

Global Resource Management for High Availability and Performance in a DSM-based Cluster

Christine Morin
IRISA-INRIA

Renaud Lottiaux
IRISA

Campus universitaire de Beaulieu 35 042 Rennes cedex (FRANCE)
{cmorin,rlottiau}@irisa.fr

Abstract

High availability and performance are two desirable properties for the execution of long-running parallel scientific applications on software DSM based clusters. Global resource management in the operating system is a way to achieve these properties. To illustrate this approach, a system integrating a paged-based shared virtual memory and a parallel file system for global management of memory and disk resources is presented. Main design issues include the optimization of disk accesses in the context of a single level storage system and fault tolerance.

1 Introduction

Due to the increasing power of microprocessors and the evolution of network technology, clusters of SMPs are now attractive for executing high performance parallel applications such as numerical simulation applications. The simplicity of the shared memory model makes it popular for programming these applications. The shared memory programming model can be supported on clusters by the implementation of a Shared Virtual Memory (SVM) [16].

Our goal is to design and implement operating system mechanisms for better resource management in a homogeneous cluster of SMPs interconnected by a System Area Network to provide both high performance and high availability to parallel applications. We adopt a global approach in resource (processor, memory, disk) management, claiming that the success of clusters strongly relies on the capacity of global resource management. A cluster user should indeed see it as a single shared memory machine rather than as a set of machines having their own resources. Although memory management (remote paging [8, 7], SVM [16, 1, 20]), disk management (RAID [5], parallel or distributed file system [6, 21, 23]), processor management (process migration [24, 22]) have already been extensively studied, very few work has been done for global resource management in which these mechanisms *cooperate* or *are integrated* towards the objective of executing high performance applications. Moreover, we are convinced that high availability is a key prop-

erty to be ensured in a cluster where resources used by a process are distributed in several nodes. It is thus essential to tolerate node failures during the execution of an application with minimal performance degradation.

Several advantages follow from our approach. Common mechanisms are only implemented once in the operating system. For example, process checkpointing and process migration mechanisms both require the extraction of a process state, the former for saving it in stable storage, the latter for sending it to a remote node over the network. Global resource management can also avoid to take conflicting decisions in the system. For example, taking into account both memory and processor loads in a process migration mechanism can avoid a negative impact of a process migration on the memory system (as would be the case if a process is migrated to a node executing only a single process with large memory requirements, the migrated process would generate many page faults). Finally, information known by a subsystem can be used to optimize another subsystem. For example, the page state (modified or not) managed by a SVM can be exploited by a checkpointing mechanism to only save modified data when a process checkpoint is established.

To illustrate our approach, we focus in this paper on global memory and disk management. In fact, we propose a system in which a SVM is integrated with a Parallel File System (PFS). This system is not only designed to achieve high performance but also to tolerate node failures. With this example, we want to show that substantial benefits may be obtained by integrating a software DSM with other operating system mechanisms.

The remainder of this paper is organized as follows. In Section 2, our system integrating a SVM and a PFS is presented. Performance issues are discussed in Section 3. Section 4 is devoted to fault tolerance issues. Related work is discussed in Section 5. Section 6 concludes.

2 Integrating a Shared Virtual Memory and a Parallel File System

2.1 Motivation

Global memory and disk management in our system has led to the design of a one level storage system [14] built from a page-based SVM system and a PFS. A PFS is indeed desirable for the execution of high performance applications in a cluster as it provides increased disk bandwidth by fragmenting a file on several disks, allowing parallel accesses to the fragments. In contrast to the *read/write* interface which is traditional in PFS, file mapping in SVM is the natural

interface of the PFS in the system we propose. This kind of interface has several advantages. File accesses are implicitly performed by memory operations. Hence, no parallel file pointer needs to be managed. The coherence of concurrent accesses to the same file are automatically managed by the SVM. Data being automatically loaded in the memory of the processor accessing them, the programmer does not have to manage data distribution. As PFS (client and disk) caches and the SVM are merged, their size is dynamically and automatically adjusted depending on memory usage. The use of prefetching allows automatic overlap of computing and data loading. Programming parallel out-of-core applications is easier since files are simply viewed as another memory level.

