For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>.
For distribution issues, contact Amy Tashbook Hall, Assistant Editor <amy.hall@eng.sun.com>.

Solaris MC File System Framework

Vlada Matena *Yousef A. Khalidi* *Ken Shirriff*

Sun Microsystems Laboratories
2550 Garcia Avenue
Mountain View, CA 94043

1 Introduction

Recent trends in the computer industry indicate that large scale server systems will be constructed as systems with distributed memory. A node in such a system will be a small scale shared memory multi-processor. The main reasons for using distributed memory are to achieve scalable throughput and high availability.

The software model for systems with distributed memory is still a subject of research. One extreme is to program and manage the system as a network of independent computers and use standard networking protocols such as TCP/IP for communication. To reduce communication overhead, new lightweight communication protocols (e.g., active messages [5]) are proposed as alternatives to the standard ones. Such networked systems provide only limited single system image.

The other extreme is to take an existing operating system for shared-memory-bus multiprocessors and run it on the distributed system. This approach requires hardware support to create the image of cache coherent memory, which can provide non-uniform memory access (CC-NUMA), or cache only memory access (CC-COMA) computer.

Between the two extremes lie a number of other possible approaches. We believe that a scalable cost- effective *multi-computer* can be built by connecting off-the-shelf nodes (e.g., SPARCservers, or Pentium Pro quad boards) through an industry standard high speed interconnect, such as the Scalable Coherent Interconnect (SCI). We argue that the multi-computer should be programmed and managed as a single computer, rather than as a collection of networked nodes, even if the interconnect does not have support for CC-NUMA or CC-COMA. Furthermore, we believe that the system should tolerate failures and remain available even if one or more nodes have crashed.

The Solaris MC research project has set out to extend the Solaris™ operating system to multi-computers with distributed memory. The goal of Solaris MC is to provide a SSI for such a multi-computer by making its system and management interfaces identical to those used in today's Symmetric Multiprocessing (SMP) systems.

Support for file systems is an important component of Solaris, and thus of Solaris MC, which extends the Solaris file system framework to the cluster.[1] A key design issue is how to provide this framework for location transparent access to files, devices, directories, and other objects that are stored in file systems. A process must be able to open a file stored anywhere in the system and access it as if the file were local. Furthermore, when a file is accessed concurrently from multiple nodes, single node file sharing semantics must be preserved across the cluster. The difference between remote and local file performance must be minimal to ensure location transparency.

The architecture of the Solaris MC file system is designed around new object-oriented interfaces that extend the Solaris vnode/VFS interface [16]. (Vnode operations are used in Solaris to integrate file systems through functions dealing with file data access, file caching, file locking, directory operations, file links, and file attributes.)

While the vnode interface has enabled several new file systems such as NFS [25], over time OS researchers have discovered several shortcomings of the interface, such as:

- no support for evolution (interface versioning)
- poor interface abstraction (vnode is a union of several distinct interfaces)
- limited extensibility.

The shortcomings became apparent during the design of the Solaris MC file system framework, motivating us to "fix" the vnode interface by making it more extensible. In effect, we have replaced the vnode interface with several new, well-structured interfaces based on the extensible file system architecture of the Spring operating system [14]. The new interfaces address not only the issue of file system distribution in a clustered system, but also the issues of file system extensibility. The new interfaces are object-oriented and are defined in the CORBA standard Interface Definition Language (IDL) [19].

For backward compatibility with Solaris, the new interfaces plug into the kernel and communicate with existing file systems (e.g., UFS) through the existing vnode interface. In the future, the vnode/VFS interface can be made obsolete and the existing file systems integrated directly through the IDL interfaces.

In this paper we describe the architecture and implementation of the Solaris MC file system. We present the file system interfaces and discuss various aspects of their integration with Solaris. The paper is organized as follows. Section 2 contains background information on Solaris MC. Section 3 discusses the requirements for the distributed file system. Section 4 discusses the problems with vnodes and how we have replaced them with a new interface. Section 5 presents the various implementation issues that we have encountered during integration with Solaris. Section 6 describes how the file system achieves high availability. Section 7 provides the implementation status as of this writing. Section 8 talks about the performance of the prototype. We compare our effort with related work in Section 9, and provide a summary in Section 10.

2 Solaris MC

Solaris MC [13] is a prototype distributed operating system for multi-computers (i.e., clusters of nodes) that provides a SSI: a cluster appears to the user and applications as a single computer running the Solaris operating system, providing the same ABI/API to the applications.

1. The terms *cluster* and *multi-computer* are synonymous in this paper.

The components of Solaris MC are implemented in C++ through a CORBA-compliant object-oriented system with all new services defined by the IDL definition language. Objects communicate through a runtime system that borrows from Solaris doors and Spring subcontracts.

Solaris MC is designed for high availability: if a node fails, the remaining nodes remain operational.

Process operations are extended across the cluster, including remote process execution and a global */proc* file system. External networks are transparently accessible from any node in the cluster.

3 Requirements

We have identified the following requirements for the design of the file system framework:

- distributed
- single node file semantics
- easy integration with Solaris
- performance
- high availability
- extensibility

As the primary goal of Solaris MC is to extend Solaris to multi-computers without shared memory, a distributed file system becomes a necessity for such systems.

The distributed file system must present single-node semantics for all file operations in order to maintain compatibility with existing Solaris applications. Specifically, the expected consistency of operations on file data must be achieved even if a file is mapped in memory by processes on multiple nodes.

The distributed file system is built on top of the existing Solaris operating system. Special care must be taken not to require major kernel changes to the underlying Solaris.

One of the main reasons for developing a multi-computer is to provide a platform with scalable performance. Since file systems are the backbone of the UNIX® operating system, file system performance is very critical to the success of the multi-computer.

A multi-computer has the potential of being a platform for high availability commercial applications, such as interactive database services. The file system must provide continuous access to data, even in the presence of node failures. Continuous availability also requires that the system be able to evolve over time without making the system unavailable, for example, during operating system upgrades.

Often there is a need to extend the functionality of an existing file system. Since Solaris does not provide any interfaces for extending file systems, we consider it important to provide support for such extensions in our architecture.

To meet the above requirements, we have made the following design decisions:

- To avoid extensive changes to the Solaris kernel, the new file system interfaces plug into Solaris using the existing vnode interface. The implementation of the new interfaces appears to the kernel as a file system called the *proxy file system* (PXFS).
- The framework provides support for coherent caching of file data, file attributes, and directory entries. The granularity of file data sharing is a single VM page on systems without hardware support.[2]
- Support for high availability and replication is built into the communication infrastructure, and is transparent to the majority of the file system code.

2. On systems with CC-NUMA hardware assist, the coherence unit can be as small as a single cache line.

- The IDL interfaces provide the support for system evolution and for coexistence of multiple versions of the same interface. This is important for providing continuous system availability.
- The file system architecture supports interposing (stacking) at the level of an individual file, or at the level of an entire file system.

4 Replacing vnodes

The vnode interface [16] provides a mechanism to plug new file systems into the operating system kernel. As vnodes were used in different projects [2,9,24,26,28], however, several problems surfaced.

