Smooth and Efficient Integration of High-Availability in a Parallel Single Level Store System

Anne-Marie Kermarrec, Christine Morin

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Smooth and Efficient Integration of High-Availability in a Parallel Single Level Store System

Anne-Marie Kermarrec — Christine Morin

N° 4099

Décembre 2000

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4099--FR+ENG

Smooth and Efficient Integration of High-Availability in a Parallel Single Level Store System

Anne-Marie Kermarrec*, Christine Morin†

Thème 1 — Réseaux et systèmes

Projets PARIS

Rapport de recherche n° 4099 — Décembre 2000 —30 pages

Abstract: A parallel single level store (PSLS) system integrates a shared virtual memory and a parallel file system thus providing programmers with a global address space including both memory and file data. Parallel single level store systems implemented in a cluster thus represent an attractive support for long running parallel applications combining both the natural shared memory programming model and a large and efficient file system.

However the need to tolerate failures in such a system increases with the size of applications. In this paper we present the smooth integration of a backward error recovery high-availability support into a parallel single level store system. Our system is able to tolerate multiple transient failures, a single permanent one, and power cut failures affecting the whole cluster without requiring any specific hardware. For this purpose, our highly-available parallel single level store system relies on a high degree of integration (and reusability) of high-availability and standard supports. We focus on the parallel file system management at checkpointing and recovery time and especially on the mirror management. A prototype integrating our high-availability support has been implemented and we show some performance results in the paper.

* Microsoft Research Cambridge

† Université de Rennes 1

Unité de recherche INRIA Rennes

IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)

Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Key-words: Parallel Single Level Store, Checkpointing, Replication, Integration, Parallel File Systems, Shared Virtual Memory

Haute disponibilité dans un système parallèle à stockage uniforme des données

Résumé : Un système parallèle à stockage uniforme des données intègre une mémoire virtuelle partagée et un système de gestion de fichiers parallèle. Un tel système offre donc aux programmeurs un espace d'adressage global incluant à la fois les données en mémoire et les données de fichiers. Les systèmes parallèles à stockage uniforme des données mis en œuvre dans les grappes de calculateurs constituent un support de choix pour l'exécution d'applications parallèles en combinant à la fois un modèle de programmation naturel par mémoire partagée et un système de gestion de fichiers de grande taille et efficace. Cependant, le besoin de tolérer les défaillances dans un tel système croît avec la taille des applications. Dans cet article, nous présentons l'intégration de mécanismes de haute disponibilité fondés sur une technique de recouvrement arrière au sein d'un système parallèle à stockage uniforme des données. Le système proposé peut tolérer de multiples fautes transitoires, une faute permanente simple et des coupures de courant affectant l'ensemble de la grappe sans avoir recours à un quelconque dispositif matériel spécifique. Dans ce but, notre système parallèle à stockage uniforme des données à haute disponibilité s'appuie sur un degré d'intégration (et de réutilisation) élevé des mécanismes standard et de haute disponibilité. Nous abordons plus particulièrement la gestion du système de gestion de fichiers parallèle au moment de la création et de la restoration de points de reprise et la gestion des données sur disque en mode miroir. Un prototype intégrant les mécanismes de haute disponibilité proposés a été mis en œuvre et nous présentons quelques résultats de l'évaluation de performance.

Mots-clés : Système parallèle à stockage uniforme des données, recouvrement arrière, réplication, intégration, système de gestion de fichiers parallèle, mémoire virtuelle partagée

1 Introduction

Clusters of SMPs represent an attractive support for the execution of long-running parallel scientific applications. Targeted applications for clusters such as large-scale numerical simulations usually rely on the simple shared memory programming model and need to perform large input/output operations as well. On one hand, distributed shared virtual memory systems (SVM) [1, 13] offer an abstraction of a shared space across distributed memories in a cluster thus providing programmers with the attractive shared memory programming model. On the other hand, parallel file systems (PFS) [19, 9] provide high disk bandwidth by fragmenting a file on several disks of different cluster nodes, allowing parallel accesses to fragments [3].

Parallel Single Level Store systems To cope with this twofold requirement, namely the shared memory abstraction and a large and efficient file system, parallel single level store systems (PSLS)[1] [11], which integrates an SVM and a PFS are very well-suited for the execution of high performance applications in a cluster. The shared memory programming model is very much appreciated for its simplicity by scientific programmers. Unfortunately, one of the major limitations of SVM systems is their size limit and it is very seldom that memories fit the total working set of large-scale scientific applications. PFSs enable to overcome this limitation while ensuring fast and distributed access to data but usually rely on complex interfaces. It is desirable that programmers benefiting from a shared memory programming model do not loose this abstraction and do not have to deal with complex interfaces which would force them to understand the underlying distribution. To provide both disk capacity of a PFS and the natural way of programming of an SVM, our system relies on a single level of addressing: a global shared virtual address space manages both memory and file data. A mapping interface enables disk data to be mapped in the SVM system. All operations, including PFS ones, are made using standard memory reads and writes. Programmers are thus released from explicitly managing data transfers between disks and memories and concurrent accesses to the same file data are automatically handled by the SVM coherence protocol [16].

[1] In the remainder of this paper, PSLS refers to a single level store e system integrating an SVM and a PFS.

Integrating high-availability and efficiency As clusters are made up of a large number of components, the probability that a failure occurs in the system is not negligible at all. Thus, tolerating failures in a PSLS system becomes more and more important as the size and execution time of applications increase. In this paper we present a highly-available PSLS which smoothly integrates the high-availability support into the standard functioning of a PSLS without requiring any specific hardware. This integration enables to combine fault-tolerance and efficiency in failure-free executions which are two statements often considered as contradictory. First the support for high-availability takes benefit of the standard features thus decreasing the additional cost and complexity traditionally inherent to any high-availability mechanism. Second, high-availability features are exploited in order to improve the standard functioning during failure-free runnings. In particular, our performance results show that using PFS mirroring implemented for high-availability purposes improves significantly the number of local accesses during failure-free executions. This paper presents the whole PSLS system and its underlying philosophy. We especially focus on the integration of standard and high-availability features and on the PFS management, the SVM part having been studied in previous papers [10, 17].

