A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability

Anne-Marie Kermarrec, Gilbert Cabillic[†], Alain Gefflaut[†], Christine Morin[†] and Isabelle Puaut[§]
IRISA
Campus Universitaire de Beaulieu
35042 Rennes Cedex - France

Abstract

Large-scale distributed systems are very attractive for the execution of parallel applications requiring a huge computing power. However, their high probability of site failure is unacceptable, especially for long time running applications. In this paper, we address this problem and propose a checkpointing mechanism relying on a recoverable distributed shared memory (DSM) in order to tolerate single node failures. Although most recoverable DSMs require specific hardware to store recovery data, our scheme uses standard memories to store both current and recovery data. Moreover, the management of recovery data is merged with the management of current data by extending the DSM's coherence protocol. This approach takes advantage of the data replication provided by a DSM in order to limit the amount of transferred pages during the checkpointing. The paper also presents an implementation and a preliminary performance evaluation of our recoverable DSM on a 56 nodes Intel Paragon.

1 Introduction

Distributed shared memory (DSM) systems implement a shared memory programming model on top of a distributed system without hardware support for shared memory (i.e. distributed memory multicomputers, networks of workstations). A DSM is attractive from the programmer's point of view, since it simplifies data partitioning and load balancing, which are two of the toughest problems when programming parallel machines.

One of the main advantages of distributed systems is their scalability. They are thus very attractive for the execution of parallel applications requiring both long running execution times and a powerful computing environment. Nevertheless, as the number of components in these systems grows, the probability that a failure occurs also increases. This behavior is unacceptable, especially when long-running applications are to be executed. As restarting a long-running application from its initial state must be avoided, it is desirable for a DSM to be recoverable and to allow an application to restart its execution from a state saved beforehand, when a failure occurs. In order to preserve the vocation of distributed systems, the implementation of such error recovery mechanisms must be done without sacrificing the system scalability.

Backward error recovery is a well-known fault tolerance technique [16]. It avoids the need of restarting an application from its initial state in case of failure by using a previous image of the system to restart the execution. Most recoverable DSM assume the presence of a stable storage device to ensure the stability of recovery data. These stable storages are usually implemented with specific hardware devices such as stable memories [3, 2], or more conventional devices such as disks. The first solution is efficient but too expensive, especially for large-scale systems, whereas the second one is cheap but may rapidly limit the system scalability. In this paper, we propose a backward error recovery implementation that allows the system to tolerate permanent single node failures and avoids the use of specific hardware by ensuring the stability of recovery data with the replication mechanisms provided by a DSM system. Moreover, this recovery scheme integrates the management of recovery data and current data by extending the standard coherence protocol used by a DSM. The paper also describes a first implementation of the proposed recoverable DSM on a 56 nodes Intel Paragon [14]. The application reconfiguration, in particular the way to restart processes after a failure, is outside the scope of this paper.

The remainder of the paper is organized as follows. The next section presents our base DSM and failure model. Section 3 then describes the integration of recovery data management in the standard coherence protocol and the way to make a DSM recoverable. Section 4 presents our implementation and some measures we have made to show the performance degradation introduced by fault tolerance mechanisms. It also studies the scalability of this proposal. Section 5 presents related works and finally, Section 6 concludes with a summary of our proposal.

[†]INRIA: Institut National de Recherche en Informatique et Automatique
[§]INSA Rennes: Institut National des Sciences Appliquées of Rennes

2 System Model

The system we consider is composed of a set of nodes connected by an interconnection network. Each node is composed of one or more processors and of a memory module. In the evaluation section, this system is an Intel Paragon but any other hardware platform could be envisaged.

This section first describes the base DSM and especially the coherence protocol which is extended and then lists up the assumptions of our scheme.

2.1 Distributed Shared Memory

In a DSM, local memories are used as large caches of the global shared address space. Pages are replicated on demand in the local memory of processors requesting them, and can be loaded either from disk or from another local memory. In this paper, for the sake of simplicity, we do not consider the swapping management and consider that the set of memories can contain all virtual pages.

Since several copies of the same page may exist in the system, coherence between these copies has to be maintained. Our DSM is managed by a static distributed manager [17] implementing a sequential consistency model. Coherence is maintained in the system by directories statically distributed among the nodes. This technique avoids the bottleneck inevitably created by a central manager in a large-scale distributed system. The coherence protocol uses a write-invalidate strategy [1].

For each page, we distinguish its *owner node* from its *manager node*. The owner of a page maintains the only valid copy of the page. It can hence read or modify this page. The manager of a page keeps a directory entry for that page containing coherence information related to it: its owner, its state (*Invalid, Shared or Modified-exclusive*) and its *copyset list* which contains the list of the nodes having a copy of the page in their local memory. The manager of a page is statically determinated by a dedicated function (for instance a modulo between the page's address and the number of nodes). When a page fault occurs, a request for the page is sent to the manager of the page which forwards the request to its owner. The manager of a page is also in charge of performing the necessary invalidations before a write access on a replicated page. The standard coherence protocol is depicted by the transition diagram of Figure 1.

