Metadata Logging in an NFS Server

Uresh Vahalia, EMC Corporation[1]
Cary G. Gray, Abilene Christian University
Dennis Ting, EMC Corporation

ABSTRACT

Over the last few years, there have been several efforts to use logging to improve performance, reliability, and recovery times of file systems. The two major techniques are metadata logging, where the log records metadata changes and is a supplement to the on-disk file system, and log-structured file systems, whose log is their only on-disk representation. When the file system is mainly or wholly accessed through the *Network File System* (NFS) protocol, it adds new considerations to the suitability of the logging technique. NFS requires that all operations be updated to stable storage before returning. As a result, file system implementations that were effective for local access may perform poorly on an NFS server. This paper analyzes the issues regarding the use of logging on an NFS server, and describes an implementation of a BSD *Fast File System* (FFS) with metadata logging that performs effectively for a dedicated NFS server.

1. Introduction

Recent years have seen improvements in CPU speeds that have not been matched by comparable improvements in disk access speeds. As a result, disk I/O has become the new bottleneck in operating system performance [Oust 90]. Traditional file systems perform poorly when run on fast machines with relatively slow disks. This has motivated the development of new file systems that seek to reduce the frequency and latency of disk access. One important technique is to use logging, which holds the promise of higher performance, greater reliability, and quick crash recovery.

Transparent access to files on other machines is a relatively new development. Consequently, most file systems are designed primarily for local access. When the same file system is used for exporting files and directories over a network, many assumptions inherent in its design are invalid, and the file system may perform poorly. In particular, the *Network File System* (NFS) protocol [Sand 85] requires the server to commit any file system modification to stable storage before returning the results of a request. Consequently, an NFS file system must perform many more disk writes, most of them synchronous, than a file system used for local access.

Recent implementations of logging in file systems require extensive changes both to the kernel algorithms and to the on-disk structures. Development costs are high, since not only the kernel code, but also utilities such as *newfs(8)* and *dump(8)* must be rewritten. To upgrade to the new file system, the system administrator must back up all partitions, boot the new kernel, reformat the disks, and finally restore all files. Their performance gains, even for local access, are not significant compared with other optimizations such as file system clustering [McVo 91]. Moreover, current logging file systems are optimized for local access, and their advantages are reduced for NFS use.

The Calaveras project [Rama 94] was an advanced development effort at Digital Equipment Corporation. Its aim was to build a dedicated, high-performance, multi-protocol file server using commodity hardware and conforming to existing software standards. Its first prototype was an NFS server running on Intel platforms. While designing the Calaveras file system, we wanted to gain the advantages of logging, namely high reliability and quick crash recovery, while retaining the on-disk layout of the BSD *Fast File System* (FFS) [McKu 84]. It was also critical to optimize the file

[1] The work in this paper was performed at Digital Equipment Corporation.

system for NFS-only access, which required addressing some of the drawbacks of existing logging implementations.

We describe here the implementation and performance of the Calaveras file system, which uses metadata logging to enhance a traditional FFS.

2. Background

The original UNIX file system [Thom 78] uses simple disk layout and algorithms, but performs very poorly. It uses small, fixed-size blocks that are allocated randomly from the disk. FFS, introduced in the 4.2BSD release, provides a major improvement. FFS uses large block sizes (typically 4K or 8K bytes), and tries to optimize disk access by intelligent placement of blocks. It divides the disk into cylinder groups, comprising a set of contiguous cylinders. Each cylinder group stores file data as well as metadata (inodes, directories, and indirect blocks), and the allocation algorithm tries to place related information in the same cylinder group. It also tries to minimize rotational delays by predicting the amount of disk rotation between consecutive read operations (the rotational delay, or *rotdelay*), and separating successive blocks of the same file by that amount.

There are several limitations to FFS performance. The *rotdelay* factor optimizes for the case when the file is accessed one block at a time. At the same time, it limits the disk performance to a fraction of raw disk access speed. For instance, if the *rotdelay* for a disk is set to one (the best case), two successive blocks of a file are separated by one unrelated block. This limits the disk bandwidth to half its raw capacity. Secondly, many file operations require several different I/O operations, some of which have to be done synchronously to preserve file system consistency. For example, a file create operation allocates and writes a new inode, and modifies the parent directory and its inode. This requires three disk accesses (more if the blocks are not already in memory), each of them potentially involving a head seek.

