Unix File Access and Caching in a Multicomputer Environment[1]

Paul J. Roy
OSF Research Institute
Cambridge, Massachusetts
roy@osf.org

Abstract

This paper describes the Unix file access and caching mechanisms in a version of the OSF/1 Unix operating system designed to run in a multicomputer environment. The multicomputer hardware platforms targeted can consist of hundreds or even thousands of individual nodes, where each node consists of one or more processors.

The multicomputer version of OSF/1 (called OSF/1 AD) uses Mach memory objects to cache data from Unix files, and relies on an in-kernel distributed shared memory implementation to maintain coherency for data cached across multiple nodes. The focus of this paper is on the modifications made to standard OSF/1 functionality to support distributed, efficient access to memory objects. Of particular interest are the introduction of a *mapped files module* for synchronizing clients and maintaining file meta data, the elimination of the traditional Unix buffer cache from the file data access path, and the implementation of a disk block reservation scheme to correctly support Unix write() semantics.

An evaluation of the technology is presented, providing insight into how it can be improved in the future, including several possible enhancements to Mach. As will be seen, most of this insight would equally apply to a single-node operating system based on Mach.

1 Introduction

Many computer vendors are now building computers utilizing a type of parallel processing known as Massively Parallel Processing (MPP). MPP computers, also known as *multicomputers*, can consist of hundreds or even thousands of nodes connected via a high-speed interconnect. Each node may contain one or more processors. In a typical multicomputer system (see Figure 1), the nodes of the multicomputer are divided into three groups: nodes used for input/output and connectivity (I/O nodes or file server nodes); nodes dedicated to parallel applications (compute nodes); and nodes for interactive use (service nodes). The number of nodes in each group may vary depending on the particular configuration.

The Open Software Foundation's Research Institute has built a multicomputer version of the OSF/1 operating system, called OSF/1 AD,[2] that provides a view of the hardware that looks like a conventional shared memory multiprocessor with an unusually large number of processors. The op-

[1]This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Material Command (AFMC).

[2]OSF/1 AD is an acronym for OSF/1 with Advanced Development (AD) extensions from the OSF Research Institute.

erating system presents this notion of a Single System Image by building Unix functionality on top of base Mach services running on each node in the multicomputer.

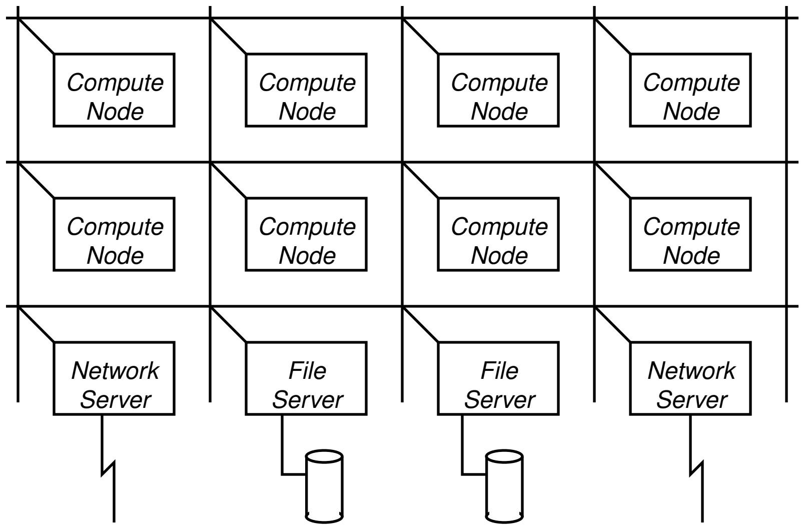

Figure 1: Multicomputer Architecture

This paper describes OSF/1 AD's Unix file access and caching mechanisms. It uses Mach memory objects to cache data from Unix files, and relies on an in-kernel distributed shared memory implementation to maintain coherency for data cached across multiple nodes. The focus of this paper is on the modifications made to standard OSF/1 functionality to support distributed, efficient access to memory objects. Of particular interest are the introduction of a *mapped files module* for synchronizing clients and maintaining file meta data, the elimination of the traditional Unix buffer cache from the file data access path, and the implementation of a disk block reservation scheme to correctly support Unix write() semantics.

An evaluation of the technology is presented, providing insight into how it can be improved in the future, including several possible enhancements to Mach. As will be seen, much of this insight would equally apply to a single-node operating system based on Mach.

After providing background information on OSF/1 AD and Mach memory objects, the remainder of the paper presents the goals and design of Unix file access, and evaluates the resulting technology. The paper concludes with sections on related work, implementation status, and ongoing work.

2 Background

2.1 OSF/1 MK Single Server

The Open Software Foundation's OSF/1 is an open operating system that incorporates advanced features while providing compatibility with industry standards and support for existing applications [18] [19]. OSF/1 is based on the Mach 2.5 operating system from Carnegie Mellon University, both of which are integrated or monolithic operating systems that implement the majority of traditional operating system functionality in the kernel.

The OSF/1 MK Single Server was created by dividing the OSF/1 monolithic kernel into the Mach 3.0 kernel and the OSF/1 MK Single Server. This was achieved by replacing OSF/1's Mach 2.5 internals with Mach 3.0, and adding a layer of compatibility code to allow the rest of OSF/1 to execute as a user-mode server, the OSF/1 MK server. Among the features of the compatibility code are a threads library that provides lightweight user threads on top of Mach's kernel threads, and an *emulator* that implements some system functionality in application address spaces. This work was based on an earlier conversion effort at CMU involving the Mach 2.5 system [8]. The resulting OSF/1 MK server is pageable, preemptable, and multithreaded.