A page p on a node n may be in one of the following states:

- *Invalid*: the local memory of n does not contain a copy of p.
- *Shared*: the local memory of n has a copy of p that can only be read. Other *Shared* copies of p may exist in other local memories.
- *Modified-exclusive*: the local memory of n has a copy of the page that can be read or written. No other copy of the page exists in the system and n is the *owner* of the page.

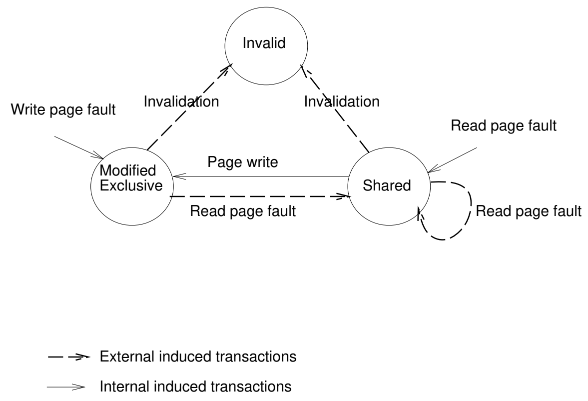

Figure 1: Base coherence protocol

The *Shared* and *Modified-exclusive* states respectively correspond to the *Read* and *Write* states in Li's terminology [17].

2.2 Assumptions

We make the following assumptions about the system we consider. Like most of the backward error recovery implementations, we first consider *fail-stop* nodes: a faulty node stops its execution as soon as its failure is detected, without interfering with the other nodes. This characteristic ensures that neither the failure-free nodes nor the contents of their memory modules, are altered by the failure of a faulty node. The failure of one of its elements leads to the unavailability of the whole node. Secondly, the interconnection network we consider is assumed to provide reliable communications between processors in the system. Finally, we consider that the system implements an efficient error detection mechanism. Our scheme supports the presence of transient and single permanent failures in a distributed system implementing a DSM. A transient failure leads to the unavailabity of the node just temporarily as it disappears without external intervention. After the rollback, the system can restart with the whole set of nodes, including the node which has caused the failure. In this paper, we focus on the checkpoint of shared data whereas the application and system reconfigurations, including in particular the migration on safe nodes of processes which were executing on a faulty node, are beyond the scope of this paper.

3 A Recoverable DSM Integrating Recoverability in the Coherence Protocol

3.1 Design Guidelines

The vocation of the scalable distributed systems considered in this paper is to support the execution of large scale parallel applications.

Backward error recovery schemes [16] consist in periodically saving a consistent system image, called a checkpoint. In the event of a failure, the system is rolled back to this checkpoint. So, backward error recovery is well-suited for long-running applications, as it avoids the need to restart an application execution from its initial state when a failure occurs. Moreover, backward error recovery may be implemented in a user-transparent way.

For parallel applications, consistent checkpointing is however much more complicated than in a sequential environment since it must take into account the interactions between processes. Different strategies have already been proposed to ensure that the set of process'checkpoints always forms a consistent recovery state [2, 15]. In this paper, for the sake of simplicity, we adopt a global coordinated checkpoint. Periodically, one of the nodes, called the initiator, broadcasts a message to initiate the recovery point establishment. The major benefit of this strategy is the simplicity of the algorithm and the fact that it provides a straightforward approach to take a complete recovery line at once. Furthermore, we chose *incremental* coordinated checkpointing: new recovery copies are required only for data that have been modified since the last recovery point. Such an approach limits the amount of data which must be saved during the establishment of a recovery point. Any backward error recovery scheme assumes that the recovery points (recovery data) are kept on a stable storage. Such a storage ensures stability properties to recovery data, that is, it ensures persistency and atomic update of recovery data. Persistency ensures that recovery data are always accessible by fault-free processors even in spite of a failure. It is usually ensured by replicating recovery data on two failure independent storage devices [4]. Atomic update of recovery data allows to tolerate a failure even during the establishment of a recovery point. It can be physically implemented by a two-phase commit protocol based on data replication [13].

Many proposed stable storage implementations use either specific hardware devices such as stable memories which are quite expensive, or more conventional devices such as disks which are not very efficient and which limit the scalability of the architecture [3, 2, 4]. Unlike these cases, our approach does not require any specific hardware. Instead, the stable storage needed for recovery data is implemented by using the replication mechanisms provided by the standard memories of a DSM. As node memories can store any shared data, recovery data persistency is easily ensured by replicating recovery data on two distinct node's memories. Atomic recovery point establishments are guaranteed by a two-phase commit protocol where pages which have been modified since the last recovery point, are first replicated before old recovery pages are discarded. Thus, even if a failure alters one of the copies, the other is still available to be restored at rollback.

