For injections of *Shared-CK* copies, a node waiting for a read copy of the line can also accept the injection.

4.2 Results Analysis

In this section, we study the performance overheads introduced by the ECP by comparing simulation results obtained with a simulator using a standard coherence protocol and a second simulator using the ECP.

4.2.1 Simulation Environment

The simulator used in this study uses the SPAM simulation kernel that allows an efficient implementation of an execution-driven simulation method [6]. The memory references are generated by parallel applications instrumented with a modified version of the Abstract Execution technique [18]. To ensure that the produced trace corresponds to the target architecture, the architecture simulator can control the execution of the traced processes. The architecture simulator is implemented with a discrete event simulation library providing management, scheduling and synchronization of lightweight processes [22].

Read miss access	Number of cycles
Fill from cache	1 cycle
Fill from local AM	18 cycles
Fill from remote AM (1 hops)	116 cycles
Fill from remote AM (2 hops)	124 cycles

Table 2: Read miss latency times

4.2.2 Architectural Parameters

Most of the physical characteristics of the simulated architecture (except network characteristics) come from the KSR1 architecture [9]. The processor uses a 20Mhz clock and only one level of cache. The data cache is a sectored 8-way associative 256 KB cache. The sector size is 2KB and the line size is 64 bytes. The instruction cache has the same characteristics but is not modeled in the architecture simulator, and we assume that the instructions have a 100% hit ratio. The AM is 16-way set associative and uses 16KB page allocation. The size of the AM is fixed to 8 MB. Each page is subdivided in 128 items of 128 bytes. The data transfer unit between nodes is an item and coherence is maintained on an item basis. The cache access time is fixed to 1 cycle. The local bus is assumed to be 64 bits wide and a cache miss serviced by the local AM takes 18 cycles. Accessing and transferring a 128 bytes item from the memory to the network controller takes 20 cycles. To limit the injection stall time, the injection acknowledgment is sent 5 cycles after the reception of the item on the node. Copying the item to memory is performed after the acknowledgment is sent.

Nodes are connected through a worm-hole routed synchronous mesh using a flit size of 32 bits. The network is made of two sub-networks, one used for requests, the other used for replies. The network fall-through time is one cycle (50 ns) resulting in a transfer rate of 76 Mbytes/s between two nodes. The simulator correctly models contention in all parts of the system. Table 2 gives the time it takes to satisfy a read miss from different levels of the memory hierarchy assuming no contention and a 4x4 mesh.

The implementation of the *create* algorithm assumes that the time between 2 injections is sufficient to identify another item to be replicated. The *commit* algorithm charges 1 cycle to test if a page is allocated and 1 cycle to test and modify the state of an item in the page (we assume SRAM for the implementation of the AMs). As in the KSR1, four independent controllers implement the AMs.

In this evaluation, we use four parallel applications from the SPLASH benchmark suite [23] representing a variety of shared memory access patterns. Table 3 describes their characteristics. For all these benchmarks, statistics are collected during the parallel phase. As the size of the AM is large compared to the size of the applications, no capacity replacements occur during the simulations.

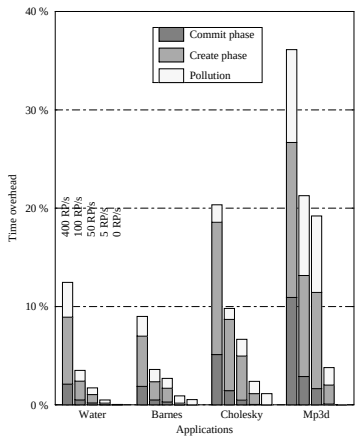


Figure 3: Time overhead

4.2.3 Overhead Study

The first overhead introduced by the ECP is a time overhead resulting in longer execution times than on the standard architecture. To identify this overhead, we compare results obtained by the architecture using the standard coherence protocol and the architecture using the ECP. The time overhead can be divided into two separate effects: (1) the time required to create/commit new recovery points (T_{create} and T_{commit} respectively), (2) a memory pollution effect caused by an increase of the number of misses and injections in the AM ($T_{Pollution}$). The execution time with the ECP can then be expressed as the sum of four components, $T_{Ft} = T_{Standard} + T_{create} + T_{commit} + T_{Pollution}$ where $T_{Standard}$ is the execution time on the standard architecture. Fig. 3 depicts these different values for each application and for various recovery point establishment frequencies ranging from 400 to 5 recovery points per second. These frequencies may seem quite high compared to other evaluations [8]. In the absence of real recovery point frequencies, they give, however, the performance degradation for different computing environments. All the simulations are sufficiently long so that several recovery point establishments occur.