Further improvements have come in three major directions. One is to modify file system algorithms to reduce and optimize disk accesses, such as I/O clustering [McVo 91] and write-gathering of NFS operations [Jusz 94]. The second is to use non-volatile read-only memory (NV-RAM) [Mora 90, Hitz 94] to delay and batch writes that normally must be synchronous. The third is to use logging as either a supplement or a substitute for normal file system writes.

The logging technique is particularly attractive since its benefits are not restricted to performance. It offers increased reliability, since the log may replicate some or all of the file system data and metadata. It also allows quick recovery after a system crash, since a log playback is usually much faster than the file system checking and patching performed by *fsck(8)* [Kowa 78] in traditional systems. This is an extremely important consideration for installations that require high availability and cannot tolerate long delays due to crashes.

2.1. Logging file systems

A file system can use logging in two ways – it can be log-structured, or log-enhanced. The former approach represents the entire file system as a single, continuous log [Finl 87, Oust 89]. It relies on a large cache to handle most read requests, and tries to write the log in large chunks, sequentially on the disk. Operations are batched as far as possible to avoid small writes. Whenever a block is modified, it is simply rewritten to the tail of the log instead of updating in place, and other data structures are updated and similarly rewritten to reflect the new position. In time, many blocks in the log become invalid, either because they have been rewritten further ahead in the log, or because the corresponding files have been deleted. The file system tracks and garbage collects these blocks, freeing up space needed when the disk is full and the log must wrap around. This requires compacting the free space, rewriting scattered active blocks to the head of the log.

[Selt 93] describes an implementation of a log-structured file system (LFS) for 4.4BSD UNIX, based on similar work for Sprite in [Rose 91]. While LFS performs better than traditional FFS, its performance gains are matched, and in some ways bettered, by simpler enhancements to FFS such as the file system clustering work of [McVo 91]. LFS also involves a major code rewrite and on-disk structures that are incompatible with FFS. Moreover, the performance benefits of LFS come directly from the ability to write the log in large chunks. This is generally not possible with NFS, which requires synchronous commits. Finally, the garbage collection and compaction costs substantially reduce the performance; in some benchmarks, the performance of LFS with garbage collection was as much as 20% worse than standard FFS.

In log-enhanced file systems, the log is a supplement to, and replicates information in, the normal on-disk structures [Hagm 87]. Typically, the log only records changes to metadata objects (inodes, directories, allocation maps, etc.), perhaps in an ordinary file in the same file system. If the system is shutdown gracefully, the log can be discarded, since the file system is up to date and consistent. In the event of a crash, however, the log is used to rebuild the file system. During normal operation, each metadata write is first written synchronously to the log, and the on-disk structures are updated later during cache flushes. Hence after a crash, the on-disk structures may contain stale data, but the log has a record of all completed operations, and can be played back to recover the file system to a consistent state.

Metadata logging may improve performance as well. On one hand, each metadata update is written to disk twice – once to the log, and once to its normal location on disk (we call this write the *in-place update*). On the other hand, since the in-place updates are delayed, they are often eliminated or batched. For example, the same inode may be modified several times before it is flushed, and multiple inodes in the same disk block are written out together. The log writes are batched as well. For a single operation such as create, the changes to the directory and the two inodes can be combined in a single log entry. Multiple operations that are temporally close to each other can be similarly batched. This reduces the total number of disk writes for metadata blocks. The overall impact on performance depends on the ratio of the metadata operations (such as create, delete, and link) to data writes. If much of the activity in a system is large file writes, the performance improvement is negligible.