Using standard memories to store current data as well as recovery ones has several advantages. It first limits the hardware development but nevertheless ensures rapid recovery point establishments. It also preserves the system scalability since the number of memories increases with the number of nodes. Finally, it allows to use the natural replication in DSM to avoid the need of creating new recovery copies. As in a DSM a page may have several copies in the system, our approach uses these replicas instead of creating new copies. This advantage leads to a decrease of the time required to take a checkpoint. For this, the DSM coherence protocol has to be modified. This is obtained by adding two states to distinguish recovery pages from current ones. Standard memories may then contain both recovery pages and current ones. Moreover, when recovery pages have not been modified since the last checkpoint, they can be read by processors, thus avoiding in some cases, page transfers between nodes.

3.2 Extension of the Coherence Protocol

The basic principle of our recoverable DSM is the integration of the management of recovery data stored in the nodes' standard memories, in the coherence protocol which manages current data. The major property which is ensured is that two copies of recovery data exist at any time in the system. Thus, if a permanent failure alters one of them, the second one is still valid and can be used to restart the system. The two recovery copies of a page just have to be stored in two distinct memory modules. In our scheme, both recovery and current data coexist in standard memories and are managed transparently by an extended coherence protocol. Two states must be added to the standard coherence protocol to manage recovery data. The two copies of a recovery page are in the same state among the following ones:

- *Checkpoint*: pages in this state are recovery copies of a page that has been modified since the last checkpoint. A current version of this page is also present in the system. Pages in *checkpoint* state are not available for consultation during fault-free execution. They belong to the recovery point and will be accessed only in the event of a failure.

- *Shared-Checkpoint*: pages in this state are recovery copies of a page that has not been modified since the last recovery point. As a result, they are readable by processors in fault-free executions. No other current version of such a page exists in the system except *Shared* copies. *Shared-checkpoint* pages only exist in the interval between a checkpoint and the first modification on the page after this checkpoint. On a write operation, *shared-checkpoint* pages are changed into *checkpoint* ones.

A recovery point consists of the set of *checkpoint* and *shared-checkpoint* pages. Figure 2 depicts the state diagram of the extended coherence protocol including the transitions caused by standard read and write operations, as well as checkpointing and rollback operations.

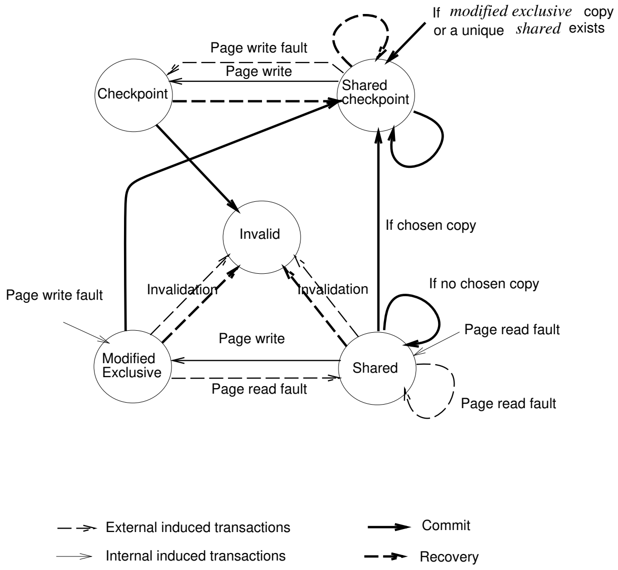

Figure 2: Extended coherence protocol

The behavior of the extended coherence protocol is the following. A write request on a *shared-checkpoint* page induces a change of the two *shared-checkpoint* pages into *checkpoint* ones and a *modified-exclusive* copy of the page is then created on the requesting node. When a page p is requested by a node n, several situations caused by the introduction of the two recovery states may occur:

Page read operation If the state of p is *Shared-checkpoint*, p is readable because the page has not been modified since the last recovery point. A *shared* copy of the page is created on the requesting node. Otherwise, the operation is handled as in the base coherence protocol.

Page write operation A write request on a *shared-checkpoint* page induces a change of the two *shared-checkpoint* pages in *checkpoint* ones; a *modified-exclusive* copy of the page is then created on the requesting node. In case of a write hit, a *checkpoint* copy and a *modified exclusive* copy may exist in the same local memory. Write requests on pages which are in standard states, are handled as in the base coherence protocol.

The main advantage of this scheme is that a part of recovery data (*shared-checkpoint* pages) may be used as standard ones and in particular may serve read page faults.

3.3 Establishment of a Recovery Point

Recovery point establishment may be a frequent operation and hence must be performed efficiently so as not to degrade performance in failure-free execution.