Create Phase Overhead T_{create} is the protocol's largest overhead. Depending on applications and recovery point frequencies, this overhead varies from 1 or 2 % in the best case to 15 % in the worst case (Mp3d with 400 recovery points per second). T_{create} principally depends on the size of the recovery data that must be

Applications	Parameters	Instructions (millions)	Reads	Writes	Shared Reads	Shared Writes
Barnes-Hut	1536 bodies 11 iterations	190	49,5 (18,4%)	28,7 (10,7%)	11,1 (4,2%)	0,27 (0,1%)
Cholesky	bcsstk14	53,1	17.6 (23,3%)	4.7 (6,2%)	14,2 (18,8%)	2,5 (3,3%)
Mp3d	50 K molecules 8 steps	48,3	10,6 (16,3%)	6,3 (9,7%)	8,6 (13,1%)	5,4 (8,3%)
Water	120/144 molecules 2 iterations	78,6	26,8 (23,7%)	7,8 (6,9%)	4,9 (4,3%)	0,58 (0,5%)

Table 3: Simulated applications characteristics

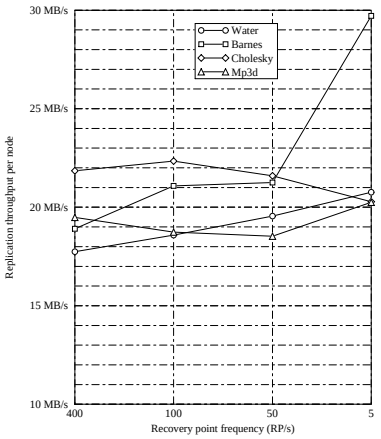


Figure 4: Per node replication throughput of modified data

transferred which is itself directly influenced by the recovery point frequency. At low recovery point frequencies, T_{create} becomes very low for all applications since items can be modified several times between two successive recovery points. With Cholesky, for example, the amount of data replicated at 400 recovery points per second is 8 times the amount of data replicated at 5. The total amount of data handled during recovery point establishments then decreases from 10 to 1.2 Mbytes.

T_{create} is also influenced by the applications characteristics. As an example, Mp3d which exhibits a high write rate (10 %), reports a larger amount of data replicated (4 Kbytes per processor for 10 000 memory references at 400 recovery points per second). Moreover, the larger the application working set is, the more data may be modified. For instance, the different T_{create} overheads of Mp3d and Barnes, both of which have about the same modification rate, may be explained by the different size of their working set (Mp3d's set is 9 times greater than that of Barnes). Finally, locality of memory accesses also influences the amount of recovery data treated.

The low T_{create} overhead may be explained by the use of a high bandwidth interconnection network to transfer recovery data. For the simulated architecture, the replication throughput during recovery point establishments is around 20 MB/s per node for all applications (see Fig. 4). Moreover, by using the existing replication, the protocol avoids some data transfers. Among the four studied applications, Barnes, which uses many mostly-read shared data, illustrates this property. At 5 recovery points per second, 52% of the items which have to be replicated during the establish phase are already replicated. The replication throughput increases to 30 MB/s per node.