Roadmap The remainder of this paper is organized as follows. Section 2 presents the design guidelines of our highly available PSLS. Section 3 presents our system in failure-free executions, including the checkpoint mechanism and Section 4 depicts the rollback of the system in the case of transient and permanent failures. Section 5 concludes. We have implemented a prototype of our system and results are depicted along the paper.

2 Smooth integration of standard and high-availability features

Our highly available PSLS system relies on a high degree of integration between standard and high-availability features. After addressing our fault-tolerance assumptions, we present our design guidelines and emphasize for each one in which way it is used for both standard and high-availability purposes.

2.1 Fault-tolerance assumptions

We are concerned with tolerating *(i)* transient failures, which do not involve the loss of memory contents, *(ii)* permanent node failures which involve the loss of both the memory content and the disk contents including the PFS part managed by the faulty node and *(iii)* power failures that might affect the whole cluster.

We consider a system of failure-independent fail-silent nodes connected by a reliable interconnection network. The permanent failure of a node component leads to the unavailability of the whole node. Our system relies on backward error recovery (BER) [12] which is a recovery technique well-suited to high performance parallel applications: a consistent system state, a checkpoint, is periodically snapshot and stored on stable storage and restored upon detection of a failure. A stable storage ensures that (1) data is not altered and remains accessible despite a failure (*permanence* property), and that (2) data is updated atomically in presence of failures (*atomicity* property).

The coherence of the checkpoint is ensured by an incremental global coordinated checkpointing policy where all nodes save simultaneously a checkpoint [2]. A two-phase commit protocol [7], where a previous checkpoint is invalidated only when the new one is validated, guarantees the atomic update of a checkpoint.

Terminology The data stored in memory belongs to one of the following categories:

- Active data is data used for computation and does not belong to a checkpoint.
- Readable recovery data represents recovery data not modified since the last checkpoint, it remains readable and can be used for standard execution as long as it is not modified. Upon modification it becomes pure recovery data.
- Pure recovery data represents recovery data, no longer usable for standard execution, this data is restored in the event of failure.

2.2 Stable storage implemented without specific hardware

In our highly available PSLS, no specific hardware is required to ensure the persistence of recovery data and this keeps the fault-tolerance mechanism to a reasonable cost. We exploit the fact that nodes are failure-independent to implement a stable storage in standard support storage both at memory and disk levels by replicating every checkpointed page in two distinct nodes.

2.3 Combining replication for efficiency and high-availability

Despite the fact that efficiency and high-availability are somewhat contradictory, they rely on the same mechanism namely *replication*: replication is used in SVM systems to exploit data locality and distribute the load between nodes and is intrinsic to any high-availability mechanism. We widely exploit this commonality in our system.

At the memory level, already existing replication, implemented by the SVM is exploited at checkpointing time to avoid replication of recovery data and data transfers across the network and conversely, created recovery data can be used afterwards to anticipate page faults in failure-free executions.

Likewise, at the PFS level, page mirroring, required to tolerate the lost of a disk, is used during failure free executions to increase the probability of local accesses. Each page stored in the PFS exists in two copies in two distinct nodes: the primary and the mirror copies. Both copies can be used to serve files accesses. How pages are mirrored on the PFS is detailed in Section 3.

2.4 Injection mechanism

To ensure as much efficiency as possible, we use a COMA[2]-like feature to limit data transfers between disks and memory. In a COMA, the last copy of a data should not be discarded from physical memory and is thus *injected* in the memory of a remote node when it is replaced. Our PSLS implements a software injection mechanism to delay as long as possible expensive disk write operations. Instead,

[2] COMAs are hardware cache-based systems.

data selected to be evicted from a local memory is preferably injected in the memory of a remote node rather than being written back onto disk.

Likewise this injection mechanism implemented for efficiency in our standard PSLS is also used to handle replacement of readable recovery data.

2.5 Multiple use of location function

Our SVM in based on a statically distributed directory [13]. For each page, an SVM *manager* is statically defined by using a simple modulo function (page number mod number of nodes). The SVM manager of a page stores the identity of the page owner[3] and receives and serves page faults related to that page. The SVM manager of a page does not necessarily store the page contents locally but is always able to locate a copy whereas the primary PFS manager of page is the node storing the primary copy of a page. The same modulo function is used to distribute the primary copies of the PFS on different nodes.

We have proposed another function, used in coordination with the modulo function *(i)* to reconfigure the SVM management upon detection of a permanent failure, *(ii)* to choose the location of the mirror PFS pages and to access them afterwards, and *(iii)* to reconfigure the PFS storage and management in the event of a permanent failure.

3 Failure-free execution

3.1 Data Management in a PSLS

Our PSLS system defines a single global address space which includes both the memory and PFS pages. Two kinds of pages are distinguished: mapped and volatile pages. Volatile pages are allocated in the SVM memory only, whose life time is the duration of the computation. Such pages do not have any counterpart in the PFS disks. They may be swapped on disk when evicted from memory. Mapped pages are mapped in the SVM from a parallel file. Such pages have corresponding disk copies in the PFS disks. A page is clean if its copies in memory are identical to the disk copy. If

[3]The page owner is the node having an up-to-date copy of the page and keeping track of the replicas location.

the disk copy of a page is not up-to-date (the up-to-date copy being in memory), this page is called a dirty page.

