An Efficient and Scalable Approach for Implementing Fault-Tolerant DSM Architectures

Christine Morin, Anne-Marie Kermarrec, Michel Banâtre, Alain Gefflaut

An Efficient and Scalable Approach for Implementing Fault-Tolerant DSM Architectures

Christine Morin, *Member*, *IEEE Computer Society*, Anne-Marie Kermarrec, Michel Banâtre, and Alain Gefflaut

Abstract—Distributed Shared Memory (DSM) architectures are attractive to execute high performance parallel applications. Made up of a large number of components, these architectures have however a high probability of failure. We propose a protocol to tolerate node failures in cache-based DSM architectures. The proposed solution is based on backward error recovery and consists of an extension to the existing coherence protocol to manage data used by processors for the computation and recovery data used for fault tolerance. This approach can be applied to both Cache Only Memory Architectures (COMA) and Shared Virtual Memory (SVM) systems. The implementation of the protocol in a COMA architecture has been evaluated by simulation. The protocol has also been implemented in an SVM system on a network of workstations. Both simulation results and measurements show that our solution is efficient and scalable.

Index Terms—Distributed shared memory, fault tolerance, coherence protocol, backward error recovery, scalability, performance, COMA, SVM.

1 Introduction

Distributed Shared Memory (DSM) architectures are attractive for the execution of high performance parallel applications since they provide the simple shared memory paradigm in a scalable way. Scalability of DSM architectures relies on their physically distributed memory and their high bandwidth interconnect, enabling their shared memory to support the bandwidth demand of a large number of processors. In this paper, we consider cache-based DSMs. Such architectures rely on the automatic migration and replication of data so that the data is local to the processors using it for computation in order to execute parallel applications at maximum speed. Scalable shared memory architectures and software-based DSMs implemented on multicomputers or networks of workstations (NOW) are examples of such architectures. These architectures implement a coherence protocol to manage the multiple copies of a data in different node memories. Due to their large number of components, and despite a significant increase in hardware reliability, these architectures may experience hardware failures. Thus, tolerating node failures becomes essential if such architectures are to execute long-running applications whose execution time is longer than the architecture Mean Time Between Failures (MTBF). Backward Error Recovery (BER) [1] is part of a fault tolerance strategy which restores a previous consistent system state after a failure has been detected. To achieve this goal, a consistent system state, made up of a set of recovery data, has to be periodically saved on stable storage. A stable storage ensures that 1) data is not altered (*permanence* property) and remains accessible (*accessibility* property) despite a failure, and that 2) data is updated atomically in presence of failures (*atomicity* property). Using BER to provide fault tolerance in DSM architectures is more attractive than using static hardware replication [1] because BER limits the hardware cost. In contrast with active replication schemes [1], it allows the use of all the processors for computation.

Our goal is to allow parallel applications executing on a DSM architecture to continue their execution despite the occurrence of a node failure. In this paper, we propose a low overhead and scalable error recovery approach based on BER to tolerate transient and single permanent node failures in cache-based DSM architectures exemplified by Cache Only Memory Architectures (COMA) and Shared Virtual Memory (SVM) architectures. We have simply extended the standard coherence protocol of a DSM. Our extended coherence protocol is designed to manage both data accessed by processors for computation (active data) and that required for recovery. Our approach has lots of advantages. First, both types of data are stored in the node volatile memories so no specific hardware needs to be developed. Implementing the protocol in a COMA only requires slight modifications to the existing cache controller, but no new functionality needs to be added. Another significant advantage of our scheme is that, on one hand, it takes benefit of the replication inherent to the DSM by using active data already replicated to avoid the need to create additional recovery data. On the other hand, recovery data can be used for normal computation until the corresponding active data is modified for the first time after a recovery point. Thus, the replication required for providing fault tolerance can be exploited during failure-free executions.

- *C. Morin and M. Banâtre are with IRISA/INRIA, Rennes, France. E-mail: {cmorin, banatre}@irisa.fr.*
- *A.-M. Kermarrec is with Microsoft Corp., Cambridge, UK. E-mail: annemk@microsoft.com.*
- *A. Gefflaut is with IBM T.J. Watson Research Center, Hawthorne, NY. E-mail: alaing@us.ibm.com.*

Manuscript received 1 Jan. 1997; revised 15 June 1998; accepted 30 Nov. 1999.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 111780.

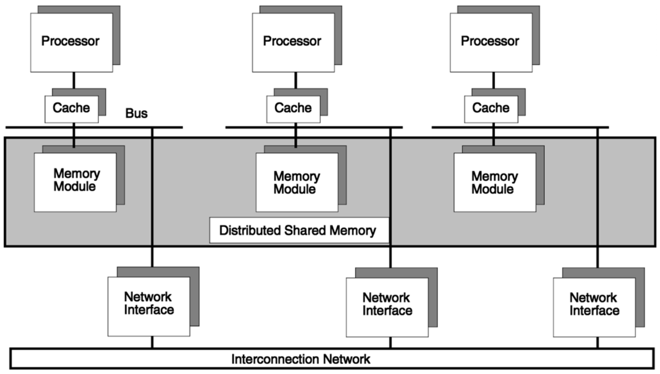

Fig. 1. Generic DSM model.

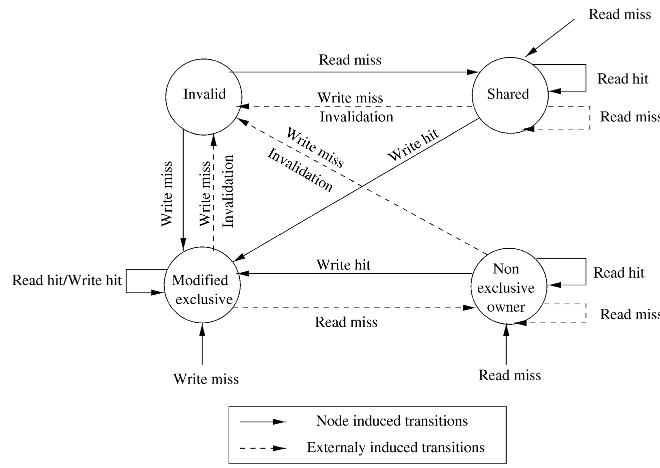

Fig. 2. Basic write-invalidate protocol.

Performance results obtained by simulation in COMAs and measured in SVM architectures implementing the proposed protocol show that our approach is both efficient and scalable. It is efficient as it limits the disturbance caused by the fault tolerance scheme on failure-free executions. Our approach also preserves the scalability of DSM architectures for the following reasons. First, when increasing the number of nodes in a DSM architecture, more memory is available and, thus, the presence of recovery data in volatile memories has less impact on the memory behavior. Second, the creation of recovery data can be processed more efficiently when recovery points are established as the number of processors processing recovery data concurrently increases with the number of nodes. Moreover, node to node data transfers can be dealt with very efficiently by the scalable interconnect of DSM architectures.

The remainder of this paper is organized as follows. Section 2 presents a common generic model of cache-based DSM architectures in which COMAs and SVM systems fit. It also introduces fault tolerance assumptions and the design guidelines of our approach to provide error recovery in DSM architectures. In Section 3, we describe our extended coherence protocol, which implements an efficient backward error recovery strategy in DSM architectures. Issues raised by the implementation of the coherence protocol in COMA and SVM architectures are discussed in Section 4. Results of the performance evaluation for the considered architectures are provided in Section 5, where we also point out the advantages of our approach. Section 6 concludes.

2 Design Guidelines

2.1 System Model

DSM systems are composed of a set of processing nodes interconnected through a scalable interconnection network. Each node is composed of one or more processors, their associated caches, and a memory. A DSM system provides a single shared address space despite distributed memories. Thus, it is particularly attractive from the programmers' point of view since it provides a simple way of programming.

In this paper, we focus on DSM architectures implementing a dynamic address space, for which a generic model is depicted in Fig. 1. Both COMAs such as DDM [2] or the KSR1 machine [3], in which the shared address space is implemented by specialized hardware, called *attractive memories*, and SVM systems, which are implemented by software on multicomputers such as the Intel Paragon [4] or NOWs [5] are based on a dynamic address space, that is to say that the memory of a node is used as a large cache for the node's processors. In such systems, data is automatically replicated or migrated on demand to the memory of the node of the processor that uses the data in a computation, in order to exploit both spatial and temporal locality and to decrease memory latencies. These transfers are managed transparently from the application's point of view.

The main characteristics of such architectures are that, first, there is no fixed physical location for data and, second, since replication of an item is allowed in several memory modules for efficiency, a coherence protocol is required to maintain consistency between the multiple copies. In write-update protocols [6], on each write on a memory item, all the nodes having a copy of the same item also perform the update. In contrast, in write-invalidate protocols [7], an invalidation of all existing copies of a replicated item is required when it is modified by a processor. Write-invalidate coherence protocols are preferred to write-update ones since a coherence operation is not required at each write. In this paper, we consider DSM systems implementing a directory-based write-invalidate coherence protocol. In directory-based protocols, a directory contains one entry for each memory item, maintaining the list of nodes holding a copy of the item. The node holding the directory entry for an item is called its manager. When a data not present in local memory is referenced, the directory is first read to forward the request to the node that is able to deal with it. Directories may be static or dynamic. In the former case, each node manages a statically defined portion of the directory. In the latter, a directory entry has no static location but is associated with the current owner of the item.

A standard write-invalidate protocol is shown by the transition diagram in Fig. 2. Such a protocol is implemented on each node. A node induced transition represents a read or write request issued by a local node processor. Externally induced transitions are related to requests from remote node processors concerning data for which the local node holds a copy in its memory. A read (resp. write) miss occurs when a processor issues a read (resp. write) request related to a data which is not present in the local memory. A read (resp. write) request which can be satisfied locally is called a read (resp. write) hit.