2.2 OSF/1 AD - A Multicomputer Version of OSF/1

OSF/1 AD extends the OSF/1 MK Single Server to a multicomputer environment [30]. Its architecture is derived directly from the target hardware architecture, in which large numbers of nodes are connected via a high-speed interconnect, and some subset of the nodes have peripheral devices attached. OSF/1 AD must allow all nodes and devices to be efficiently utilized and, in particular, must distribute OS functionality to avoid bottlenecks.[3]

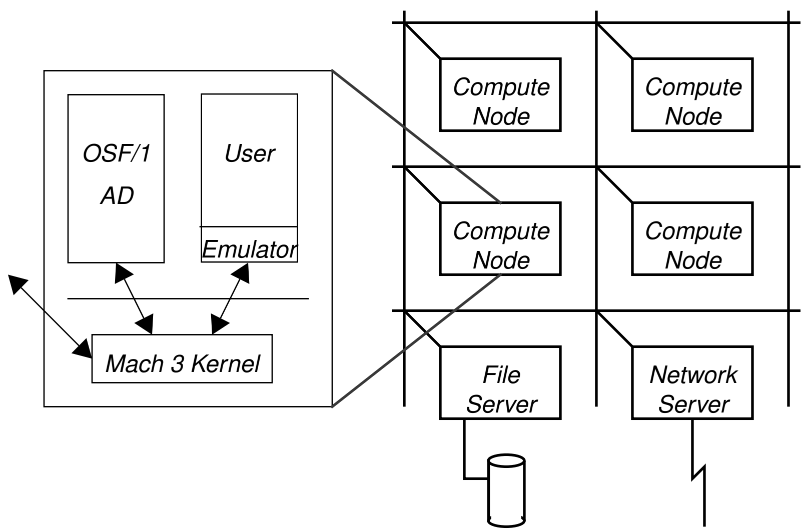

Figure 2: OSF/1 AD Architecture

These considerations lead to the following architectural model (see Figure 2):

- The Mach kernel runs on all nodes of the multicomputer, providing generic task/thread management, memory management, communication services, and device access.
- Specialized user-space servers implement Unix functionality, such as file service, process management, and networking.
- Disk and networking devices are managed by servers typically co-located on the same node with the device, although co-location is not mandatory.
- Process management functionality is distributed across most or all nodes on which application processes run.
- An *emulator* [8] [13] is available in each process's address space to provide some Unix functionality and perform system call-to-message conversion. It also contains a *callback thread* to receive messages from servers in support of interruptible system calls and file caching.

[3] Scalable process management functionality is provided by Transparent Network Computing (TNC) extensions from Locus Computing Corp.

- Process management system calls that require client-server interaction are converted by the emulator to messages to the process management server, which normally resides on the same node as the process. File management system calls that require client-server interaction are converted to messages to the file server managing the particular file being manipulated.

OSF/1 AD implements the entire OSF/1 Application Programming Interface (API). Although OSF/1 AD assumes no physical shared memory between nodes, it supports shared memory between processes residing on different nodes in software (see Section 2.4). Inter-node communication is provided by a modified version of the Mach network IPC presented in [1].

2.3 OSF/1 AD File System

OSF/1 AD extends the standard OSF/1 file system to support the integration of multiple file servers into a coherent whole. Each file server provides file service for one or more file systems (partitions), and mount operations are used to assemble multiple file systems across multiple file servers into a single file name space (Figure 3).

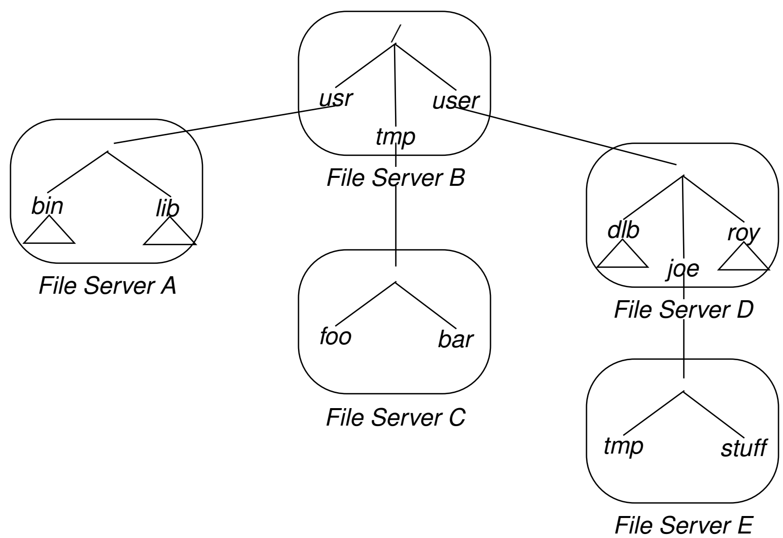

Figure 3: File Name Space

As in standard OSF/1, processes access files via system calls. In a multicomputer environment, it is extremely important to implement these system calls in a manner that minimizes IPC messages and server context switches. OSF/1 AD accomplishes this using the Mach system call redirection mechanism [8] to implement functionality in a per-process emulator residing in each process's address space. In particular, the emulator:

- contains the per-process file descriptor table, where each table entry has a Mach port right mapping to the open file structure on a file server. Hence, operations on open files communicate with at most one server.

- contains send rights to the Mach ports representing the root and current working directories for the process. Hence, most (but not all) operations on pathnames (e.g., open()) communicate with one server.

- provides a location to implement mapped access to open files and accrue the benefits of node-local file data caching. It is this aspect of the system that is the primary subject of this paper.