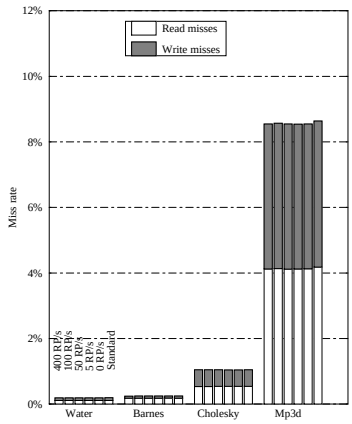


Figure 5: Node miss rate when varying the recovery point frequency

Commit Phase Overhead T_{commit} depends only on the number of pages allocated on the architecture nodes. Thus, its overhead is directly related to the working set size and the recovery point frequency. Applications with a large working set (Mp3d, Cholesky) show the largest T_{commit} time overhead. Solutions using a node recovery point counter, incremented each time a new recovery point is confirmed, and recovery point counters associated with each memory item could be used to avoid scanning the AMs during the commit phase and would nullify T_{commit}.

Pollution Effect The $T_{Pollution}$ overhead is induced by the ECP which keeps the recovery item copies in the AMs. Storing recovery data in the AMs has two effects: (1) a variation of cache and AM misses, (2) a creation of new item injections[3]. Whatever the recovery point establishment rate, this pollution overhead appears to be quite limited and ranges from approximately 10% in the worst case to less than 2%. This limited pollution effect is mainly the consequence of a low increase in the number of misses.

In the caches, only the number of write misses slightly increases since cached modified data, flushed to memory when a recovery point is established, remain in the cache and can still be read by processors. Fig. 5 shows the

[3]These two effects are combined for read and write accesses on *Inv-CK* copies.

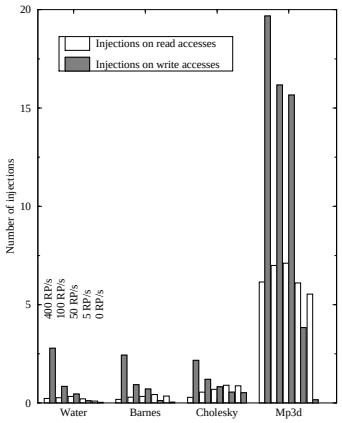


Figure 6: Node average number of injections (on read or write accesses) for 10 000 memory references when varying the recovery point frequency

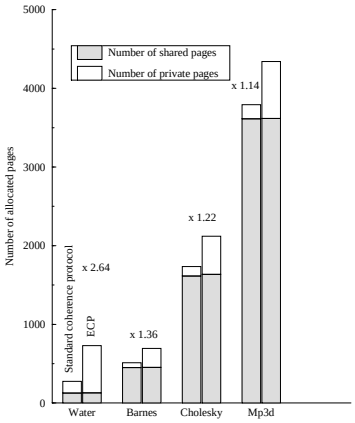


Figure 7: Page allocation comparison between an architecture using ECP and one using a standard protocol

variation of the average AM miss rate for the different recovery point frequencies. The miss rate variation is negligible whatever the recovery point frequency. The constant read miss rate confirms one of the ECP advantages, which is to allow processors to read recovery data that have not been modified since the last recovery point. For Barnes and 400 recovery points per second, the number of *Shared-CK* items reads represents 33% of read requests coming from the cache. In some cases, the read miss rate may even decrease with increasing recovery point frequencies if the application exploits the replication created by the recovery point establishments. As an example, the read miss rate of water decreases from 1.13% to 1.09 % at 400 recovery points per second. The weak variation of write misses may be explained by the fact that the applications often use migratory data that generate write misses anyway.

However, the slight write miss rate increase of the AMs does not explain the pollution effect shown in Fig. 3. This effect is in fact due mainly to new items injections (see Table 1). Fig. 6 shows the variation of the number of injections per AM for 10 000 memory references for various recovery point frequencies. The total number of injections is low, at most 25 for 10 000 references. The number of injections on read accesses is roughly independent of the recovery point frequency. As in the case of the constant read miss rate, this is due to the fact that the ECP allows processors to read unmodified recovery item copies. Most of the new injections are actually caused by write accesses on recovery copies. Their number increases with the recovery point frequency because current item copies are frequently transformed into recovery item copies. At 400 recovery points per second, the number of injections caused by write accesses on *Shared-CK1* copies represents, depending on the applications, 88% to 98% of the total number of injections on write accesses for a node. These injections are the principal cause of the pollution effect.

The processor and network are low-performance compared to current multiprocessor architectures. [10] reports results obtained with a 100 MHz processor and a network similar to Flash one. This study shows that the performance degradation decreases for all applications.