We assume the concept of unique ownership [7], the owner of the item being the unique processor allowed to serve write misses. A memory item b of the shared address space may be in one of the following states in a node memory:

- *Modified exclusive*: The memory of node n contains the unique copy of item b. It can be read or written.
- *Nonexclusive owner*: The memory of n contains a copy of b which can only be read. Other copies may exist in the system in state *Shared*.
- *Shared*: The memory of n contains a read-only copy of b. One copy in state *Nonexclusive owner* must exist in the system.
- *Invalid*: The memory of n does not contain an up-to-date copy of b.

A node whose memory contains an item in *Modified exclusive* or *Nonexclusive owner* state is its owner.

A detailed knowledge of COMAs and SVM systems is not required to understand the design of the error recovery approach that we propose for DSM architectures. However, the specific characteristics of each of these architectures have a significant impact on the implementation and the efficiency of our approach. A precise description of COMA and SVM systems is delayed to Section 4.

2.2 Fault Tolerance Assumptions

Our goal is to tolerate multiple concurrent transient node failures, which do not involve loss of memory contents, and a single permanent one in DSM architectures. The unit of failure is the node: The permanent failure of a node component leads to the unavailability of the whole node. Nodes are failure-independent. It is assumed that nodes operate in a *fail-silent* fashion. They are connected by a high throughput low latency interconnection network. For SVM systems implemented on top of NOWs, high-speed networks based on technologies, such as Ethernet 100 Mb/s, ATM, or Myrinet are assumed. The network is assumed to be reliable.

2.3 Exploitation of DSM Features for Backward Error Recovery

In this section, we consider the DSM architecture depicted in Fig. 1. We propose a new approach for implementing BER in such architectures, other aspects of fault tolerance such as error detection and confinement being beyond the scope of this paper. BER is a well-known fault tolerance technique [1]. One way to implement BER consists of periodically saving a global consistent state of the system on stable storage and restoring it if a failure occurs.

For the sake of simplicity, we assume in this paper a global checkpointing approach [1] where all processors synchronize their execution for the establishment of a recovery point in order to ensure that the set of recovery data forms a global consistent snapshot of the entire system. Moreover, the implemented scheme is incremental. New recovery data is created only for data that has been modified since the last recovery point.

Several methods have been proposed in the literature for implementing a stable storage. A stable storage ensures the permanence, accessibility, and atomic update of recovery data and provides a way to identify and localize it. Stable storage can be implemented with dedicated hardware. For instance, a stable memory, implemented with two memory banks, is used in the FASST architecture [8] and MEMSY [9]. Recovery data is stored in the stable memory and localizing and identifying it is particularly easy. Such an approach is expensive in hardware development, but allows low latency accesses to the stable storage. Another solution is to exploit the organization of a memory hierarchy by storing current and recovery data at two different levels. The fault-tolerant shared memory multiprocessor SEQUOIA [10] uses the cache/memory hierarchy to store and identify recovery data. Memories contain recovery data, whereas caches contain modified data items. The permanence of recovery data is ensured by replicating it on two memory modules. The recoverable SVM systems proposed in [11] uses the memory/disk hierarchy to store recovery data. Recovery data is stored exclusively on the disks, and so is easy to localize, and is identified by its state on the disk.

An alternative to these solutions is to store recovery data in volatile memories. This solution is adopted in the recoverable SVM systems proposed in [12], [13], and [14]. The feature of our approach is to take advantage of the considered DSM architectures' properties to implement an efficient stable storage at a low cost. We thus chose this latter solution, as explained in the remainder of this section. As two nodes are failure-independent in a DSM system, according to our failure assumptions, their memories can be used to implement stable storage. The permanence of data is ensured by its replication in the memories of two distinct sites. With such an approach, the system scalability is preserved. Moreover, it ensures a fast recovery point establishment as data is transfered between low latency high throughput volatile memories through a high-speed and low latency network. This solution to implement a stable storage is more efficient than the use of disks and less expensive than dedicated hardware.

Moreover, in the DSM architectures we consider, memory items have no fixed physical location and are automatically migrated or replicated in local memories. Our approach takes advantage of these two features for the management of recovery data which is stored in the node local memories.

As node memories are failure-independent, a DSM architecture gives the opportunity to store both active and recovery data in node memories while still ensuring the recovery data permanence property. Because a memory item has no fixed location in memory, the current copy of an item can be stored in one node memory, whereas its recovery copies can be kept in the memory of any other nodes. The atomic update of recovery data can be ensured by a traditional two-phase commit protocol similar to the one used in [15]. The implementation of this protocol is simplified by the fact that DSM replication mechanisms allow the creation of as many copies of a memory item as needed. The absence of fixed physical locations also greatly simplifies the reconfiguration step necessary after a failure. Lost memory items can indeed be reallocated on any valid node of the architecture without any address modification.

Finally, as items in node memories are managed with the states of the coherence protocol, identification of recovery data can be ensured by adding new states to the coherence protocol. Localization of recovery items is dealt with as for active data.

This approach is both low cost and efficient. The hardware development is limited as no specific device is required to store 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 [16]. Moreover, recovery data is stored in the node memories. This ensures a fast access and a high throughput when a recovery point has to be established. Another advantage is that already existing copies of memory items can be used to avoid data transfers during the establishment of a new recovery point. Finally, as recovery copies are stored in the node memories, they can still be accessed by the processors as long as they are identical to the active ones 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.

3 A New Approach for Implementing Backward Error Recovery in DSM Architectures

Implementing the previous ideas in a DSM architecture can be realized by extending the coherence protocol of the architecture to transparently combine the management of active and recovery data. Two new states are added to the basic coherence protocol depicted in Fig. 2 to identify recovery data. The *Shared-CK*[1] 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 *Invalid-CK* (*Inv-CK*) 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.

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. In the presentation of the protocol, the *Shared-CK* (respectively *Inv-CK*) state is meant to represent the two *Shared-CK1* and *Shared-CK2* (respectively *Inv-CK1* and *Inv-CK2*) copies.

Two new transitions, *Establish (create/commit)* and *Recovery*, represent the establishment and restoration of a recovery point. The resulting protocol, illustrated in Fig. 3, is called the Extended Coherence Protocol (ECP) in the remainder of this paper. It ensures that, at any time after the first recovery point establishment, 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 just the situations 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, the *Shared-CK1* copy, as reflecting the ownership, is used to serve the request. The requesting node receives a copy of the item and sets its local state to *Shared*. On a write hit on a *Shared-CK* item, meaning that a node attempts to write a read-only item present in its local memory, the two *Shared-CK* copies change their state into *Inv-CK* and all possible *Shared* copies change their state into *Invalid*. Moreover, a copy is created locally from the local *Shared-CK* one in state *Modified exclusive*. A write miss on an item not modified since the last recovery point is handled in an almost identical way as a write miss in the standard protocol. Invalidation messages are sent to all nodes with a copy of the item and the node having the *Shared-CK1* copy sends a copy of the item to the requesting node. As a result, the two *Shared-CK* copies change their state into *Inv-CK* and all possible *Shared* copies change their state into *Invalid*. At the end of the transaction, the requesting node owns the only valid active copy of the item, in state *Modified exclusive*. The architecture now contains an active 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* items are only kept for a possible recovery.

3.1 Establishing a Recovery Point

A traditional two-phase commit protocol [15] is executed on each node to establish a new recovery point. As a global checkpointing approach is used, all nodes are synchronized to execute this protocol. One node is statically designed as the coordinator and is responsible for initiating the establishment of a recovery point and controlling each phase. When the coordinator is faulty, a new one is defined. The goal of the first phase, called the *create* phase, is to create the new recovery point, while the second phase, called the *commit* phase, aims at discarding the previous recovery point and confirming the new one. Before starting the *create* phase, each node first terminates all pending requests. The use of an incremental scheme ensures that two copies of new recovery data are created only for those items that have been modified since the last recovery point (those in state *Modified exclusive* or *Nonexclusive owner*) during the *create* phase. Another transient state is added, the *Precommit* state, to distinguish new recovery data from the old one during this phase. The first recovery copy is obtained by simply changing the state of *Modified exclusive* and *Nonexclusive owner* copies to *Precommit* and the second one by replicating the item in state *Precommit* to any other node's memory. For replicated *Nonexclusive owner* items (that is to say, with existing *Shared* copies), an optimization consists of choosing one of the already existing replicas to become the second recovery copy (in state *Precommit*), thus avoiding a data transfer and the creation of an additional

