A Cluster Operating System Based on Software COMA Memory Management

Renaud Lottiaux and Christine Morin
IRISA/INRIA, Campus de Beaulieu
35042 Rennes Cedex, France
{rlottiau,cmorin}@irisa.fr

Abstract

Clusters of SMPs are attractive for executing shared memory parallel applications but reconciling high performance and ease of programming remains an open issue. A possible approach is to provide an efficient Single System Image operating system giving the illusion of an SMP machine. In this paper, we present such a system focusing on global management of the memory resource. We introduce the concept of container at the lowest operating system level to build a COMA-like memory management subsystem. Higher level operating system services such as virtual memory system and file cache can be easily implemented based on containers and transparently take benefit of the whole memory resource available in the cluster.

1. Introduction

Due to the increasing power of microprocessors and the evolution of network technology, clusters of SMP's are now attractive for executing high performance parallel applications. However, the lack of dedicated software make clusters difficult to use. The management of resources (memories, disks, processors) which are distributed through the cluster nodes is a burden to the programmer. An efficient management of these resources is difficult and introduces a significant overhead in term of development time and decreases the software reliability. The work presented in this paper relates to Gobelins operating system which is designed to offer the view of a unique shared memory multiprocessor on top of a cluster. Gobelins supports the execution of multiple parallel applications consisting of a set of threads sharing memory. Threads may be scheduled on any node of the cluster and may share data whatever their location.

Global memory management is a key feature to make the cluster appear as a shared memory multiprocessor. Existing software Distributed Shared Memory (DSM) [9, 1], implemented at user level, are not sufficient to offer the view of a shared memory multiprocessor. Indeed, they only allow several threads of an application to share memory areas as specified by the programmer. However, they generally support to a limited extend the concurrent execution of multiple applications and inter-application data sharing. Moreover, interactions between software DSM and file systems are not taken into account.

Several systems have made a step ahead by virtualizing not only memories but processors or disks. Thread migration [8, 17], parallel file system [10], cooperative caching [18] and fault tolerance [12] on top of a DSM have been studied. However, each project focuses on a specific problem which is solved by using complex and specific mechanisms. The integration of all these mechanisms in the same operating system to offer a real single system image managing each cluster resources is complex due to the high software overhead of each proposed solution. In this paper, we present the Gobelins operating system. Gobelins use a unified low level memory management mechanism which allows to easily implement all the previously presented mechanisms.

In Gobelins the node physical memories are merged to form a single memory shared by all processors, thus providing a global and unified management of the memory resources. Low level memory management software in Gobelins makes a node local memory act as a cache of the shared memory space for the processors of the node. Similarly to a Cache Only Memory Architecture (COMA) [6], a page has no fixed physical location in the cluster and is migrated and/or replicated in the node memory of the processors referencing it. The consistency of the multiple copies of a data that may exist in the cluster is managed by a coherence protocol as in a COMA. The global low level memory management that we propose in Gobelins is based on the concept of *container* introduced in this paper. A container consists of a set of pages that may be located in any node of the cluster and may be shared by different threads whatever their location. Higher level operating system services such as the virtual memory system, a shared virtual memory system, a cooperative caching system for a distributed, thread

migration or parallel file system can be easily implemented relying on containers.

The remainder of this paper is organized as follows. In section 2, we give an overview of Gobelins operating system and introduce the container concept. In section 4, we describe the design of containers and their use for the design of different Gobelins services: (shared) virtual memory system and (cooperative) caching system for file systems. Section 5 presents the current Gobelins implementation. Section 6 discusses related work and section 7 concludes.

2. Gobelins Operating System

The goal of Gobelins operating system is to offer the view of a unique shared memory Symmetric Multi Processor (SMP) machine on top of a cluster of PCs. We use a global and integrated management of disks, memories and processors. Gobelins offers a shared memory programming model. A parallel application is executed by a *g-process*, which contains a number of threads called *g-threads*. The execution of a *g-process* is distributed among the cluster nodes, its *g-threads* being automatically migrated to idle nodes to increase parallelism. This programming model is the one provided by operating system designed for SMP machines. With Gobelins, we extend this model to clusters of SMPs.