4.2.4 Memory Overhead

Another overhead introduced by the ECP is the need of additional memory space for storing the recovery copies. This overhead varies with time. After the end of the *commit* phase, the minimum number of copies for an item is two. 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 *create* phase, the minimal number of copies for a modified item is four since two recovery points are kept. For shared data, these values do not represent the real memory overhead since the standard protocol already replicates shared memory items on several nodes.

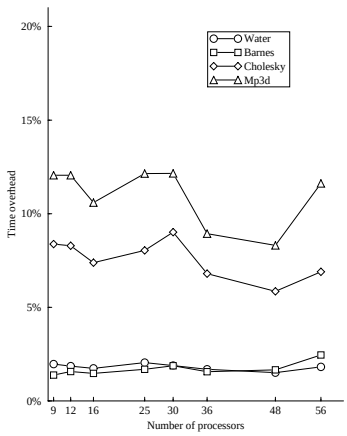


Figure 8: Phase 1 cost when varying the number of processors

Fig. 7 shows the memory overhead measured in the simulations. For each application, the number of pages allocated by the standard architecture as well as the number of pages allocated by the fault tolerant application, are presented. The memory overhead ranges from 1.1 to 2.6. Shared memory pages do not produce any memory overhead even if four copies are statically allocated for each data page in the ECP. This result is due mainly to the large page size (16KB) used in the architecture, which favors false sharing and hence results in the allocation of a page in several memories[4]. Private memory pages are normally allocated on a single node. With the static four page allocation strategy, the over-

[4]When a processor references an address not found in its AM, a *page* is allocated. The contents of the newly created page are filled as needed, one *item* at a time. Hence, there is room for item recovery copies in already allocated pages.

head induced by private pages is four times the number of private pages allocated in the standard architecture.

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. These results present, however, a favorable situation since, because of the small size of the applications, no page replacement occurs in the memories with the architecture using a standard protocol. Consequently, most of the shared pages are allocated on several nodes of the architecture and the four necessary copies for the ECP are already allocated with the architecture using a standard protocol. The memory overhead generated by the ECP is then practically nil for such pages. With larger applications, the memory overhead could be more important. The exact amount of memory necessary for the ECP is however hard to evaluate. The ability of the ECP to use memory space allocated but not used, to store recovery data, makes us think that it should be inferior to four times the memory necessary for an architecture using a standard coherence protocol. To perform well, the ECP will however require a non-negligible amount of additional memory which might then represent one of the principal overheads introduced by the ECP.

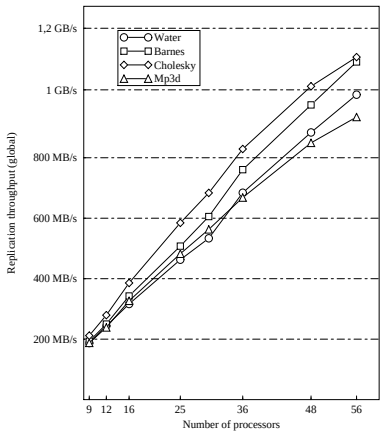


Figure 9: Recovery data throughput when varying the number of processors

4.2.5 Scalability

When designing the ECP, one of the goals was to preserve the architecture's scalability. In this section, we evaluate the scalability of an architecture using the ECP by varying the number of nodes from 9 to 56 and measuring T_{create} and $T_{Pollution}$ overheads with the recovery point frequency fixed to 100 recovery points per second. T_{commit} is not considered here since it does not depend on the number of processors.

Fig. 8 shows T_{create} overhead as a function of the number of processors. The time overhead is constant and even decreases when the number of processors increases. Two reasons explain this behavior. The first is that with fixed-size applications and a larger number of processors, the amount of recovery data treated by each processor at each recovery point decreases. For Mp3d, the size ranges from 9.6 KB with 30 processors to 6.8 KB with 56 processors. The second reason is a nearly linear increase of the recovery data replication throughput (see Fig. 9). For Cholesky, the aggregate replication

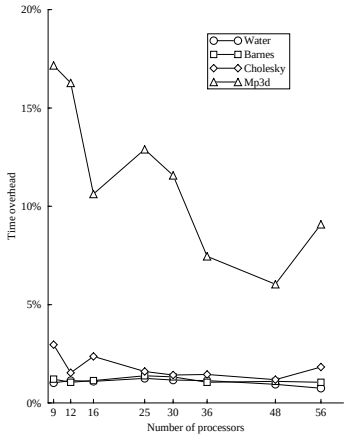


Figure 10: Pollution effect when varying the number of processors

throughput grows from 211 MB/s with 9 processors to 1.1 GB/s for 56.