1. CK = Checkpointed.

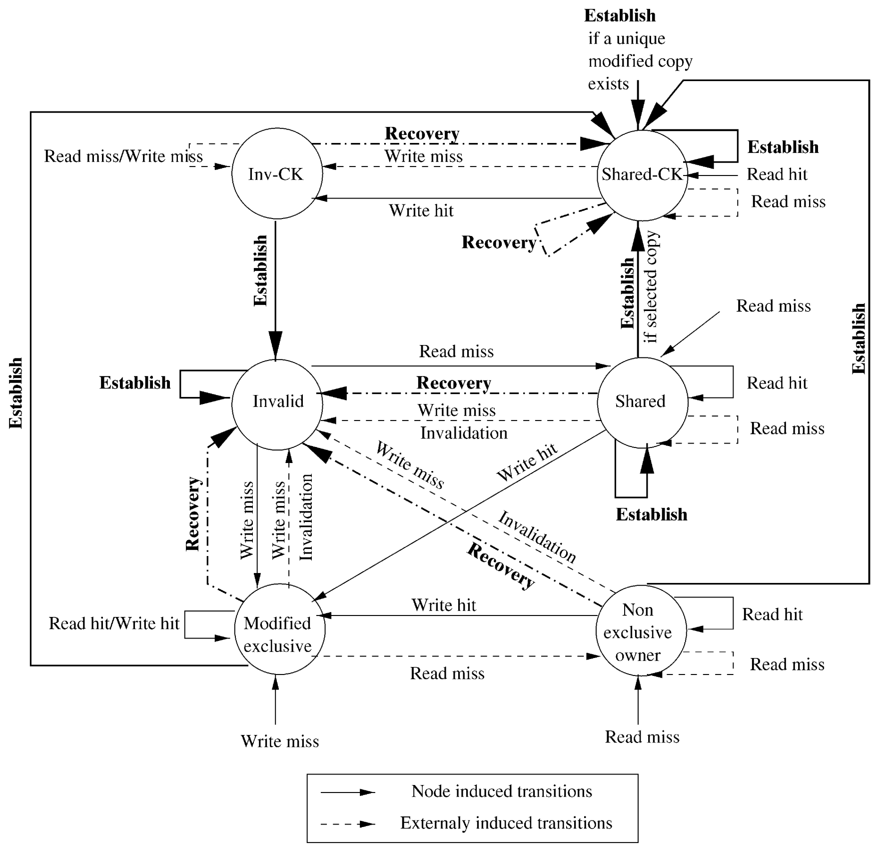

Fig. 3. Extended coherence protocol state transition diagram.

copy. Once all nodes have terminated the *create* phase, the *commit* phase, local to each node, can start. Each node scans its memory and simply sets all its *Inv-CK* items into *Invalid* and all its *Precommit* items into *Shared-CK*. After a successful recovery point establishment, every item has two recovery *Shared-CK* copies and possibly other read-only *Shared* 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 (composed 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 *Precommit* and *Shared-CK* copies) is complete and persistent since all items are already replicated. Since this phase is local, any failure occurring 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.

3.2 Restoring a Recovery Point

After a node failure has been detected, the purpose of the restoration phase is to recover to the previous recovery point consisting of all *Shared-CK* and *Inv-CK* items. All other items must be invalidated. The recovery is global. As soon as a failure is detected, a broadcast message informs the nodes that recovery must be performed. Each node scans its local memory and invalidates all active items (in state *Shared*, *Modified exclusive*, or *Nonexclusive owner*) as well as *Precommit* ones. Note that *Shared* items must be invalidated because there is no way to know whether they correspond to active or recovery items. *Inv-CK* items are restored to *Shared-CK* since they belong to the consistent global system state currently being restored. No action is required for *Shared-CK* items since they belong to both active and recovery data. 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 node failure, a memory reconfiguration must also be performed in order to again tolerate new failures immediately. This reconfiguration duplicates lost data which was located on the faulty node so that the persistence property is satisfied again. After the recovery phase, only *Shared-CK* items exist. To reconfigure the architecture, each *Shared-CK* item has to check whether its replica is still alive or not. If not, a new *Shared-CK* copy has to be created on any other remaining node.

4 Implementation Issues

In this section, we discuss issues related to the implementation of the extended coherence protocol. In order to show that our fault tolerance mechanism is a generic one for DSM architectures, we have implemented it in two different architectures: a nonhierarchical COMA and an SVM system implemented on a network of workstations.

Although a generic architecture was sufficient to describe our fault tolerance protocol, the distinction between the two considered architectures becomes essential when implementation issues are addressed. Thus, before describing the implementations, we briefly present

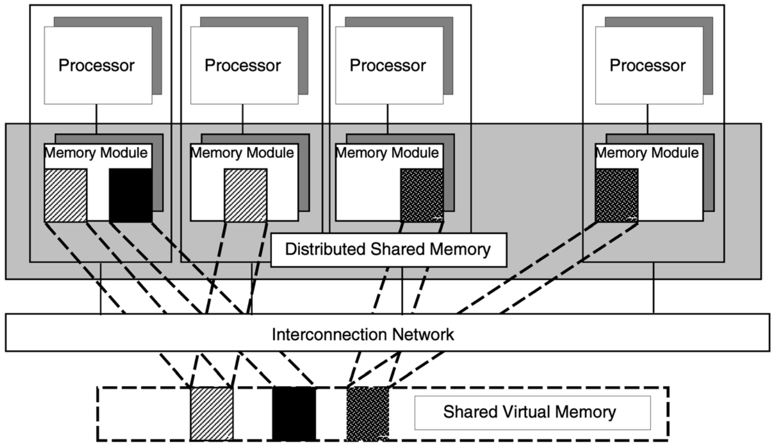

Fig. 4. Shared virtual memory.

the two architectures, in which way they differ, and the impact of their differences on our fault tolerance protocol implementation.

4.1 COMA versus SVM

Both COMA and SVM systems rely on the automatic replication or migration of data to the local memory of the node requesting it in order to exploit the locality of parallel applications and to decrease access time latencies. The main similarity between the two considered architectures is that data has no fixed physical location. Such a characteristic is essential for our fault tolerance scheme. However, COMA and SVM architectures differ in several ways we detail below.

4.1.1 Hardware versus Software Implementation

The main difference between the two considered architectures is whether the implementation of the DSM is in hardware or in software. This difference leads to a different memory management and different computing times. COMAs are organized as a set of processing nodes connected by a high speed interconnection network. The memory associated with each node is organized as a large cache called an attractive memory. COMAs are implemented in hardware by extending the basic caching mechanisms to attractive memories.

Coherence is usually maintained on a memory item (also called a line) basis by a hardware coherence protocol, typically a directory-based write-invalidate protocol [17]. The first COMA machines, such as the DDM [2] or the KSR1 [3], 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 nonhierarchical organization was first proposed in COMA-F [18] and more recent COMA proposals [19] are based on general interconnection networks. From a fault tolerance point of view, a nonhierarchical organization is preferable as the loss of an intermediate node in a hierarchy could cause the loss of the whole underlying subsystem, resulting in multiple failures. COMA-F uses a flat directory organization in which the directory entries are statically distributed among the nodes. The coherence protocol is close to the generic one depicted in Fig. 2.

In contrast, SVM (Fig. 4) systems are implemented in software and rely on the virtual memory of the operating system. SVM systems were introduced with IVY [20]. Since then, several prototypes have been developed both on top of multicomputers, such as Koan [21], implemented on the IPSC/2 and on top of NOWs, such as TreadMarks [22]. In contrast to the shared address space implemented in dedicated hardware in a COMA, the shared memory in an SVM exists only *virtually*, is implemented in software, and relies on the standard address translation hardware (Memory Management Unit). A software unit implemented in the operating system of a node maps parts of the local address space onto the global shared address space. As interconnection networks of NOWs are not usually designed to implement snooping, the coherence protocols in SVMs are also usually directory-based. In order not to limit the scalability of the system, distributed directories are preferred over centralized ones. The most frequent implementation of write-invalidate coherence protocol in an SVM is a statically distributed directory scheme, similar to the one presented in Fig. 2.

This different implementation between COMAs and SVM systems has a large impact on the time needed to solve misses. Software implementations have the advantage over COMAs of using standard components, but COMAs are much more efficient to transfer data from one node to another and to implement coherence operations.

4.1.2 Memory Management

The second main difference concerns the memory management which intrinsically depends on the hardware/software implementation.

First, the granularity of sharing is different in the two considered architectures. As COMAs are implemented in hardware, they are managed at a granularity of a memory line (hundreds of bytes). For instance, the size of a memory line in the KSR1 is 128 bytes. It is larger than a cache line in order to limit the amount of coherence information to keep. In contrast, SVMs are implemented in software on top of the operating system. Thus, the most frequent unit of sharing is the page, whose size typically ranges from 1 to 8 KBytes to fit in with the operating system memory management. Such a large size is bound to induce some *false sharing*, which arises when several processors alternatively write in the same page at different addresses. Conversely, spatial locality may be better exploited in SVM systems than in COMAs.

Because COMAs are implemented in hardware and rely on a caching mechanism, memory is managed at two levels. The operating system is responsible for allocating pages in memory and the hardware distributed shared memory is responsible for replicating or migrating items. Consequently, replacements of items must be handled with care in a COMA: No physical memory backs up the attractive memories. The hardware replacement strategy of an attractive memory must make sure that there is always at least one valid copy of every item in the system. In the COMA-F [18], 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 attractive memory generate an *injection* which ensures that the item copy is first transferred to another node's attractive memory before being replaced. We describe the injection mechanism in detail in the following section. On the contrary, in an SVM, the distributed shared memory and the allocation of pages are both implemented at the operating system level. Such a system does not require a similar injection mechanism.

The last difference between the two architectures is the associativity degree of their memory. This has a large impact on the implementation of the fault tolerance protocol. Attractive memories in COMAs are generally *n-way associative*. Such an organization speeds up the search for a memory line which is faster in a set of the memory than in the whole memory. This also means that an item must be stored in a dedicated set statically defined based on its address, which usually has a small size (typically four or eight items). Such an implementation does not support the presence of two copies, with different values but identical addresses, of an item in the same memory because it would be in the same set. This situation may arise when a write is requested on a *Shared-CK* item. Then, the local *Shared-CK* copy is transformed into an *Inv-CK* copy and a *Modified exclusive* copy is created. In a COMA, these two copies of a same item must not cohabit in an attractive memory. To avoid this situation, which leads to a set of the memory monopolized by two copies of a same item, one being a recovery copy and thus useless during failure-free execution, we take the benefit of the injection mechanism already implemented in a COMA to inject the recovery copy when a new copy of the item is created in a memory where a former copy of the item exists. As memories are fully associative in an SVM, two copies of an item, a recovery one and the corresponding active one, are allowed to be stored in the same memory. We study the impact of the associativity during the performance evaluation. Moreover, this implementation avoids miss conflicts which may arise in a COMA.