2.1. Existing Operating Systems

In existing operating systems designed for SMP machines, memory management is the core of the system. It ensures two main goals: allowing processors to share data through memory and storing data coming from disk. Physical memory is organized as a set of page frames. Page frames can be allocated, deallocated or swapped to disk. This physical memory is used as a page cache for the implementation of process virtual memory and file cache.

The virtual address space of a process is divided into several segments: text, data, stack, etc. Each segment is a set of virtual pages linked to a set of page frames. A segment is characterized by a beginning and ending address, an access right and in some cases an associated file. These segments (except stack segment) are shared between threads of the same application. Thus, threads can share data through physical memory, thanks to segment sharing. Moreover, several processes can share memory thanks to a special shared memory segment (System V segment, for instance).

File management in modern operating systems is designed on top of a file cache organized as a set of pages. Each entry in the cache is stored in a page frame. Each file access passes through this file cache. For instance, when a read is issued the file system checks the presence of the corresponding page in the cache. If the page is not present, a disk access is performed to load the data in physical memory. Then, data is replicated in user space. To avoid data replication and ease file access, a file can be mapped in virtual memory. In this case, cache entries corresponding to the file are mapped in the virtual address space of the process mapping the file.

2.2. Containers

The lack of shared physical memory in a cluster makes it difficult to implement services offered by a traditional operating system. In order to share physical memory between nodes in a cluster, we introduce the concept of *container*. A container is a virtual entity consisting of a sequence of bytes structured as a set of pages. It allows to store and share data between several nodes in a cluster. Containers are identified in the cluster by a unique identifier. Access to a container is done thanks to dedicated manipulation functions allowing to get a copy of a page or to modify its content.

Containers allow the implementation of a software COMA-like memory management at the operating system level. Similarly to COMA architectures, containers use local memories as caches but with a memory page management granularity. They ensure the migration or replication of pages between nodes at physical memory level. Thus, a container can be handled only by the operating system. The operating system does not distinguish a standard page frame from a page frame coming from a container. All operations that can be applied to standard page frames can also be applied to page frames belonging to a container. Notably, it is possible to map them in the address space of one or more processes.

There are three kinds of containers: *volatile*, *file* or *linked*. A *volatile* container holds temporary data which is not linked to any device. These containers generally have a lifetime equal to the lifetime of applications using them. They exactly behave as a standard software DSM. *File* containers are associated to a parallel file. They contain data coming from parallel files, allowing to access it through physical memory, thanks to container handling functions. They allow to map parallel files in DSM, greatly easing their access and use. They can survive to the death of applications using them, then behaving as cooperative parallel file caches. Finally, *linked* containers are associated to a device (disk, CD-Rom reader, etc.) located on a given node. They make it possible to any node to access a device located on a remote node through physical memory. They can also survive to the death of applications using them.