2.2. Considerations for NFS access

The behavior and performance of a file system are very different when it is accessed locally and when it is accessed by remote clients using a file access protocol such as NFS. NFS is a stateless protocol, and neither the server nor the clients are required to maintain state information about the other (although both usually maintain some state for performance reasons). When a server crashes and recovers, the client has no way of knowing it, and the effect to the client is similar to that of a network delay. For such a protocol to work consistently, the server is required to commit all file system modifications to stable storage before returning the results of an operation.

This condition has a great impact on file system behavior. For local file systems, the kernel delays most disk writes until it needs to flush its cache. This has many advantages. Multiple writes to the same block between cache flushes are all committed by a single disk write. The disk driver can effectively reorder the writes to minimize head seeks. Many writes can be eliminated altogether – if a user creates a temporary file, writes some data into it, and deletes it shortly thereafter, the data blocks may never be written to disk.

NFS requests, on the other hand, require frequent synchronous disk writes. Each write request that increases the size of a file causes at least two disk writes – one to write the data block, and one to write the updated inode. Additional writes are necessary if an indirect block must be created or modified. NFS access thus generates a lot more disk I/O and offers less room for traditional optimizations such as reordering of disk requests. Consequently, a file system that is suitable for local access may perform poorly if used mostly or wholly for NFS access.

NFS server performance is characterized by two metrics – latency and throughput. Latency measures the average time taken for each NFS request, while throughput is the maximum load the server can bear, measured in NFS operations per second. The two are interrelated; as the load on a server increases, so does the average latency. There is also room in the design for a tradeoff. A log-structured file system, for instance, achieves high performance by batching writes. This results in increased throughput. On the other hand, since the write must be committed to disk before replying to the NFS request, such batching can result in unacceptably high latency.

High latency causes several problems. Users see the system as slow and unresponsive. If a request takes very long to complete, the client may time out and retransmit the request. If the server cannot detect or handle these retransmissions effectively, it may perform a lot of duplicate processing. This further escalates the performance degradation, and also causes numerous correctness and consistency problems [Jusz 89].

3. Calaveras file system design

We began by porting the UNIX file system (ufs)[2] from DEC OSF/1 to the Calaveras kernel. This required some changes since the storage interface,

[2] ufs refers to the implementation of FFS under the vnode/vfs interface [Klei 86]

scheduling, and buffer cache of Calaveras [Rama 94] were different from OSF/1. The on-disk structures were identical to those of FFS. Once we had a working prototype, we decided to enhance it by adding metadata logging.

The primary motivation for logging was to provide quick crash recovery. We wanted to eliminate the need for *fsck*, which can take tens of minutes on a file server with a large number of disks. We also wanted equal or better performance – in particular, it was essential that the addition of logging not increase the latency of the server.

An important constraint was that the on-disk layout of the file system remain unchanged, so that users could migrate existing disks to the server without needing to back up and restore the file system. Finally, we wanted to restrict and isolate the changes to the file system, and avoid a large development effort or extensive code rewrite. These considerations precluded a log-structured file system approach.

An important decision was whether to use an *undo-redo* or a *redo-only* log [Moha 92]. An undo-redo log records, for each modified object, both its old and new values. The advantage of this method is that it has looser consistency requirements governing the order of log writes and in-place updates [Chut 92]. On the other hand, it doubles the size of the log and of each write to it. The recovery algorithm is also more complex, since logged transactions can be either replayed or rolled back. A redo-only log only records the new value of each modified object. This imposes stricter ordering of operations (described below), but has a simple recovery algorithm and smaller log size. We decided to adopt a redo-only log.

A related decision was to use physical block addressing, as opposed to logical block addressing or operations logging. Our log entries identify blocks by their physical disk locations, rather than by their logical names in the file system. This makes recovery simple, since the log contains the destination address of each item. For systems that use a logically addressed log, it is generally incorrect to replay the log from the beginning if the system crashes during recovery. Hence the recovery process must itself log its progress. This makes recovery slow and complex. The same problem occurs with systems that log operations rather than the new value of the data.

Our basic approach is similar to that in the Cedar file system [Hagm 87]. Instead of storing the log in the same file system, however, we use a separate partition (preferably, a separate disk) for the log. This does not have to be a very large disk, since a metadata-only log does not take up too much space. Further, as we shall demonstrate, the log disk can be relatively slow, yet not degrade overall performance. This allows a system administrator to dedicate a small, inexpensive disk for the log. There is only one log in the system; it records changes to objects in all the file systems.