4.2 COMA Implementation

As implementing the protocol within a COMA architecture requires slight hardware modifications, we simulated a COMA architecture. Nevertheless, we address implementation issues and describe how our fault tolerance mechanism may be implemented within a real architecture.

TABLE 1
New Injections Causes Introduced by the ECP

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

We consider an architecture similar to the COMA-F [18]. Each computing node contains a processor, a cache, an attractive memory, and a network interface. The interconnection network connecting the nodes is a two-dimensional mesh. The coherence protocol is similar to the one used in the COMA-F.

On a node, all the hardware modifications introduced by the ECP are related to the implementation of the attractive memory and its associated controller implementing the coherence protocol. The processor and its cache do not need to be modified. Encoding the new states related to recovery data requires additional bits per item. The coherence protocol must be augmented to deal with read and writes accesses to *Shared-CK* copies and with establishment and restoration of a recovery point. The extended coherence protocol must allow read accesses to *Shared-CK* copies. Read and write requests on a *Shared-CK1* copy are handled in a similar way as read and write requests on a *Nonexclusive owner* copy. The only difference is that, in the write request, the *Shared-CK1* copy changes its state into *Invalid-CK1* and an invalidation is also sent to the *Shared-CK2* copy, which changes its state into *Invalid-CK2*.

In a standard COMA, injections are used only when a copy in state *Modified exclusive* or *Nonexclusive owner* has to be replaced in the cache by a more recently accessed item when there is not enough space to store the referenced item. Our extended coherence protocol introduces five new cases of potential injections (see Table 1). Two of them occur when a copy in state *Shared-CK* or *Inv-CK* must be replaced from an attractive memory. As these states represent recovery data, copy in state *Shared-CK* or *Inv-CK* must not be removed from the memory. Thus, these copies are treated in the extended coherence protocol in a similar way as *Modified exclusive* or *Nonexclusive owner* copies in the standard coherence protocol. The others may occur when a node wants to access an item already laying in a recovery state in its local memory. In such cases, we choose to trigger off injections in order to avoid two instances (recovery and active) of an item in the same set of an attractive memory. Injections of recovery data must be handled with care since they could result in the loss of a recovery item copy. To accept an injection, an attractive memory can only replace one of its *Invalid* or *Shared* lines.[2] If the injection cannot be accepted, the node forwards the injection to another node. Injections are accomplished in two steps. In a first step, an

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

injection message is sent to find a victim line on a remote node. When the remote 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 caused by fault tolerance 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 attractive memories. In the KSR1, this problem is solved by allocating an irreplaceable page for each page allocated in the architecture [3]. 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.

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.

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

4.3 SVM Implementation

We have implemented a recoverable SVM [23], based on the ECP on the ASTROLAB platform. This platform consists of a set of Pentium PC (133 MHz) running CHORUS microkernel [24], interconnected through an ATM (155Mb) local area network. Our recoverable SVM is implemented as an extension of a basic SVM implementing Li and Hudak's statically distributed coherence protocol [20] apart from the following difference. In our recoverable SVM, distributed directories only contain the identity of the owner of each page whereas, in Li and Hudak's scheme, they contain all the coherence information about pages. In our recoverable SVM, coherence information (state and copysetlist) is left to the owner of a page. This feature simplifies the establishment of a recovery point. As the copysetlist is stored on the owner node, no access to the manager is required to exploit an already existing copy at checkpointing time. The granularity is a CHORUS page of 4 KBytes. As the coherence protocol is implemented in software in an SVM, the introduction of new states is very easy and does not imply any hardware modification.

Our recoverable SVM is implemented as a CHORUS actor on each node and consists of five threads:

- The *user thread* runs the application;
- The *mapper thread* handles page faults generated by the CHORUS microkernel and is in charge of solving them;
- The *communication thread* is responsible for managing communication with other nodes;
- The *fault tolerance thread* is responsible for periodically establishing a recovery point;
- The *rollback thread* is responsible for rolling processes back to the last recovery point if a failure is detected.

Nodes are numbered and node 0 is chosen to initiate the global recovery point establishment algorithm. As nodes are synchronized to establish a recovery point, all user threads are suspended at checkpoint time. The approach is to wait for the completion of pending transactions before proceeding to the establishment of the recovery point.

5 Performance Evaluation

Our extended coherence protocol has been evaluated for the COMA and SVM architectures described in the previous section. Performance results have been obtained by simulation for COMA and by measurements performed on the recoverable SVM described above. The reader will not find in this section a detailed performance study nor a deep analysis of the protocol behavior in a particular architecture. These studies are reported in [25], [37], [27]. In this paper, we mostly want to demonstrate that the ECP can be implemented at a low cost and is efficient and scalable in both COMA and SVM architectures. We first present the context of evaluation for the two considered architectures. Results are analyzed in Section 5.2.

5.1 Context of Performance Evaluation

5.1.1 *Simulation Environment for Evaluating the ECP in a COMA*

Our simulator uses the SPAM simulation kernel [28] that allows an efficient implementation of an execution-driven simulation method [29]. The memory references are generated by parallel applications instrumented with a modified version of the Abstract Execution technique [30]. 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 [31].

Most of the physical characteristics of the simulated architecture (except network characteristics) come from the KSR1 architecture (see [3] for more details). 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 implementation of the *create* phase assumes that the time between two line injections is sufficient for a memory to identify the next item to be replicated.

In this evaluation, we use four parallel applications from the SPLASH benchmark suite [32] representing a variety of shared memory access patterns. Table 2 describes their characteristics. For all these benchmarks, statistics are collected during the parallel phase. As the size of the attractive memory is large compared to the size of the applications, no capacity replacement occurs during the simulations.

As the recovery point establishment frequency is mainly influenced by the number of operations coming from or

TABLE 2
Simulated Applications Characteristics

Applications	**Parameters**	**Instructions** (millions)	**Reads** (millions)	**Writes** (millions)	**Shared Reads**	**Shared Writes**
Barnes-Hut	1536 bodies 11 iterations	190	49.5	28.7	11.1	0.27
Cholesky	bcsstk14	53.1	17.6	4.7	14.2	2.5
Mp3d	50 K molecules 8 steps	48.3	10.6	6.3	8.6	5.4
Water	120/144 molecules 2 iterations	78.6	26.8	7.8	4.9	0.58

TABLE 3
Memory Operations in Standard Execution

Application	**Number of nodes**	**Read Miss**	**Write Miss**	**Invalidations**	**Writes on read-only items**
Matmul 256	2	192	384	0	0
	4	480	384	0	0
Matmul 512	2	768	1536	0	0
	4	1920	1536	0	0
MGS	2	512	512	0	0
	4	1536	512	0	0
Radix	2	18	15176	15140	4
	4	43	18859	18827	6

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 compared to other evaluations [15]. In the absence of real recovery point frequencies, they give, however, the performance degradation for different computing environments and allow us to evaluate our scheme at its limits.

5.1.2 Measurement Environment for Evaluating the ECP in an SVM Architecture

In this experiment, we used four PCs[3] with three applications, different from those used for COMA simulations: Matmul, MGS, and Radix. This choice ensures a wide range of behaviors from a memory point of view. Matmul is a matrix multiplication algorithm; the two source matrices are produced by a node and then read by all the others. Each node is then responsible for writing a part of the result matrix. We use two working set sizes, 256*256 and 512*512, double float elements per matrix. The Modified Gram-Schmidt (MGS) algorithm produces from a set of vectors an orthonormal basis of the space generated by these vectors. We consider a base of 512 vectors of 512 double float elements. Radix is a SPLASH-2 [33] application. The integer radix sort kernel is iterative and performs one iteration for each radix r digit of the keys.

Table 3 depicts the memory operations generated by the considered applications during standard execution, without any fault tolerance. The chosen applications exhibit very different behaviors from memory point of view. This feature is particularly interesting for the validation of the ECP. Indeed, Matmul shares pages through read operations, whereas Radix generates a lot of false sharing on a small working set.

CHORUS 3.5 has been used for the implementation of the recoverable SVM. This release suffers from a major drawback: The kernel manages only 8 MBytes out of the 32 MBytes available on each node. This feature explains the small size of the applications.

Frequency of checkpointing in SVM systems ranges from one recovery point per second to one recovery point every 150 seconds. The checkpoint frequency in SVM systems is much lower than in COMAs since communications are much more expensive and, consequently, the time in the two kinds of architecture cannot be considered on the same scale. We observe a difference of checkpoint frequencies between applications executed in a SVM system as well. This is due to the fact that the size and execution time change considerably from one application to another.

5.2 Results

Two types of overhead are introduced by the ECP in both architectures, resulting in longer[4] execution times than with the standard architecture: a *performance degradation* due to the establishment of recovery points and an *impact on the memory behavior* due to changes in the state of memory items. These two overheads give insight into the protocol efficiency and cost. To identify the time overhead, we compare results obtained by the architecture using the standard coherence protocol and the same architecture using our extended coherence protocol.

3. The relatively small configuration size is due to the fact that we had only four workstations equipped with an ATM interface board in the Astrolab platform.