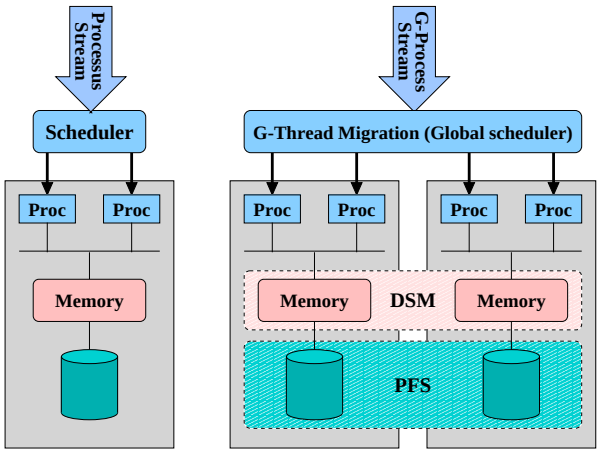
2.3. Gobelins Overview

To offer a shared memory multiprocessor view, Gobelins extends traditional operating system memory management to a cluster scale (see figure 1). To allow memory sharing between nodes, a Distributed Shared Memory (DSM) mechanism is used. This offers the view of a unique and shared memory to each node of the cluster. To hide disk multiplicity and increase global disk bandwidth, we implement a Parallel File System (PFS) [2, 13, 7]. In such a system, file data is split in sets of blocks stored on several disks of the cluster nodes. The overall disk bandwidth is thus increased by reading or writing several disks in parallel. Finally, a threads migration [19] mechanism allows a transparent use of all cluster processors. *G-thread* are migrated from one processor to another to balance the global computing load and increase parallelism.

In Gobelins, these subsystems are integrated to avoid redundancy of common mechanisms. This integration also avoids to take conflicting decisions in the system. For example, taking into account both memory and processor load 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.

The address space of each *g-process* is divided in several segments called Shared Memory Area (SMA). As segments rely on physical memory in traditional operating systems, SMAs rely on *containers*. SMAs are shared between *g-threads* of the same application. Thus, *g-threads* can share data through SMAs. Moreover, several *g-processes* can share memory through a special SMA mapped in the virtual address space of these *g-processes* extending mechanisms such as *IPC system V* segments to a cluster scale in a DSM context.

Gobelins is composed of a standard UNIX file system and a Parallel File System. These systems are implemented on top of a cooperative cache [4, 18]. Memory of all nodes is used to cache file data. Each node can put data in this global cache and can access to any data cached by any node. When a node is running out of memory, cache data is injected in the memory of a remote node. When a read is issued, the file system checks the presence of the corresponding page in the file cache. The page can be retrieved from local memory or remote memory. If the page is not present in file cache, it is loaded from disk through Gobelins PFS or traditional UNIX file system. The implementation of a PFS and a DSM in the operating system has led us to design a single level storage system. In this system, file mapping is the basic file system interface [11]. When a file is mapped in memory, an access to this memory area behaves as if it has been done to the corresponding file area. By suppressing the traditional *read/write* file system interface, the programmer task can be significantly eased specially for the design of out-of-core applications (i.e. applications working on a data set greater than the available memory).



a) SMP Machine *b) Cluster of SMPs running Gobelins*

Figure 1. *Comparison between an SMP machine and a cluster running Gobelins*

3. Design

In this section, we detail the design of containers and their use to implement higher operating system mechanisms.

3.1. Containers design

A container is structured as a set of pages. When a page of a container is referenced, it is placed in physical memory inside a page frame, its address being returned to the operating system. Thus, nodes physical memory is used like a container page cache. Four functions are provided for accessing this page cache:

- **FindPage (container id, page number)**: search for a page in local memory.
- **GetPage (container id, page number)**: get a page copy in the local memory.
- **GrabPage (container id, page number)**: get write access to a page.
- **FlushPage (container id, page number)**: eviction of a given page from a local memory.

The physical memory being managed as a container page cache, pages can not be directly accessed. To access a page, the operating system should first check if the page is present in the cache thanks to the *FindPage* function. When this function is called, the physical memory is scanned to check the presence of the requested page. If the page is present in memory, its physical address is returned to the operating system, or else an exception is triggered.

In this last case, the *GetPage* function is used to obtain a copy of the page. When this function is called, a copy of the requested page coming from a node owning the page is placed in local physical memory. If there is no copy in the cluster, the page is created, loaded from disk or from a remote device, depending on the container type.

The containers semantic imposes to use the *GrabPage* function before writing to a page. This function ensures that the copy present in local memory is the only one existing in the cluster, allowing writes without any coherence problem. When this function is called, every existing copy in the cluster is invalidated.