A fundamental problem of vnodes is that they combine multiple interfaces into a single interface. They provide both a naming service and data provision. The data provision of vnodes includes file operations, paging operations, and locking. Combining these interfaces has several disadvantages: 1) objects that only support part of the interface (e.g., pipes) don't fit vnodes well, 2) these operations cannot be split across nodes, 3) and the system loses flexibility in how it is structured.

Second, vnodes have limited extensibility. The main problem is that they do not provide interfaces for consistent caches that are split between vnodes. Many applications, such as stacking vnodes or building distributed file systems, require vnodes to cache data in multiple locations. With vnodes, the caching code is located with the data provider. Stacked vnodes cannot share a cache, since pages are cached according to their vnode and offset.

Third, vnodes are structured around a simple data path where data is moved through the vnode. They cannot support more flexible I/O paths, such as moving data directly into a client cache for efficiency while preserving consistency.

Fourth, vnodes are a kernel-level interface only. They do not support user-level file systems.

Finally, vnodes do not provide interface versioning or other support for evolution. Thus, vnode-based file systems must be released concurrently with operating system releases that modify the vnode interface.

4.1 An object-oriented replacement for vnodes

We replaced the vnode interface with object-oriented interfaces based on the Spring operating system [14]. The new interfaces are defined in the CORBA standard IDL [29].

The object-oriented approach has several advantages for the interface specification. First, it is extensible, since interfaces can be extended through inheritance. Second, IDL is an industry standard for defining the interfaces independently of the language and operating system.

We put strong emphasis on the separation of interface from implementation. Interface inheritance is crucial to system extensibility; implementation inheritance is a convenient way to share implementation code. These two concepts should not be confused in a system design.

4.2 The Solaris MC interfaces

We have decomposed the vnode operations into multiple distinct interfaces. The hierarchy of file system interfaces is illustrated in Figure 1. An arrow represents the *inherits-from* relationship.

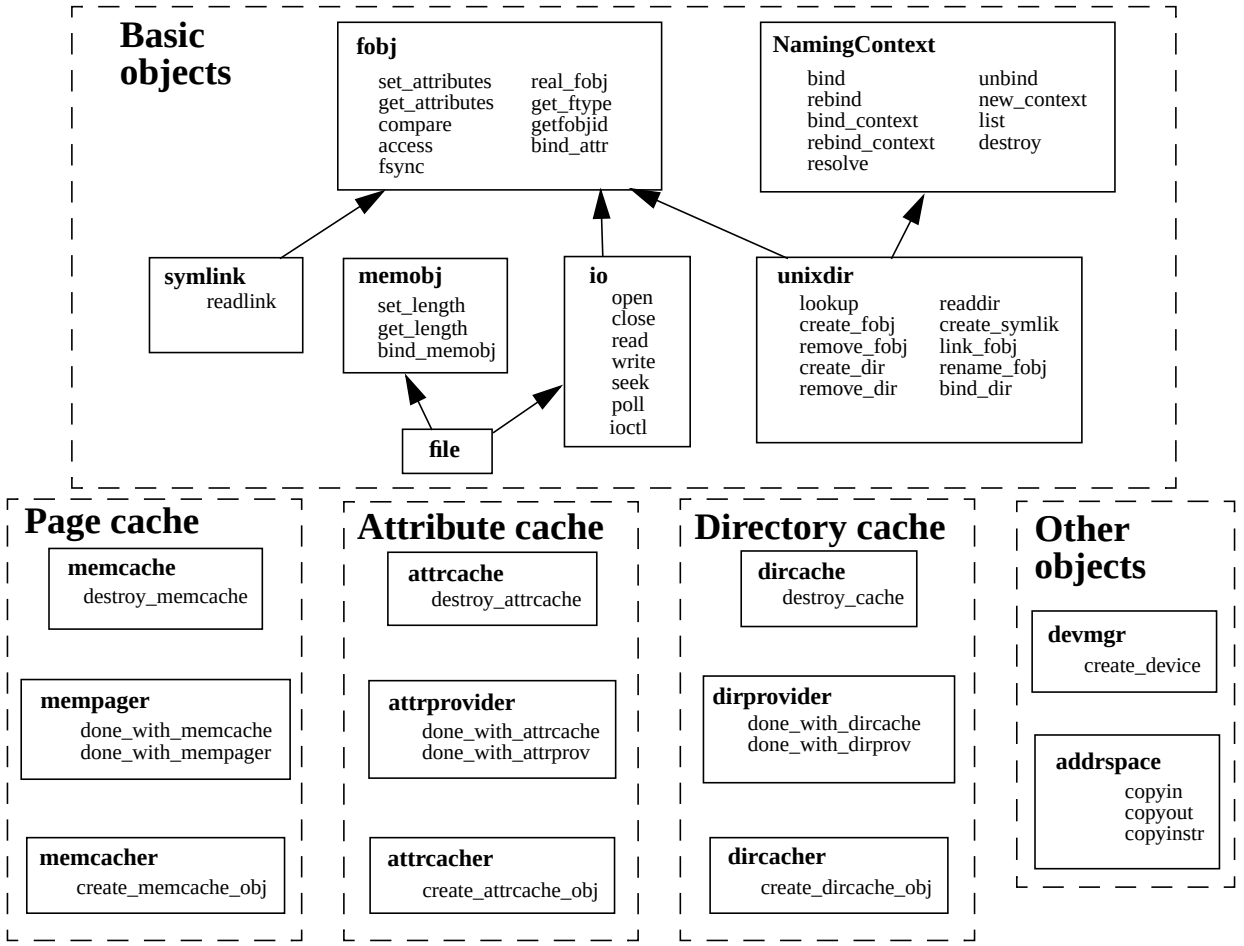


FIGURE 1. File system interfaces

These interfaces form the heart of the Solaris MC file system as they define the communication between file system objects, both between nodes and on a single node.

The fundamental structural difference between the Solaris MC interfaces and vnodes is that the Solaris MC interfaces break vnodes into three main parts: File objects, naming objects, and cache objects (which are subdivided into page, directory, and attribute caches).

4.3 File object interfaces

The basic interfaces are designed to support the vnode operations other than those used by caching. The requirement of backward compatibility with Solaris dictates the arguments of the operations.

At the top of the hierarchy is the ***fobj*** interface. The *fobj* interface defines the operations that are common to all the objects that are stored in file systems (files, directories, device nodes, named pipes, doors, and symbolic links).

The *set_attributes* and *get_attributes* operations are used to set and get object attributes. *Compare* tests if two objects are equal. *Access* tests permissions to perform a given operation. *Fsync* is used to force object's data and/or metadata to stable storage. *Real_fobj* is provided for compatibility with Solaris (to implement the "real vnode"

concept for devices). *Get_ftype* returns the object type (e.g., "unixdir"). *Getfobjid* returns a file handle that can be used for example by NFS. *Bind_attr* binds object attributes to an attribute cache.