2.2 Overview

In the proposed system, nodes' memory and disks are managed in a global and integrated way by a SVM system merged with a PFS. File mapping is the natural PFS interface and makes disk accesses transparent to the programmer. With file mapping in virtual memory, files are not accessed using standard input/output operations but by direct read and write operations in virtual memory. A memory area is allocated to store file data. 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 page replacement or at the end of the application.

Our system is based on a COMA-like memory management. The nodes' memories are used as large caches. Pages, which represent the unit of transfer and coherence, are automatically and transparently migrated or replicated in the memory of the node of the processor which references them. When a page has been loaded in memory, existing copies are used to serve subsequent page faults thus avoiding disk accesses.

Our system is based on a write-invalidate coherence protocol similar to Berkeley protocol [10] implemented by statically distributed directories. The *owner* of a page is distinguished from its *manager*. 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. The owner of page p owns an up-to-date copy of a page and is the only node allowed to give access rights to this page. It also manages the copyset of the page p , *i.e.* the list of nodes holding a copy of this page. The copyset is managed by the owner rather than the manager as it allows a more efficient implementation of fault tolerance mechanisms as well as efficient management of page replacement. The manager of a page is also responsible for the serialization of requests.

As PFS caches and write buffers are merged with the SVM, page states are used not only to manage page sharing as in a traditional SVM but also to manage their status regarding the file system (prefetched, in cache, in write buffer...). Hence, the Berkeley protocol is extended with new states associated with data managed by the PFS part of the system.

A *prefetched* state is associated with a page copy that has been prefetched. Such a state is useful to provide feed back to the *prophet*. PFS optimization is indeed based on a prophet which guides the prefetching mechanism. The prophet, itself, relies on information provided by an access pattern detector (see Section 3). The page copy in *prefetched* state is changed to a state of the standard coherence protocol as soon as a processor accesses it.

Another additional state is associated with a page copy

which is in the write buffer of the PFS. For a given page, there is only one such copy located in the node having the disk holding the corresponding page. This state is associated with the last copy of a page which has been modified in memory since it has been loaded. If a process accesses such a page before it has been copied to disk its state is changed to Berkeley protocol *modified* state.

Each page of a file which is mapped in the SVM has a synoptic state which is maintained in the page directory entry. It summarizes the state of the page copies in the cluster. This state is useful for minimizing the number of memory/disk transfers. It indicates if the most up-to-date copy of the page is in memory or if a prefetched copy of the page exists or if the page has to be written to disk (that is to say belongs to the PFS write buffer).

A page synoptic state may be updated

- by the PFS manager in the event of a prefetch operation or when the page is loaded from or copied to disk;
- by the SVM manager upon detection of the first write access on the page;
- by the SVM manager when the last copy of the page is replaced.

Concerning the page replacement strategy, the idea is to keep at least a copy of a page in global memory as long as there is a chance that it is referenced by a processor. When page frames are needed in a node, the number of copies of a given page may be decreased or a page copy may be injected in a remote node.

The disk paging mechanism implemented in most modern operating systems works by evicting pages from memory according to a Least Recently Used (LRU) selection scheme. In our system, the selection scheme uses page state information to better select pages to evict from a local memory. Page replacement is selectively performed depending on these states.

Redundant copies of a page shared by several nodes have the highest priority of eviction. They are simply discarded on replacement. When the last copy of a page has to be replaced, it is injected in a remote node having memory space available. If there is no space available in global memory it is copied to disk. A page belonging to a mapped file is copied to its disk counterpart. Other pages allocated in the SVM are copied to the local swap disk.

Several criteria may be taken into account to choose the node where to inject a page copy. For a modified page, it may be sent to the node storing it on disk. Thus, a subsequent disk write operation on this page may be done without requiring another page transfer in the network. Another criterion is the application data access pattern. By taking into account the application data access pattern, a node to node page transfer on a subsequent page fault may be anticipated. A page may be sent to a node having a high probability to reference it in the near future. Such a strategy requires a mechanism for predicting the application data access pattern. We envisage to implement in software a mechanism inspired from the *memory sharing predictor* presented in [13] for a hardware implemented DSM. This is another illustration of the interest of integrating different operating system mechanisms as such a memory access prediction mechanism would also be profitable for the optimization of the PFS.