When there is no more free page frames in a node memory, the operating system replacement algorithm chooses a page to evict and calls the *FlushPage* function. If there exists other copies of the selected page in the cluster, the local copy is simply discarded. If this copy is the only one in the cluster, it is injected into the memory of a remote node. If there is no memory space left in the cluster and the chosen page belongs to a *volatile* container, the page is discarded after being written to disk. Finally, if the chosen page belongs to a *file* or *linked* container, it is discarded after updating the device copy if needed.

3.2. Virtual Memory Design

In the Gobelins operating system, memory sharing between nodes is provided through SMAs, which are implemented by containers (see figure 2). Each SMA is a container mapped in the virtual address space of a *g-process*. Each page of the *g-process* address space is associated to a page of the corresponding container. The link between a page frame holding container data and a virtual page is carried out by the operating system virtual memory manager. However, when a page referenced by a *g-thread* is not present in the physical memory of its execution node, the local node contacts the corresponding container to obtain a copy of the page. This page is then mapped in the *g-process* virtual address space.

The *G-process* data SMA is associated to a *volatile* container. This SMA is equivalent to a traditional software DSM since it just allows to share memory between threads executing on different nodes. Memory sharing between several applications is possible, since a container can be mapped in the address space of several *g-processes*. In this case, each *g-process* sharing a memory area holds in its virtual memory a special SMA linked to a shared *volatile* container.

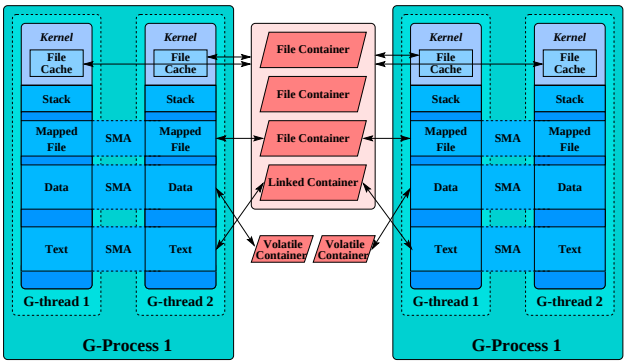


Figure 2. *Global view of the Gobelins memory management (here, two instances of the same application)*

3.3. File System Cache

Containers are also used by the operating system to share kernel data between several nodes, such as file cache data. In modern operating systems, this file cache is structured as a set of pages. The basic unit of file management is the memory page. Each file access passes through file cache, which caches pages from both mapped files and files accessed through normal *read/write* calls. Thus, it is possible to use containers to implement this cache, making it possible to increase the overall system performance by offering a large and shared cache. As a node can use data cached by another node, the cache hit rate is increased. Moreover, cache data chosen by the replacement algorithm can be kept in global memory if it can be injected into the memory of a remote node. This file cache is also used to cache data from PFS files. Each data coming from disk, from standard UNIX files or parallel files, are stored in the file cache. The key point is that no cooperative file cache is implemented in the PFS. Caching is entirely managed by the underlying containers.

3.4. Parallel File Mapping

Parallel files and UNIX files can be mapped in the virtual memory of *g-processes*. For each mapped file, there is a dedicated SMA in the virtual address space of the *g-processes* mapping the file. For instance, the text SMA is associated to a container *linked* to the file holding the application code. A mapping SMA is associated to a *file* container associated to a parallel file. Containers used to map files are also part of the operating system file cache.

3.5. Advantages

The use of containers to globally manage memory in a cluster offers many advantages. First, it offers a unique mechanism for the implementation of shared memory mechanisms, file mapping and cooperative file caching. By unifying all these mechanisms within a single system, software complexity is decreased, increasing the system robustness.

The use of containers to manage the file cache makes it possible to obtain an efficient cooperative cache [4, 18] without doing a dedicated implementation in the file system. The management of file cache data is ensured by the container management mechanism. This mechanism ensuring the management of every data located in memory, it is possible to offer the best possible balance between the amount of cache data and volatile data. The system automatically adjusts the balance between several types of data according to applications needs.