2.4 Mach Memory Objects

The address space of a Mach task is represented as a collection of mappings from addresses to offsets within memory objects. A Mach memory object represents a single source of memory (e.g., the file from which an executing program was loaded). The kernel manages physical memory as a cache of memory object contents; access to the actual memory (i.e., backing storage) is via a Mach port to which messages can be sent containing data or requesting that it be supplied [28] [29]. This allows memory objects to be implemented by user-state managers such as a file server or database application. The interface between the kernel and managers is known as the *External Memory Management Interface.*

To support access to memory objects on a multicomputer system, the XMM (eXtended Memory Management) subsystem has been added to the Mach kernel [2]. XMM supports access to memory objects from any node in the multicomputer, implementing a multiple-reader/single-writer policy for managing inter-node cache coherency. The policy is maintained on a per-page basis. The result is that pages from a single memory object may be cached across multiple nodes, with pages moved between nodes as necessary to maintain coherency in the face of ongoing memory object accesses throughout the multicomputer.

XMM is part of every Mach kernel on a multicomputer. Its implementation makes the collection of Mach kernels on a multicomputer behave as if they were a single kernel when communicating with memory object managers. This removes the complexity of distributed shared memory functionality from memory object managers. XMM also includes support for copy on reference between nodes for lazy evaluation of memory inherited between tasks on different nodes.

3 Goals

The primary goal of the Unix file access architecture was to enable efficient caching of file data on compute nodes. Achieving this goal would largely satisfy the performance and scalability requirements of a multicomputer file system because file data is frequently re-accessed by Unix applications.

Another goal was to support flexible allocation of memory cache resources to different types of data, depending on the behavior of applications. For example, a compute node running applications accessing large amounts of file data should have a much larger portion of its memory dedicated to cached file data versus, say, a compute node running applications with very large text and heap requirements.

Lastly, the design had to be amenable to a user-space implementation layered on top of the Mach kernel.

It is important to note that the time available for implementation was a constraint. For this reason, caching of other file system information, such as pathnames and attributes, was not implemented. Also, it was not a goal of this work to satisfy the needs of many supercomputer applications which require very high bandwidth access to vast amounts of data (but see Section 9).

4 Unix File Access and Caching

Because Mach memory objects have the ability to cache data provided by extermal, user-space managers, they are well suited to meet the goals outlined in Section 3. However, memory objects are only cache repositories; they don't provide full Unix file semantics. Hence, an additional level of functionality must be provided. Specifically:

Client synchronization. Each Unix read() and write() system call must be performed atomically with respect to other reads and writes of the same data.

File meta data. An open file's seek pointer, length, and its accessed and modified status must be maintained separately from the file's data.

Cache control. Cleaning and invalidating memory object-resident data must be done at appropriate times on behalf of some Unix system calls. For example, sync() must cause dirty file data to be written back to disk.

After describing the role of memory objects in Unix file access and caching, each of these items is discussed. In addition, changes to the underlying backing storage manager (the Unix File System) are presented. Figure 4 shows the relevant modules for this discussion.

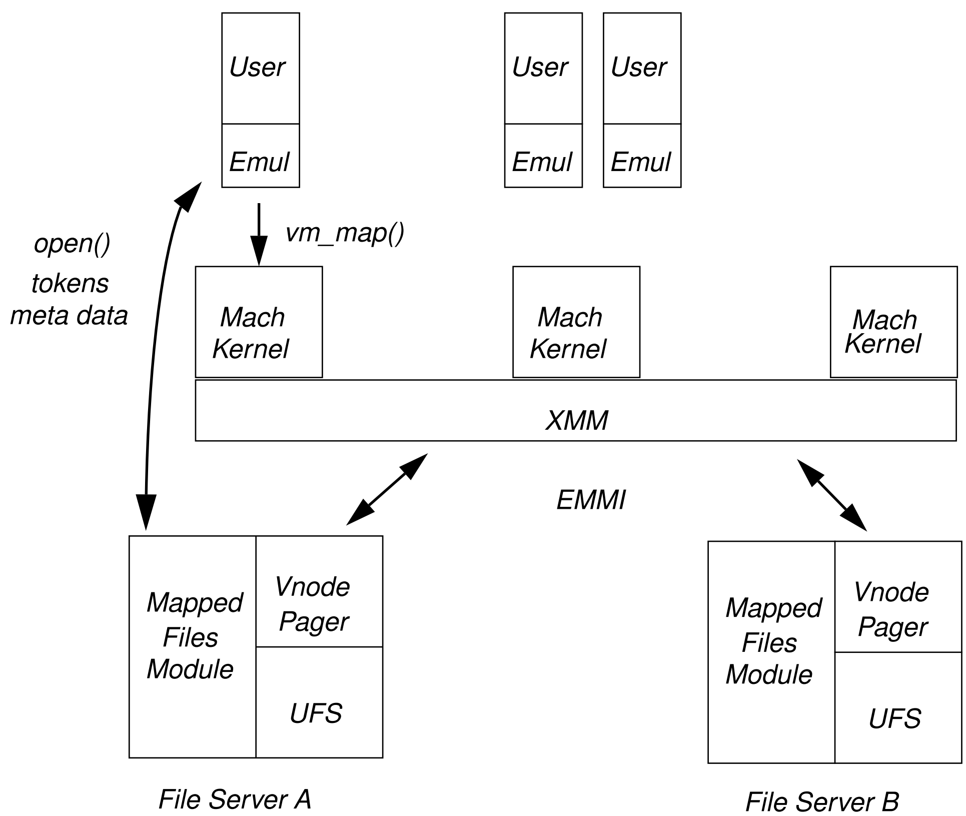

Figure 4: File Access and Caching Architecture

4.1 The Role of Memory Objects

A Mach memory object is used to cache data from a regular Unix file in the following manner:

- A file's memory object port is obtained at open() time from the file server and stored in the emulator's file descriptor table.
- read() and write() system calls are converted by the emulator to mapped accesses to the memory object. These accesses are accomplished by mapping the memory object port, and using bcopy() to access the mapped region.
- Cache hits to already-resident data are satisfied without file server interaction. Also, because mapped regions are retained across successive Unix system calls, these cache hits may not involve any page faults or TLB misses.