3 Performance Evaluation

In traditional PFS, programmers can specify, through the *read/write* interface, data access pattern to help the operating system to optimize disk accesses. In the proposed system, file mapping makes disk accesses transparent to the programmer. A data is loaded from disk on a page fault. Disk writes may only occur in the event of a page replacement or at the end of the application. Thus, it is essential to optimize data loading performance. However, as the PFS does not have any information about the application data access pattern, optimization of disk accesses is more difficult than in a traditional system.

In order to check the feasibility of such a system, we have developed a simple prototype. In this section, we describe this prototype and results obtained during preliminary performance evaluations.

3.1 Prototype description

A first prototype of the proposed system has been developed on the PACHA machine[3]. This machine is a cluster of PCs interconnected by an SCI network and running Linux SMP. Each node is composed of two processors PENTIUM II 266 MHz, 128 MB memory and a 2 GB local hard drive. The hard drive bandwidth is 5 MB/s and the network bandwidth is 200 MB/s.

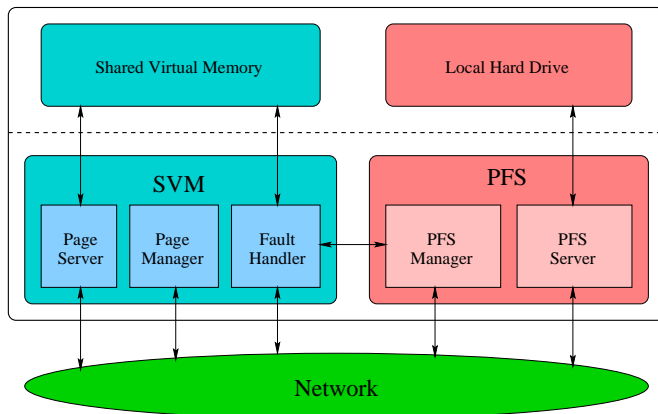


Figure 1: *Software architecture of the prototype*

This prototype consists of a sequential consistency SVM, a basic PFS and is implemented at user level. In our PFS, the chunk size is equal to the size of a memory page (4 KB). We have implemented a *one block lookahead* prefetching mechanism: when a page is loaded, the next n pages are prefetched ($n = \text{number of disks used to store the file} - 1$).

A write-invalidate coherence protocol similar to [16] is implemented. A *right access* is associated to each page. There are three possible right accesses : *invalid*, the page is not accessible by the processor; *read*, the page can be accessed for read and multiple copies may exist; *read/write*, the page can be accessed for read and write and only one copy of the page exists. When a processor tries to write in a page which is not in *read/write* mode, all the copies this page are invalidated and the right access changes to *read/write*.

3.2 Experiments

Our experiments consist of n nodes reading a file stored on d nodes (i.e. disks). We made three types of experiments. In the first one, one node ($n=1$) reads a file stored on several disks ($d=2,3,4$) in a sequential way. In the second one, several nodes ($n=d=2,3,4$) read a file in a sequential way. In the last experiment, several nodes ($n=d=2,3,4$) read a file in a random way.

In the case of a sequential read made by several nodes, each node accesses to an independent and sequential part of the file. For example, in the case of a read made by 2 nodes, the first one reads the first half part of the file and the second one reads the second half part of the file.

To read the file, each process accesses the memory area where the file is mapped. For each experiment the file size varies from 10 MB to 200 MB, and the disk cache is flushed before each run thus, data is really loaded from disks.

3.3 Results

The first experiment is a sequential read made by one node. Results are presented on Figure 2.a. The best bandwidth which is achieved is 14 MB/s when the file is stored on 4 disks. This bandwidth is close to 3 times the bandwidth of one disk, which is a good result for such a simple prototype but higher performance can be obtained. With a mapping interface when a page is touched, it should be loaded immediately. This leads to small disk accesses and so, to a low bandwidth. A prefetching mechanism allowing to prefetch more than one page per disk should offer higher performance, by increasing the size of disk accesses.