Establishing a recovery point consists in creating two recovery copies of every page which has been modified since the last recovery point. A two-phase commit protocol is required to ensure the atomicity of the recovery point establishment. The new recovery point is created during the first phase, called *establish phase*, while the old one is discarded in the second phase, the *commit phase*. The first phase is initiated by a node (initiator node) which broadcasts a *begin-establish* message that triggers the checkpoint to all other nodes. So, two of the current copies of a page become recovery data using the temporary *precommit* state, while recovery data are maintained in their state. The following actions are performed by the manager of each page during the first phase:

- Two selected *shared* pages must be changed into *precommit* pages. We must ensure that a second recovery copy of that page exists on another node. Two cases may arise: if several nodes have a replica of that page, the second recovery copy is chosen randomly among the copyset of the page and its state is changed to *precommit*. If the copyset is empty, a second copy is created on another node.
- A page in *modified-exclusive* state becomes *precommit*. As a *modified-exclusive* page is not replicated in physical memory, another copy is created in *precommit* state on another node.

When a node has finished the *establish* phase, it sends an acknowledgment to the initiator. When the initiator node has received all acknowledgments, it broadcasts a *begin-commit* message. At this step, *checkpoint* pages belong to the old checkpoint, *precommit* pages belong to the new checkpoint and *shared-checkpoint* pages belong to both checkpoints. Thus, even if a failure occurs during the establishment of a recovery point, at least one of the two possible checkpoints is available. During the *commit* phase, *precommit* pages are changed into *shared-checkpoint* pages. *Shared-checkpoint* pages, which correspond to pages not modified since the last recovery point are kept in the same state as they also belong to the new recovery point. *Checkpoint* pages are invalidated, as they only belong to the previous recovery point. At the end of this phase, only *shared-checkpoint* and *shared* pages exist in the memories of the system. Figures 3 and 4 describe the two phases of the recovery point algorithm.

```
Establish Phase{
  For each entry in the local directory{
   case (entry.state){
    Modified-Exclusive :
       /* There is a single current copy in memory */
       Create a copy of the page in a another memory
       by sending the page and a precommit message
       Entry-state = Precommit
    Shared :
       case(entry.copyset_list.nb){
       1 :
          /* There is a single copy in memory */
          Create a copy of the page in a another
          memory by sending the page and a
          precommit message
          Entry-state = precommit
       2 or + :
                selection of two copies of the page
                by sending a precommit message
                Entry-state = precommit
    }
    Shared-Checkpoint:
    Checkpoint :
       skip
         }
         }
   Send end_establish message to the initiator

} End of Establish Phase
```

Figure 3: Establish phase of the recovery point establishment algorithm

```
Commit Phase{
   For each entry in the local directory{
       case (entry.state){
              Precommit :
                 entry_state = Shared-Checkpoint
              Checkpoint :
                 entry_state = Invalid
                 /* Discarding of the old recovery point */
              Shared-Checkpoint:
              Shared :
                   skip
       }
   }
} End of Commit Phase
```

Figure 4: Commit phase of the recovery point establishment algorithm

```
Rollback algorithm
{
  For each entry in the local directory{
      case(entry.state){
            CHECKPOINT: /* recovery data that must be
restored*/
                        entry.state = SHARED-CHECKPOINT
                        Send update message to home site
            SHARED-CKECKPOINT: /* recovery data */
                             Skip
            PRECOMMIT : /* current or intermediate copies */
            SHARED :
            MODIFIED-EXLUSIVE:
                             entry.state = INVALID
                             } }
 }
```

Figure 5: Rollback algorithm

3.4 Restoration of a Coherent State

When a failure is detected, a message indicating the identity of the faulty node is broadcast. Upon the receipt of such a message every node locally performs the rollback algorithm. All current pages, i.e. pages which state is either *modified-exclusive* or *shared*, as well as *precommit* pages are invalidated. To restore the previous recovery point, *checkpoint* pages are restored to *shared-checkpoint* ones and *shared-checkpoint* pages, which have not been modified since the last recovery point establishment, remain unchanged. At the end of the rollback operation, only *shared-checkpoint* pages exist in the system. At this time, one of the two nodes whose memory contains a *shared-checkpoint* page is randomly designated as the owner of the page. Figure 5 describes the rollback algorithm.

After a rollback, processes can immediately restart their execution if the failure is a transient one. However, if the detected failure is permanent, pages which are stored on the faulty node are definitively lost. Hence, a reconfiguration must be performed in order to ensure again that two copies of every recovery page exist, and that a new failure can be tolerated.

In the reconfiguration step, every node checks for every recovery page if the second recovery copy was stored on the faulty node or not. In this former case, a second copy of the recovery page is immediately created on another node from the existing recovery page and a new owner is designated among the two nodes whose memory contains a recovery version of the page. Two methods can be used to perform this check. The first one is to maintain, during failure-free execution, a directory entry per recovery page pointing to the second one. Every node checks its directory at the reconfiguration time and immediately detects recovery pages which second copy was on the faulty node. This method implies some overhead during normal execution and requires careful management of directory updates. Another method is to use broadcasting to check if the corresponding copy of each recovery page was on the faulty node or not. This second method makes reconfiguration more expensive by heavy communication costs but implies no overhead during normal execution.

In addition, if a permanent failure occurs, two cases must be considered: the failure of a manager node and the failure of an owner node.