- Cache misses are handled by the Mach kernel using the External Memory Management Interface to request data from the file server.
- The memory object cache is effectively "write-back." The Unix sync() system call causes dirty data to be written back to the file server (again via the External Memory Management Interface), which in turn writes it to disk.
- Mach's XMM kernel subsystem provides inter-node cache coherency of file data (XMM is discussed in Section 2.4).

Note that the Mach 3.0 4.3BSD Single Server and Chorus SVR4 systems [3] [6] have implemented similar file caching schemes for single-node systems.

4.2 Client Synchronization

Implementing POSIX file I/O semantics [10] requires that read() and write() system calls be performed atomically with respect to other reads and writes of the same data. In a single-node environment, this has historically been provided by a low-level file system lock (e.g., the inode lock). In a distributed environment, another mechanism is needed for synchronizing clients, while also minimizing the amount of necessary communication.

OSF/1 AD implements a module, known as the *mapped files module* (not to be confused with mmap() functionality), for synchronizing distributed clients. This module exports the notion of a *token*, which provides its holder the right to access a particular file's data with either read or read/write capability. An instance of the mapped files module is implemented within each file server and supports the files managed by that server. Message-based interfaces are provided for acquiring and releasing tokens.

A client emulator wishing to access a file must first acquire a token from the file's file server (with read or read/write capability), at which point access to the corresponding memory object may proceed. Subsequent I/O's to the same file may find the token still held. However, it is also possible the token has been revoked, in which case a message to the file server is necessary to reacquire a token.

A file server revokes a token by sending a message to an emulator's callback thread, instructing it to release the token. Token revocation occurs whenever an emulator attempts to acquire a token specifying a capability conflicting with already outstanding tokens. Other reasons for revoking tokens are associated with file meta data (see Section 4.3).

Tokens are implemented using Mach ports. In the event of an abnormal process abort, in which case tokens may not be released properly, file servers will receive no-more-senders notifications from the kernel [14] for all outstanding tokens, allowing them to perform necessary clean up.

4.3 Maintaining File Meta Data

In addition to synchronizing client data access, tokens are used to implement caching of file meta data to enable node-local file data caching, while still correctly implementing read() and write() semantics. Specifically, possession of a token allows a client emulator to cache an open file's:

- seek pointer
- length
- accessed and modified status

The seek pointer determines the offset within the memory object to read or write. The length is necessary to implement append-mode writes, and to prohibit reading beyond the end-of-file. Retaining whether a file has been read or written is necessary so that proper accessed and modified times may be recorded.

When a token is released by an emulator (either voluntarily or because of a revocation), the cached state associated with the token is written back to the file server. If an application closes a file, the application's emulator voluntarily releases the token.

There are cases when a file server itself needs access to state that may be cached. For example, truncate() requires exclusive access to the file and its length, sync() must determine if there is modified file data to be written back to disk, and stat() must determine whether the file has been accessed or modified so that proper times may be reported. The mapped files module supports such cases by exporting well-abstracted interfaces to the rest of the server's file system code, thus hiding the fact that the implementation is revoking tokens as necessary.

Note that POSIX semantics [10] for updating a file's accessed and modified times only require that they be updated during the last close() of the file, or if the file's attributes are read (due to stat() or fstat()). Thus, emulators retain booleans indicating whether a file has been accessed or modified, relying on file servers to revoke tokens and update times as necessary. In addition to the requirements specified by POSIX, sync() also revokes tokens so that times may be updated. This results in behavior equivalent to a traditional UFS where booleans are similarly maintained in inodes.

Ideally, caching a file's meta data would support a multiple-reader/single-writer policy for accessing the file's memory object, with the exception of the seek pointer which necessitates a mutual exclusion policy for all processes sharing an open file. However, at the current time, the mapped files module's implementation is not yet optimized to distinguish these cases, resulting in mutual exclusion for all processes accessing a file's memory object. Exclusion lasts for the duration of time it takes to access/update the memory object on behalf of any given read() or write(). (This problem is being rectified as part of ongoing work - see Section 9).

4.4 Memory Object Cache Control

Although Mach memory objects are used to hold cached file data, *most* of the control of the cache is the responsibility of file servers. In particular, a file server will invalidate cached data on behalf of a truncate() system call, and cause dirty pages to be written back to disk on behalf of a sync() or fsync() system call. In addition, a file server may choose to mark a memory object as "temporary," preventing write-backs of dirty data to the file server when the memory object is no longer active. When unlinking files, this technique avoids write-backs of data that will just be deallocated anyways.

This server-based cache control is performed by the mapped files module in close cooperation with another file server module, the *vnode pager.* The vnode pager interfaces directly with the Mach External Memory Management Interface.

Cache control as it relates to inter-node cache coherency of file data is the responsibility of the XMM kernel subsystem. It implements a multiple-reader/single-writer policy on a per-page basis. Also, per-node page cache management (e.g., page reclamation) is still under the jurisdiction of the local Mach kernel.

4.5 Enhancements to the Unix File System (UFS)

The Unix File System (UFS) is used to provide disk backing storage for the system. Some important changes have been made to the standard OSF/1 UFS to provide better performance and correct behavior in OSF/1 AD.

No-buffer-cache. A performance-motivated change was the elimination of the traditional Unix buffer cache from the access path for file data. The buffer cache is still used for directory and meta data. Thus, when file data is read or written through the UFS, the UFS provides allocation of disk blocks and translation from logical file offsets to physical disk block addresses. The UFS provides no file data caching.

