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

Anne-Marie Kermarrec[1] and Christine Morin[2]

[1] Microsoft Research
St George House, 1 Guildhall StreetCambridge CB2 3NH, UK
Tel: +44 1223 724 823, Fax: +44 1223744 777
annemk@microsoft.com
[2] Irisa/Université de Rennes 1
Campus Universitaire de Beaulieu 35042 Rennes Cedex, France
Tel: +33 2 99 84 72 90, Fax: +33 2 99 84 71 71
cmorin@irisa.fr

Abstract. A parallel single level store (PSLS) system integrates a shared virtual memory and a parallel file system representing an attractive support for long running parallel applications in a cluster. In this paper we present the smooth integration of a backward error recovery high-availability support into a PSLS system. Our highly-available PSLS system relies on a high degree of integration and re-usability between high-availability and standard supports. We focus on the parallel file system management at checkpointing and recovery time. A prototype has been implemented and we show some performance results.

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. To cope with this twofold requirement, namely the shared memory abstraction and a large and efficient file system, parallel single level store systems (PSLS) [5], which integrate a shared virtual memory (SVM) [1] and a parallel file system (PFS) [12] are very well-suited for the execution of high performance applications in a cluster.

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 such as all operations, including PFS ones, are made using standard memory reads and writes. Concurrent accesses to the same file data are automatically handled by the SVM coherence protocol.

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

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 752–763, 2001.

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 high-availability support 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 to improve the standard functioning during failure-free executions. The remainder of the paper is organized as follows: we present the design guidelines of our highly available PSLS in Section 2 and the system itself in failure-free executions (Section 3) and at rollback (Section 4). Section 5 concludes. Results are depicted along the paper.

2 Smooth Integration of Standard and High-Availability Features

2.1 Fault-Tolerance Assumptions

Our system is able to tolerate *(i)* multiple transient failures, which do not involve the loss of memory contents, *(ii)* a single permanent node failure involving the loss of both the memory contents 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. Our system relies on backward error recovery (BER) [6]: a consistent system state, a checkpoint, is periodically snapshot and stored on stable storage and restored upon detection of a failure. The coherence of the checkpoint is ensured by an incremental global coordinated checkpointing policy where all nodes save simultaneously a checkpoint. A two-phase commit protocol guarantees the atomic update of a checkpoint.

2.2 Design Guidelines

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.

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.

To ensure as much efficiency as possible, the SVM part of the PSLS implements a software injection mechanism to delay as long as possible expensive disk write operations. Instead, 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.

Our SVM in based on a statically distributed directory [7]. For each page, an SVM *manager* is statically defined by using a simple modulo function. The SVM manager of a page is always able to locate a copy. The primary PFS manager of a page is also 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. **Volatile** pages are allocated in the SVM memory only, their 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 and **dirty** if the disk copy of a page is not up-to-date, Data replication in an SVM leads to the presence of several copies of a page in different memories. A sequential consistency model [2] implemented with a write-invalidate protocol managing both mapped and volatile pages is implemented in order to ensure the consistency of multiple copies.

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: *(i)* 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 3.2). The checkpoint area is never checked on a page reference during failure-free execution; *(ii)* The **swap area** is used to

swap active[1] or readable recovery[2] 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 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. Figure 1 represents the architecture of a 2-nodes PSLS.

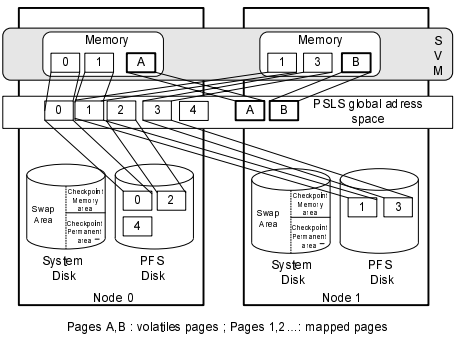

Fig. 1. Example of a PSLS system

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 dual-processors running Linux based on the Scalable Coherent Interface (SCI) [3] 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 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

[1] Active data is data used for computation and does not belong to a checkpoint

[2] 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.

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.

3.2 Checkpointing

Two types of checkpoints are considered in the system. A **memory checkpoint** consists in establishing a checkpoint in memories only. 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 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. A memory and a permanent checkpoints may cohabit as long as the permanent one is older.

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 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. A complete description of the memory checkpoint algorithm can be found in [9]. In our experiments of the MGS algorithm, between 55% and 83% of recovery pages comes from existing page copies.

Since it is not worthwhile keeping pure recovery copies in memory since they are useless for failure-free executions, they can be moved into the checkpoint memory area. A memory checkpoint is thus 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.

Permanent Checkpoint Algorithm A permanent checkpoint algorithm consists in ensuring that two copies of every page are present on two disks. The checkpointing algorithm follows a two-phase commit algorithm, 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. 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 dirty mapped pages are injected rather than being written back onto disk.