Data replication in an SVM leads to the presence of several copies of a page in different memories. A coherence protocol managing both mapped and volatile pages is implemented in order to ensure the consistency of multiple copies. Our PSLS is based on the sequential consistency model [5] implemented with a write-invalidate protocol.

Each node is equipped with two disks.

1. The PFS disk is used to store user files and is managed by the PFS part of the PSLS system.
2. The system disk is viewed as an extension of the local memory and is composed of two distinct areas:
   - The checkpoint area consists itself of two zones: the memory area is used to store pure recovery copies of (volatile or mapped) pages that belong to the current checkpoint; the permanent area is used to store permanent recovery copies of volatile pages (see Section 3.3). The checkpoint area is never checked on a page reference during failure-free execution.
   - The swap area is used to swap active or readable recovery copies of volatile pages. Pages belonging to a mapped file are not swapped in the *system* disk when they are evicted from memory but copied to their disk counterpart in the PFS disks. The swap area is checked upon a page reference.

The interface between the SVM and the PFS is file mapping which makes disk accesses transparent to the programmer. Files are not accessed using standard and complex input/output operations but by direct read and write operations in virtual memory. A memory area, called mapping area, is allocated to store file data. The PFS primary manager of a page stores the primary copy in its PFS disk and is in charge to serve requests regarding this page. When a processor first references a data in the mapping area, a page fault occurs and the corresponding data is automatically loaded from disk. Disks writes only occur in the event of a modified page replacement or at the end of the application. A memory page granularity is used by the PFS to access disks. In the SVM, the nodes memories

are used as large caches. Pages, which represent the unit of transfer and coherence in an SVM are transparently replicated in the node memory of the processor which references them. The owner of a page owns a page copy in its memory and its SVM manager is a node statically designated to serve requests regarding this very page.

As we previously mentioned, a software injection mechanism is provided in order to delay swapping to disk which is an expensive operation. The injection mechanism is used to keep as long as possible a useful page in the SVM. Thus if an injected page is used afterwards, it is loaded from a remote memory rather than from the local disk.

Figure 1 represents the architecture of a 2-nodes PSLS. Pages A, B represent volatile pages whereas pages 1, 2, ... represent PFS pages mapped into the global address space. Both type of pages cohabit in the global shared address space. Page A for example is physically mapped into the memory of node 0, whereas the page B is mapped, and present in the physical memory of the node 1. Nevertheless, these pages are virtually accessible by both nodes[4]. As regards to the PFS pages, page 1 is managed by the node 1 which is its primary PFS manager and is stored on its PFS disk, and is mapped in the global address space. Besides this page is also physically replicated in the memory of both nodes.

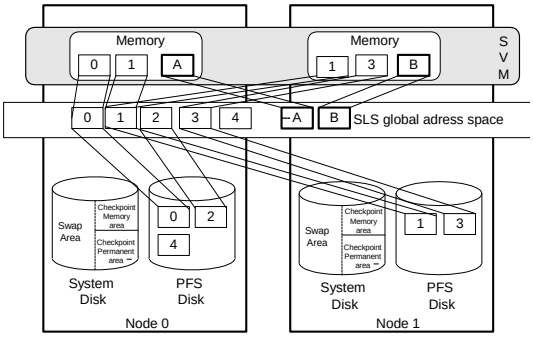

Figure 1: Example of a PSLS system

[4]Note that the SVM manager of pages 0, 2 and 4 is node 0 whereas node 1 manages 1 and 3.

3.2 Implementation

The considered mechanisms are illustrated along this paper with performance results obtained from an implementation of our PSLS. Our prototype has been implemented on a 4-nodes cluster of bi-processors running Linux based on the Scalable Coherent Interface (SCI) [8] interconnection technology. Nodes are based on Intel Pentium II (450 MHz) and have a 256 M-byte local memory. The SCI network has a latency of about 5 microseconds and a throughput of about 60 M-bytes per second. The PSLS implements a sequential consistency SVM and a PFS. The coherence management unit size in the SVM and the PFS striping unit size is equal to the size of the memory page (4KB). Performance results have been obtained from the execution of two applications: Modified Gram Schmidt (MGS) and matrix multiplication algorithms. The MGS algorithm produces from a set of vectors an orthonormal basis of the space generated by these vectors. We consider a base of 1024 vectors of 1024 double floats elements. The matrices used in the matrix multiplication algorithm contains 1024 x1024 double float elements. The SVM size for all the experiments is 64 M-bytes.

3.3 Checkpointing

Two types of checkpoints are considered in the system. A memory checkpoint consists in establishing a checkpoint in memories only. Such a checkpoint exploits efficient communication channel between nodes and avoids disk accesses for storing checkpoints. Saving as long as possible checkpoints in memory without any disk access, we keep the cost of a checkpoint reasonable. Nevertheless, this efficient implementation of a checkpoint is not sufficient to handle power failures. To this end, we define a permanent checkpoint where pages (mapped and volatile) are checkpointed on disk.

For efficiency reasons, permanent checkpoints are much less frequent than memory ones. Several memory checkpoints may occur between two permanent checkpoints. Upon detection of a failure, the last checkpoint is restored, whether it is a permanent or a memory one. Note that a permanent checkpoint invalidates the previous memory checkpoint whereas a memory checkpoint does not invalidate the previous permanent checkpoint. A memory and a permanent checkpoints may cohabit as long as the permanent is older.

In this section we successively present the memory and permanent checkpoints. We then address the issue of data replacement and the PFS mirroring mechanism.