The ***io*** interface adds the operations used for device and file access. The *open* and *close* operations perform device specific open and close protocols. The *read* and *write* operations transfer data between process buffer and the device. *Seek* is provided for compatibility with Solaris. *Poll* is used by the distributed implementation of the *poll()* system call. *Ioctl* performs an ioctl operation on the underlying object.

The ***file*** interface inherits from the *io* and *memobj* interfaces. The ***memobj*** interface specifies the operations for an object that the VM system can map into process's virtual memory. The *bind_memobj* operation is used to set up a two way communication channel between a cache and a provider (pager).

4.4 Naming interface

The ***unixdir*** interface defines the operations on a file system directory. *Unixdir* inherits from the *fobj* and the Common Object Services Specification (COSS) *NamingContext* [20] interfaces. *Unixdir* adds operations for creation and deletion of the various objects that are stored in file systems. The *bind_dir* operation binds the directory to a directory cache. Supporting the standard COSS naming interface allows inclusion of the file system directory tree in the global name space. A directory appears as a naming context in the global name space [14]. Objects in a file system can be located using the *NamingContext::resolve* operation in addition to using the low level *lookup* operation. *Lookup* traverses a name path one element at a time, and is called from the Solaris *lookuppn* path traversal function.

4.5 Caching interfaces

Stateful protocols are required to support coherent caching in order to meet our performance requirements. While other distributed file systems, such as VMS [7], LOCUS [22], DFS [12], and Calypso [17] rely on an external lock or token manager to achieve cache coherence, our design integrates the coherence protocol with the data movement protocol. Such integration results in improved performance, both in speed and space.

The caching framework has been adopted from the Spring architecture for object caching [14,18]. The framework is applied to caching of file data, file attributes, and directory entries.

The file caching interfaces are illustrated in Figure 2. The figure shows a file that is being cached by the VM subsystem on two nodes. Unlike a vnode, the caching components in Solaris MC are split into three parts: the mempager, the memcacher, and the memcache. With this flexibility, caching can be divided across nodes or layers.

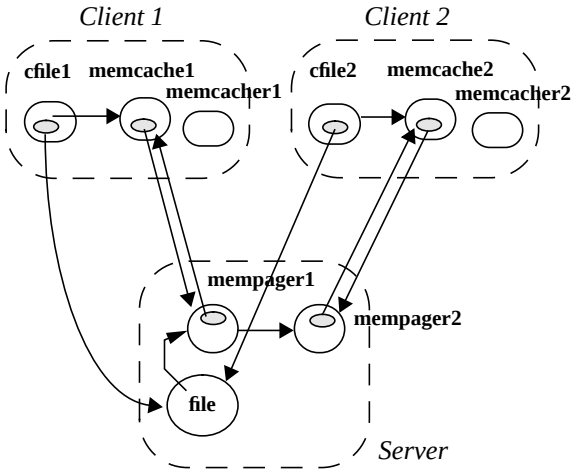


FIGURE 2. Interfaces for caching of file data

The server creates a ***mempager*** object for each caching domain. Each *mempager* object

communicates with its associated ***memcache*** object located in the caching domain. *Memcache* calls the *mempager* interface to request data. *Mempager* calls the *memcache* interface to send invalidate and downgrade requests to the cache. The *mempager* and *memcache* objects are created by the bind protocol described in [15]. A ***cfile*** object represents a cached file. A *cfile* object implements the *file* interface by delegating file operations to the *memcache* object.

The file system framework does not prescribe any particular caching protocol between the mempager and memcache objects. Rather, an implementation defines the most suitable caching protocol for the given system through interface inheritance. In fact, a single file implementation may support multiple caching protocols, each using a different *mempager.*

The memcache/mempager operations for the Solaris MC prototype file system were chosen to meet the following requirements:

- multiple-reader/single-writer policy
- sharing data with page level granularity
- sequential consistency of data access
- support for fault isolation and high availability
- independence from VM page size

The Solaris MC ***mempager*** and ***memcache*** methods are shown in Figure 3.

Page_in is called to request data in a specified range[3] for read-only or read-write access. *Page_upgrade* is called to upgrade the read-only caching rights to read-write. *Page_zero* is called to advise the mempager that pages will be created in the cache with read-write access rights. *Page_out*, *write_out*, and *sync* send modified file data to the pager. After the operation completes, the cache maintains no rights, read-only rights, and read-write rights, respectively.

3. All operations specify the range by *offset* and *length*.

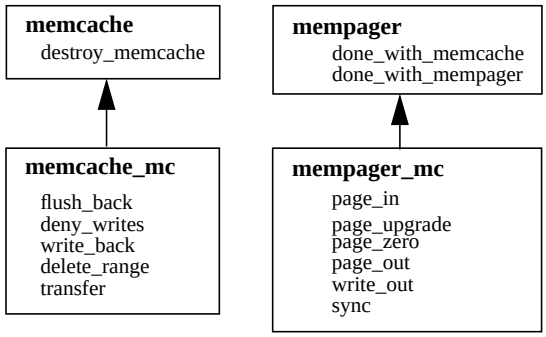


FIGURE 3. Solaris MC data coherence operations

The operations on the *memcache_mc* are invoked by the pager if another cache requests data in a conflicting mode. *Flush_back* instructs the cache to send modified data to the pager and give up all access rights to the specified range. *Deny_writes* instructs the cache to send modified data and downgrade its access rights to read-only. *Write_back* requests that modified data be sent to the pager without downgrading the cache's access rights. *Delete_range* instructs the cache to remove data in the specified range from the cache and give up all access rights to the range. The *transfer* method is invoked to copy cached data into another cache.

Spring allowed the *flush_back*, *deny_writes*, and *sync* operations to return data as output arguments. The Solaris MC protocol instead requires that the ***memcache*** invoke the *page_out*, *write_out*, and *sync* operations respectively to send data to the ***mempager*** object. This "push" model is necessary to recover from failures in the system without a data loss. With the push model, the ***memcache*** holds the data until the call to the ***mempager*** completes**.**

4.5.1 Caching file attributes

The caching protocol is implemented by a pair of objects similar to those shown in Figure 2 for data caching. The objects are called ***attrcache_mc*** and ***attrprov_mc,*** and are derived from the generic attribute caching interfaces as shown in Figure 4.

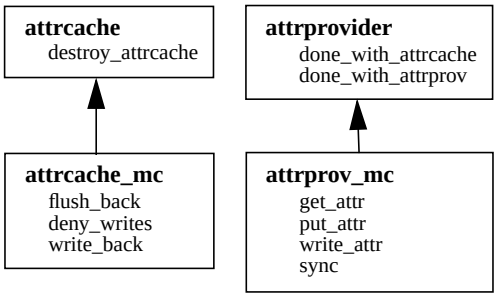


FIGURE 4. Interfaces for caching file attribute

The multiple-reader/single-writer policy is used for caching all attributes as a single unit. A single caching right (token) covers the file attributes: user and group IDs, file access control list (ACL) if any, file size, and access and modification times

We considered assigning caching tokens to individual attributes, but we could not find any performance advantages from this fine grain token scheme for typical multi-user file access workloads.