4. Lower execution times are observed in some cases for the recoverable SVM.

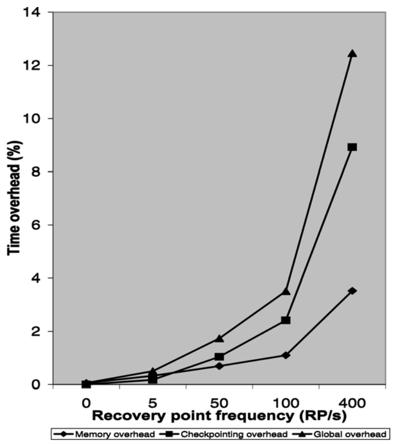

Fig. 5. Overhead for Water in a fault-tolerant COMA.

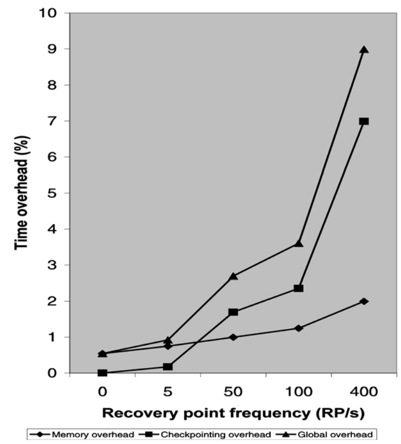

Fig. 7. Overhead for Barnes in a fault-tolerant COMA.

5.2.1 Performance Degradation

The time overhead can be divided into two separate effects: 1) The time required to establish new recovery points and 2) the change of memory behavior caused by the introduction of recovery data in volatile memory.

The execution time with the ECP can then be expressed as the sum of three components,

$$T_{ECP} = T_{standard} + T_{checkpoint} + T_{memory},$$

where $T_{standard}$ is the execution time on the standard architecture. $T_{checkpoint}$ is the time required to establish recovery points. It is composed of the time needed to synchronize nodes during the two-phase commit algorithm and the time required to replicate items between nodes. T_{memory} is related to the impact of the extended coherence protocol on the memory behavior. $T_{checkpoint}$ is measured during simulations or experiments, whereas T_{memory} is deduced from $T_{standard}$ and $T_{checkpoint}$. Consequently, it is possible to get negative memory overhead (T_{memory}).

Performance degradation in COMAs. In a COMA, the memory overhead is caused by an increase of the number of misses and injections. Figs. 5, 6, 7, and 8 depict the time overhead compared to a standard architecture for various applications and recovery point establishment frequencies. Depending on the application and recovery point frequencies, this overhead varies from 1 or 2 percent in the best case to 15 percent in the worst case (Mp3d with 400 recovery points per second). $T_{checkpoint}$ principally depends on the size of the recovery data that must be transferred which is itself directly influenced by the recovery point frequency. At low recovery point frequencies, $T_{checkpoint}$ 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 five recovery points per second. The total amount of data handled during recovery point establishments then decreases from 10 to 1.2 MBytes.

$T_{checkpoint}$ is also influenced by the applications characteristics. As an example, Mp3d which exhibits a high write rate (10 percent), shows 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 $T_{checkpoint}$ 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 larger than that of Barnes). Finally, locality of memory accesses also influences the amount of recovery data treated.

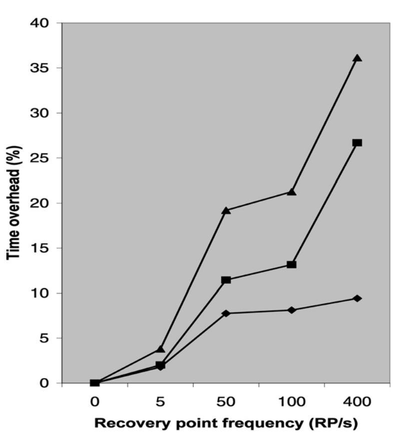

Fig. 6. Overhead for Mp3d in a fault-tolerant COMA.

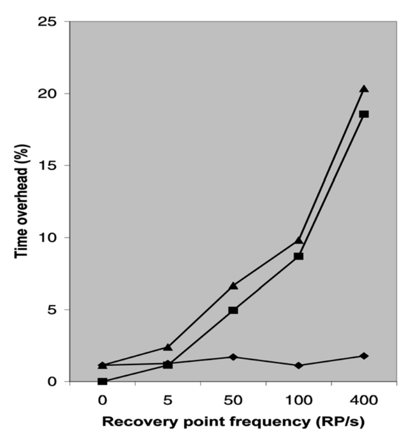

Fig. 8. Overhead for Cholesky in a fault-tolerant COMA.

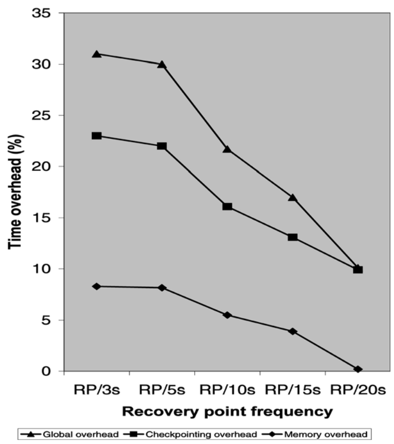

Fig. 9. Overhead for Matmul 256*256 on two nodes in our recoverable DSM.

The low $T_{checkpoint}$ 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. 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 five recovery points per second, 52 percent of the items which have to be replicated during the create phase are already replicated. The replication throughput increases to 30 MB/s per node.

The processor and network are low-performance compared to current multiprocessor architectures. Gefflaut [34] reports results obtained with a 100 MHz processor and a network similar to the one used in Flash [35] and shows that the performance degradation decreases for all applications.

Performance degradation in SVM systems. Figs. 9, 10, 11, 12, 13, 14, 15, and 16 depict the time overhead in several configurations for the three considered applications. RP/ys means one recovery point established every y second. This difference of recovery point frequency between COMA and SVM is appropriate according to the hardware versus software implementations.

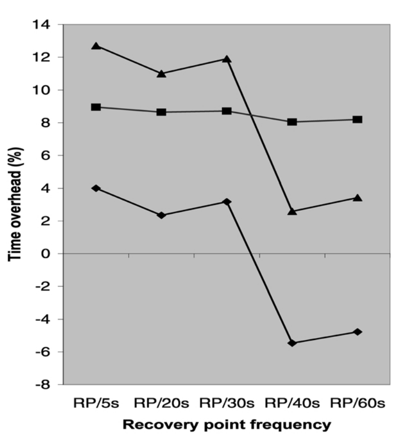

Fig. 10. Overhead for Matmul 512*512 on two nodes in our recoverable DSM.

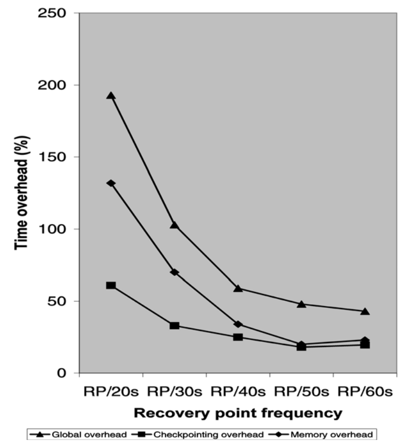

Fig. 11. Overhead for MGS on two nodes in our recoverable DSM.

For Matmul, the largest overhead is the time needed to synchronize nodes. Moreover, most pages are already replicated after the first recovery point establishment (at least the two source matrices). Thus, the global overhead decreases with the recovery point frequency. For MGS (Figs. 11 and 15) the time overhead becomes significantly higher for higher recovery point frequency because of the important memory overhead generated by the changes of memory state. It is significantly better in the four-nodes configuration than in the two-nodes one. Checkpointing overhead decreases with the recovery point frequency. The evolution of the replication overhead can be explained by the fact that a fixed number of pages are modified between two recovery points, whatever the recovery point frequency.

However, results are particularly favorable for the two-nodes configuration of Radix execution as the performance gain for our recoverable SVM reaches 35 percent over the basic SVM. This is due to the exploitation, during failure-free executions, of pages replicated for the needs of fault tolerance. This phenomenon is also observable on four-nodes configuration with Matmul 256*256 and 512*512. In the four-nodes configuration Radix execution, the time overhead is not negative because the replication, needed for fault tolerance, cannot be exploited for normal execution as it is in the two-nodes one. Indeed, when pages are

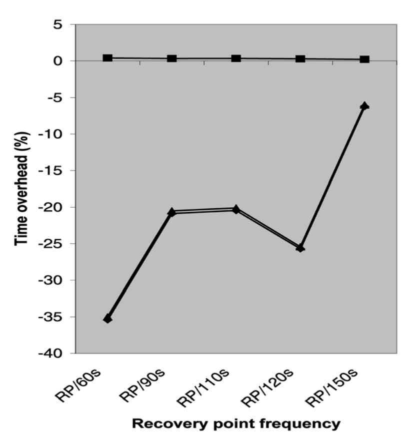

Fig. 12. Overhead for RADIX on two nodes in our recoverable DSM.

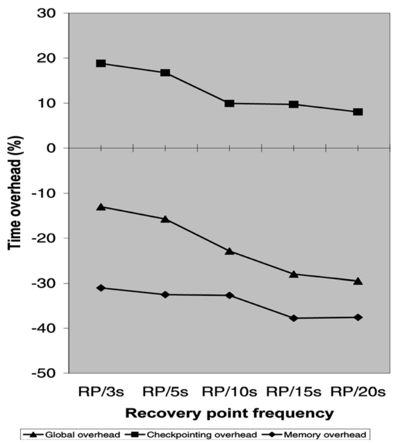

Fig. 13. Overhead for Matmul 256*256 on four nodes in our recoverable DSM.