3.3.1 Memory checkpoint algorithm

The memory checkpoint algorithm consists in ensuring that two copies of each page modified since the last checkpoint exist in two distinct node memories. The algorithm works as follows:

- The single memory copy of each dirty page unique in the SVM (unique either because the page is writable or because it is not yet replicated) is transformed into a readable recovery copy and a second copy is created in a distinct node. These copies remain readable and can be used afterwards during failure-free executions. Upon the first write access to a readable recovery copy, the two recovery copies are transformed into pure recovery copies no longer usable during failure-free execution. These copies are restored in the event of a failure.
- Two already existing copies of each dirty shared and already replicated page are transformed into readable recovery copies, thus avoiding page creation and transfer at checkpointing.

On one hand, this algorithm takes benefit of the replication inherent to the SVM by using data already replicated to avoid the need to create additional page copies to store recovery data. On the other hand, recovery data remains readable between two checkpoints as long as the corresponding page has not been modified since the last checkpoint. They can be used for anticipating page faults during failure-free executions. In [10], we showed indeed that using these copies, created for checkpointing purposes, later on has a great impact on the performance.

Figure 2, shows the percentage of recovery data that comes from the exploitation of already existing page copies in constrast to recovery data copies created at checkpointing time for various checkpoint frequencies for the MGS algorithm. The call to the checkpoint primitive is placed at the beginning of the main loop of MGS. So, the frequency is given as an interval between checkpoints expressed in number of vectors. A frequency of 250 means that a checkpoint is saved each time 250 vectors have been computed (every 4 seconds). We observe up to that more than half the recovery

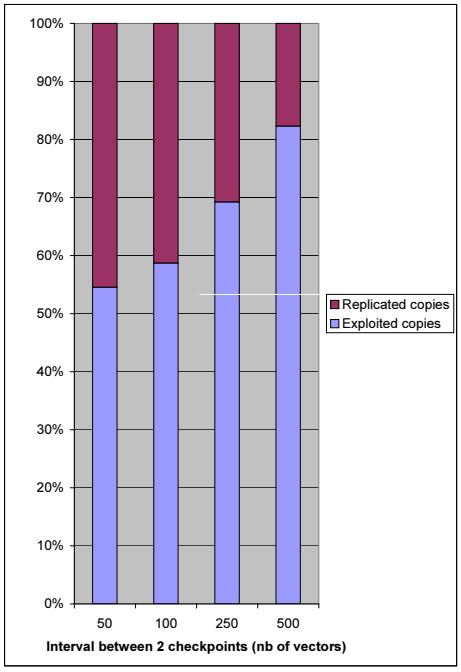

Figure 2: *Exploitation of existing page copies in the memory checkpoint algorithm for MGS*

copies comes from existing page copies. For a 500 checkpoint frequency, only 17% of recovery data copies are created at checkpointing time.

A complete description of the memory checkpoint algorithm can be found in [15].

It is not worthwhile keeping pure recovery copies in memory since they are useless for failure-free executions. In order to free the memories, they can be moved into the checkpoint memory area. Besides, dirty mapped pages (modified during their stay in memory) are injected rather than being written back onto PFS disks. This avoids to compromise the coherence of a checkpoint.

A memory checkpoint is composed of *(1)* Volatile and mapped recovery page copies in memory, *(2)* Volatile readable recovery page copies in the swap area of the system disks, *(3)* Volatile and mapped pure recovery copies in the checkpoint memory area of the system disks and, *(4)* Mapped pages on the PFS which do not have corresponding recovery data of previous type (1) and (3) in memory.

3.3.2 Permanent checkpoint algorithm

A permanent checkpoint algorithm consists in ensuring that two copies of every page (mapped or volatile) are present on two disks. The checkpointing algorithm follows a two-phase commit algorithm as well, thus ensuring the atomic update of the primary and the mirror copies of a page. The algorithm is presented in Algorithm 1 and works as follows: *(i)* recovery data associated to volatile pages are created in the checkpoint permanent area of two system disks since they do not have counterpart on the PFS disks, and *(ii)* recovery data corresponding to mapped pages are mirrored on the PFS disks.

Volatile pages are replicated onto different system disks using the injection mechanism (request to inject onto disk rather than onto memory). Each mapped page has to be replicated in two different PFS disks. The copy hold by the primary manager is used as a first replica and a mirror manager is chosen to host the mirror copy (see Section 3.3.3). A permanent checkpoint is composed of the volatile page recovery copies stored in the permanent zone of the checkpoint area, and of all pages stored on the PFS. This is actually in order to stick to this checkpoint composition that disrty mapped pages are injected rather than being written back onto disk. Table 1 summarizes the treatment applied to pages depending on their location in the system when a memory and a permanent checkpoint are established.

As depicted in Figure 3, the memory checkpoint algorithm is much more efficient than the permanent checkpoint algorithm. That is the reason why permanent checkpoints are much less frequent that memory checkpoints. This difference is due to the fact that memory checkpoints avoid at most data transfers on the network and do not perform any disk access whereas several remote disk accesses are performed requiring page transfers on the network during a permanent checkpoint.

3.3.3 Mirror function

Our goal while defining a PFS mirror manager is twofold: *(i)* being able to easily locate from a page number the mirror node of a page in order to send the request either to the primary manager or to the mirror manager and *(ii)* not attributing the same mirror manager for all the pages having the same manager in order to distribute the load if this initial manager fails permanently.