While other distributed file systems use a separate token for achieving atomicity of read and write operations that span multiple pages, we use the attribute caching token for this purpose. This works well because the writer needs to get exclusive access to the file attributes anyway to update the modify time and file length.

POSIX file semantics require that attribute changes after certain operations be made durable in stable storage. When the client releases the token on the attributes after such a file operation, the new values of the attributes are written through to the server.

4.5.2 Directory caching protocol

The directory caching protocol is also based on the cache/provider mechanism. Figure 5 illustrates the methods of the ***dircache_mc*** and ***dirprovider_mc*** interfaces. The interfaces are derived from the generic directory caching interfaces and have methods optimized for caching directories in the clustered system.

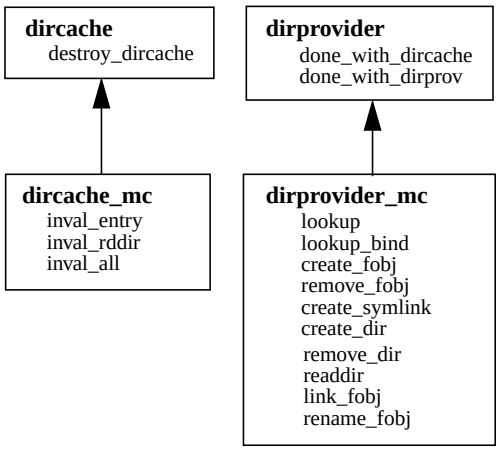


FIGURE 5. Directory caching interfaces

The directory cache is a *write-through* cache. Operations that change directory contents are first forwarded to the server before the local cache is updated. We chose this policy to preserve the semantics of UFS directory operations: directory changes must be forced synchronously to stable storage.

4.6 Other objects

The ***devmgr*** object is used to create a device node. The ***addrspace*** object is used to support ioctls on remote devices.

4.7 File system extensibility

The protocols for caching file data, attributes, and directory information described in the previous sections were chosen to work optimally in the current Solaris MC cluster environment. While these interfaces provide a complete distributed file

system with UNIX semantics, they should be considered only one possible application of the generic file system framework specified by the interfaces in Figure 1.

The object-oriented framework and the caching architecture provide a great deal of flexibility for extending file system in the following areas:

- multiple implementations of an interface
- system evolution by interface versioning
- user or kernel level implementations
- multiple caching protocols
- interconnect specific data transfer optimizations
- file system stacking
- interoperability with foreign file systems

The ***/proc*** file system illustrates how the object-oriented framework is conducive to having multiple implementations of an interface. The ***/proc*** pseudo file system in Solaris provides a file-based interface to processes for use by ps, dbx, and other programs that need to access process state. In Solaris MC, ***/proc*** is extended to provide a view of the processes running on the entire cluster, by merging together local /procs into a global picture.

Interface inheritance provides a partial solution to interface evolution. A backward compatible revision of an interface is accomplished by inheriting interface version N+1 from version N. The object type system assures that an object of version N+1 can be used anywhere where an object of version N is expected. The framework, however, does not provide much help for incompatible versions of an interface. If an implementation object is to support multiple incompatible interfaces, the implementation object must provide the multiple interfaces as individual objects.

The Solaris MC IDL file interfaces are independent of the implementation environment. It is possible to implement any component either in user space or in the kernel without any impact on the other components.

The generic caching interfaces (e.g., the *memcache*/*mempager* objects) do not have any methods related to a cache coherence protocol. The methods used for cache coherence must be supplied by interfaces derived from the generic ones. While one protocol implements data coherence at the page level (as in the current Solaris MC prototype), another protocol may require that the entire file be sent to the client node that wants to access file data. If the hardware interconnect provides an assist for cache coherence (e.g., CC-NUMA), the caching protocol can let the hardware enforce data coherence at the cache line level. The methods on the ***memcache*** and ***mempager*** objects for these protocols will look quite different. It would be architecturally wrong to define a "generic" caching protocol that would attempt to be a superset of all possible caching protocols.

The Solaris MC file system retains the extensibility of the Spring file system. All the forms of file stacking, watchpoints, and other file system extensions described in Spring [14,15] are also possible in the Solaris MC file system. Our caching architecture even allows a single file to support multiple caching protocols. If the server supports multiple caching protocols, the "best" protocol is selected at *bind* time by negotiation between the cacher and the server.

4.8 Benefits of the Solaris MC interfaces

The file system interfaces of Solaris MC provide an object-oriented replacement for vnodes. These interfaces can co-exist with vnodes and vnode-based operating systems, while fixing the key problems of vnodes.

The Solaris MC interfaces separate different file system functionality into distinct interfaces, instead of the single interface provided by the vnode interface. In particular, the

naming interface is separated from I/O, and caching has separate interfaces.

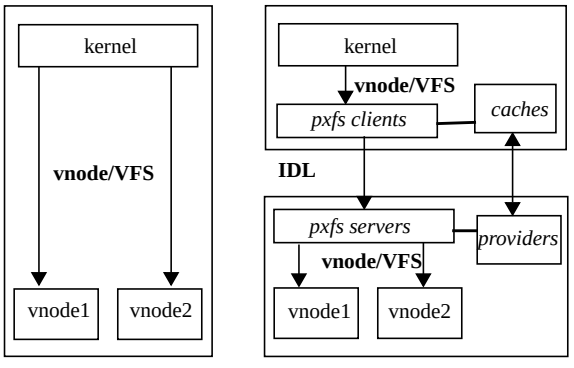
The interfaces also support complex data paths, where data is moved directly from disk to a cache on the client, for instance. The low-level support for this is discussed in Section 5.3.

File systems can be built at user level or kernel level with the Solaris MC framework, since the object communication works transparently among kernel, user, and remote domains.

Finally, Solaris MC provides versioning support through its object-oriented interfaces. Many file system interface changes can be added through interface inheritance, and released without changes to the operating system.

5 Implementation

We integrated the Solaris MC file system framework with the underlying base Solaris at the vnode/VFS interface to avoid extensive modification to the kernel (see Figure 6). It would be possible to remove the vnode interface from the system and call the new interfaces directly (at the price of modifying the existing file systems and the kernel).



(a) Standard Solaris (b) Solaris MC

FIGURE 6. Integration at Solaris vnode interface

On the client side, processes invoke file operations through *proxy vnodes*. A proxy vnode is a C++ class derived from the generic vnode structure. Each UNIX file type has its corresponding proxy vnode type. The classes *pxreg*, *pxdir*, *pxchr*, *pxblk*, *pxlink*, and *pxstr* correspond to the vnodes for a regular file, directory, character special device, block special device, symbolic link, and Stream head, respectively.