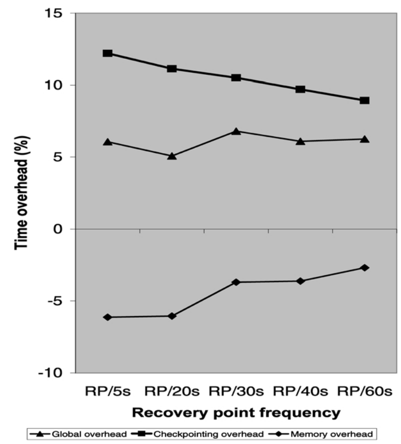

Fig. 14. Overhead for Matmul 512*512 on four nodes in our recoverable DSM.

replicated at checkpoint time in the two-nodes configuration, it is equivalent to a broadcast of the whole working set. Nevertheless, the performance degradation does not exceed 1 percent because, first, the working set is reduced and, second, a small number of pages are modified between two recovery points.

Except for Radix, which benefits from the fault tolerance replication, a lower checkpointing frequency implies significantly lower time overhead.

5.2.2 Impact on Memory Behavior

Impact on memory behavior in COMAs. The T_{memory} overhead is induced by the ECP, which keeps the item recovery copies in the memories. Storing recovery data in the attractive memories of a COMA has two effects: 1) a variation of cache and attractive memory misses, and 2) a creation of new item injections.[5] Whatever the recovery point establishment rate, this memory overhead appears to be quite limited and ranges from approximately 10 percent in the worst case to less than 2 percent. This limited memory overhead is mainly the consequence of a limited increase in the number of misses.

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.

However, the slight write miss rate increase of the attractive memories does not explain the memory overhead. This effect is, in fact, due mainly to new items injections (see Table 1). Fig. 17 shows the variation in the number of injections per attractive memory for 10,000 memory references for various recovery point frequencies. The total number of injections is low, at most 20 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 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 copies are frequently transformed into recovery ones. 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 percent of the total number of injections on write accesses for a node. These injections are the principal cause of the memory overhead in a COMA.

Impact on memory behavior in SVM systems. The establishment of a recovery point is likely to introduce three particular situations: *write on Shared-CK1*, *read on Shared-CK2*, and *write on Shared-CK2*. As in an SVM system, solving a page fault is much more expensive than in COMA, messages and, potentially, a page must transit over the network, the impact of memory behavior is important to consider.

- *Write on Shared-CK1*: This first situation has a negative effect and reduces performance. It is related to write hits to read-only pages. Indeed, pages which were in state *Modified-exclusive*, therefore writable, before the establishment of a recovery point are changed to *Shared-CK1* and become read-only. If such pages are referenced again, which seems reasonable due to the temporal locality of references, this situation involves additional write hits to read-only pages that would not have arisen in standard

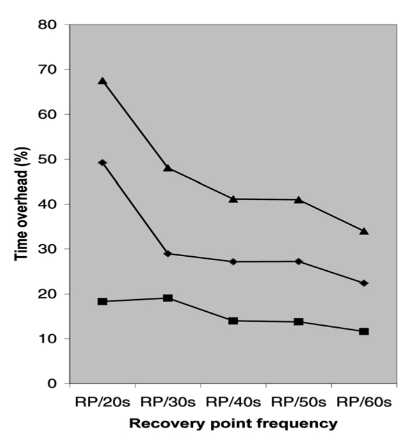

Fig. 15. Overhead for MGS on four nodes in our recoverable DSM.

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

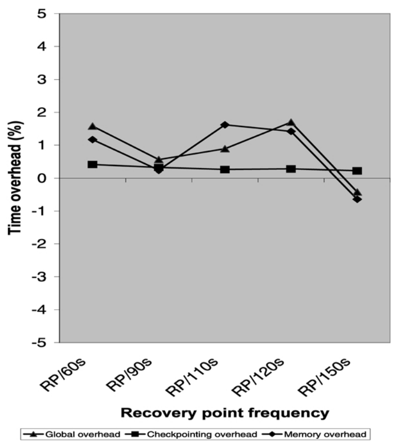

Fig. 16. Overhead for RADIX on four nodes in our recoverable DSM.

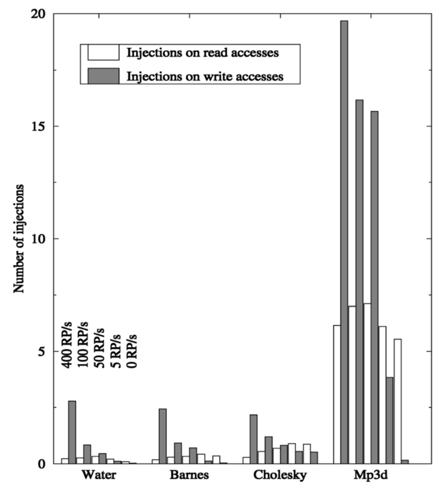

Fig. 17. Average number of injections per node (on read or write accesses) for 10,000 memory references when varying the recovery point frequency.

execution and are exclusively due to the recovery point establishment.

- *Read on Shared-CK2*: This situation has a positive effect on the performance because it anticipates page faults. It arises when pages in state *Shared-CK2*, replicated for the fault tolerance needs, are later read during failure-free execution. A replication of a *Shared-CK2* copy in the framework of the global checkpoint is less expensive than a basic read miss. This is due to the fact that a replication during a global checkpoint is less expensive than solving a read page fault since many replications are done simultaneously at that time. Furthermore, the systematic way of replication at checkpointing time avoids message exchanges between manager, owner, and so on needed to resolve a page fault.
- *Write on Shared-CK2*: This situation has a positive effect on the performance because it also anticipates page faults. Writes on *Shared-CK2* pages are treated like a hit, but require additional message exchanges, first, to transfer the ownership from the node holding the *Shared-CK1* copy to the requesting node and, second, to invalidate the two *Shared-CK* copies and the potential *Shared* pages. Moreover, a local copy in state *Nonexclusive owner* has to be created. Nevertheless, because a write on a *Shared-CK2* page does not require a page transfer over the network, this operation is less expensive than a traditional page fault. Such situations arise when pages *Shared-CK2*, replicated for the fault tolerance needs, are then requested through write operation during failure-free execution.

Tables 4 and 5 show that for Matmul two of these situations arise, one favorable (read on *Shared-CK2*) and the other not (write on *Shared-CK1*). In the two-nodes configuration, the number of writes on *Shared-CK1* increases with the recovery point frequency, whereas the number of read on *Shared-CK2* stagnates. Thus, the time overhead increases also with the recovery point frequency. However, in the four-nodes configuration, we observe a negative time overhead. This is due to the impact of the number of reads on *Shared-CK2*, which is much more important in a larger working set and which compensates for the negative effect of writes on *Shared-CK1*.

For MGS, we can now explain the high overheads depicted in Figs. 11 and 15 which are caused by the large number of writes on *Shared-CK1* (see Table 6). Such situations would not have arisen in standard execution without fault tolerance. Each time a page is written before and after a recovery point, the second write leads to the invalidation of the *Shared-CK2* page. So, the whole memory overhead is due to the occurrence of such operations and its impact decreases with the recovery point frequency. A solution to this problem is described in [23]. These results also explain the difference between the two and the four-nodes configurations since fewer writes on *Shared-CK1* pages occur in the latter configuration, thus decreasing the overhead compared to the former one.

For the two-nodes Radix execution, we observed a performance gain of more than 35 percent. This favorable situation arises because a large number of writes on *Shared-CK2* are reached out to the expense of writes misses (Table 7). As the former operation is less expensive than the latter, this situation improves the execution time despite the synchronization and replication times.

5.2.3 Scalability

DSM architectures are scalable, that is to say that their power increases with the number of nodes. It is important that our approach to fault tolerance preserves the scalability property of DSM architectures. In this paper, we only present scalability results for COMAs. In fact, the current implementation of the recoverable SVM on a four node configuration does not allow us to evaluate the system scalability. However, in a previous study [36], we have implemented the ECP protocol on a 56 nodes Intel Paragon machine running Mach microkernel. In this framework, experiments attest to the scalability of our approach in SVM systems.

The scalability of a COMA implementing the ECP protocol has been evaluated by varying the number of nodes from nine to 56 and measuring performance

TABLE 4
Memory Overhead for Matmul 256*256

Frequency	RP/3s	RP/5s	RP/10	RP/15s	RP/20s
	Two Nodes				
Write on SHARED-CK1	534	529	377	239	149
Read on SHARED-CK2	63	63	63	63	63
	Four Nodes				
Write on SHARED-CK1	489	354	146	63	32
Read on SHARED-CK2	120	146	136	136	135

TABLE 5
Memory Overhead for Matmul 512*512

Frequency	RP/20s	RP/30s	RP/40	RP/50s	RP/60s
	Two Nodes				
Write on SHARED-CK1	2081	2074	2067	2060	2060
Read on SHARED-CK2	455	438	407	369	337
	Four Nodes				
Write on SHARED-CK1	2207	2089	2071	1962	1308
Read on SHARED-CK2	640	564	330	288	242

TABLE 6
Memory Overhead of MGS

	Two Nodes				
Frequency	RP/10s	RP/15s	RP/20	RP/30s	RP/40s
Write on SHARED-CK1	8287	2634	1332	752	588
	Four Nodes				
Frequency	RP/20s	RP/30s	RP/40	RP/50s	RP/60s
Write on SHARED-CK1	1640	1044	828	708	691

TABLE 7
Memory Overhead of Radix

Frequency	RP/60s	RP/90s	RP/110	RP/120s	RP/150s
	Two Nodes				
Read Miss	22	21	21	19	25
Write Miss	9661	11389	11225	12005	14148
Write on SHARED-CK1	92	74	74	56	56
Write on SHARED-CK2	5515	3787	3987	3174	998
	Four Nodes				
Read Miss	50	46	53	48	53
Write Miss	19015	18817	18922	19083	18695
Write on SHARED-CK1	162	121	102	102	82
Write on SHARED-CK2	25	20	14	24	17

degradation and memory overhead with the recovery point frequency fixed to 100 recovery points per second. Fig. 18 depicts the performance degradation 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 KBytes with 30 processors to 6.8 KBytes with 56 processors. The second reason is a nearly linear increase of the recovery data replication throughput (see Fig. 19). For Cholesky, the aggregate replication throughput grows from 211 MB/s with nine processors to 1.1 GBytes for 56. Fig. 20 presents the memory overhead, T_{memory}, for different numbers of processors. This effect remains the same or decreases with the number of processors.