1 Permanent checkpoint (two-phase commit algorithm)

{This algorithm is performed on each node}

for each volatile active page *p* in memory or in swap area **do**

write_back(*p*, local permanent_checkpoint_area);

injection_disk(*p*, remote permanent_checkpoint_area);

{The remote system disk is on the same node as the one which would have been chosen for memory checkpoint (neighbor node in the implementation}

end for

for each page copy *p* in checkpoint memory area **do**

invalidation(p);

end for

for each readable recovery copy of volatile page *p* in memory or in swap area **do**

write_back(*p*, local permanent_checkpoint_area);

end for

for each dirty mapped page *p* in memory and each mapped page *p* having readable recovery copies in memory **do**

write_back(*p*, PFS_Primary_Manager(*p*));

write_back(*p*, PFS_Mirror_Manager(*p*));

end for

{The memory is not modified at all. This enables the application to carry on in the same configuration, no access pattern is lost due to the checkpoint}

Location	State of a page before a checkpoint	Treatment in a memory checkpoint	Treatment in a permanent checkpoint
Volatile data	active	readable recovery	copied to checkpoint permanent area
in memory	readable recovery	unchanged	copied to checkpoint permanent area
	pure recovery	discarded	discarded
Mapped data	clean active	readable recovery	unchanged
in memory	dirty active	readable recovery	written back and mirrored on PFS
	readable recovery	unchanged	written back and mirrored on PFS
	pure recovery	discarded	discarded
System disk	active in swap area	readable recovery	copied to checkpoint permanent area
	readable recovery in swap area	unchanged	copied to checkpoint permanent area
	checkpoint memory area	discarded	discarded
	checkpoint permanent area	unchanged	discarded
PFS		no action	no action

Table 1: Summary of operations for a memory and a permanent checkpoint

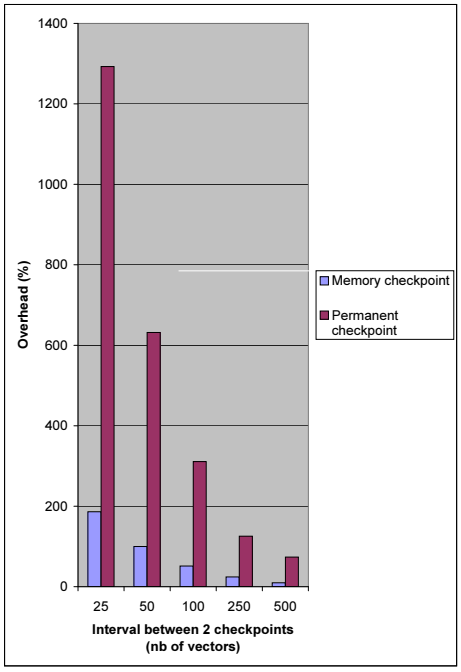

Figure 3: *Comparison between memory and disk checkpoints in MGS*

We use a function, called *PFS_Mirror_Manager*, to distribute uniformly the replication of pages previously managed by a faulty node. It was first defined in ICARE, a recoverable shared virtual memory system [10] and used to define a spare SVM manager in the event of a permanent failure.

When a page p is referenced and needs to be loaded in memory, the node originating the request can easily find the primary manager of the page by using a simple modulo function considering a system with N nodes, numbered from 0 to $N-1$:

$$PFS\_Primary\_Manager(p) = p \bmod N.$$

Let us consider that each node y manages P pages pages are $p_{0,...P-1}$ such as $p_k = y + kN,\ k \in \{0...P-1\}$. We want to mirror in an uniform way the P pages managed by y on the $N-1$ other nodes.

The mirror function must establish a one-to-one mapping between $\{0, 1, 2...y-1, y+1, ...N-2\}$ and $\{0, 1, 2...y-1, y+1, ...N-1\}$.

So, $PFS\_Primary\_Manager(p_k) = f[k \bmod (N-1)]$ with

$$f: \quad \{0,1,2...,N-2\} \overrightarrow{bij.} \overbrace{\{0,1,2,...,y-1,y+1,...N-1\}}^{(N-1) \text{ remaining nodes}}$$
$$k \bmod (N-1) \longmapsto f[k \bmod (N-1)]$$

That is to say $f(x) = (x + y + 1) \bmod N$. The mirror manager $PFS\_Mirror\_Manager(p)$ of a page p is then obtained by the following uniform redistribution function:

$$PFS\_Mirror\_Manager(p) = [k \bmod (N-1) + PFS\_Primary\_Manager(p) + 1] \bmod N$$

where $p = M(p) + kN$, is the address of the considered page, $M(p) \in \{0, ..., N-1\}$ is the primary manager of p, and $k = \frac{M(p)}{N}$ is the ordering number of the considered page in the page list initially managed by $PFS\_Primary\_Manager(p)$. The proof can be found in [10].

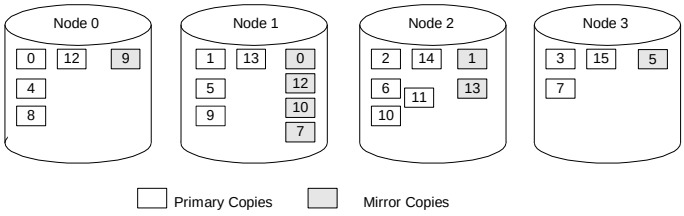

Figure 4: Exemple of Node 1 mirroring management

Figure 4 depicts the example of a 4-nodes system and 15 pages. In this picture, primary copies of all pages are represented. For the sake of clarity, we focus on Node 1 management. Node 1 is the primary manager of page 1, 5, 9, 13. Mirroring is uniformly distributed: Page 1 is mirrored on Node 2, page 5 on Node 3, page 9 on Node 0 and page 13 on Node 2. Besides Node 1 hosts mirror copies of pages 0, 12, 10 and 7. Other mirror copies are not depicted.

3.4 Using mirrored copies for PFS efficiency

Henceforth, using the *PFS_Mirror_Manager* function, it is as straightforward to find the primary manager of page *p* than to find its mirror manager. Each PFS page has a mirrored page. When a node initiates a request on a page *p*, it computes simultaneously the primary and the mirror managers of the page and sends the request to either one. If the node is itself one of the two nodes, the request is served locally. Enabling mirrored copies to be used to serve requests as well as primary copies increases by two the probability that a request is served locally. The impact of this optimization clearly depends on the access patterns of the application as regards to the PFS pages distribution among nodes.