Fig. 10 presents the pollution effect for different numbers of processors. As before, this effect remains the same or decreases with the number of processors. Results for the average number of injections per node

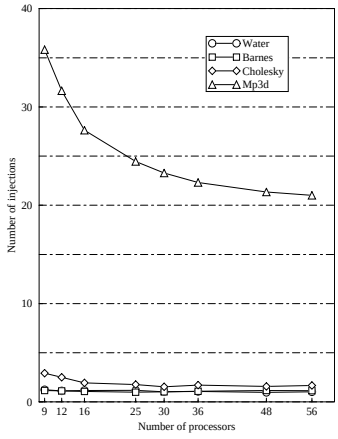


Figure 11: Average number of injections (on read and write accesses) per node for 10 000 memory references when varying the number of processors

shown in Fig. 11 explain this behavior. The number of injections per node is either constant when its value is low or decreases when its value is already high (Mp3d). While the number of injections on writes remains constant, the number of injections on reads decreases when the number of processors grows. For Mp3d, the number of injections on reads for 10 000 references decreases from 16.6 with 9 processors to 3.3 with 56 processors. This decrease is caused by shared items that have a greater probability of occupying an unused memory location, benefiting from a greater number of page copies when the number of processors increases.

5 Conclusion

Building fault tolerant SSMM is attractive to meet the requirements of high performance long-running parallel applications. We have demonstrated that a COMA is a sound architectural basis to build a fault tolerant

SSMM. COMA have indeed a natural advantage over CC-NUMA since their replication mechanisms facilitate the implementation of a BER strategy for tolerating node failures. Recovery data can be stored in AMs and transparently managed by the coherence protocol extended for this purpose. Hence, only limited hardware modifications are required to already proposed COMA. Measurements performed by simulation show that this approach is both efficient and scalable. The presence of recovery data in standard memories slightly disturb the normal behavior of the basic coherence protocol. However, overheads depend on application characteristics and on the frequency of recovery data update operations.

Our approach is not limited to non-hierarchical COMAs. The extended coherence protocol can also be implemented with snooping coherence protocols [11]. It is particularly well-suited to new architectures such as FLASH [16] where the coherence protocol is implemented by software. Our approach is more generally applicable to architectures implementing a shared memory on top of distributed physical memories. In particular, it can be used to implement a recoverable distributed shared virtual memory (DSVM) on top of a multicomputer or a network of workstations. We have already implemented a recoverable DSVM based on the ECP on the Intel Paragon multicomputer and on a network of workstations running Chorus micro-kernel [15].

Acknowledgment

We are grateful to André Seznec for his valuable comments on an earlier draft of this paper.

References

[1] Agarwal, A., Chaiken, D., Johnson, K., Kranz, D., Kubiatowicz, J., Kurihara, K., Lim, B., Ma, G., and Nussbaum, D. The MIT Alewife machine : A large-scale distributed memory multiprocessor. Research report MIT/LCS/TM-454, MIT Laboratory for Computer Science, June 1991.

[2] Bartlett, J., Gray, J., and Horst, B. Fault tolerance in Tandem computer systems. In *The Evolution of Fault-Tolerant Computing*, A. Avizienis, H. Kopetz, and J. Laprie, Eds., vol. 1. Springer Verlag, 1987, pp. 55–76.

[3] Bernstein, P. Sequoia: A fault-tolerant tightly coupled multiprocessor for transaction processing. *IEEE Computer 21*, 2 (February 1988), 37–45.