On the server side, a set of implementation objects implements the ***fobj***, ***unixdir***, ***io***, ***file***, and ***symlink*** interfaces, which implement generic file system objects, I/O devices, files, directories, and symbolic links, respectively. Each implementation object contains a pointer to the vnode in the underlying file system. Using this approach, any Solaris vnode-based file system works with the new framework without modification. In the future, the underlying file system could implement the Solaris MC interfaces directly.[4]

5.1 Caching

This section provides the details of how the caching of file data is integrated with the Solaris virtual memory system. One of our goals was to avoid substantial changes to the complex Solaris VM system to integrate the new file system framework. Figure 7 illustrates our approach.

The three objects on the client side are:

- pxreg—proxy vnode for regular file
- attrcache—attribute cache
- pxmemcache—page cache

These objects communicate with corresponding server-side objects:

- file—interface to the underlying file
- attrprov—provider for file metadata
- mempager—provider for file data

4. Our prototype uses this direct implementation for UFS. About 100 lines of code were added to UFS and several hundred were removed.

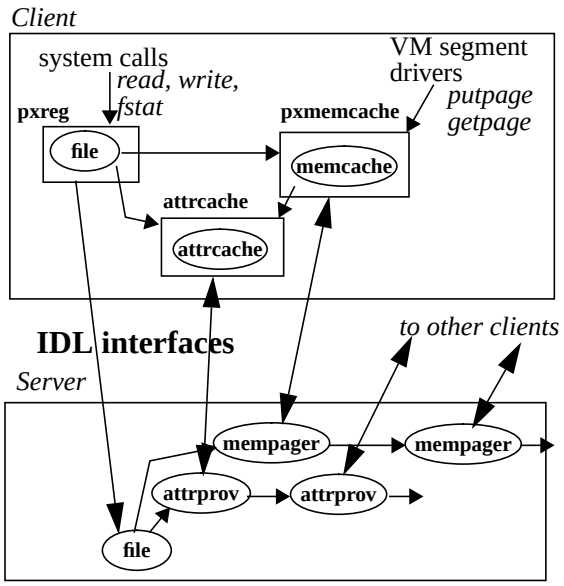


FIGURE 7. Integration with Solaris VM on the client

The ***pxreg*** and ***attrcache*** objects are created when a file is looked up in a file system directory. The directory *lookup_bind* method provides a highly optimized protocol that looks up an object and creates the attrcache/ attrprov attribute coherence channel in a single object invocation.

The ***pxmemcache*** object and the corresponding ***mempager*** object are established on the first access to the file data (i.e, *read*, *write*, or *mmap*). The ***pxreg***, ***attrcache***, and ***pxmemcache*** objects are linked together as shown in Figure 7.

From the perspective of our object-oriented framework, a proxy vnode (pxreg) is a representation of a cached file. A pxreg object can be passed anywhere where the ***file*** interface is expected. This property is essential to file stacking.

Since a pxreg object also supports the vnode interface, client kernel system calls that act on vnodes (such as read and write) can be handled directly by the pxreg object.

Pxmemcache is an object that is both a vnode and an implementation of the ***memcache_mc*** interface (described in Section 4.5). By being a vnode, the pxmemcache object is easily integrated into the Solaris VM system, since the VM treats it like any other vnode and can send the GETPAGE and PUTPAGE operations to the pxmemcache object. Pxmemcache is responsible for maintaining the state of the client side of the data coherence protocol. The following are the rules that govern client's access to a page:

- R1: if a page exists in the cache, the cache has been granted at least the read-only access rights.
- R2: a bit in a page structure field reserved for file system use is set if the cache has been granted the read-write access rights.

The ***mempager*** object maintains two state bits for each page. These two bits encode the access rights that the cache is currently granted.

5.2 An example: file read

To illustrate the interaction among the file system components, we will walk through the *read* system call (refer to Figure 7).

A process invokes the read system call. The kernel on the client side dispatches the operation to the pxreg vnode. If the file is marked as non-cacheable, pxreg invokes directly the *read* method on the *file* object on the server side. The server sends the requested data to pxreg. If other nodes cache the file, the server makes sure that the read returns the latest version of the data by forcing the other caches to write back the data in the requested range before returning the data to pxreg.

If the file is cacheable on the client (the default case), pxreg invokes a method on the attrcache object to acquire a read token on the file attributes. Holding the token assures atomicity of the read operation and protects

the file size from changing. While holding the token, pxreg delegates the read operation to the pxmemcache object.

Pxmemcache maps the file pages in the requested range into the kernel mapping segment (segmap) and calls *uiomove* to copy data into the process buffer. Uiomove may generate a page fault. The VM system dispatches the page fault to the GETPAGE method on the pxmemcache object. If the faulted page is present in the cache, pxmemcache does not have to call the server because rule R1 from the previous section guarantees that the cache has at least the read-only rights to the page and that the page contains the most recent file data.

If the page does not exist in the cache, GETPAGE allocates a page frame and calls the *page_in* method on its associated mempager object. Any optimizations for sequential file access (prefetch or disk block clustering) are done at this point. The invoked mempager scans all other mempagers for the file to make sure that no other cache caches the requested page in a conflicting mode. If another node caches the page in the read-write mode, the cache is called to write back the data and downgrade its caching rights to read-only. Then, the mempager object records that the memcache has been granted the read-only rights to the page and the page is returned to pxmemcache.

After the read from pxmemcache completes, pxreg releases the token on the file attributes.

5.3 Optimizations of data transfer

While the global file system can work on a variety of hardware configurations ranging from a network of workstations connected through Ethernet to a CC-NUMA computer, the file system architecture has been optimized for a cluster. We consider that the optimal cluster interconnect has the following performance characteristics:

- 100-1,000 Mbyte/sec node-to-node bandwidth
- support for direct DMA between a disk and any node's memory
- low latency (cross node remote procedure calls complete in 50-100 microseconds)

We believe that these characteristics make it possible to almost eliminate the differences between local and remote file performance. To reduce the penalties for the remote case, it is necessary that the file system be able to transfer data directly from a remote disk into a page frame in the client file cache without intermediate buffering on the server node (see Figure 8).

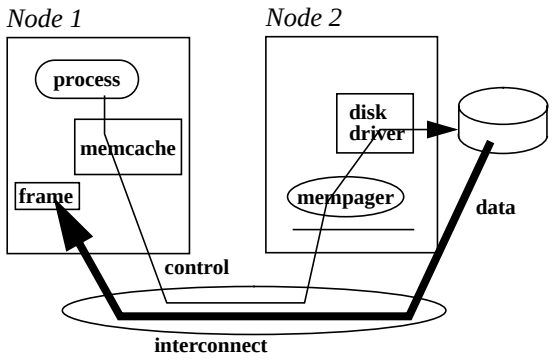


FIGURE 8. Direct data transfer into remote page frame

Although the support for data transfer into remote memory is very specific to each underlying interconnect hardware, we did not want to "hard code" the support for any particular interconnect into the architecture of the file system. Furthermore, the file system code must remain portable to off-the-shelf interconnects that do not support direct memory transfers and require using intermediate buffers (e.g., Ethernet).

Our object-oriented framework allows us to entirely hide the specifics of the interconnect from the file system architecture and implementation. It also supports multiple data handlers to permit different data movement

strategies to be transparent to the higher-level implementation.