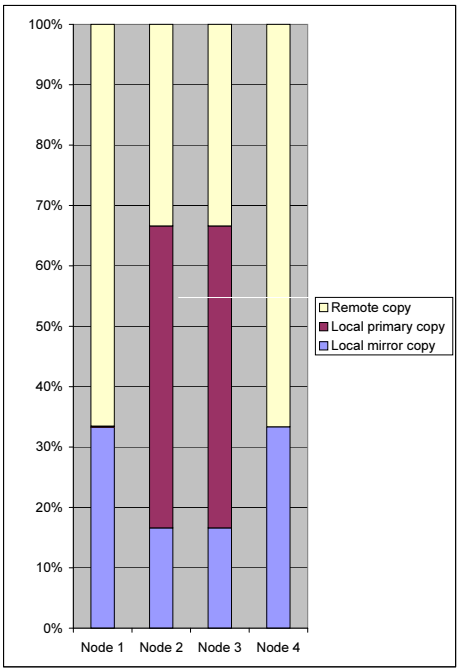

Figure 5: *Utilization of local primary and mirror disk page copies*

Figure 5 depicts the proportion of local accesses to the PFS, distinguishing between primary and mirror copies versus the number of remote accesses. This plot shows that the use of mirroring, im-

plemented for high-availability purpose, implies that 50 % in average of disk accesses are performed locally. More than 30% of the total number of accesses performed by some nodes are served using a local mirror copy in the MGS application.

4 Rollback recovery

4.1 Handling transient failures

When a transient failure occurs, the last checkpoint must be restored. If the last checkpoint is a memory checkpoint, pure recovery copies (from memory, swap, and checkpoint memory areas) are restored in readable recovery pages. All active copies are discarded, except clean copies of mapped pages that are still valid (the copy on disk is identical).

If a permanent checkpoint is restored, memories are emptied except for clean active mapped pages, checkpoint memory and swap areas are emptied, volatile page copies in the permanent area are copied back to memory whare they are transformed into readable recovery copies.

A summary of rollback operations depending on the page state and location is presented on Table 2. No further reconfiguration is needed.

4.2 Reconfiguration after a permanent failure

Once a permanent failure has been detected, the previous checkpoint must be restored the same way as in the case of a transient failure. However, the contents of the memory and disks of the faulty node have been lost. Thus the PSLS must be reconfigured as in order to be able to tolerate right away another failure. We assume in this section a crash-stop model where a node permanently crashed and never recovers. The reintegration of a repaired node is tackled in the next section. At the end of the rollback, each page should have two readable recovery copies. The aim of the reconfiguration is to duplicate lost data which was located on the faulty node so that the persistence property is satisfied again. Algorithms 2 and 3 present the whole reconfiguration process.

Location	State of a page before a rollback	Rollback to a memory checkpoint	Rollback to a permanent checkpoint
Volatile data in memory	active	discarded	discarded
	readable recovery	unchanged	discarded
	pure recovery	readable recovery	discarded
Mapped data in memory	clean active	unchanged	unchanged
	dirty active	discarded	discarded
	readable recovery	unchanged	discarded
	pure recovery	readable recovery	discarded
System disk	active in swap area	discarded	discarded
	readable recovery in swap area	unchanged	discarded
	checkpoint memory area	readable recovery	discarded
	checkpoint permanent area	unchanged	readable recovery
PFS		no action	no action

Table 2: Summary of rollback

SVM Reconfiguration In the SVM, each node is the *manager* of a statically defined set of pages and manages for each page a directory entry containing the identity of its owner and the replicas location. A new recovery copy must be created for each page which had one of its two recovery copies located in the faulty node. Moreover, a spare SVM manager has to be defined for pages previously managed by the faulty node. The same function as the *PFS_Mirror_Manager* is used to define a spare manager.

PFS Reconfiguration From the PFS point of view, the faulty node acted as a primary manager and a mirror manager for two different sets of pages. A new function has to be applied to define a *PFS_(mirror)*^{2}*_Manager*. At reconfiguration time, each node checks if it stores on its PFS disk a page for which either the *PFS_Primary_Manager* or the *PFS_Mirror_Manager* is the faulty node. To define a *PFS_(mirror)*^{2}*_Manager* we iterate on the *PFS_Mirror_Manager* function.

PFS_(mirror)^2_Manager The *PFS_(mirror)^2_Manager* function must have the ability for a given page to avoid both the faulty node (either the primary manager or the mirror manager) and the node holding the other disk copy of the page.

Consider the following situation where z becomes faulty, all the pages present on the PFS disk of each node y must be considered, namely:

- the pages p for which the *PFS_Primary_Manager* is z and y is the *PFS_Mirror_Manager*: $p = kN + z$ and $PFS\_Mirror\_Manager(p) = y$
- the pages p for which z is the *PFS_Mirror_Manager* and y is the *PFS_Primary_Manager* : $p = kN + y$ and $PFS\_Mirror\_Manager(p) = z$

The same *PFS_Mirror_Manager* function is applied to find a spare (primary or mirror) manager but since two nodes must be ignored (the node itself y and the faulty node z), the applied modulo is $(N - 2)$.

The function $PFS\_Mirror\_Manager_z^2(p) = [k \bmod (N-2) + z + 1] \bmod N$ enables us to dismiss z and $(z - 1)$ whereas we want to dismiss z and y. To achieve this, 1 is added when $PFS\_Mirror\_Manager_z^2(p)$ belongs to the interval $\{y..(z-2)\}$.

$$PFS\_Mirror\_Manager_{y,z}^2(p) =$$
$$PFS\_Mirror\_Manager\_z^2(p) + 1_{y,(z-2)}[PFS\_Mirror\_Manager\_z^2(p)]$$

where $PFS\_Mirror\_Manager_z^2 = [k \bmod (N-2) + z + 1]$

and $1_{y,(z-2)}[PFS\_Mirror\_Manager_z^2(k) = 1$ if $PFS\_Mirror\_Manager_z^2(k) \in \{y...(z-2)\}$ else 0.