[4] Birman, K. Replication and fault-tolerance in the ISIS system. In *Proc. of 10th ACM Symposium on Operating Systems Principles* (Washington, December 1985), pp. 79–86.

[5] Chaiken, D., Fields, C., Kurihara, K., and Agarwal, A. Directory-based cache coherence in large-scale multiprocessors. *IEEE Computer 23*, 6 (June 1990), 49–58.

[6] Davis, H., Goldschmidt, S., and Hennessy, J. Multiprocessor simulation using Tango. In *Proc. of 1991 International Conference on Parallel Processing* (August 1991), vol. II, pp. 99–107.

[7] Din, M., Grygier, A., Hessenauer, H., Hildebrand, U., Hönig, J., Hohl, W., Michel, E., and Pataricza, A. Fault tolerance in distributed shared memory multiprocessors. In *Parallel Computer Architectures* (1994), A. Bode and M. Cin, Eds., vol. 732 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 31–48.

[8] Elnozahy, E., Johnson, D., and Zwaenepoel, W. The performance of consistent checkpoint. In *Proc. of 11th Symposium on Reliable Distributed Systems* (October 1992), pp. 39–47.

[9] Frank, S., Burkhardt, H., and Rothnie, J. The KSR1 : Bridging the gap between shared memory and MPPs. In *Proc. of spring COMPCON'93* (February 1993), I. C. Society, Ed., pp. 285–294.

[10] Gefflaut, A. *Proposition et évaluation d'une architecture multiprocesseur extensible à mémoire partagée tolérante aux fautes*. PhD thesis, Université de Rennes I, January 1995.

[11] Gefflaut, A., Morin, C., and Banâtre, M. Tolerating node failures in cache only memory architectures. In *Proc. of Supercomputing'94* (November 1994).

[12] Hagersten, E., Landin, A., and Haridi, S. DDM - a cache-only memory architecture. *IEEE Computer 25*, 9 (September 1992), 44–54.

[13] Harrison, E., and Schmitt, E. The structure of SYSTEM/88, a fault-tolerant computer. *IBM Systems Journal 26*, 3 (1987), 293–318.

[14] Janssens, B., and Fuchs, W. Experimental evaluation of multiprocessor cache-based error recovery. In *Proc. of 1991 International Conference on Parallel Processing* (August 1991), vol. I, pp. 505–508.

[15] Kermarrec, A., Cabillic, G., Gefflaut, A., Morin, C., and Puaut, I. A recoverable distributed shared memory integrating coherence and recoverability. In *Proc. of 25th International Symposium on Fault-Tolerant Computing Systems* (Pasadena, USA, June 1995), IEEE Computer Society Press, pp. 289–298.

[16] Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M., Gupta, A., Rosenblum, M., and Hennessy, J. The Stanford FLASH multiprocessor. In *Proc. of 21th Annual International Symposium on Computer Architecture* (Chicago, Illinois, April 1994), pp. 302–313.

[17] Lampson, B. Atomic transactions. In *Distributed Systems and Architecture and Implementation : an Advanced Course*, vol. 105 of *Lecture Notes in Computer Science*. Springer Verlag, 1981, pp. 246–265.

[18] Larus, J. Abstract execution : A technique for efficiently tracing programs. *Software Practice and Experience 20*, 12 (December 1990), 1251–1258.

[19] Lee, P., and Anderson, T. *Fault Tolerance: Principles and Practice*, second revised ed., vol. 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer Verlag, 1990.

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

[21] Saulsbury, A., Wilkinson, T., Carter, J., and Landin, A. An argument for simple COMA. In *Proc. of 1st IEEE Symposium on High-Performance Computer Architecture* (January 1995).

[22] Schwetman, H. CSIM user's guide, rev. 2. Tech. Rep. ACT-126-90, Rev. 2, MCC, July 1992.

[23] Singh, J., Weber, W., and Gupta, A. SPLASH : Stanford parallel applications for shared-memory. Tech. Rep. CSL-TR-91-469, Computer Systems Laboratory, Stanford University, April 1991.

[24] Stenstrom, P., Joe, T., and Gupta, A. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *Proc. of 19th Annual International Symposium on Computer Architecture* (May 1992), pp. 80–91.