5.3 Comparison

To evaluate the effectiveness of the ECP, we have studied its impact on failure-free executions in two different architectural frameworks: by simulating a COMA architecture and by implementing an SVM on a NOW. In both architectures, the already existing replicas are exploited to avoid the data transfers at recovery point establishment, thus limiting the cost of the fault tolerance mechanism. However, the results

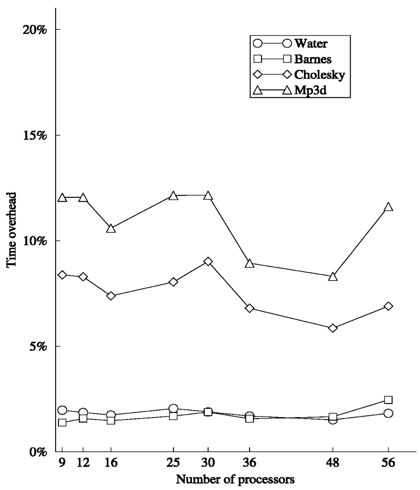

Fig. 18. Performance degradation when varying the number of processors (100 recovery points per second).

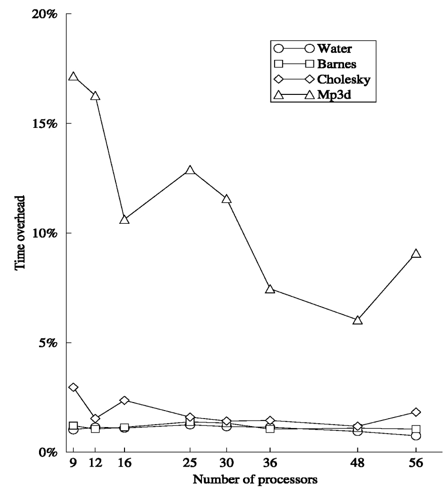

Fig. 20. Memory overhead when varying the number of processors in a fault-tolerant COMA (100 recovery points per second).

we have obtained are quite different in the two architectures in terms of time overhead. Moreover, the recovery point frequencies are very different between COMA and SVM because of the different execution times of applications and characteristics of the architectures.

Indeed, time overhead in COMAs does not exceed 35 percent, but is always above 1 percent according to the set of experiment described here. In contrast, in the recoverable SVM, the time overhead reaches 70 percent with MGS in the worst case but ranges down to -35 percent in the best case. This wide variation is due to the cost of solving a page fault in an SVM system compared to a miss resolution in a COMA. In a COMA, the coherence protocol, implemented in hardware, is faster than the protocol of an SVM architecture based on a message-passing system. So, SVM systems suffer much more than COMAs from the additional write hits to read-only copies induced by the ECP, whereas the impact of anticipated misses is larger in an SVM than in a COMA. In this particular case, the implementation difference between these two kinds of DSM becomes important because the time overhead is

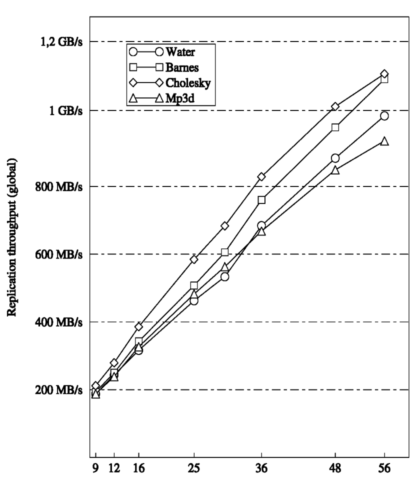

Fig. 19. Recovery data throughput when varying the number of processors in a fault-tolerant COMA (100 recovery points per second).

widely conditioned by the memory overhead. Moreover, different network characteristics (throughput) in the two architectures impact the creation phase overhead.

The difference in memory associativity between COMAs and SVM systems also has an important influence on the memory overhead. The set associativity of COMA systems leads to the need for additional injections since an active copy and a recovery one related to the same page are not allowed to cohabit in the same node memory. The full associativity, combined with its software implementation, simplifies the memory management in an SVM system using our extended coherence protocol.

Our extended coherence protocol does not have exactly the same impact on COMAs and SVM systems in terms of memory costs. As mentioned before, in COMAs, a static allocation of item must be done to be sure that there will always be enough space in attractive memory to store four copies of the same item (two *Precommit* and two *Inv-CK* copies of an item may be required at some point). Despite this static allocation, the overhead is not that high since we noticed that most of the statically allocated items would have been replicated anyway in a non-fault-tolerant DSM due to the item sharing. However, a static allocation is not necessary in an SVM system since the coherence protocol is tightly coupled with the operating system memory management. Thus, the traditional disk swapping mechanism can be used. In [37], we proposed an extended swapping mechanism which deals with the recovery states of the extended coherence protocol to choose the most relevant pages to be replaced.

6 Related Work and Conclusions

The main contribution of this paper is a new generic approach providing highly available DSM architectures. Our scheme allows us to tolerate transient as well as single permanent node failures by using a backward error recovery mechanism. As this approach is based on the extension of the coherence protocol, it represents a generic approach that can be applied to both hardware and software implementations of a DSM. Although most fault-tolerant hardware DSMs rely on dedicated hardware, many software recoverable DSMs have been proposed. Most of them rely on the use of disks to implement stable storage [38], whereas we take the benefit of both the high throughput of local area networks and the low access times of memory to avoid high access latency of disks. Nevertheless, the recoverable DSM proposed in [14] also relies on the storage of recovery data in volatile memory. However, in this scheme, recovery data cannot be used for computation during failure-free execution as our extended coherence protocol allows it in order to increase the whole efficiency. The DSM proposed in [39] also takes the benefit of the DSM mechanism, particularly on the entry coherence protocol, to propose a message logging scheme. But, likewise, recovery data is not exploited to increase efficiency during failure-free executions. This algorithm has also been applied to TreadMarks [40].

Our recoverable DSM scheme combines advantages of all these approaches: Stable storage is implemented in volatile memories, DSM features are exploited to limit the fault tolerance cost while allowing recovery data to be used for computation. Moreover, it can be used both in hardware and software DSMs.

Our extended coherence protocol has been evaluated by simulation in a COMA. This study shows that a COMA is a sound architectural basis to build a fault-tolerant scalable shared memory multiprocessor. The extended coherence protocol has also been implemented in an SVM system and integrated with a process checkpointing mechanism on top of a network of workstations running Chorus microkernel [41]. This is an interesting result of our work as not that many real implementations of recoverable shared virtual memory systems exist. Our recoverable SVM prototype also implements various optimizations that are not described in this paper. These optimizations aim at increasing the efficiency behavior of the recoverable SVM by judiciously controlling recovery data replication to decrease the number of page faults both in normal functioning and after a permanent failure [37]. Moreover, a dedicated disk paging mechanism has been designed and experimented in order to overcome the problem of memory overhead while not losing the advantage of creating recovery data in local memories when a recovery point is established.

Performance results show that the extended coherence protocol is cheap, efficient, and scalable in the two studied architectures. This solution is cheap since the hardware[6] and the memory overheads are low. Only limited hardware modifications are required to already proposed COMAs. It is also efficient since the performance degradation remains low even for relatively high recovery point frequencies. Nevertheless, the performance degradation is very application-dependent. The presence of recovery data in standard memories slightly disturbs the normal behavior of the basic coherence protocol. However, overheads depend on applications characteristics and on the frequency of recovery point establishment operations. This low performance degradation is mainly due to the use of memories to store recovery data and to a low memory overhead. We even observed in the recoverable SVM system a positive effect due to the management of recovery data. 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 [8]. Recovery point establishment performed in parallel with the execution of the application [15] could hide the time required by this operation.

Our approach is not limited to nonhierarchical COMAs and SVM architectures on top of workstations. We also had previously implemented the ECP on an Intel paragon multicomputer [36]. The extended coherence protocol can also be implemented in COMA with snooping coherence protocols [25]. It is also particularly well-suited to new architectures such as FLASH [35], where the coherence protocol is implemented in software. Furthermore, the ECP can also be considered in a CC-NUMA which implements a dynamic page placement strategy [42], [43].

6. There is no hardware overhead in an SVM implementation.

Acknowledgments

We would like to thank Pete Lee for his suggestions on improving this paper. We are also grateful to the anonymous reviewers for their valuable and helpful comments on the organization of this paper. The work presented in this paper was conducted while all authors were with IRISA, Rennes, France. This work was supported by DRET (research contract number 93.34.124.00.470.75.01).

References

[1] P.A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice,* second revised ed. Springer Verlag, 1990.

[2] E. Hagerstern, A. Landin, and S. Haridi, "Ddm—A Cache Only Memory Architecture," *Computer,* vol. 25, no. 9, pp. 44-54, Sept. 1992.

[3] J. Franck, H. Burkhard III, and J. Rothnie, "The KSR1: Bridging the Gap between Shared Memory and MMPs," *Proc. Compcon93 38th IEEE CS Int'l Conf.,* pp. 285-294, Feb. 1993.