This *no-buffer-cache* enhancement was made for several reasons. First, because memory objects already cache data, it avoids a double caching effect. Second, it eliminates data copies into and out of the buffer cache. Third, it supports a mode of file access where data caching of any sort, even memory object caching, is undesirable. This mode is useful for satisfying the high bandwidth requirements of many supercomputer applications (see Section 9).

Besides data caching, the traditional buffer cache provides other functionality, namely read-ahead and write-behind. In OSF/1 AD, these functions were implemented and appropriate synchronization provided to avoid access to stale data. This involves blocking read requests while conflicting asynchronous disk writes are in progress because the kernel doesn't provide ordering guarantees, and synchronizing incoming write requests with the read-ahead mechanism. An important distinction between this read-ahead mechanism and that provided by the traditional buffer cache is that a hit on read-ahead data actually removes the data from the read-ahead cache, thus avoiding the unwanted copy and double caching effect.

Another important aspect of this design is that it will enable the UFS to coalesce contiguous file system blocks into larger disk reads and writes, thus obtaining greater disk bandwidth. Other systems have accrued this same benefit via extents [5], or extent-like mechanisms [16]. This optimization has not yet been implemented.

Disk block reservation. When write() is called, the system must guarantee that enough disk space exists to satisfy the write, and if not, return an ENOSPC error. For newly written data (i.e., no disk space yet exists), a memory object cache miss is guaranteed to occur, which provides the file server with an opportunity to allocate disk space. However, the UFS can't simply allocate disk blocks and enter them into the inode map because a subsequent crash would make those blocks appear to be valid portions of the file when in fact they don't yet contain valid data.

This problem necessitates some form of disk block reservation. The possibility of simply decrementing the free block count for the partition has appeal, but was rejected because of the inability to determine how much to add back to the free block count when a file with holes is truncated. Also rejected was the use of in-core data structures to keep track of all reserved blocks for all files, because of their potential to grow very large, and because of the desire to avoid allocating additional memory resources at a time when the system is already under memory pressure due to data being written.

The solution chosen relies on stealing the high bit of the disk address (daddr) fields stored in inodes to indicate that the rest of the daddr field represents a "reserved" disk address. In other words, setting this bit indicates that the disk address is valid but the corresponding disk blocks do not yet contain valid data. Hence, at the time of write(), disk addresses are reserved, but then converted to real disk addresses (by masking off the high bit) when the data is actually written back from the kernel to the UFS.

This scheme required a couple hundred lines of additional code to the UFS, but they are very well contained within the allocation and truncation routines. A few lines of changes were also needed to ufs_fsck so that it would treat reserved disk addresses as invalid when the partition is salvaged after a crash.

Since daddr's address file system fragments, this change decreases the maximum amount of addressable fragments in a partition. But, conservatively assuming a 1 Kb fragment size, this results in the maximum partition size decreasing from 4 terabytes to 2 terabytes, which is tolerable, especially given that this limit can be overcome by either using a larger fragment size or partitioning a disk into more partitions.

5 Evaluation

This section evaluates the system, providing insight into how it can be improved in the future, including several possible enhancements to Mach. The particular areas addressed are:

- Mach's External Memory Management facility.
- The mechanisms provided by Mach for accessing memory object data.
- Variable-sized caching for different types of data.
- Inter-node coherency of file data.
- The mapped files module.
- Read-ahead for memory object-based files.
- The emulator.

5.1 External Memory Management

The External Memory Management functionality provided by Mach has proved to be well suited to a modular file system implementation. Of particular note is the fact that caching functionality can be cleanly separated from management of backing storage. Alternative designs where caching and backing storage are tightly integrated can result in intricate virtual memory/file system interaction [16], and are not well suited to a user-space implementation of backing storage managers [3].

An effect of the clean separation is that it necessitates an extension to the backing storage manager (in this case, UFS) to reserve disk blocks so that correct ENOSPC semantics for the write() system call can be implemented. However, these changes are quite well encapsulated and are expected to also be useful for implementing pre-allocated and real-time files.

A difficulty with using the External Memory Management functionality is maintaining the correct length of a Unix file. Because Mach memory objects do not have a notion of length, write-backs of data from the kernel to the backing storage manager are of page size granularity. As described previously, the OSF/1 AD file system is able to maintain the exact length (in bytes) separately, but not without complexity.

In particular, when a file is being grown due to the write() system call, the memory object and length of the file must be updated atomically by the emulator. Unfortunately, while a memory object is being updated it is possible for the page(s) being modified to be written back to the manager (e.g., due to high memory demand from other applications), blocking the thread performing the update. The manager, in turn, may then have to send a callback to the emulator to determine the exact length of the file so that it may write the proper amount to disk. To avoid deadlock, the execution of this callback must avoid blocking on the blocked thread, but doing so violates the atomicity requirement of the file grow operation.

The solution chosen for this problem allows the atomicity violation to occur, and because the emulator updates the file length before the memory object, can result in the length of the on-disk file temporarily being too large relative to the amount of valid data it contains. However, the system guarantees that the page(s) in question will be written back again, this time containing enough valid data so that the length of the on-disk file accurately reflects the amount of valid data.

In practice, this solution works because 1) the chances of a page being written back while it is being modified is unlikely, and 2) even then, a crash would have to occur to expose the invalid data (zero's) at the end of the file to applications. Nonetheless, additional support at the Mach level to prevent pages from being written back while they are being modified would eliminate the problem altogether and avoid much complexity in the Unix layer. Although Mach's vm_wire() call [14] could be used to wire the pages involved, a cheaper mechanism is needed (such as the one proposed in Section 5.2)

Perhaps the most attractive solution, from the perspective of Unix file access, would be for Mach to maintain a notion of *length* of memory objects, thus enabling write-backs of data on a byte-size granularity and reducing the burden on the Unix layer to maintain the exact length of cached data. However, this represents a fundamental change to the Mach memory object abstraction that must not be taken lightly.