Finally, process migration raises the problem of remote accesses to files located on a particular node disk. If a process which has opened a file located on a disk local to its execution node is to be migrated, the system must ensure remote access to this file. This implies the implementation of complex mechanisms linking a migrated process and the file opened on a remote disk. The use of containers to manage the file cache solves this problem without implementing extra mechanisms. Thanks to containers, the file cache is accessible from any node in the cluster. A migrated process can access a remote disk by accessing the global file cache.

4. Implementation Issues

Implementation of Gobelins has been carried out using the Linux operating system. Our experimentation platform is a 28 dual-processors PCs cluster. We use three kinds of network technology in the cluster: Scalable Coherent Interface (SCI), 100 Mb Ethernet and Gigabit Ethernet.

Gobelins is implemented at kernel level without modifying the Linux core kernel. Gobelins functionalities are added using *kernel modules*. A module is a piece of code which can be easily added and removed from the kernel core at execution time. Hence, the programming cycle is speed up and the crash risk during tests is decreased.

4.1. Overview of the Implementation

The actual implementation consists of 3 modules (see fig. 3). A communication layer has been implemented on top of Ethernet network adapters. It offers high level communication primitives like *send*, *receive* and *active messages* at the kernel level. This library has been developed on top of a modified version of Gamma [3]. Gamma is a low level Ethernet communication library offering very low latency and high bandwidth. In our system, Gamma has been modified to make it usable into Linux kernel.

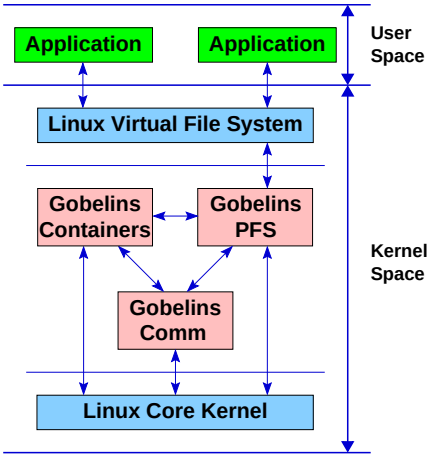


Figure 3. *Integration into Linux*

A module implement *containers*. The container system consists of 3 components: a *container manager*, a *page manager*, and a *page server* (see fig. 4). The *container manager* implements the interface used by the operating system services. It receives requests from the kernel and forwards them to the *page manager*. The *page manager* manages a table containing the owner of each page, i.e. the node which was the last one to write the page. The *page server* handles requests from remote nodes. It sends copy of local pages or invalidates local page copies.

Finally, the parallel file system is a new Linux file system, usable like any other file system. On each node, the PFS is composed of a *PFS manager*, a *PFS server* and a *disk manager* (see fig. 4). The *PFS manager* deals with a list of open files and their corresponding mapping address. It makes the correspondence between a page number in the SVM and its disk location. The *PFS server* handles requests from remote nodes. The *disk manager* accesses data of the local disk. It is used as an interface between storage units (through the Linux disk manager) and *PFS servers*. To minimize the implementation complexity, Gobelins PFS relies on *Ext2* file system. We kept most of the existing code and have only modified some mid-level functions like *readpage*. A parallel file is thus implemented as a set of local *Ext2FS* files.

4.2. Basic Data Structures

To manage containers, the table *containerList* is replicated on each node. Each entry of this table contains information on a given container: container identifier, size, type, identifier of the linked parallel file or device, location