In the former case, a new manager must be defined for pages whose manager node is the faulty node. This spare manager is determined by a static *recovery function* known by every node. After a failure the spare manager of a page has no information about the state and the owner of the page. So, when it receives the first request for the page, it broadcasts a request in the system to identify the page owner and to update the corresponding directory entry. In this case, every node checks its table and the owner of the concerned page replies to the manager. This mechanism avoids to manage a backup manager node during normal execution. Once a failure has occurred, every node must use the spare function to identify the manager of pages which were managed by the faulty site.

In the latter case, when the manager node of a page points to a faulty node which was the owner of the page at the time of the failure, it must determine the new owner of that page. It then broadcasts a message to which the new owner replies, thus permitting the manager node to update its table. An alternative is to update the manager tables with the new owner at the reconfiguration time. As the reconfiguration has not yet been implemented, we have no result to know which method is the most efficient.

This protocol has been proven in [12]. The extended protocol's coherence and stability properties have been verified using a dedicated coherence protocol verification technique based on a symbolic expansion of the states accessible by a system made of an arbitrary number of caches [10].

4 Performance Evaluation

In this section, we present the first performance results we have obtained with our recoverable DSM implementation. We focus in particular on the overhead of consistent checkpointing on failure-free executions of parallel applications and study the scalability of our proposal. To obtain our measures, the extended coherence protocol is integrated in a standard DSM system, MYOAN [6], implemented on top of a 56 nodes Intel Paragon. We briefly describe the system, the implementation of our recoverable DSM and show the preliminary results we obtained with two parallel applications, matrix multiplication (MatMul) and Modified Gram-Schmidt (MGS). Matmul consists of multiplications between two square matrices of 256*256 doubles. Two data structures are accessed through read operations as the third is written. There is no false sharing in this application. The MGS algorithm produces, from a set of vectors, an orthonormal basis of the space generated by these vectors. The size of the MGS problem is 128 vectors of 256 doubles. This application execution induces false sharing and this phenomenon increases with the number of processors on which the application is executed.

4.1 Implementation

Measures have been done on an Intel Paragon supercomputer [14] equipped with 56 nodes linked by a high speed interconnection network. Each node is composed of two i860 processors and runs a copy of the Paragon operating system, based on the Mach OSF/1 operating system [18]. The first node is used to run applications and the second is specifically used for communications. Due to the operating system version used in our implementation, we were not able to use the second processor to manage the communications. All measures have been taken with only one processor running on each node. The interconnection network is a 2 dimensional grid. Our implementation is an extension of MYOAN [6], a DSM mechanism implementing K. Li's static distributed coherence protocol on the Intel Paragon.

Compared to a standard DSM implementation, the integration of the extended protocol is not very complex. In MYOAN, the DSM is composed of a set of Mach memory managers, one per node, that are implemented as Mach tasks. The checkpoint algorithm is implemented as an additional thread in each of them. We have chosen the node 0 to be the coordinator. The establishment of a recovery point is initiated by the coordinator and all nodes are synchronized by it. Indeed, every phase of the protocol can begin as soon as all nodes have finished the previous phase and this nodes synchronization is managed by the coordinator.

An optimization of the recovery point establishment algorithm presented in figure 4, has been implemented. Instead of scanning twice the manager's directory, first in order to create *precommit* pages and secondly, during the *commit* phase, in order to change pages from *precommit* state to *shared-checkpoint* state, we implement one counter per page and one global counter per node. During the *establish* phase, recovery pages are directly changed into *Shared-checkpoint* state and their counter is incremented to the value of the global counter plus 1. At this step, the old recovery point is composed of the set of *shared-checkpoint* and *checkpoint* pages which have a counter equal to the global counter; the new recovery point is composed of *Shared-checkpoint* pages whose counter is superior to the global counter. The global counter is incremented during the *commit* phase in order to validate the new recovery point. This mechanism avoids a scan of the manager's directory during the *commit* phase.

In our implementation, each manager has to scan its directory and consults for each page, its copyset list. When other copies of the page exist, the page stored in the memory in the first node of the list is chosen to become a recovery page. A request is then sent to this node which sends an acknowledgement directly to the owner of the page. If the copyset list is empty, a request is sent to the owner of the page which is in charge of creating a recovery page from its own page on a neighboring node.

4.2 Evaluation

In this section, we present preliminary results. We first study the impact of checkpointing on the performance of parallel applications and secondly we focus on the impact of data replication. In both cases, the scalability of our approach is detailed.

4.2.1 Overhead of checkpointing

A lot of consistent checkpointing proposals consider that the checkpointing interval can be very long (several minutes). They usually do not consider external operations that an application can perform and which can force the saving of a new checkpoint each time they occur. For instance, a recovery point may be saved every time a write I/O operation occurs. We believe that such operations can occur quite frequently. Our preliminary results have been obtained with a frequency of 1 checkpoint every 3 seconds. Lower checkpointing intervals would report higher performance degradations.