The second experiment is a sequential read made by several nodes. Results are presented in Figure 2.b. In this case, performance collapses: the best bandwidth obtained is 6 MB/s when a 200 MB file is stored on 4 disks. This bandwidth is close to the bandwidth of a single disk, so we don't get benefit from the use of a PFS. This poor bandwidth is a result of the concurrent accesses made by the different processes. When a process accesses a page in the mapping area, the process is blocked until the loading of corresponding data. Thus, disk requests cannot be buffered and must be achieved without delay. If several processes access simultaneously to different parts of the file, the PFS must access successively to different disks cylinders, leading to many disk seeks and then to poor performance.

The third experiment is a random read made by several nodes. Results are presented in Figure 2.c. The bandwidth is low, which is typical for a random access, but we could have had lower performance. In a traditional file system the bandwidth obtained for a random read with this kind of hard drive is typically from 500 to 1100 KB/s. This relatively high performance is due to cache effects. Higher performance can be obtained with an adaptable prefetching. When the system detects a random access, prefetching is disable avoiding wrong page prefetches.

3.4 Design of a new architecture

Preliminary results obtained by performance measurements on our prototype have highlighted specific problems of the file mapping interface in a parallel architecture. We have identified two major problems. The first one is the small disk accesses induced by the mapping interface. The second one is the high number of seeks caused by simultaneous accesses to a file by several processes.

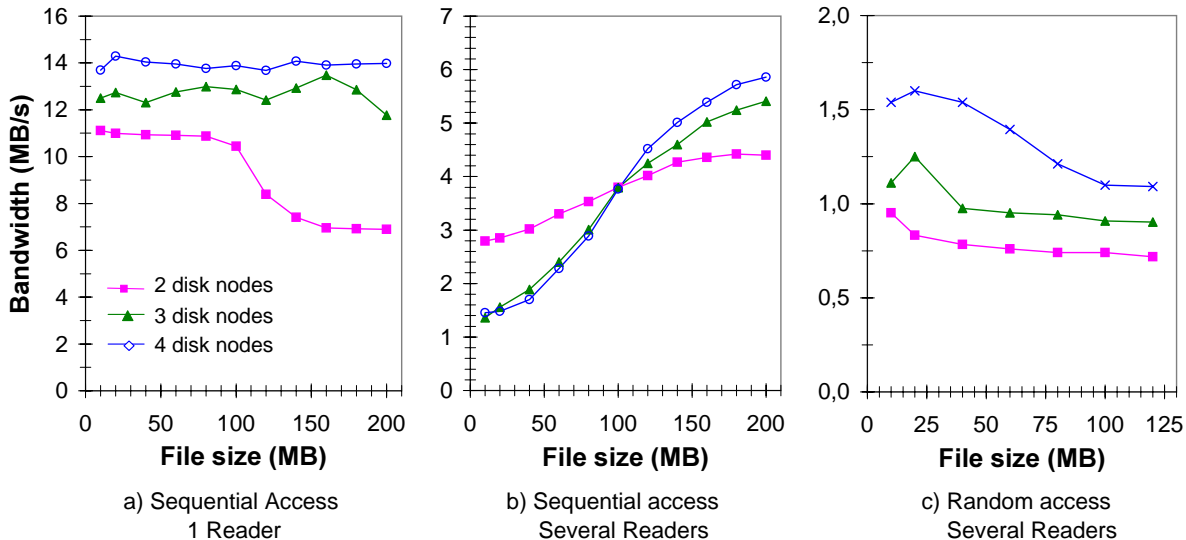


Figure 2: *Experimental results*

To solve these problems, implementation of efficient prefetching and disk scheduling strategies is required. But these mechanisms need high level information on the application data access pattern. With a mapping interface, this information is not explicitly given by the application programmer. So, we are currently working on a mechanism to automatically find a close approximation of a parallel application logical data access pattern at execution time. To achieve this automatic pattern detection, we investigate different approaches, in particular artificial neural networks and Markov chains [19]. The use of *on-line* statistics on disk accesses can also help our system to prefetch data. Information collected by the automatic pattern detector is used by a *prophet* to make predictions on future disk accesses and these predictions are sent to the prefetching mechanism.