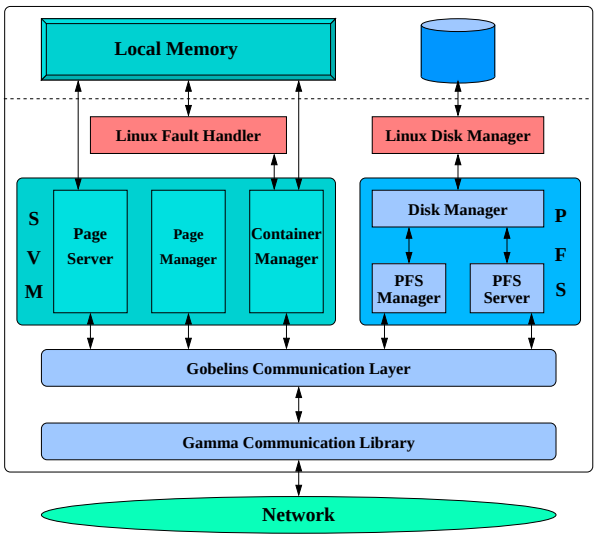


Figure 4. *Software organization*

of this device and list of *g-threads* sharing the container on the local machine.

For each container, we also use two tables to store state and location of pages in the cluster. The first one is a statically distributed table called *pageInfo* used to locate page copies. This table contains for each page the location of its master copy (i.e. the last modified copy). A second table called *ctnrPageTable* exists on each node. It contains for each page present locally, its coherence state (*invalid*, *read* or *read/write*) to implement a write invalidate coherence protocol and the list of existing copies in the machine (if the page is the master copy).

Finally, the table *PFSfile* is replicated on each node. It contains the list of open parallel files. Each entry contains the correspondence between our PFS file identifier and the local *Ext2FS* file identifier.

4.3. Implementation of Container Manipulation Functions

When the *findPage* function is called, the *container manager* checks in the table *ctnrPageTable* of the corresponding *container* if the requested page is present in local memory. If the page is present, its physical address is returned. Else, a null pointer is returned.

When the *GetPage* function is called, the *container manager* sends a page copy request to the concerned *page manager*. This one determines if there is a copy of the requested page in the memory of a node. If there exits a copy in a remote node, the request is forwarded to the *page server* of this node, which sends back a copy to the requesting node. If there is no existing copy in the cluster, several actions are possible according to the container type. If the page belongs to a *volatile* container, a request is sent back to the requesting node, which creates a new empty page. If the page belongs to a *file* container, a request is sent to the PFS manager to load the page in memory and send it to the requesting node. Lastly, if the page belongs to a *linked* container, a request is sent to the *page server* of the node on which the linked device is located. Data is retrieved from the device and sent to the requesting node.

When the *GrabPage* function is called, the *container manager* sends an invalidation request to the *page manager* which forwards it to the owner of the page. This one invalidates its copy from memory and transmits an invalidation request to each node holding a copy. On these nodes, the page is discarded from physical memory and removed from the virtual address space of each *g-threads* attached to the corresponding container. If the page is a *file* or *linked* page, the corresponding entry in file system cache data structure is invalidated.

4.4. Implementation of SMA

The virtual address space of a Linux process is split in several segments called *Virtual Memory Area* (VMA). Each VMA corresponds to a specific segment (text, data, stack, etc) and is managed by a set of functions. These functions can be replaced by user specific functions.

SMA are implemented by modifying VMA management functions and adding a link to a container. Functions diverted by *Gobelins* are *no_page*, called when a page is touch for the first time and *wp_page*, called to perform a copy on write. Our functions are very simple. They mainly consist of calls to appropriate container functions and manipulations of the *g-process* page table.

When a process is created, each page in a *volatile* SMA is invalidated. An access to one of these pages cause a fault triggered by the linux memory manager. The *no_page* or *wp_page* Gobelins function is then called through VMA manipulation functions. If the fault is a read fault, these functions check the presence of the requested page in local memory thanks to *findPage*. If the page is not present, the *getPage* function brings a copy back in local memory. This page is then inserted in the virtual address space of the faulting process by the Linux memory manager. If the fault is a write fault, the *grabPage* function is used to invalidate remote copies and the page access right is changed to write.

File and *linked* SMAs work in the same way. The only difference come from the management performed by corresponding containers.