Figure 6 and Table 3 depict an example showing the results of Node 1 reconfiguration after the permanent failure of Node 2. As a primary manager, each node checks if the mirror node of its primary copies is the faulty one. As a mirror manager, each node checks whether the primary manager of its mirrored copies is the faulty node. For every page which primary or mirrored node is faulty, the node has to define a (mirror)^2 manager and replicate the page on this node. In the

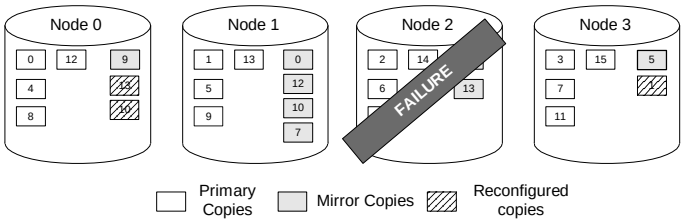

Figure 6: Node 1 reconfiguration after the failure of Node 2

Page number $p = m + kN$	PFS_Primary_Manager m	PFS_Mirror_Manager	PFS_Mirror^2_ Manager
1	1 = **1** + 0*N	0 + 1 + 1 = **2**	0 + 2 + 1 + 0 =**3**
13	13 = **1** + 3*N	0 + 1 + 1 = **2**	1 + 2 + 1 + 0 = **0**
10	5 = **2** + 2*N	2 + 2+ 1 = **1**	0 + 2 + 1 +0 = **3**

Table 3: PFS managers for pages managed by node 1 and affected by node 2 failure

example, Node 1 has to take care of pages 1 and 13 as a primary manager, whose mirror manager was Node 2, they are respectively replicated on nodes 3 and 0, and of page 10 as a mirror manager, primary manager of page 10 being Node 2. Page 10 is replicated on Node 3.

4.3 Reintegrating a repaired node

We consider the situation where a system has been victim of a permanent failure, has been reconfigured according to the previous algorithm and where the faulty node is subsequently reintegrated in the system.

The system must be reconfigured such as the faulty node takes back its function as an SVM manager and PFS manager (primary and mirror). Each node checks its (SVM and PFS) directories and for each page it manages as a *PFS_Mirror^2_Manager* or a SVM spare manager for the account of the reintegrated node, moves the directory information to the recovering node. No further action is taken as regards to the memory contents of the SVM since the memory is going to be filled out from

2 Memory and disk reconfiguration in case of permanent failure

{This algorithm is applied locally on node n. The faulty node is f.}

{Memory reconfiguration}
if (last_checkpoint == memory_checkpoint) **then**
  for each page p in memory **do**
    if (p== pure_recovery_page) **then**
      p = readable_recovery
    else
      if ((p != recovery_page) && (p != clean mapped page)) **then**
        invalidation(p)
      end if
    end if
  end for
  for each volatile page p in swap area **do**
    if (p== readable_recovery_page) **then**
      Copy back p to memory
    else
      invalidation(p);
    end if
  end for
  for each page p in the checkpoint memory area **do**
    p = readable_recovery_page;
    copy back p to memory
  end for
end if

3 Memory and disk reconfiguration in case of permanent failure (continued)

```
if (last_checkpoint == permanent_checkpoint) then
  for each page p in memory do
    if (p!= clean mapped page) then
      invalidation(p);
    end if
  end for
  for each volatile page p in swap area do
    invalidation(p);
  end for
  for each page p in the checkpoint memory area do
    invalidation(p);
  end for
  for each page p in the checkpoint permanent area do
    copy_back_into_memory(p);
  end for
end if
{SVM reconfiguration (replication of lost pages and spare manager )}
for each page p (mapped or volatile) do
  if (recovery-replica belongs to f) then
    replication(p, remote memory);
  end if
  if ((manager(p) == f) && (n == p.owner)) then
    new_manager = spare_manager(p);
  end if
  updateManager(p.owner, new_manager);
  {Update the new manager of the page with owner information}
end for
{PFS reconfiguration}
for each page p on the PFS disk do
  if ((PFS_Primary_Manager(p)== f) or (PFS_Mirror_Manager(p)==f)) then
    PFS_Mirror2_Manager == PFS_Mirror2_Manager(p);
    Mirror (p, PFS_Mirror2_Manager);
  end if
end for
```

the application accesses. From the PFS point of view, each node acting as a *PFS_Mirror*^{2}*_Manager* has to replicate this page on the recovering node ansd dismisses it locally.

4.4 Power failure

Recovery after a power failure is performed from last permanent checkpoint. The memory is emptied and filled from the permanent checkpoint area of the system disk for volatile pages and from the PFS regarding mapped pages. It is assumed that the private state of each process of the application has been checkpointed atomically with PSLS checkpoint.

5 Conclusion and related work

Very few other work has been done on file mapping in SVM with a PFS. [14] is one representant but this system is not designed to tolerate node failures. Several recoverable shared virtual memory systems have been proposed [18]. However, to our knowlegde, all these memory management systems do not consider issues related to the interactions between the memory management system and a (parallel) file system. In the system described in [21], memory checkpoints are also proposed in addition to permanent checkpoints for efficiency reasons. However, in contrast to our system, recovery data stored in memory cannot be used as read-only data for the computation even if their contents is identical to active copies contents. Moreover, a permanent checkpoint is always preceded by a memory checkpoint, the latter being copied to disk in background. [4] describes a reliable remote paging mechanism for a network of workstations relying on a parity mechanism to tolerate node failures. Data redundancy is not exploitable for failure-free execution whereas the use of recovery pages (memory or disk) is one of the major contribution of our system. GMS [6] also globally manages the cluster memory resource. However, a modified page cannot be exported to a remote node without being first copied into the local swap disk in order to tolerate node failures.