To analyse the checkpointing overhead, we compare the execution of the two applications first on the Intel Paragon using a standard DSM and second on the same architecture using our recoverable DSM. We decompose the global overhead in three distinct overheads: (1) the time required to synchronize processes during a checkpoint establishment, (2) the time required to replicate pages during a checkpoint establishment and (3) the time due to other overheads and especially the time required to solve additional page faults induced by the fault-tolerance mechanisms implementation. Figure 6 presents these overheads for the matrix multiplication and figure 7 presents those for the MGS algorithm compared to the execution of same application with a standard DSM.

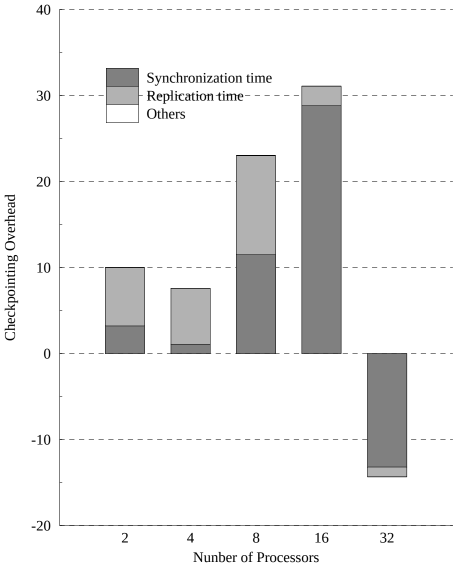

Figure 6: Checkpointing overhead of MatMul

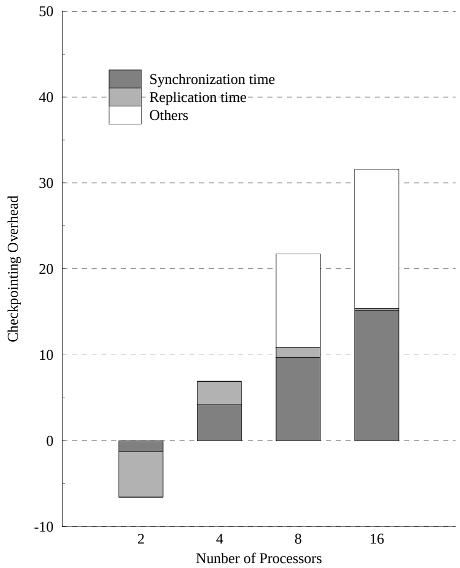

Figure 7: Checkpointing overhead of MGS

We are not interested in quantitative results but in the behavior during the checkpointing and the scalability of our approach. Globally the overhead due to checkpointing ranges from 35% to less than 5%. With MatMul, there is no other overhead than synchronization and replication overheads. The major part of the overhead is due to the synchronization time and its proportion increases with the number of processors. This situation is strongly dependent on the synchronization implementation used but could be improved by using, on each node, the second processor dedicated to communications. On the contrary, the replication time decreases with the number of processors and this fact is an interesting feature for the scalability of our approach. With the MGS algorithm, the checkpointing overhead increases with the number of processors. The major part of this overhead is composed of the synchronization time and the other overhead. This overhead is mainly due to the resolution of additional write page faults induced by the implementation of the fault tolerance mechanisms. Indeed, at checkpointing, writables pages (*modified-exclusive* pages) are changed in *shared-checkpoint* pages which are only readable. Nevertheless, results show that the overhead due to the page replication at checkpointing time decreases with the number of processors.

4.2.2 Data replication

Several aspects are to be considered in data replication. The first one is that replication is required in any backward error recovery scheme. Figure 8 shows the throughput of data replication for a number of processors varying from 2 to 32. Results demonstrate that our approach is scalable for the considered parallel applications. Indeed, in MatMul, the replication throughputs range from 2.5 Mbytes/s for 2 processors to 13.9 Mbytes/s for 16 processors. Similarly, in MGS, the replication throughputs range from 2.2 Mbytes/s for 2 processors to 13.6 Mbytes/s for 16 processors. Moreover, these throughputs increase almost linearly with the number of processors.

The second important point is that the replication inherent to the concept of DSM is used to avoid the replication of pages during recovery point establishments. In MatMul, the rate of pages which are already replicated in memories and for which the replication of recovery pages at checkpoint time is avoided varies from 22% for 2 processors to 35% for 32 processors of the total amount of pages which have been modified since the last checkpoint and have to be replicated. Moreover, this rate increases with the number of processors and this fact attests that our approach is scalable and takes advantage of the standard behavior of a DSM.