5. Related Works

In the Tempest project [14], a set of mechanisms is offered to support several parallel programming styles. More specifically, it enables an application's user-level code to easily and efficiently support fine grain shared memory. Blizzard-S [16] is a software implementation of Tempest, running on top of a set of workstations. A part of the physical memory is managed in a COMA like way to support shared memory. Fine grain access control is carried out by modifying the executable program. Special code is inserted before all LOAD and STORE that could access to a shared segment.

Shasta [15] is another system offering a COMA like fine grain shared memory. It allows to transparently execute unmodified multiprocessor applications.

Blizzard-S and Shasta are implemented inside an operating system kernel. However, they do not offer a fully single system image. Creation of shared memory regions pass through explicit memory allocation functions. Moreover, there is no support for (parallel) file access, swapping, process migration, and fault tolerance. Finally, the system is limited to a single user. To fill these gaps, a deep modification is needed since shared memory is only supported through virtual memory. Sharing data between several processes of several users, caching (parallel) file data and allowing swap require a deeper operating system memory management modification.

A low level memory management has been proposed in GMS [5] to use memory of remote machines. GMS is a system designed for cluster of workstations to allow a machine to inject pages in the memory of remote machines. This mechanism is implemented at the lowest operating system level, allowing the injection of any kind of page: text, data, and so on. The main difference with our work is the lack of support for the execution of parallel applications. In contrast to Gobelins, GMS does not offer the view of a unique shared memory multiprocessor machine.

6. Conclusion

In this paper we have presented the concept of container and described how it can be used for global distributed memory management in Gobelins operating system targeted for clusters. Gobelins memory subsystem unifies mechanisms needed to implement complex operating system services like distributed shared memory, thread migration, parallel file mapping in shared memory and management of cooperative caches.

An implementation of Gobelins in the Linux kernel is in progress. This implementation makes minor modifications to core kernel. Gobelins functionalities are added in the kernel using modules. The actual implementation does not include thread migration and fault tolerance. Fault tolerance mechanisms are currently studied on a user level prototype. Thread migration on top of containers is actually studied and will be implemented in the next few month.

Our future work includes experimental evaluation with a set of representative applications and introduction of fault tolerance mechanisms to tolerate node and disk failures.

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, Feb. 1996.

[2] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 285–302. ACM Press, Dec. 1995.

[3] G. Ciaccio. Optimal communication performance on Fast Ethernet with GAMMA. *Lecture Notes in Computer Science*, 1388:534–??, 1998.

[4] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the First Symposium on Operating System Design and Implementation*, Nov. 1994.

[5] 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, Dec. 1995.

[6] E. Hagersten, A. Landin, and S. Haridi. DDM — A cache-only memory architecture. *IEEE Computer*, 25(9):44–54, Sept. 1992.

[7] J. Huber, C. Elford, D. Reed, and 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.

[8] A. Itzkovitz, A. Schuster, and L. Shalev. Thread migration and its applications in distributed shared memory systems. *The Journal of Systems and Software*, 42(1):71–87, July 1998.

[9] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *IEEE Transactions on Computers*, 7(4):321–359, Nov. 1989.

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

[11] 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.

[12] C. Morin and I. Puaut. A survey of recoverable distributed shared memory systems. *IEEE Transactions on parallel and Distributed Systems*, 8(9), Sept. 1997.

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

[14] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and typhoon: User-level shared memory. In *Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA'94)*, pages 325–337, Apr. 1994.

[15] D. J. Scales and K. Gharachorloo. Towards transparent and efficient software distributed shared memory. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*, Oct. 1997.

[16] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *Proc. of the 6th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVI)*, pages 297–307, Oct. 1994.

[17] K. Thitikamol and P. Keleher. Thread migration and communication minimization in DSM systems. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):487–497, Mar. 1999.

[18] G. Voelker, E. Anderson, T.Kimbrel, M. Feeley, J. Chase, A. Karlin, and H. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 33–43, June 1998.

[19] A. B. S. G. R. Wheeler. *The MOSIX Distributed Operating System*, volume 672 of *LNCS*. Springer Verlag, 1993.