1 Permanent checkpoint (two-phase commit algorithm) 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 contents 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}
```

The memory checkpoint algorithm is much more efficient than the permanent checkpoint algorithm (approximatively 6 times less overhead). This difference is due to the remote disk accesses performed during a permanent checkpoint.

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 primary manager in order to distribute the load if this initial manager fails permanently.

We use a function, called *PFS_Mirror_Manager*, to distribute uniformly the replication of pages previously managed by a faulty node. 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 $N-1$: $PFS\_Primary\_Manager(p) = p \bmod N$ as well as its mirror manager using the following 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 $PFS\_Primary\_Manager(p)$. This function ensures a uniform mirroring of pages managed by a node between the remaining nodes. The proof can be found in [4]. Figure 2 depicts the example of a 4-nodes system and 15 pages.

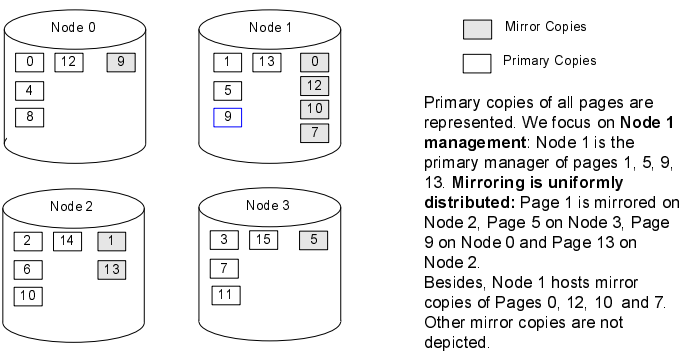

Fig. 2. Example of Node 1 mirroring management

3.3 Using Mirrored Copies for PFS Efficiency

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

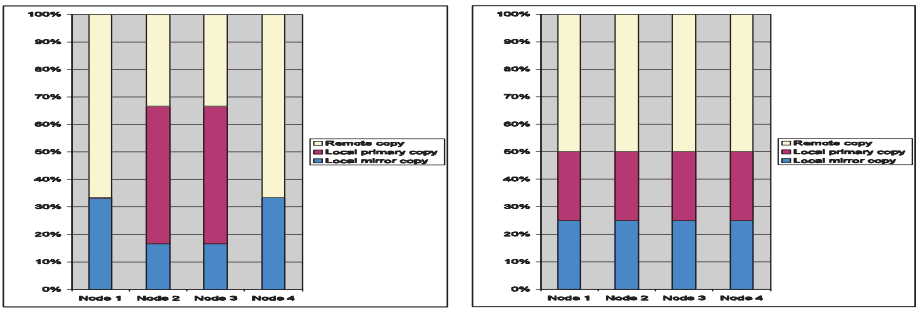

Fig. 3. Utilization of local primary and mirror disk page copies in MGS and matrix multiplication

this optimization clearly depends on the access patterns of the application as regards to the PFS pages distribution among nodes.

Figure 3 depicts the proportion of local accesses to the PFS, distinguishing between primary and mirror copies versus the number of remote accesses. These plots show that the use of mirroring, implemented for high-availability purpose increases significantly the number of local disk accesses. 50 % in average of disk accesses are performed locally both for MGS and matrix multiplication.

4 Rollback Recovery

Handling Transient Failures When a transient failure occurs, the last checkpoint must be restored. If it is a memory checkpoint, pure recovery copies are restored in readable recovery pages. All active copies are discarded, except clean copies of mapped pages that are still valid.

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 where they are transformed into readable recovery copies.

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 a crash-stop model where a node permanently crashed and never recovers. 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.

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}. 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} we iterate on the *PFS_Mirror_Manager* function.

PFS_mirror^2*_Manager* The *PFS_mirror*^{2} function must have the ability for a given page to avoid both the faulty node 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 to define PFS_Mirror^2 is the following:

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

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

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. [8] is one such system but it is not designed to tolerate node failures. Several recoverable

2 Memory and disk reconfiguration after a permanent failure applied locally on node n

```
{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
      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
```

shared virtual memory systems have been proposed [11]. However, to our knowledge, all these memory management systems do not consider issues related to the interactions between the memory management system and a file system.

XFS [13] 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 are needed.

Our highly-available PSLS tolerates multiple transient, unique permanent and power cut failures without requiring any specific hardware. Every single feature in this system is based on re-usability and integration. 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 [10]. Another contribution is the use of a function, used in conjunction with the modulo function which ensures a well-balanced PFS mirroring mechanism. 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.

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
  update_Manager(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_Mirror^2_Manager == PFS_Mirror^2_Manager(p);
    Mirror (p, PFS_Mirror^2_Manager);
  end if
end for
```

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. 752
2. M. Dubois, C. Scheurich, and F. A. Briggs. Synchronization, coherence and event ordering in multiprocessors. *IEEE Computer Survey, Tutorial Series*, February 1988. 754
3. D. B. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro*, pages 10–21, February 1992. 755
4. 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. 758
5. P. J. Leach, P. H. Levine, B. P. Douros, J. A. 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. 752
6. P. A. Lee and T. Anderson. Dependable computing and fault-tolerant systems, vol. 3. In J. C. Laprie A. Avizienis, H. Kopetz, editor, *Fault Tolerance : Principles and Practice*. Springer Verlag, New York, 1990. 753
7. K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989. 754

8. 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. 760
9. C. Morin, A.-M. Kermarrec, M. Banâtre, and A. Gefflaut. An efficient and scalable approach for implementing fault tolerance architectures. *IEEE Transactions on Computers*, 49(5):414–430, May 2000. 756
10. 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. 761
11. C. Morin and I. Puaut. A survey of recoverable distributed shared memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(9), September 1997. 761
12. N. Nieuwejaar and D. Kotz. The Galley parallel file system. In *Proceedings of the 10th International Conference on Supercomputing*, pages 374–381, August 1996. 752
13. 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. 761