FFS has five different types of metadata objects – inodes, directories, allocation bitmaps, indirect blocks, and cylinder group summaries. We log modified inodes in their entirety. For directories, we record the 512-byte chunk that contains the modified data. In the case of allocation bitmaps, we only log the changed bits. Cylinder group summaries can be quickly computed from the allocation bitmaps, and hence are not logged. Logging indirect blocks would have required a substantial change to several functions, and the benefits are small, since this is a relatively infrequent operation. We decided to defer it in the first implementation.

The logging code only affects those NFS requests that can potentially modify the file system. We call these requests (*setattr*, *write*, *link*, *symlink*, *create*, *remove*, *mkdir*, *rmdir*, and *rename*) *intrusive*. Each *intrusive request* generates a log entry, which records all metadata changes made by that request. The server must write that entry to the disk before replying to the request. The in-place updates of the metadata wait for the next *sync* operation, which happens every 30 seconds in Calaveras.

When all metadata objects described by a log entry are *sync*'ed to disk, the entry is obsolete and can be overwritten. The log is circular, wrapping around when it reaches the end. If the log is large enough, the *sync* operations will keep it clean, and no separate garbage collection is necessary. We found that a 10-Mbyte log was sufficient for a server with 12 disks, each with about 375 Mbytes of active data, running at a load of 650 NFS operations per second. We therefore mandated a minimum log size of 32 Mbytes (way more than enough for the loads our server could support), and restricted garbage collection code to merely track the log usage, and to panic if active data was overwritten.

The initial performance results were extremely poor, due to the overhead of writing one log entry for each intrusive NFS operation. We devised a solution that allowed automatic batching of log entries without increasing the latency. We also added batching to the recovery algorithm. These

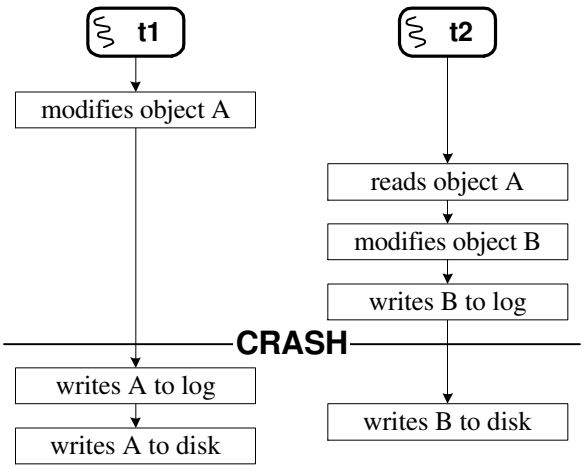

Figure 1: Race condition in a redo-only log

enhancements, described in the following sections, allowed us to meet our performance goals.

3.1. Log consistency

Crash recovery with a redo-only log is effected by replaying the log, writing all objects back to their correct locations on disk. This requires that the log copies of the objects are more current than the on-disk copies. Hence during normal operation, any change to a metadata object must be committed to the log before the in-place update of the object. The file system satisfies this condition by not releasing the buffers containing the metadata objects to the cache until the log write completes.

There is, in fact, a much stronger requirement. It is incorrect to for an intrusive request to even read any object that has not been written out to the log. Figure 1 illustrates a potential problem. Thread **t1** modifies object **A**, and is about to write it out, first to log and then to disk. Before it can do so, thread **t2** reads object **A**, and based on that, modifies object **B**. It then writes **B** to the log, and is about to write it to the disk. If the system were to crash at this instant, the log contains the new value of **B**, but the new value of **A** is neither in the log nor on disk. Since the change to **B** depends on the change to **A**, this situation is potentially inconsistent.