5.2 Memory Object Access

The only way to access a Mach memory object is to map it and access the mapped region (e.g., via bcopy()). Although there is some overhead associated with mapping, OSF/1 AD maps sufficiently large windows into memory objects to amortize the cost across many (smaller sized) read() and write() system calls. And, once a particular page has been faulted in and installed in the TLB, subsequent accesses to the same page can be very fast because faults and TLB misses are avoided.

Unfortunately, accessing memory objects via page faults has a serious problem: information is lost across the Mach boundary. With respect to Unix file access, this problem is manifested in the following ways:

Size of reads/writes. Because Mach treats all page faults as a request for one page, it has no knowledge of the amount of data actually being read/written. In the case of reading, it is thus unable to request larger amounts from the backing storage manager. Although the fault path *could* guess and read-ahead, it is suboptimal (consider large, non-sequential reads). Mach's inability to intelligently "cluster" data can have a significant performance impact due to extra disk I/O's.

Note that standard OSF/1 has implemented a form of clustering [4], where Mach's notion of cluster size is set on a per-object basis by the file system, using the External Memory Manager Interface. This is sufficient for access to anonymous memory where the optimal cluster size is mainly a function of the optimal disk access time, and hence is known to the file system at the time the object is initialized. It is also sufficient for access to mmap() regions where the cluster size can be initialized and later changed based on advice from the Unix madvise() system call.

However, for Unix file access the cluster size should be a function of the size of the particular Unix read() or write() system call in progress. But, changing the cluster size via the External Memory Manager Interface during the execution of read() or write() would be prohibitively expensive, especially if the application is remote from the file server. And, it would suffer from the inability to handle the case of multiple applications simultaneously accessing a memory object, each with a different optimal cluster size.

Also, in the common case of growing Unix files, the associated write faults will make one request to the backing storage manager for each page (to find out if data for the page exists, and if not, zero-fill it). Many of these requests could effectively be "batched" if Mach was able to contact the backing storage manager on behalf of larger sizes.

Error returns. There is no way to return error codes from a page fault. This is especially problematic for implementing ENOSPC semantics of the write() system call, which requires that an ENOSPC error code be returned if there is no more disk space left to satisfy the write(). OSF/1 AD handles this by passing ENOSPC to the memory_object_data_error() kernel interface, which in turn generates an exception. The exception handler in the file server then modifies the program counter of the thread blocked accessing the memory object, forcing it to "jump" to a new location to pick up an ENOSPC error code. The location to jump to must have previously been "registered" with the exception handler. Obviously, a simpler mechanism would be preferable.

Type of access. The fault path is unable to distinguish between faults due to read() and write() versus those due to accesses to mmap() regions. A key difference in semantics for these cases is that writes beyond the end-of-file should grow the file whereas attempts to update mmap() regions beyond the end-of-file should not. Barring changing mmap() semantics (to prevent mapping a file beyond the end-of-file), an efficient solution to this problem is difficult (if not impossible) without additional information regarding the type of access.

These examples justify experimentation with new Mach interfaces for explicitly copying data to and from memory objects. These interfaces would provide the necessary channels through which access size and other information could be passed. They could also be brought to bear on the problem with file length handling described in Section 5.1 because the pages involved could be temporarily wired while they are being accessed An important aspect of this experimentation would be to quantify the negative performance impact of the resulting additional Mach system calls.

5.3 Variable-Sized Caching

Because memory objects are used for caching both file data and program data (text, heap, etc.), the amount of memory allocated to the different types of data can vary, depending on the particular load. The Sprite system has demonstrated the benefit of such variable-sized caching [17]. Indeed other systems are unifying caching in this way [3] [7].

A problem is that Mach treats all memory equally when making page reclamation decisions, with the exception of deactivating-behind pages (making them better candidates for reclamation) from sequentially accessed memory objects. And, in all cases, Mach doesn't know whether pages belong to a memory object containing file data versus a memory object containing other types of data. This precludes biasing reclamation decisions based on the type of data.

There are several approaches to this problem. One is to provide knowledge to the kernel about the type of data and/or access behavior, but still allow it to make the ultimate page reclamation decisions (the Sprite system falls in this category). Other approaches allow external managers to affect all page management decisions to varying degrees [9] [15] [23]. For the particular problem at hand, providing additional information to the kernel (e.g., type of object, expected access behavior) so that it may make more intelligent page reclamation decisions is probably sufficient.

5.4 Inter-Node Coherency of File Data

XMM has proved adequate as a mechanism for providing inter-node coherency of file data. An aspect of this approach, however, is that XMM only has semantic knowledge of memory objects. While this is often sufficient for distributed shared memory implementations, its lack of information regarding Unix file sharing behavior can have a performance impact. In particular, when a file is cached in one node and, say, written on another node, XMM is able to move pages between nodes but does so one page at a time while they are faulted on.

A possible optimization is to provide inter-node cache coherency at the Unix layer. It would be a straightforward extension to perform appropriate flushing or invalidation as part of the token revoke mechanism, thus accruing the benefits of operating on multiple pages at a time and avoidance of XMM machinery. Currently, though, flushing or invalidating data can only be done by the file server because it holds the control port for the memory object, which would involve extra messages. A new Mach interface would be needed to support this type of cache control using only the memory object port.

Also, XMM is "involved" for all memory object messages whether it is needed or not. Although this provides multi-node transparency from the perspective of backing storage managers, there is some cost for objects not needing its capabilities. Indeed, given that most Unix files are *not* shared, a design that assumes all files *are* shared seems suboptimal. Further investigation is needed to quantify the costs imposed by XMM.

A last note is that XMM is not yet able to process multi-page write-backs of data. As a result, many write-backs to backing storage are artificially broken up into single page units, thus impacting disk write bandwidth. This must be rectified in future versions of XMM (see Section 9).