Another problem related to file mapping as a new PFS interface is the scheduling of disk accesses. In our system, each thread issues a single disk read request at a time generated by a page fault. If several threads of the same application access simultaneously different areas of the same file located in a single disk, this leads to many disk seeks. A good scheduling policy should permit a low number of disk seeks without decreasing the potential computing/loading overlap.

4 Fault Tolerance Issues

In the proposed system, tolerating a single disk failure or a single node reboot or failure is essential as memory and disk management is distributed. In this section, we first give fault tolerance assumptions. We then present the checkpointing strategy used to tolerate node failures. Finally, we describe mechanisms provided to tolerate disk failures.

4.1 Fault Tolerance Assumptions

We assume that nodes operate in a *fail-silent* fashion. Nodes connected by a reliable interconnection network are considered. The unit of failure is the node: the failure of a node

component leads to the unavailability of the whole node. Our system is designed to tolerate transient failures or a single permanent one.

4.2 Tolerating Node Failures

Tolerating node failures during the execution of long-running applications can be achieved using backward error recovery techniques [15]. Backward error recovery is part of a fault tolerance strategy which enables an application that is affected by a node failure to restart its execution from a prior state. To achieve this goal, a consistent system state made up of a set of recovery data and called a *checkpoint* has to be periodically saved in stable storage. Programmers of parallel scientific applications usually manage checkpoints manually by periodically saving the computation state. When using a SVM, efficiently establishing a checkpoint at the application level is complex because of the lack of knowledge on data location in memories.

In the proposed system implementing file mapping, data location in memory and on disk is known by the system which can thus offer checkpointing primitives. Moreover, the mapping system has a detailed knowledge of data state (modified or not modified), allowing an *incremental* checkpointing strategy where new recovery data are saved only for modified data when a checkpoint is established.

There are two main issues when implementing backward error recovery. First, it must be ensured that the set of communicating processes' checkpoints forms a consistent system state. Our system is based on *global coordinated checkpointing* policy, where all nodes save simultaneously a checkpoint [4]. Thus, only one checkpoint needs to be maintained. The previous checkpoint is discarded when a new one is created.

Second, recovery data must be saved in stable storage. A stable storage combines two properties: persistence and atomic update. Persistence ensures that data saved in stable storage cannot be altered by a failure and remains accessible despite the occurrence of a failure. Atomic update guarantees that updates made to data in stable storage are

either successful operations or leave the data in their previous state.

Our system relies on an adaptation of Icare recoverable SVM described in [11] and in which checkpoint data are saved in memory. The permanence of recovery data is ensured by their replication in two distinct nodes memory. A two-phase commit protocol [9] ensures the atomic update of recovery data. During the first phase, called the *creation* phase, a new checkpoint is created, the old one being kept for recovery purpose. During the second phase, called the *commit* phase, the old checkpoint is discarded and the new one is confirmed.

Exploitation of SVM features for implementing a backward error recovery strategy has several advantages. First, as memory modules are used to store recovery data thus a fast checkpoint establishment is ensured where both the network high bandwidth and memory throughput are exploited. Second, as pages have no fixed physical location in a SVM they may be stored and restored anywhere in the system, thus greatly simplifying the checkpointing and recovery mechanisms. Finally, pages replicated by the standard SVM are simultaneously used as recovery data, thus limiting the cost of a checkpoint establishment by avoiding the creation of recovery data and requiring no extra memory for recovery data.

Conversely, SVM efficiency may benefit from the data replication required for fault tolerance since recovery data remains readable by processors as long as they have not been modified since the last checkpoint. So, two kinds of recovery data are distinguished in memory: those which can be read by processors as they are identical to the corresponding current data (we denote these copies as *shared-checkpoint* copies) and those which are kept only for recovery purpose and cannot be accessed by processors in normal functioning (these copies are denoted *invalid-checkpoint* copies). Moreover, by creating recovery data preferably in a node having a high probability to access the page, page faults may be anticipated.