XFS [20] is a highly available parallel file system which implements cooperative caching. In contrast to our system, memory and disk management is not fully integrated resulting in a worse usage of the cluster memory resource. Moreover, XFS provides a standard read/write interface and

implements RAID-5 rather than mirroring to ensure the high availability of files. To efficiently implement a distributed RAID-5 mechanism, complex mechanisms (management of write logs) are needed. The load generated by read operations cannot be balanced on several nodes as it is possible in a system implementing mirroring.

In this paper we have presented an high-availability support to enable a PSLS to tolerate multiple transient, unique permanent and power cut failures without requiring any specific hardware. Every single feature in this system is based on reusability and integration. First of all, stable storage in implemented in memory by replicating recovery copies in two distinct memories and at the PFS level with mirroring. SVM replication is used to avoid the creation of recovery data and checkpointing replication in the SVM or PFS is exploited afterwards for standard failure-free execution, thus increasing the load-balancing and efficiency. We implement a two-level checkpointing algorithm: a memory checkpoint is established very efficiently and a permanent checkpoint is established on a much lower frequency basis but enables to tolerate permanent cut failures. Moreover a permanent checkpoint can also be used when memories are saturated to clean the memories [17]. Finally, another contribution is the use of a function, used in conjunction with the modulo function which ensures a well-balanced PFS mirroring mechanism and which is also used to reconfigure the SVM in a balanced way when a permanent failure occurs. It is also possible to iterate this function to reconfigure the PFS in the event of a permanent failure.

We have implemented a prototype of our highly-available PSLS and results show that the integration of standard and high availability supports results in a very efficient system. In particular, the number of accesses to PFS pages are performed locally doubles in average, due to the exploitation of copies mirrored for the fault-tolerance purposes. Future work includes the study and experimentation of a larger set of memory hierarchy management strategies as well as a complete rollback implementation including the processes private context. We also plan to integrate the studied mechanisms in Gobelins, a distributed operating system designed to give the illusion of a virtual shared memory multiprocessor on top of a cluster [16].

References

[1] 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*, pages 18–28, February 1996.

[2] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3 (1):63–75, February 1985.

[3] P.F. Corbett and D.G. Feitelson. Design and implementation of the Vesta parallel file system. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 63–70, 1994.

[4] E. Marcatos G. Dramitinos. Adaptative and reliable paging to remote main memory. *Journal of Parallel and Distributed Computing*, 1999. to appear.

[5] M. Dubois, C. Scheurich, and F.A. Briggs. Synchonization, coherence and event ordering in mu ltiprocessors. *IEEE Computer Survey, Tutorial Series*, February 1988.

[6] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, and H. M. Levy. Implementing global memory management in a workstation cluster. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP-15)*, pages 201–212, December 1995.

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

[8] D. B. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro*, pages 10–21, February 1992.

[9] J.V. Huber, C.L. Elford, D.A. Reed, and A.A. Chien. PPFS: A high performance portable parallel file system. In *Conference proceedings of the 1995 International Conference on Supercomputing*, pages 385–394. ACM Press, July 1995.

[10] A.-M. Kermarrec, C. Morin, and M. Banâtre. Design, implementation and evaluation of icare: an efficient recoverable dsm. *Software Practice & Experience*, 28(9), July 1998.

[11] P. J. Leach, P. H. Levine, J. A. Douros, B. P. a nd Hamilton, D. L. Nelson, and B. L. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications*, 1(5):842–857, November 1983.

[12] P.A. Lee and T. Anderson. Dependable computing and fault-tolerant systems, v ol. 3. In J.C. Laprie A. Avizienis, H. Kopetz, editor, *Fault Tolerance : Principles and Practice*. Springer Verlag, New York, 1990.

[13] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[14] Qun Li, Jie Jing, and Li Xie. BFXM: A parallel file system model based on the mechanism of distributed shared memory. *ACM Operating Systems Review*, 31(4):30–40, October 1997.

[15] C. Morin, A.-M. Kermarrec, M. Banatre, and A. Gefflaut. An efficient and scalable approach for implementing fault tolerance architectures. *IEEE Transactions on Computers*, 49(5):414–430, May 2000.

[16] C. Morin and R. Lottiaux. Global resource management for high availability and performance in a dsm-based cluster. In *Proc. of 1st workshop on Software Distributed Shared memory*, June 1999.

[17] C. Morin, R. Lottiaux, and A.-M. Kermarrec. High-availability of the memory hierarchy in a cluster. In *19th IEEE Symposium on reliable Distributed Systems*, pages 134–143, Nurnberg, Germany, October 2000.

[18] C. Morin and I. Puaut. A survey of recoverable distributed shared memory systems. *IEEE Transactions on Parallel and Distributed Syst ems*, 8(9), September 1997.

[19] N. Nieuwejaar and D. Kotz. The Galley parallel file system. In *Proceedings of the 10th International Conference on Supercomputing*, pages 374–381, August 1996.

[20] T. Anderson M. Dahlin J. Neefe D. Patterson D. Roselli R. Wang. Serverless network file systems. In *proc. of 15th ACM Symposium on Operating Systems Principles*, December 1995.

[21] T.J. Wilkinson. *Implementing Faut-Tolerance in a 64 bit distributed operating system*. PhD thesis, City University, London, 1993.

Unité de recherche INRIA Rennes

IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifi que

615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur

INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)

http://www.inria.fr

ISSN 0249-6399