5.5 Mapped Files Module

The mapped files module hides the implementation of tokens, cached file meta data, cache control, etc. from the rest of the Unix code via well-abstracted interfaces. For example, there are interfaces for cleaning, truncating, getting a file's size, making a file temporary, etc. For implementation expediency, invocation of its interfaces are sprinkled in code that is otherwise unmodified from standard OSF/1 code. But, because the semantics of the mapped files module is effectively a caching module layered *above* the disk storage manager, it appears well-suited to a stacked vnode implementation [25]. In fact, the current implementation is already quite well layered and should easily transition to a stacked implementation.

At the current time, the mapped files module only supports mutually exclusive access to a file's memory object. This must be optimized to support a multiple-reader/single-writer policy.

5.6 Read-ahead

As described previously, the UFS no-buffer-cache code implements read-ahead so that the disk wait time for subsequent data requests from the kernel can be reduced or eliminated. Although Unix has historically performed read-ahead at this layer, it is suboptimal for a variety of reasons.

First, it is making policy decisions about *when, where,* and *how much* to read-ahead, although such decisions could more flexibly be provided at a higher layer in support of application-specific read-ahead policies. Second, initiating reads from deep within the UFS is problematic for systems needing to intercept all reads, such as hierarchical storage systems (see [26]). Third, the UFS doesn't have knowledge of what data is already cached, and hence may perform wasted read-aheads. Fourth, because memory object cache hits do not involve any interaction with the UFS, no read-aheads will be initiated in these cases. Fifth, even when data is found in a read-ahead buffer the latency for the application can be quite long, especially if the application is remote from the file server.

All of these deficiencies can be rectified by initiating read-aheads from a higher layer and reading the data directly into memory objects. The kernel would perform such reads (using the External Memory Manager Interface) but the policy of *when*, *where*, and *how much* to read-ahead could be left to a layer above the kernel. Again, such a scheme would require that additional information be provided to Mach, perhaps as extension to the explicit memory object access interfaces suggested in Section 5.2.

5.7 Emulator

The main benefit of the emulator is that it is a place to implement system functionality "near" the application (i.e., it is quick to get to when an application makes a system call). However, it has problems. One is that it has become quite large which, because it is per-process, has memory consumption and process creation overhead ramifications. But, perhaps most importantly, its unprotected nature results in a fair amount of complexity and overhead in servers to "defend" themselves against the possibility of corruption in the emulator's state, and often results in an artificial partitioning of code and state based on protection concerns rather than real modularity concerns.

Future revisions of the OSF/1 AD system will utilize an alternative to the emulator where system calls are redirected to a local server using the exception mechanism [22].

6 Relevance to Single-Node Systems

Although OSF/1 AD is targeted for multicomputers, most of the work described in this paper would equally apply to a single-node operating system based on Mach. In fact, the entire evaluation (Section 5) is relevant, excluding the discussion on inter-node data coherency (Section 5.4). And, the interfaces to the mapped files module would remain largely unchanged, although its implementation could be simplified, especially in a system without the emulator (see Section 5.7) because all file access would be centralized in the server, thus obviating the need for tokens.

A key aspect of OSF/1 AD's file caching design is the clean separation between caching functionality and management of backing storage. This separation would equally apply to a single-node operating system, and would result in a more modular system that should enable easier integration of new backing storage managers.

7 Related Work

The Mach 3.0 4.3BSD Single Server and Chorus SVR4 systems [3] [6] have implemented Unix file caching schemes based on unifying the cache management of file data and program data, although these systems are restricted to single-node environments. SunOS [7] also unifies caching but is restricted to a single-node, monolithic kernel environment.

Some early work on using Mach memory objects for Unix file caching was done in previous Mach systems [24].

Sprite [21] is a research system designed for a network of workstations. It provides a single file system image throughout all nodes in the cluster [27], and unifies the cache management of file data and program data [17].

Multics [20] and Apollo DOMAIN [13] are early examples of non-Unix operating systems providing the notion of a single level store, where access to all data is by mapping objects. Multics runs on single-node systems whereas DOMAIN runs across a network of workstations.

8 Implementation Status

OSF/1 AD is the base for the operating system for the Intel Paragon XP/S Supercomputer [11]. Typical configurations of the Paragon range from fewer than 100 nodes to more than 1000 nodes. Additionally, OSF/1 AD runs on clusters of i386 and i486 PC's on an Ethernet, simulating a multicomputer.

All of the work discussed in this paper is available from the OSF Research Institute in its OSF/1 AD prototype which is available as part of the standard OSF/1 license.

9 Ongoing Work

All of the suggested improvements in the evaluation (Section 5) are being pursued, with initial emphasis on enhancing the mapped files module to support a greater degree of concurrency, modifying XMM to enable processing of multi-page write-backs, and improving the memory object access and variable-sized caching mechanisms. In addition, future revisions of the architecture will not have an emulator and the mapped files module will be better integrated with the rest of the standard OSF/1 code.

Performance measurement and tuning is also underway. An accurate performance evaluation of the overall architecture will not be possible, however, until the system is able to perform multi-page data transfers across the External Memory Management Interface (see Sections 5.2 and 5.4).

There is also work in progress to support the needs of many supercomputer applications which require very high bandwidth access to vast amounts of data. A file access path bypassing all data caching will be provided, leveraging the existing UFS no-buffer-cache code.

10 Conclusion

This paper has presented the Unix file access and caching architecture in OSF/1 AD, a version of the OSF/1 operating system designed to run in a multicomputer environment. OSF/1 AD uses Mach memory objects to cache data from Unix files, and relies on an in-kernel distributed shared memory implementation to maintain coherency for data cached across multiple nodes.