An example of a object handler is the efficient bulk I/O handler, which avoids copying of data pages into a marshal buffer, and will perform zero-copy transfers if the interconnect allows. Bulk I/O data is passed through the file system interfaces as "serverless" objects with the *bulkio* object handler [14]. The bulkio object handler that is invoked by the object framework during marshaling and unmarshaling of RPC arguments encapsulates the interconnect specific code. The following fragment from the pxmemcache code illustrates how bulkio objects are used:

```
page_t *pp = ... ;  /* allocate page frames */
bulkio_pglist pgl_obj = bulkio_pglist(pager_obj, pp);
pager_obj->page_in(offset, length, pgl_obj, ... );
```

First, pxmemcache allocates the page frames from the virtual memory manager. Second, the pgl_obj object with the bulkio handler is created. Since the constructor is passed the pager object and the local page frames, the constructor can do any necessary interconnect Memory Management Unit (MMU) setup on the client. Third, the mempager is invoked passing the pgl_obj as an *inout* argument. The argument is inout because conceptually the client passes an empty frame to the server and the server returns a frame that is filled with the data.

When the server receives the *page_in* operation, the unmarshal method on the bulkio handler sets up the mappings in the interconnect MMU to allow direct data transfer. Then, the *page_in* method is invoked. *Page_in* looks as follows:

```
void page_in(pxfs::offset_t offset, pxfs::size_t length,
      bulkio::inout_pages_ptr& pgl_obj, ... )
{
      page_t *pp = bulkio_impl::conv(pgl_obj);
            ...
      /* read data into page frame list pp */
}
```

First, the pgl_obj is converted into a list of page frames suitable for local disk I/O. Then, the disk device driver strategy routine is invoked, passing the page frame list as an argument. The data is transferred directly from disk into the memory on the client.

As illustrated in the code fragments, the interconnect specific code is localized in the bulkio handler and is transparent to the file system code.

The code also works when the client and server are on the same machine. The page frame list created by pxmemcache is passed directly to the server without the overhead of the marshal/unmarshal operations.

5.4 Object-oriented programming

Both the client and server sides of the file system are written in C++. All interactions between the components are specified as IDL interfaces. Since IDL maps well to C++, and IDL allows location transparent programming, structuring the file system as distributed objects has turned out to be an elegant implementation approach.

Importantly, due to location-independence, the code contains no statements that explicitly provide separate code paths for local and remote cases. For example, the client code does not know or care whether or not the server object is local. If the server object happens to be local, invocations on the object are reduced to C++ function calls. If the object is remote, the client invokes a proxy (stub) that performs remote procedure calls to the server object.

Our implementation of the object framework includes a distributed protocol for object reference counting [14]. The framework automatically keeps track of the number of references to each object. The framework notifies an object via the *unreferenced* method when the reference count drops to zero.

Automatic reference counting simplifies the file system implementation. For example, if a

file is to be removed while it is still accessed by processes, the server object simply delays the removal of the file until *unreferenced* is called.

The distributed reference counting algorithm is fault- tolerant with respect to node failures. Using the *unreferenced* method as the notification mechanism for failures of remote components turned out to be a powerful mechanism to implement failure recovery and high availability. Our framework does not require an additional event notification mechanism, such as the fault-tolerant distributed lock manager used in the VAXcluster [7] for failure recovery. Our framework also avoids putting timer code in the file system to detect failures.

6 High Availability

Support for high availability (HA) has been a major requirement for the design of the file system. Our design has the following characteristics:

- no single node failure makes data unavailable
- both client and server failures are transparent to processes accessing files
- file access semantics are preserved across failures
- the support for HA is transparent to the client side of the file system implementation
- only minimal changes are required on the server side of the implementation
- minimal performance overhead incurred by HA
- minimal impact of failover on performance

Our design assumes that the underlying platform allows disks to be attached to multiple nodes, as illustrated in Figure 9.

The file system is stored on mirrored disks that are attached to at least two nodes. At any point in time, one of the attached nodes acts

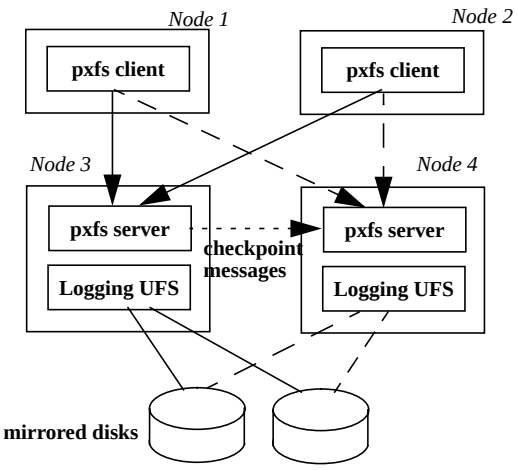


FIGURE 9. High availability configuration

as the *primary* replica. One of the other nodes with access to the disks is designated as the *secondary* replica. A synchronous primary/secondary replication protocol is used by the file system. The clients communicate only with the primary server. The primary server carries out the operation and sends a checkpoint message to the secondary. Read-only operations do not require checkpoint messages. Some non-idempotent operations require that the stable state on the disk and the state on the secondary be updated as a single transaction. We have developed a highly optimized protocol that we call a *mini-transaction* for this purpose.

If a client node crashes, the framework detects that node membership has changed [1]. The framework notifies the ***mempager*** objects on the server side through the *unreferenced* call.[5] The mempagers perform cleanup such as releasing caching rights held on pages cached on the crashed node.

Recovery after a crash of the primary server node is more complicated. As shown in Figure 9, the object references to server side objects held by the client side are connected

5. Each ***memcache*** object has a separate ***mempager*** object.

to both the primary and secondary objects via two xdoors (an xdoor [1] is a communication end-point). The multiple xdoors are held in the object handler which makes server replication transparent to the client code.

If the primary server crashes, the framework for replicated services notifies both the secondary server and the replica handlers at the clients. The secondary server becomes a new primary to which the clients send their requests. Note that the framework automatically redirects object invocations to the secondary so that the replication of objects and the switchover on failure is entirely transparent to the client application, and doesn't require changes to the code. Mini-transactions assure that non-idempotent calls are executed exactly once, even when a request was resent after failover.

Logging UFS is Sun's extension to UFS to provide fsck-less recovery through write-ahead logging of file system updates. After the crash of the primary, the secondary quickly recovers the state of the file system from the log before becoming the new primary.

We do not require modifications to the underlying file system to support replication. Our technique would work with any underlying file system in which file system operations are atomic and testable (an operation is *testable* if the secondary can determine during recovery whether the operation has completed [8]).

Server failover is fully transparent to processes accessing files. Since the volatile state of the client side (e.g., page cache contents) is preserved across failover, performance degradation due to cold cache effects after failover is minimal.

7 Implementation Status

At the time of this writing, we have a functioning prototype of the distributed file system. The prototype includes an implementation of all the interfaces shown in Figure 1. The caching protocols for file data, attributes, and directory operations described in this paper have been implemented. High availability is still under development.