The checkpointing mechanism is implemented as an extension of the coherence protocol of the basic SVM in order to combine recovery and active data management. The extended coherence protocol ensures that at any time there exists two recovery copies for each page in two distinct memories.

Disk write operations need to be dealt with carefully in the context of our system implementing file mapping between the SVM and a PFS. There are essentially four issues that were not studied in [11]:

- recovery of disk write operations,
- management of write buffer data at checkpointing time,
- disk update,
- replacement of recovery data.

Recovery of Disk Write Operations In the event of a rollback, the system must be able to undo disk write operations performed between the last checkpoint and the failure. Implicitly, data on disk belongs to the current checkpoint. When a page is loaded in memory, its current value remains identical to its associated recovery data till the first modification. As soon as a page is modified in memory, its current value becomes different from its recovery data. Due to the mapping, such a page may be copied to disk at anytime (a disk write operation will occur if the last copy of the page

is evicted from global memory) leading to the loss of the page recovery data. Thus, when a page is first modified in memory, two recovery copies are created for this page in memory. These recovery copies will be used to restore the page disk copy in the event of a rollback if the disk copy has been updated between the last checkpoint and the failure. To do so, a list of disk write operations performed since the last checkpoint is maintained. This list is used at recovery time to undo disk write operations. It is emptied each time a new checkpoint is successfully established.

Checkpointing Write Buffer Data In our system, the PFS write buffer is integrated with the SVM. Data belonging to the write buffer is data that has been modified since the last checkpoint and is waiting to be copied to the disk. When a checkpoint is saved, this data has to be checkpointed. Recovery copies of a page belonging to the write buffer could be created either in memory or on disk. As such a data was intended to be copied to disk, the system takes the opportunity of a checkpoint establishment to perform the disk write operation. So during the creation phase of the two-phase commit protocol, every page copy present in the write buffer is copied to disk. No recovery data is created in memory for such a page. During the commit phase of the two-phase commit protocol, every page copy which was in the write buffer is discarded. If a failure occurs during the creation phase, a rollback is initiated. The disk copy of the page can be restored from the previous checkpoint stored in memory.

Disk Update In our system, the owner of a page is responsible for updating the disk copy of a page upon replacement of its last copy or at the end of the application. In the standard coherence protocol the memory of the owner node of a page contains an up-to-date copy of the page. In the extended protocol, the memory of the owner of a page may only contain one of the two recovery copies of a page. This is the case if a page has not been modified since the establishment of the last checkpoint. Such a recovery copy cannot be discarded from memory without being first copied to disk. The node which is the owner of the first recovery copy is responsible for updating the corresponding disk copy if this recovery copy is to be replaced or at the end of the application.

Replacement of Recovery Data As recovery data are stored in memory, they use page frames that would be used otherwise to contain data useful for the computation.

Invalid-checkpoint copies are evicted from memory in priority as they are only kept for recovery purpose. They are locally copied in a dedicated log-structured file system (called *checkpoint-lsfs* in the remainder of this paper) ensuring efficient disk copies. The *checkpoint-lsfs* is stored in a disk distinct from the one used to store the application files¹ to avoid disturbing the applications disk scheduling. Moreover, using a local log-structured file system eases the garbage of data belonging to stale checkpoints. Indeed, as each local *checkpoint-lsfs* only contains *invalid-checkpoint* data, it is easy to discard these data each time a new checkpoint is established (in the commit phase). If the current checkpoint needs to be restored, data of *checkpoint-lsfs* is restored in

¹Each node is equipped with at least two disks: one used to store swapped data and the local *checkpoint-lsfs* and the others to store applications' files.

memory and is used to restore disk writes in mapped files if necessary.