The focus of the paper was on the modifications made to standard OSF/1 functionality to support distributed, efficient access to memory objects. These modifications include the introduction of a mapped files module for synchronizing clients and maintaining file meta data, the elimination of the traditional Unix buffer cache from the file data access path, and the implementation of a disk block reservation scheme to correctly support Unix write() semantics.

A key aspect of OSF/1 AD's file caching design is the clean separation between caching functionality and management of backing storage. This separation results in a more modular system that should enable easier integration of new backing storage managers.

An evaluation of the system has identified many possible areas for improvement. Suggested enhancements being pursued, include:

- New Mach interfaces for explicitly copying data to and from memory objects.
- More intelligent page reclamation decisions within Mach.
- Provision of inter-node cache coherency of file data in the Unix layer rather than relying on Mach's XMM functionality.
- An improved read-ahead scheme for memory object-based Unix files.

Although OSF/1 AD is targeted for multicomputers, most of the work described in the paper, including the suggested enhancements, would equally apply to a single-node operating system based on Mach.

11 Acknowledgments

Paul Dale, George Feinberg, and Alan Langerman provided useful comments on the contents and presentation of this paper.

References

[1] Barrera, J. A Fast Mach Network IPC Implementation. In *Proceedings of the USENIX Mach Symposium*, November 1991.

[2] Barrera, J. Copying, Caching, and Sharing in a Distributed Virtual Memory System. Ph.D. dissertation, Carnegie-Mellon University, to appear 1993.

[3] Batlivala, N., Gleeson, B., Hamrick, J., Lurndal, S., Price, D., Soddy, J., Abrossimov, V. Experience with SVR4 over Chorus. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Architectures*, April 1992.

[4] Black, D., Carter, J., Feinberg, G., MacDonald, R., Mangalat, S., Shienbrood, E., Van Sciver, J., Wang, P. OSF/1 Virtual Memory Improvements. In *Proceedings of the USENIX Mach Symposium*, November 1991.

[5] Cheriton, D., Roy, P. Performance of the V Storage Server: A Preliminary Report. In *Proceedings of the 1985 ACM Computer Science Conference*.

[6] Dean, R., Armand, F. Data Movement in Kernelized Systems. In *Proceedings of the USENIX Workshop on Microkernels and Other Architectures*, April 1992.

[7] Gingell, R., Moran, J., Shannon, W. Virtual Memory Architecture in SunOS. In *Proceedings of the Summer 1987 USENIX Conference*.

[8] Golub, D., Dean, R., Forin, A., Rashid, R. Unix as an Application Program. In *Proceedings of the Summer 1990 USENIX Conference*.

[9] Harty, K., Cheriton, D. Application-Controlled Physical Memory using External Page-Cache Management. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. October, 1992.

[10] IEEE. Portable Operating Systems Interface (POSIX) - Part 1: System Application Program Interface. IEEE Std 1003.1, 1990. ISO/IEC 9945-1.

[11] Intel Corporation. Intel Paragon XP/S Supercomputer Spec Sheet. 1992.

[12] Julin, D., Chew, J., Stevenson, M., Guedes, P., Neves, P., Roy, P. Generalized Emulation Services for Mach 3.0 - Overview, Experiences, and Current Status. In *Proceedings of the USENIX Mach Symposium*, November 1991.

[13] Leach, P., Levine, P., Hamilton, J., Stumpf, B. The File System of an Integrated Local Network. In *Proceedings of the 1985 ACM Computer Science Conference*.

[14] Loepere, K. (editor). *Mach 3 Kernel Interfaces*. Open Software Foundation and Carnegie Mellon University.

[15] McNamee, D., Armstrong, K. Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies. In *Proceedings of the USENIX Mach Symposium,* October, 1990.

[16] McVoy, L., Kleiman, S. Extent-like Performance from a Unix File System. In *Proceedings of the Winter 1991 USENIX Conference*.

[17] Nelson, M. Physical Memory Management in a Network Operating System. Ph.D. dissertation, University of California at Berkeley, November 1988. UCB/CSD 88/471.

[18] O'Reilly & Associates, Inc. Guide to OSF/1: A Technical Synopsis. 1991.

[19] Open Software Foundation. The Design of the OSF/1 Operating System. 1990.

[20] Organick, E. The Multics System: An Explanation of Its Structure. M.I.T. Press, 1972.

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

[22] Patience, S. Redirecting System Calls in Mach 3.0: An Alternative to the Emulator. In *Proceedings of the USENIX Mach Symposium*, April 1993.

[23] Sechrest, S., Park, Y. User-Level Physical Memory Management for Mach. In *Proceedings of the USENIX Mach Symposium*, November 1991.

[24] Tevanian, A. Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach. Ph.D. dissertation, Carnegie Mellon University, December 1987. CMU-CS-88-106.

[25] Unix International, Stackable Files Work Group. Requirements for Stackable Files. Revision 3.6, February, 1993.

[26] Webber, N., Operating System Support for Portable Filesystem Extensions. In *Proceedings of the Winter 1993 USENIX Conference*.

[27] Welch, B. Naming, State Management, and User-Level Extensions in the Sprite Distributed File System. Ph.D. Dissertation, University of California at Berkeley, April 1990. UCB/CSD 90/567.

[28] Young, M., Tevanian, A. Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., Baron, R. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, November 1987.

[29] Young, M. Exporting a User Interface to Memory Management from a Communication--Oriented Operating System. Ph.D. dissertation, Carnegie Mellon University, November 1989. CMU-CS-89-202.

[30] Zajcew, R., Roy, P., Black, D., Peak, C., Guedes, P., Kemp, B., LoVerso, J., Leibensperger, M., Barnett, M., Rabii, F., Netterwala, D. An OSF/1 Unix for Massively Parallel Multicomputers. In *Proceedings of the Winter 1993 USENIX Conference*.