To take a concrete example, suppose **t1** is deleting a file from a directory, while **t2** is creating a file by the same name in the same directory. **t1** deletes the file name from block **A** of the directory. **t2** finds that the directory does not have a file by that name, and proceeds to make a directory entry in block **B** of the directory. When the system recovers from the crash, it has the old block **A** and the new block **B**, both of which have a directory entry for the same file name.

To ensure consistency, the server must lock all metadata objects until their log entries are written out. This lock only affects intrusive NFS operations. It is perfectly valid to read uncommitted data if no

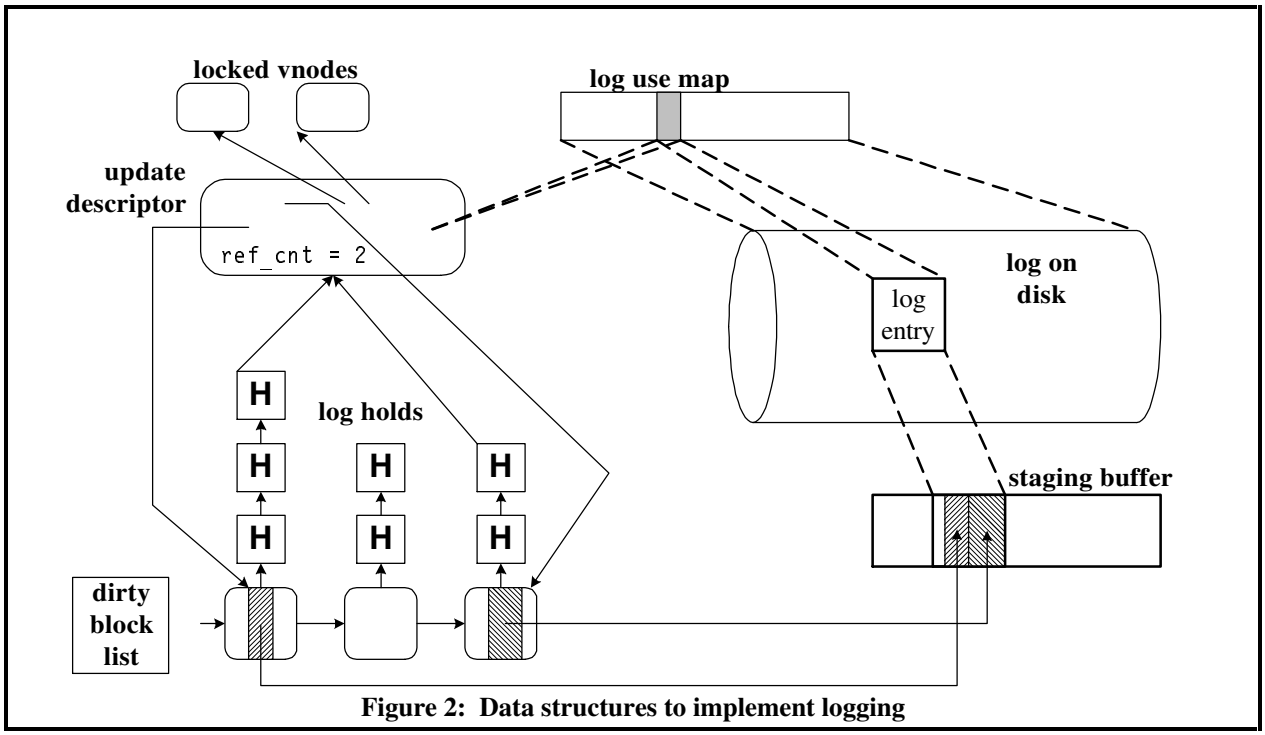

Figure 2: Data structures to implement logging

modifications are made based on that. A separate *logging lock* has been added to vnodes to implement this synchronization. Only the intrusive requests acquire this lock.

4. Implementation

Figure 2 describes the data structures used to implement logging. There are five main objects:

- an *update descriptor* tracks all operations and synchronization associated with a single NFS request.
- a *log entry* contains the actual data that is written to the log for one NFS request.
- the *log use map* tracks which parts of the log are active and which are free.
- *staging buffers* allow automatic batching of log writes.
- *log holds* prevent premature release of an update descriptor.