[4] Intel Corporation, *Paragon User's Guide,* 1993.

[5] T.E. Anderson, D.E. Culler, and D.A. Patterson, "The Berkeley Networks of Workstations (NOW) Project," *Proc. COMP Spring,* pp. 322-326, 1995.

[6] C.P. Thacker, L.C. Stewart, and E.H. Satterhwaite, "Firefly: A Multiprocessor Workstation," *IEEE Trans. Computers,* vol. 37, no. 8, pp. 909-920, Aug. 1988.

[7] R.H. Katz, S.J. Eggers, D.A. Wood, C.L. Perkins, and R.G. Sheldon, "Implementing a Cache Consistency Protocol," *Proc. 12th Ann. Int'l Symp. Computer Architecture,* 1985.

[8] M. Banâtre, A. Gefflaut, P. Joubert, C. Morin, and P.A. Lee, "An Architecture for Tolerating Processor Failures in Shared-Memory Multiprocessors," *IEEE Trans. Computers,* vol. 45, no. 10, Oct. 1996.

[9] M.D. Cin, A. Grygier, H. Hessenauer, U. Hildebrand, J. Höonig, W. Hohl, E. Michel, and A. Pataricza, "Fault Tolerance in Distributed Shared Memory Multiprocessors," *Parallel Computer Architectures,* pp. 31-48, Springer-Verlag, 1994.

[10] P.A. Bernstein, "Sequoia: A Fault Tolerant Tightly Coupled Multiprocessor for Transaction Processing," *Computer,* vol. 21, no. 2, pp. 37-45, Feb. 1988.

[11] K.L. Wu and W.K. Fuchs, "Recoverable Distributed Shared Virtual Memory," *IEEE Trans. Computers,* vol. 39, no. 4, Apr. 1990.

[12] B.D. Fleisch, "Reliable Distributed Shared Memory," *Proc. Second Workshop Experimental Distributed Systems,* pp. 102-105, 1990.

[13] M. Stumm and S. Zhou, "Fault Tolerant Distributed Shared Memory Algorithms," *Proc. Parallel and Distributed Processing,* pp. 719-724, 1990.

[14] T.J. Wilkinson, "Implementing Fault Tolerance in a 64-bit Distributed Operating System," PhD thesis, City Univ., London, July 1993.

[15] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel, "The Performance of Consistent Checkpointing," *Proc. 11th Symp. Reliable Distributed Systems,* pp. 39-47, Oct. 1992.

[16] K.L. Wu, W.K. Fuchs, and J.H. Patel, "Error Recovery in Shared Memory Multiprocessors Using Private Caches," *IEEE Trans. Parallel and Distributed Systems,* vol. 1, pp. 231-240, Apr. 1990.

[17] D. Chaiken et al., "Directory-Based Cache Coherence in Large-Scale Multiprocessors," *Computer,* vol. 23, no. 6, pp. 49-58, June 1990.

[18] P. Stenström, T. Joe, and A. Gupta, "Comparative Performance Evaluation of Cache Coherent NUMA and COMA Architectures," *Proc. Int'l Symp. Computer Architecture,* pp. 80-91, May 1992.

[19] A. Saulsbury, J. Carter, and A. Landin, "An Argument for Simple COMA," *Proc. First Symp. High-Performance Computer Architecture,* Jan. 1995.

[20] K. Li and P. Hudak, "Memory Coherence in Shared Memory Systems," *ACM Trans. Computer Systems,* vol. 7, no. 4, pp. 321-359, Nov. 1989.

[21] Z. Lahjomri and T. Priol, "Koan: A Shared Virtual Memory for the ipsc/2 Hypercube," *Proc. Compar/VAAP92,* Sept. 1992.

[22] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "Treadmarks: Shared Memory Computing on Networks of Workstations," *Computer,* vol. 29, no. 2, pp. 18-28, Feb. 1996.

[23] A.-M. Kermarrec, "Une approche globale fondée sur la réplication pour la disponibilité et l'efficacité des systèmes extensibles à mémoire partagée," PhD thesis, Université de Rennes 1, 1996.

[24] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Hermann, P. Leonard, S. Langlois, and W. Neuhauser, "Chorus Distributed Operating Systems," *Computing Systems,* vol. 1, no. 4, pp. 305-370, Oct. 1988.

[25] A. Gefflaut, C. Morin, and M. Banâtre, "Tolerating Node Failures in Cache Only Memory Architectures," *Proc. Supercomputing '94,* Nov. 1994.

[26] A.-M. Kermarrec, "Contrôle de la réplication des données dans une mémoire virtuelle partagée recouvrable efficace," *Technique et Science Informatiques,* vol. 15, no. 5, May 1996.

[27] C. Morin, A. Gefflaut, M. Banâtre, and A.-M. Kermarrec, "COMA: An Opportunity for Building Fault-Tolerant Scalable Shared Memory Multiprocessors," *Proc. 23rd Ann. Int'l Symp. Computer Architecture,* May 1996.

[28] A. Gefflaut and P. Joubert, "Spam: A Multiprocessor Execution Driven Simulation Kernel," *Int'l J. Computer Simulation,* vol. 6, no. 1, pp. 69-88, Jan. 1996.

[29] H. Davis, S.R. Goldsmidt, and J. Hennessy, "Multiprocessor Simulation Using Tango," *Proc. 1991 Int'l Conf. Parallel Processing,* Aug. 1991.

[30] J. Larus, "Abstract Execution: A Technique for Efficiently Tracing Programs," *Software Practice and Experience,* vol. 20, no. 12, pp. 1,251-1,258, Dec. 1990.

[31] H. Schwetman, "Csim User's Guide," Rev. 2 ACT-126-90, MCC, 1992.

[32] J. Singh, W. Weber, and A. Gupta, "Splash: Stanford Parallel Applications for Shared-Memory," Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford Univ., 1991.

[33] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Ann. Int'l Symp. Computer Architecture,* pp. 24-36, June 1995.

[34] A. Gefflaut, "Proposition et évaluation d'une architecture multiprocesseur extensible à mémoire partagée tolérante aux fautes," PhD thesis, Rennes Univ., Jan. 1995.

[35] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Henessy, "The Stanford Flash Multiprocessor," *Proc. 21st Ann. Int'l Symp. Computer Architecture,* Apr. 1994.

[36] A-M Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut, "A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability," *Proc. 25th Int'l Symp. Fault-Tolerant Computer Systems,* June 1995.

[37] A.-M. Kermarrec, C. Morin, and M. Banâtre, "Design, Implementation and Evaluation of Icare: An Efficient Recoverable DSM," *Software Practice and Experience,* vol. 28, no. 9, pp. 981-1,001, July 1998.

[38] C. Morin and I. Puaut, "A Survey of Recoverable Distributed Shared Memory Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 8, no. 9, Sept. 1997.

[39] N. Neves, M. Castro, and P. Guedes, "A Checkpoint Protocol for an Entry Consistent Shared Memory System," *Proc. 13th ACM Symp. Principles of Distributed Computing,* Aug. 1994.

[40] M. Costa, P. Guedes, M. Sequeira, N. Neves, and M. Castro, "Lightweight Logging for Lazy Release Consistent Distributed Shared Memory," *Proc. Symp. Operating Systems Design and Implementation,* Nov. 1996.

[41] A.-M. Kermarrec and C. Morin, *An Efficient Recoverable DSM on a Network of Workstations: Design and Implementation,* chapter 7, pp. 123-153, Kluwer Academic, 1997.

[42] W. Bolosky, R. Fitzgerald, and M. Scott, "Simple but Effective Techniques for NUMA Memory Management," *Proc. 12th ACM Symp. Operating Systems Principles,* Dec. 1989.

[43] A. Cox and R. Fowler, "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with Platinum," *Proc. 12th ACM Symp. Operating Systems Principles,* Dec. 1989.

Christine Morin received her engineering degree from the Institut National des Sciences Appliquées (INSA) of Rennes (FRANCE) in 1987, and her MS and her PhD degrees in computer science from the University of Rennes I in 1987 and 1990. In 1998, she got her habilitation à diriger des recherches in computer science from the University of Rennes I. Since 1991, she has held a researcher position at INRIA/IRISA. Her research interests include shared memory multiprocessor architectures, fault tolerance, operating systems, clusters, and high performance computing. She is a member of the ACM and the IEEE Computer Society.

Anne-Marie Kermarrec received her PhD degree in computer science from the University of Rennes, France, in 1996. In 1997, she worked as a postdoctoral researcher in the Department of Computer Science of Vrije Universiteit in Amsterdam, The Netherlands. She recently held an assistant professor position at the University of Rennes, France. She joined Microsoft Research in Cambridge, United Kingdom, in March 2000. Her research interests include distributed systems, shared memory, large-scale information sharing, replication, and high-availability.

Michel Banâtre received the "Docteur es-Sciences" degree from the University of Rennes in 1984. Since 1986, he has held a "research director" position at INRIA (Rennes). He is currently leading the Solidor research team working on the construction of distributed applications and systems. His research interests include parallel object-oriented languages, distributed operating systems, fault-tolerant architectures, and multimedia applications.

Alain Gefflaut received an engineering degree from the Institut National des Sciences Appliquées of Rennes, France, in 1991. He holds MS and PhD degrees in computer science from the University of Rennes, France. From 1996 to 1998, he was a development engineer at Siemens AG, Munich, Germany, where he worked on telecommunication products. He is currently employed as a research staff member at the IBM T.J. Watson Research Center, where he is involved in a project targeting the design and the implementation of a highly composable operating system. His research interests include operating systems, multiprocessor architectures, shared memory, networking, ubiquitous computing, and fault tolerance.