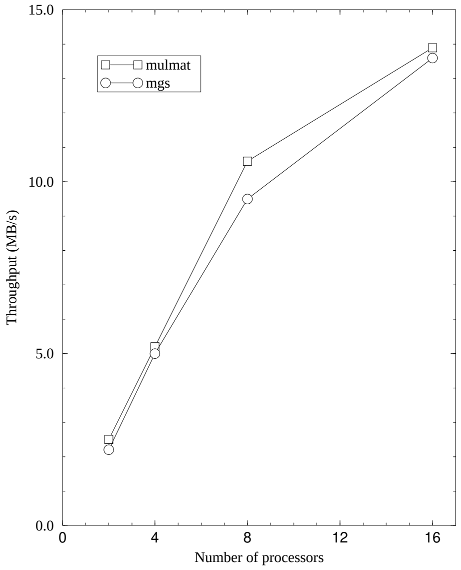

Figure 8: Replication throughput

Finally, we can note in Figures 6 and 7 that in two cases, the 32 processor-execution with MatMul and the 2 processor-execution with MGS, the execution time using the recoverable DSM is lower than in the standard case. This is due to the replication at checkpointing time of pages which are further used during the execution of the application. In these cases, the replication required to ensure data persistency decreases the number of page faults during normal execution. In Matmul, this case arises in the 32 processor-execution whereas it arises in the 2 processor-execution in MGS. This difference depends on the application's behavior. Indeed, the efficiency of Matmul execution, as there is no false sharing, increases with the number of processors whereas in MGS, the false sharing increases with the number of processors leading to a lower efficiency. In MGS, each processor writes one result matrix's column by reading the previous column provided by the neighboring node. So the case of negative overhead arises only in the 2 processor-execution as the recovery data are replicated on the two nodes and as each node is the neighbor of the other.

This negative overhead is due to the higher cost of page misses compared to the cost of page replications. Indeed, solving a page miss is more expensive than replicating a page on a neighboring node. In fact, when a processor references a page which is not in its local memory, the manager is first requested and then the request is forwarded to the page owner. Moreover, the page owner, which provides the page to the requesting node is not necessarily its neighbor. On the contrary, when a recovery point is established, modified pages are necessarily replicated in a neighbor node memory and this limits the communications between nodes. Moreover, the owner and the manager may be requested simultaneously in the recoverable DSM, whereas in a page miss, the manager is first requested and then the owner. Indeed, the replication policy is known by every node: when the copyset of a page is empty, the page is replicated on the neighboring node. Finally, at checkpointing, a lot of pages are simultaneously replicated and this feature decreases the average cost of a page replication. For all these reasons, using for computation a page replicated for fault tolerance reasons is cheaper than solving a page miss. This feature explains that when, this phenomenon arises, the execution time with the recoverable DSM is lower than in the standard case.

We must note that this favorable behavior of negative overhead has been measured for two applications with small problem sizes. Larger sets of data may induce swapping on disks and then the performance degradation due to the fault tolerance mechanisms would be more important.

5 Related Work

Many recoverable DSM systems that have been proposed [23, 21, 8, 7, 19] are usually based either on the implementation of stable storages which require specific hardware, or on the use of disks to store recovery data. Our approach is similar to Wu and Fuchs' one by integrating fault tolerance and memory management. A key difference with their work is that we store recovery data in standard memories. In [23], the authors focus on two points: memory coherence and disk storage. Their checkpointing scheme, which avoids rollback propagation, is integrated with the shared virtual memory management. A single checkpoint is maintained in the system but no global coordination is required when a process establishes a checkpoint. Establishing a process checkpoint consists of copying the process dirty pages on a central shared twin-page reliable disk. An incremental checkpointing scheme is used as disk activities required at checkpoint time can be spread over the entire checkpoint interval (no atomic update). At recovery time, no explicit I/O are required to restore the disk state as recovery data are restored in memory on demand.

In [20] four basic DSM algorithms are extended to tolerate single host failures. Our approach is similar to the 4th algorithm of Stumm and Zhou as recovery data are stored in memory but does not suffer its main disadvantages. Management of shared writable data is less expensive in our scheme as the only overhead is changing the state of the two *shared-checkpoint* copies to *checkpoint* when the first write occurs after a commit. No other page is concerned. We only keep the two necessary copies of recovery data in memory whereas in the scheme of Stumm and Zhou the number of copies is not bounded. Thus we have no need of a garbage collection mechanism. Moreover, our rollback procedure is local whereas up-to-date data have to be localized in Stumm's scheme, requiring many intersite communications. In our scheme, each site which possesses a *shared-checkpoint* copy has to check in its directory if the other copy is on the faulty site. If it is the case, a new *shared-checkpoint* copy has to be created. This method is more effective than the one proposed in [20].

In [5] a scalable fault-tolerant DSM algorithm is presented, based on the idea of a snooper for every page. As in our approach both current and recovery data are stored in standard memories but the algorithm is complex and has not been implemented. Further study is required to know the overhead of this scheme on the performance. Similarly, in [22], nodes memories are used to store current as well as recovery data. However, instead of using the coherence protocol to identify recovery data, this solution requires the use of a directory-based coherence protocol. As a result, this solution is less portable and more complex than ours.

Finally, in [9], a proposal to merge coherence and recoverability is also made by integrating a coherence support for data sharing to an existing logging mechanism which ensures data persistency. This approach is dual from ours as we integrate recoverability in the coherence protocol by exploiting the replication existing in any DSM whereas the approach proposed in [9] implements coherence in a logging mechanism which ensures data recoverability.