A *Shared-checkpoint* data has a weaker priority of eviction than an *invalid-checkpoint* data. If such a copy is selected for eviction from a local memory, it is injected in a remote memory. When too many *shared-checkpoint* copies are kept in memory without being accessed anymore by processors, they are copied to disk. In fact, such a copy is replicated into the PFS write buffer on the node maintaining the corresponding disk copy. At the same time, the two *shared-checkpoint* copies change their state to *invalid-checkpoint*. The *invalid-checkpoint* copies having a very high priority for being selected for eviction from memory, they will be eventually transferred in *checkpoint-lsfs* (on disk).

Storing recovery data in memory does not allow the system to tolerate power failures affecting all the nodes of the clusters. Thus, our system also provides a primitive to the programmers to transfer a copy of the current checkpoint from memory to disk.

4.3 Tolerating Disk Failures

To tolerate disk failures, mechanisms inspired from the RAID technology [5] are used. RAID employs two orthogonal concepts: data striping for high performance and redundancy for improved reliability. Files are composed of *stripes*. Data of a stripe is interleaved on the disks of the cluster in blocks of fixed size called a *striping unit*. In our system, the striping unit is made up of a fixed number of pages. As in RAID, the large number of disks lowers the overall reliability. Assuming independent failures, 100 disks have collectively only 1/100 of the reliability of a single disk. Thus disk data redundancy is necessary to tolerate disk failures and allows continuous operation without data loss. RAID-1, RAID-4, RAID-5 could be implemented by software in our system to achieve both high performance and high availability. These disk array organizations differ from one another over the method and pattern in which the redundant information is computed and distributed across the disks of the cluster. In RAID-1 organization, also called mirroring, a striping unit is replicated on two disks. The only drawback of this organization is the disk capacity needed (twice the capacity of a non fault tolerant PFS). RAID-1 organization has several advantages. First, a page can be read from any of the two disks thus balancing the load on two nodes. Moreover, a page can be preferably read from the disk with the shorter queuing. Write operations imply writing the page on the two disks. If a disk fails, the other one can be used to serve all the requests in degraded mode thus resulting in a low performance degradation.

The only advantage of RAID-4 and RAID-5 strategies over RAID-1 strategy is that they need a lower disk capacity. RAID-4 and RAID-5 are similar. A parity block is computed for each stripe. In RAID-4, a disk is dedicated to the storage of all parity blocks while in RAID-5 parity blocks are uniformly distributed over all the disks thus eliminating the bottleneck of RAID-4 parity disk.

When a page is read, a single data disk is read. Write request must update the requested block and must compute and update the parity block. Providing efficient software distributed management of RAID-5 mechanisms which are normally implemented by a centralized controller is a real issue. It has been previously investigated in xFS [23]. As xFS is a log-structured file system, efficient write operations can be implemented in software. In our system (which is not a log-structured file system) a different mechanism would

have to be designed to efficiently perform write operations. This mechanism could take benefit from data redundancy in memory needed to tolerate node failures for efficiently implementing the *read-modify-write* procedure needed for small write requests.

In the event of a disk failure, the lost data can be computed from the other stripe data and parity but at the expense of a very high performance degradation.

Despite its cost in disk capacity, a RAID-1 strategy seems more adequate than a RAID-5 strategy in our system which primarily aims at high performance. It is all the more reasonable that current disks offer a high storage capacity at a low cost.

5 Related Work

Very few other work has been done on file mapping in software DSM with a PFS [12, 18, 17]. In [12], mapped files are proposed as the basis of a PFS but the standard file system interface (*read/write*) is kept. In contrast to our system, there is no single level storage system. Moreover, there is no integration with a software DSM as the proposed PFS has been designed for Hector scalable shared memory multiprocessor architecture implementing a DSM in hardware.

[18] presents a PFS designed for Treadmarks software DSM. Simplistic assumptions are made. For instance, it is assumed that accessed data are ideally placed in the disk of the accessing processor node. Moreover, in this system, a file is entirely loaded in memory before the beginning of the computation phase thus narrowing the advantages of coupling a DSM with a PFS.

In [17], the design of BFXM, a PFS relying on a DSM mechanism, is roughly presented. To the best of our knowledge, BFXM has not been implemented.