Our development environment is a cluster composed of four SPARCstations 20. Each SPARCstation is configured with two processors. The nodes are currently connected via a 100 Mbit/sec Ethernet hub. We exercise the system by a variety of workloads. Our demo applications include a commercial parallel database system, pmake, and scalable HTTP server. These applications are used to stress the cache coherence protocols of the file system.

The file system prototype consists of about 16,000 lines of moderately commented C++ code. The client side is about 6000 lines, the server is about 3000, bulkio is about 1500, the new part of **/proc** is 700 lines, and the remainder is miscellaneous.

We are currently focusing on two major areas: performance and high availability.

The Ethernet interconnect does not allow us to implement the optimized data transfers described in Section 5.3. Therefore, we are moving to a faster interconnect based on SCI. We are currently evaluating the available SCI interface cards.

We are adding support for high availability, as described in Section 6. This effort appears to be relatively straightforward because the Solaris MC infrastructure for high availability and server replication does most of the hard work. We only have to add checkpoint messages from the primary to the secondary replica.

8 Performance

We planned originally to report on our prototype performance. Specifically, we wanted to compare the performance of remote file access to NFS. But we felt that such a comparison could be misleading because our goal is not to beat the NFS performance (or that of any other network file system): we want the performance of both local and remote files access in the cluster to be comparable to the local file performance of a non-clustered system.

	user	system	elapsed	overhead
UFS local	307	121	497	0%
PXFS local	311	133	521	5%
PXFS remote	314	136	559	12%
NFS remote	321	137	619	25%

Table 1. Performance of *make* (all times in seconds)

Table 1 shows the file system performance of the current untuned prototype when executing *make* on a subset of the Solaris MC sources. The "overhead" column compares the elapsed time with that of the UFS local case.

The results and those from our other informal macro level benchmarks are very positive:

- Local file performance is within 5% of local UFS performance. This is not surprising, because when the client and server are collocated, an object invocation amounts to a local procedure call.
- The performance of remote file access is better than NFS on most macro benchmarks. This improvement is mainly due to the stateful caching done by Solaris MC (NFS validates caches at file open time by a making a remote procedure call).

For some benchmarks, the Solaris MC prototype is slower than NFS for the following reasons:

- The current implementation of pxmemcache does not have all the optimizations that have accumulated over years in the UFS and NFS code. Specifically, our prototype is not complete in the area of prefetch and write-behind for files that are accessed sequentially.
- All I/O operations are synchronous in our prototype. This impedes, for example, the performance of the pageout daemon.
- Our Ethernet-based transport has not been tuned yet, and therefore is slower than the kernel level RPC used by NFS.

We were tempted to fix the above limitations and show that Solaris MC outperforms NFS on all our benchmarks. However, such work would distract us from our original goal for the following reasons:

- First, we are in the process of switching to an SCI- based interconnect and it would make little sense to invest in improving the Ethernet-based transport.
- Second, our architecture is optimized for a cluster, and not for a network environment. Thus, certain performance optimizations should be done slightly differently for a cluster. For example, unlike NFS, the Solaris MC server does not cache file data to avoid double caching and data copies. If a cache requests data from the server, the server either transfers the data from another cache (this happens often with read-only files that are accessed from multiple nodes), or the server initiates a disk operation that transfers data directly from the disk into the remote page frame on the client node. These optimizations cannot be implemented as efficiently on the Ethernet-based transport as on the "ideal" cluster interconnect.

9 Related Work

The file system interfaces in Solaris MC were motivated by the extensible file system framework in Spring [14,15]. The main contribution of Solaris MC over Spring is in the following areas:

- seamless integration with a commercial OS (Solaris)
- support for high availability
- caching protocols
- careful optimization of data transfers

The VAXcluster file system [7] provides cluster-wide coherent access to files and devices. The fundamental design differences are:

- Solaris MC does not require an explicit distributed lock manager for synchronization. Synchronization is integrated with data transfer. This saves the overhead of additional remote procedure calls (RPCs) and eliminates the overhead of the resource and lock structures used in VMS. As a result, the per-page overhead in Solaris MC is only two bits, whereas VMS requires about a hundred bytes [6].
- While the VAXcluster uses a symmetrical distributed approach to file service, Solaris MC designates a single node to be the server for a file system at any point of time. The client/server approach was required to avoid rewriting existing Solaris file systems for the cluster.

There have been several distributed file systems developed since the introduction of vnodes [16]. Network file systems such as NFS [25], RFS [23], DECorum [12], Sprite [21], AFS [10], and Calypso [17] are the most popular examples. Solaris MC differs from these network file systems by:

- strong file-sharing semantics (same semantics as in single node UFS)
- support for high availability
- sophisticated caching
- protocols optimized for cluster interconnect rather than for a slow network

Several existing products provide high availability for NFS file systems through a system failover to a hot standby node [4,11]. Solaris MC integrates the support for HA directly into the operating system and provides much stronger file access transparency across a node failure. (Solaris MC fully preserves the UFS semantics across a failure.)

LOCUS [22,29] is another system that attempts to provide single system image for clusters. We differ from LOCUS by having well-defined interfaces in the file system, support for high availability, sophisticated caching, extensibility, and support for system evolution.

The Sprite file system [21] also provided strong file sharing semantics and sophisticated caching. We differentiate from Sprite by being based on object-oriented interfaces rather than RPC and supporting high availability, extensibility, and system evolution.

Our approach to file system extensibility is based on Spring: file systems are extended by providing multiple implementations of the file system interfaces and by using interface inheritance. Most other efforts to extend vnodes [9, 24, 26] try to achieve implementation inheritance (e.g., code sharing). While Solaris MC also shares code through implementation inheritance, we consider this feature to be of secondary importance.

Watchdogs [2] are a limited form of extending a file system through stacking. Our framework supports the functionality of watchdogs.

Our approach to high availability is similar to Tandem's disk process pair [3]. Since Solaris is a non-transactional system, each file system operation is treated as an mini-

transaction in Solaris MC to achieve the exactly-once operation semantics.

10 Summary

We have described the architecture and implementation of the Solaris MC file system. The file system interfaces effectively replace the existing vnode interface in Solaris. The new interfaces overcome many limitations of the vnode interface. The new interfaces enable, for example, file system stacking and coherent caching of files.

The design is optimized for clustered systems in which multiple nodes are connected by a high speed interconnect. The design does not assume that the nodes can share memory using hardware, but it will take advantage of such support if the platform provides it.

File data and metadata are shared through software caching protocols similar to those used in network file systems. However, the assumption of a high bandwidth and low latency interconnect led to certain significant differences from network file systems. For example, our file server object does not cache data to avoid double caching and data copying.

All the file system interfaces are specified using the CORBA IDL language and implemented using the Solaris MC distributed object framework. The interfaces have been carefully designed to combine synchronization with data transfer to avoid the overhead of an external token or lock manager.

The object model allows us to optimize the file system implementation for a specific hardware interconnect by customizing only the transport and bulkio object handler.