6 Concluding Remarks

We have presented the design and implementation of a recoverable DSM which exhibits the following advantages. First, since memories are used to store both recovery and current data, no specific hardware, except the assumption of using fail-stop processors, is required in our scheme. Secondly, as replication is needed in any backward error recovery scheme, we take benefit of the data replication inherent to a DSM to limit the performance degradation. Thirdly, current and recovery data managements are fully integrated within the coherence protocol extended by just adding two states to distinguish recovery data from current ones. This approach allows the DSM to tolerate multiple transient node failures as well as single permanent node failures.

We have implemented our global incremental coordinated checkpointing mechanism on an Intel Paragon by extending the MYOAN DSM. We have measured the rate of recovery data that can be treated per second and the performance degradation due to the introduction of fault tolerance mechanisms by comparing our recoverable DSM and MYOAN on the same architecture. Preliminary results show that the performance degradation does not exceed 35%. Moreover, our measures demonstrate that our approach is scalable as the replication rate increases with the number of processors. We have also shown that the natural replication of DSM is really exploited to avoid the replication of shared pages at checkpointing time. Synchronization and other overhead are tightly dependent on the operating system of the Intel Paragon. They may be reduced and do not depend on our scheme. Further study is needed to decrease these overheads. Nevertheless, our recoverable DSM is efficient and scalable in terms of replication.

Finally, this extended protocol may be implemented in different forms of DSM as soon as a coherence protocol is used to manage multiple copies of a page. In particular, it can also be used with little hardware modifications in a Cache Only Memory Architecture, which implements a DSM in hardware at a finer granularity than a software implemented DSM [11].

Acknowledgements

We would like to thank Benoit Dupin for carefully reading preliminary drafts of this paper. The design of MYOAN is supported by Intel SSD under an External Research and Development Program (INRIA contract n*o* 193C214313180120). The work presented in this paper is partially funded by the DRET research contract n*o* 93.34.124.00.470.75.01.

References

[1] J. Archibald and J. L. Baer. Cache Coherence Protocols : Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

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

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

[4] P.A. Bernstein. Sequoia: A Fault Tolerant Tightly Coupled Multiprocessor for Transaction Processing. *IEEE Computer*, 1988.

[5] L. Brown and J. Wu. Dynamic Snooping in a Fault-Tolerant Distributed Shared Memory. In *The 14th International conference on Distributed Computing Systems*, 1994.

[6] G. Cabillic, T. Priol, and I. Puaut. Myoan: an Implementation of the KOAN Shared Virtual Memory on the Intel Paragon. Technical Report 2258, INRIA, April 1994.

[7] K-M. Chew, A. J. Reddy, T. H. Romer, and A. Silberschatz. Kernel Support for Recoverable-Persistent Virtual Memory. In *USENIX Mach III Symposium*, pages 215–234, April 1993.

[8] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. The Performance of Consistent Checkpointing. In *Proc of the 11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[9] M.J. Feeley, J.S. Chase, V.R. Narasayya, and H.M. Levy. Integrating Coherency and Recoverability in Distributed Systems. In *Proc of the First Symposium on Operating Systems Design and Implementation*, November 1994.

[10] F.Pong and M.Dubois. A New Approach for the Verification of Cache Coherence Protocols. Technical report, Department of Electrical Engineering - Systems, University of Southern California, 1993.

[11] A. Gefflaut, C.Morin, and M.Banâtre. Tolerating Node Failures in Cache Only Memory Architectures. In *Proc. of Supercomputing'94*, November 1994.

[12] Alain Gefflaut. *Proposition et Évaluation d'une Architecture Multiprocesseur Extensible à Mémoire Partagée Tolérante aux Fautes.* PhD thesis, Rennes University, January 1995.

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

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

[15] G. Janakiraman and Yuval Tamir. Coordinated Checkpointing-Rollback Error Recovery for Distributed Shared Memory Multicomputers. In *13th Symposium on Reliable Distributed Systems*, 1994.

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

[17] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–357, November 1989.

[18] K. Loepere. Osf Mach Kernel Principles. Technical report, Open Software Foundation and Carnegie Mellon University, 1993.

[19] Satyarayanan (M.), Mashburn (H.), Kumar (P.), Steere (D.) and Kistler (J.). – Lightweight Recoverable Virtual Memory. *In : Proc. of 14th ACM Symposium on Operating Systems Principles*, pp. 146–160. – Asheville, North Carolina, December 1993.

[20] M. Stumm and S. Zhou. Fault Tolerant Distributed Shared Memory Algorithms. In *Proc of Parallel and Distributed Processing*, pages 719–724, 1990.

[21] V.O. Tam and M. Hsu. Fast Recovery in Distributed Shared Virtual Memory Systems. In *Proc. of 10th International Conference on Distributed computing Systems*, pages 38–45, June 1990. Paris, France.

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

[23] K.L. Wu and W.K. Fuchs. Recoverable Distributed Shared Virtual Memory. *IEEE Transactions on computers*, 39(4), April 1990.