6 Conclusion

We have no doubt that software DSM implemented on clusters will move to the mainstream in the future. However, we are also convinced that solving the two following issues will help software DSM gaining wider acceptance.

First, the interactions of a SVM system intended to manage the memory resource with other operating system mechanisms dealing with other resources should be jointly studied in a cluster. In this paper, we have shown that global disk and memory management in the context of a SVM-based system has several advantages. In the same way, we investigate global management of other resources in the system such as processor and memory which can also bring benefits [2].

Second, *both* high availability and high performance should be achieved to allow proper execution of high performance applications on a SVM-based cluster. A lot of work has already been done on the performance side but not enough work has been performed for reconciling high availability [20] and high performance. In this paper, we propose a system in which fault tolerance mechanisms are tightly integrated with the DSM system allowing to combine both high performance and high availability.

Acknowledgement

The authors would like to thank Thierry Priol for helpful discussions about this work and David Mentré for his useful comments on the first draft of this paper.

References

- [1] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, pages 18–28, February 1996.
- [2] F. André, C. Morin, and M.-T. Segarra. Mechanisms for global processor and memory management on a NoW. In *Proceedings of the HPCN Europe'98 International Conference*, pages 324–336, Amsterdam, The Netherlands, April 1998. LNCS 1401.
- [3] P. Beaugendre, T. Priol, G. Allion, and D. Delavaux. A client/server approach for HPC applications within a networking environment. In *HPCN'98*, LNCS 1401, pages 518–525, April 1998.
- [4] 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.
- [5] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.
- [6] J. M. del Rosario and A. Choudhary. High performance I/O for parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, March 1994.
- [7] E. Marcatos G. Dramitinos. Adaptive and reliable paging to remote main memory. *Journal of Parallel and Distributed Computing*, 1999. to appear.
- [8] 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 Symposium on Operating Systems Principles*, 1995.
- [9] J. Gray. Notes on database operating systems. *Lecture Notes in Computer Science*, 60, 1978. Springer Verlag.
- [10] R.H. Katz, S.J. Eggers, D.A. Wood, C.L. Perkins, and R.G. Sheldon. Implementing a cache consistency protocol. In *Proc. of 12th Annual ISCA*, 1985.
- [11] A.-M. Kermarrec, C. Morin, and M. Banâtre. Design, implementation and evaluation of ICARE: an efficient recoverable DSM. *Software Practice and Experience*, 28(9):981–1010, July 1998.
- [12] O. Krieger, K. Reid, and M. Stumm. Exploiting mapped files for parallel i/o. In *SPDP Workshop on Modeling and Specification of I/O*, October 1995.
- [13] An-Chow Lai and Babak Falsafi. Memory sharing predictor: The key to a speculative coherent dsm. In *Proc. of the 26th Annual International Symposium on Computer Architecture*, pages 172–183, May 1999.
- [14] P. J. Leach, P. H. Levine, B. P. Douros, J. Hamilton, D. L. Nelson, and B. L. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications*, SAC-1(5):842–856, November 1983.
- [15] 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.
- [16] K. Li and P. Hudack. Memory coherence in shared memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.
- [17] 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.
- [18] S. C. Mac, C. K. Shieh, J. C. Ueng, and L. M. Tseng. Design and implementation of a parallel file subsystem on treadmarks. In *Proc. of the International Computer Symp.*, pages 256–263, December 1996.
- [19] Tara M. Madhyastha and Daniel A. Reed. Exploiting global input/output access pattern classification. In *Proceedings of SC97: High Performance Networking and Computing*. ACM Press, November 1997.
- [20] C. Morin and I. Puaut. A survey of recoverable distributed shared memory systems. *IEEE Transactions on parallel and Distributed Systems*, 8(9), September 1997.
- [21] Nils Nieuwejaar and David Kotz. The Galley parallel file system. *Parallel Computing*, 23(4):447–476, June 1997.
- [22] F. Douglass J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software Practice and Experience*, 21(8):757–785, 1991.
- [23] 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.
- [24] A. Barak S. Guday R. Wheeler. *The MOSIX Distributed Operating System*, volume 672 of LNCS. Springer Verlag, 1993.