4.1. The update descriptor

An update descriptor baby-sits an NFS request, and holds all temporary information required to successfully complete it. Each intrusive NFS request first acquires an update descriptor, and passes it as an additional argument to many of the functions it calls. The descriptor is released not when the request completes, but when all the metadata objects that it has modified have been successfully updated in-place. At that time, the log entry on disk for this request becomes obsolete, and the corresponding bits in the log use map are cleared. This is the last step in processing an NFS request.

The fields of the update descriptor include

- a list of vnodes whose logging lock is held by this request. When the log entry is committed to disk, these locks are released.
- a reference count of the number of log holds pointing to it. The descriptor is released when this count drops to zero.
- base and size of the log entry. This determines the bits to clear in the log use map when the descriptor is released.
- a list of items to log. Each item is identified by an item type (inode, directory chunk, inode allocation bitmap, or block allocation bitmap), pointer to the cached data, an inode number, the address of the block on disk and the offset of the object in that block.[3] This information is used to create the log entry.

4.2. Log management

A log entry holds all the metadata changes for a single NFS request. It comprises a header and a table of contents, followed by the actual inodes, directory chunks, etc. The entry is padded to a 512-byte boundary. The table of contents describes the type and disk location of each object in the entry.

The header contains the number of items in the table of contents, as well as the total size of the entry. It has two other fields – *recno* and *curhead*. *recno* is a monotonically increasing record number, such that *recno* modulo the size of the log (in 512-byte sectors) equals the position of this entry in the log. *curhead* is the *recno* of the first active record in the log at the time the entry was generated. These fields are used for recovery. The tail of the log is the entry with the highest *recno* value. The *curhead* field of that entry identifies the head of the log. The recovery algorithm replays all entries between the head and the tail.

The log use map is a bitmap with one bit for each 512-byte sector of the log. When an entry is active, the bits for the corresponding sectors are checked. The bitmap allocates new entries to NFS requests, and frees them when their update descriptor is released. The map tracks the current head and tail, and returns the information along with each allocated entry. New entries must be allocated at the tail, since the log must always be contiguous. If the bits at the tail are busy (the tail wraps around and catches up with the head), the log is considered full, even if there are free regions in the middle. This results in a *panic*.

A log hold is simply a reference to an update descriptor. It contains pointers to chain the holds, and a pointer to the descriptor. In Calaveras, the file system maintains a dirty block list of modified metadata blocks; this list is periodically traversed and flushed by the *sync* daemon. The buffer headers for these blocks contain a linked list of log holds. Whenever an NFS request modifies a metadata object, it adds a log hold to the corresponding buffer, and increments the reference count of the update descriptor that the hold points to.

When *sync* successfully writes the block to disk, it traverses the list of log holds, releases each of

[3] For the allocation bitmaps, the item record contains the offset and value of the modified bits in the map.

them, and decrements the reference counts on the update descriptors. When the count reaches zero, the descriptor and its log entry are freed.

4.3. Normal operation

When an *nfsd* thread receives an intrusive request, it first allocates an update descriptor. It passes this descriptor to each vnode operation and onward to other functions that may need it. During the processing of the request, the thread acquires logging locks on the vnodes of any files or directories it accesses. Whenever it modifies a metadata object, it makes changes to the cached copy of the object. It does not release the corresponding buffer to the dirty block list, since it is not yet safe to write it to disk. It adds an entry in the update descriptor (in the list of items to log). This entry identifies the buffer and acts as a reference to it.

When all the processing for the request is complete, the thread calls the *processUpdateDescriptor()* routine, which performs the following tasks:

- Goes through the list of items to log, and computes the size of the log entry.
- Reserves disk space for the entry from the log use map.
- Reserves space in the staging buffer to write the entry.
- Traverses the list again, copying each item (inode, directory chunk, or allocation bits) to the staging buffer.
- Calls *writeLog()* to write the entry to disk.
- Puts modified blocks on the dirty block list.
- Adds a log hold on each modified block, and increments the reference count on the descriptor.
- Releases all logging locks.