The file system was designed with continuous availability in mind. Server objects can be configured to checkpoint their state changes to a secondary server. The interface inheritance provided by our object framework provides for system evolution and coexistence of multiple interface revisions in the same system.

We have a relatively complete prototype that works on a cluster of nodes connected through a 100 Mb/sec Ethernet. We are currently adding checkpointing protocols for high availability and developing transport handlers for an SCI interconnect. Our goal is to achieve file system performance comparable to high-end SMP servers while providing near continuous service availability.

11 Acknowledgments

We would like to acknowledge the contribution of the other Solaris MC team members: Remzi Arpaci, Jose Bernabeu, Francesc Munoz, Madhu Talluri, Moti Thadani, and Keith Vetter.

We would like to thank Sam Cramer, Billy Fuller, Declan Murphy, John Ousterhout, Glenn Skinner, Jim Voll, and Brent Welch for their reviews of this paper. Steve Kleiman encouraged us to obsolete vnodes.

12 References

[1] Bernabeu, J., V. Matena, and Y. Khalidi, “Extending a Traditional OS Using Object-Oriented Techniques,” 2nd Conference on Object-Oriented Technologies and Systems (COOTS), June 1996.

[2] Bershad, B., and C. Pinkerton, “Watchdogs: Extending the UNIX File System,” *USENIX Winter Conference*, February, February 1988.

[3] Borr, A., “Robustness to Crash in A Distributed Database - A Non Shared-Memory Multi-Processor Approach,” *Proceedings of the 10th International Conference on Very Large Databases*, August 1984.

[4] Borr, A., and C. Wilhelmy: “Highly-Available Data Services for UNIX Client-Server Networks: Why Fault-Tolerant Hardware Isn't the Answer,” *Hardware and Software Architectures for Fault Tolerance: Experiences and Perspectives, Lecture Notes in Computer Science*, Vol. 774, pp. 285-304, Springer-Verlag, Berlin, 1994.

[5] von Eicken, T., D.E. Culler, S.C. Goldstein, and K.E. Schauser, “Active messages: a mechanism for

integrated communication and computation," *19th Annual International Symposium on Computer Architecture*, Computer Architecture News, May 1992, 20(2), pp. 256-266.

[6] Goldenberg, R., L. Kenah, *VAX/VMS Internals and Data Structures:* Version 5.2, Digital Press, 1991.

[7] Goldstein, A., "The Design and Implementation of a Distributed File System," *Digital Technical Journal*, September 1987.

[8] Gray, J., A. Reuter, *Transaction Processing: concepts and techniques*, Morgan Kaufman Publishers, San Mateo, CA, 1993.

[9] Heidemann, J., G. Popek, "Performance of Cache Coherence in Stackable Filing," *ACM Symposium on Operating Systems Principles*, December 1995.

[10] Howard, J., M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, 6(1), February 1988, pp. 51-81.

[11] IBM, *High Availability Cluster Multi-Processing/6000*, System Overview, Fourth Edition, IBM, March 1993.

[12] Kazar, M., et al., "DEcorum File System Architectural Overview," *Proceedings of the USENIX Summer 1990 Conference*, June 1990.

[13] Khalidi, Y., J. Bernabeu, V. Matena, K. Shirriff, M. Thadani. "Solaris MC: A Multicomputer Operating System," *Proceedings of Usenix 1996*, January 1996, pp. 191-203. Sun Microsystems Laboratories, SMLI TR-95-48.

[14] Khalidi, Yousef A., and Michael N. Nelson, "Extensible File Systems in Spring," *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP)*, Asheville NC, December 1993.

[15] Khalidi, Yousef A., and Michael N. Nelson, "A Flexible External Paging Interface," P*roceedings of the Usenix conference on Microkernels and Other Architectures*, September 1993. Sun Microsystems Laboratories, SMLI TR-93-20.

[16] Kleiman, Steven R., "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *Proceedings of '86 Summer Usenix Conference*, pp. 238-247, June 1986.

[17] Mohindra, A., M. Devarakonda, "Distributed Token Management in Calypso File System", *Proceedings. Sixth IEEE Symposium on Parallel and Distributed Processing*, Oct. 1994, pp. 290-297.

[18] Nelson, Michael N., Graham Hamilton, and Yousef A. Khalidi, "A Framework for Caching in an Object-Oriented System," Sun Microsystems Technical Report, SMLI TR-93-19, 1993.

[19] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Revision 1.2, December 1993.

[20] Object Management Group, *CORBA Services: Common Object Services Specification*, OMG document 95-3-31.

[21] Ousterhout, J., A. Cherenson, F. Douglis, M. Nelson, and B. Welch, "The Sprite Network Operating System," *IEEE Computer*, February 1988.

[22] Popek, G., and B. Walker, *The LOCUS Distributed System Architecture*, MIT Press, 1985.

[23] Rifkin, A., et al., "RFS Architectural Overview," *USENIX Conference Proceedings*, Summer 1986.

[24] Rosenthal, D., "Evolving the Vnode Interface," *Proceedings of the USENIX Summer 1990 Conference*, June 1990.

[25] Sandberg, R., D. Goldberg, S. Kleiman, Dan Walsh, B. Lyon, "Design and Implementation of the Sun Network File System," *USENIX Conference Proceedings*, Portland, Oregon. June 1985.

[26] Skinner, Glenn C., and Thomas K. Wong, "Stacking Vnodes: A Progress Report." *Proceedings of the Summer 1993 Usenix Conference*, 1993.

[27] Webber, N., "Operating Systems Support for Portable Filesystem Extensions," *Proceedings of the Winter 1993 Usenix Conference*, January 1993.

[28] Welch, B., "A comparison of the Vnode and Sprite file system architectures," *Proceedings of the USENIX File Systems Workshop*, 1992, pp. 29-44.

[29] Zajcew, Roman, *et al.*, "An OSF/1 UNIX for Massively Parallel Multicomputers," *Proceedings of Winter '93 USENIX Conference*, January 1993.

About the Authors

Vlada Matena is a Senior Staff Engineer at Sun Microsystems, Inc. His interests include database systems, operating systems, distributed computing, high availability, and transaction processing. He is the chief architect of the NetISAM, SunDBE, SPARCcluster Parallel Database, Java Transaction Service, and Enterprise Java Beans products. He is one of the key architects of the Solaris MC and Solaris Cluster projects.

Yousef A. Khalidi is a Distinguished Engineer at SunSoft. He is the chief architect of the Solaris Clustering product line. Yousef was one of the principal designers of the Spring and Solaris MC distributed systems, and has worked in several areas including high availability, memory management, and high-speed networking. His interests include operating systems, distributed object-oriented software, and computer architecture. Yousef was a co-winner of Sun's President Award in 1993. He has a Ph.D. in Information and Computer Science from Georgia Institute of Technology, where he was one of the principal designers of the Ra and Clouds operating systems.

Ken Shirriff is a Staff Engineer at Sun Microsystems, Inc. His interests include operating systems, distributed computing, cryptography, and fractals. He received a B. Math degree from the University of Waterloo and a Ph.D. in Computer Science from UC Berkeley.