Multiple requests may modify a buffer between flushes. Each removes it from the dirty block list and replaces it after the log write completes. This results in multiple holds on each block. A block could also be modified more than once by a single request, for instance, when a request modifies two inodes in the same disk block. This causes multiple holds on the same block referencing the same descriptor.

Eventually, the *sync* daemon removes the block from the list and flushes it to disk. It then releases all holds for the block, and decrements the reference counts on the corresponding descriptors. If the reference count on a descriptor reaches zero, its log entry is marked obsolete (by clearing the bits in the log use map), and the descriptor is released as well.

5. Batching log writes

Writing each log entry to disk individually is expensive, and causes overall degradation of server performance. All logging systems rely on batching of log writes to obtain decent throughput. At the same time, the batching requires that some writes be delayed till sufficient data has been collected for a large write. Since NFS requests cannot be replied to until the write completes, this causes an increase in latency, which is usually unacceptable. For this reason, many logging file systems are unsuitable for an NFS server.

We devised a solution to this problem, to get automatic batching without any increase in latency. The basic principle is that under heavy load, the log disk should always be busy. No write should be kept waiting if the disk is idle; the batching is restricted to entries that accumulate while another log write is

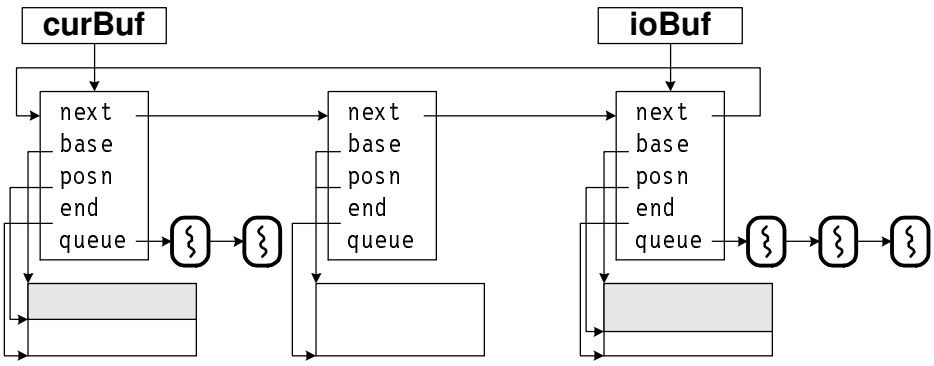

Figure 3: Batching writes through staging buffers

in progress.

The staging buffers are used to create and populate the log entry before writing it to disk, and they also provide the mechanism for automatic batching. We need a minimum of two staging buffers, so that one is filled while the other is being written to disk. Normally, we use three buffers, linked on a circular list. The buffer size is large, since it must hold all the data transferred in a single log write. We use 64-Kbyte buffers, since that was the size of a single disk track. In practice, we never write more than 16 Kbytes at a time.

Figure 3 describes the organization of staging buffers. Each buffer header contains pointers to the *next* buffer, to the *start* and *end* of the data area, and to the first free byte in the buffer (*posn*). It also contains a queue of the threads that are blocked waiting for the buffer to be written to disk. There are two global pointers – *curBuf* points to the buffer currently being populated, and *ioBuf* points to the buffer currently being written to disk.

The *processUpdateDescriptor()* routine allocates the space needed for the log entry from the buffer pointed to by *curBuf*, starting at *posn*. If there is not enough space between *posn* and *end* for this entry, the next buffer becomes *curBuf*, and the entry is allocated from that. *processUpdateDescriptor()* then copies the metadata into the entry, and calls *writeLog()* to write the entry to disk.

writeLog() checks to see if another write is in progress (*ioBuf* is non-NULL). If not, it sets *ioBuf* to this buffer, advances *curBuf* to the next buffer, and initiates a write for this buffer. This will write out all log entries in this staging buffer. When the write completes, it wakes up all threads blocked on this buffer's queue.

If another write is already in progress, *writeLog()* adds the thread to the staging buffer's queue, and blocks till this buffer is written out. In this way, each time a log write completes, all pending writes are batched into a single I/O operation. The batching does not impose any additional latency.

This technique has some interesting properties. We define the average batching efficiency as

batching efficiency

= *log entries written / num of write ops*

= *intrusive requests per second /*

log writes per second

If the disk is constantly busy, the number of log disk writes per second is bounded by the raw performance characteristics of the disk, and is usually smaller due to the processing delays between writes. The efficiency then is the average number of intrusive requests that arrive in the time taken for one log write (including the associated processing plus idle time).

Under light loads, the efficiency is close to one, since there are few instances of multiple requests arriving in the before a disk write completes. As the incoming load increases, so does the batching efficiency. On a heavily loaded system, we measured efficiencies of up to 1.6. This translates to a 40% reduction in the total number of log writes.

The remarkable property of this implementation is that the speed of the log disk does not substantially impact the performance of the system. If the disk is slower, the number of log writes per second decreases, resulting in greater batching efficiency. Hence while a slow log disk may slightly increase the latency of each operation, the overall throughput is not impacted, and may in fact improve.

6. Log recovery

Recovering the log is conceptually straightforward – the log must be played back from start to end, writing back each metadata object in it to its in-place location. The two main implementation problems are how to find the start and end of the log, and how to reduce the total I/O needed for the recovery.

The log is divided into 64-Kbyte segments, whose only property is that log entries may not cross segment boundaries. The log allocation routines enforce this by padding the last entry of a segment to the boundary. Thus the beginning of a segment always coincides with the start of a new log entry. The entry's header contains its size, so the recovery algorithm can sequentially traverse entries in a segment.

The *recno* field in the entry is a monotonically increasing record number (such that *recno % log size* equals the position of the entry in the log). Hence the tail of the log is the entry with the highest *recno* value (the log may wrap around several times, but *recno* continues to increase).

The recovery procedure first uses a binary search of the segments (looking only at the first entry of each segment) to find the segment containing the highest *recno* (tail entry). It then reads the segment into memory, and scans it linearly to locate the tail entry. The *curhead* field in that entry identifies the

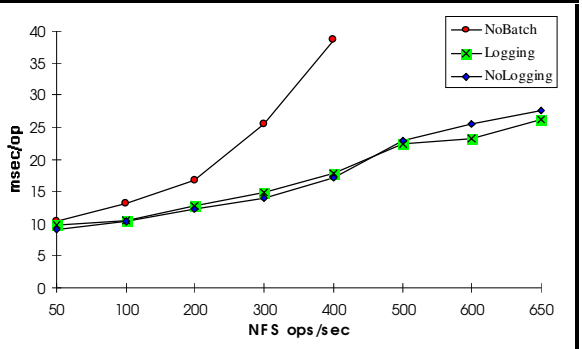

Figure 4: LADDIS benchmark results

head of the log. It then recovers the log one segment at a time, starting at the head.

To recover a segment, it reads the whole segment into memory, and scans all entries to eliminate items obsoleted by later items in the same segment. For instance, several filenames may have been written to a directory chunk; the last instance of the chunk in the segment contains all the changes (to that point). It then sorts the remaining items based on file system id and block number. Finally, it copies the items back from the sorted list to the in-place disk locations. This way, it can combine all entries in a segment that modify the same block.

This algorithm yields substantial savings. For a 32-Mbyte log (512 64-Kbyte segments), the binary search locates the head segment in 9 reads, as opposed to 256 reads for a linear search). By eliminating duplicate items, and sorting and combining the rest, we reduce the number of writes by about a factor of 8 to 10 in typical cases.

7. Performance

We made our measurements on a DECpc 560ST, a 60 MHz Pentium machine with 192 Mbytes of RAM, an EISA bus, a DEFEA FDDI controller, and two Adaptec 1740 controllers. For the logging benchmarks, we used 10 Digital RZ-26 disks for file systems, and one more for the log. For the non-logging tests, we used all 11 disks for the file systems. We ran the Spec LADDIS benchmark [Witt 93], using DEC Alpha systems running OSF/1 as clients. We added code to the server to track the peak active log size and batching efficiency.

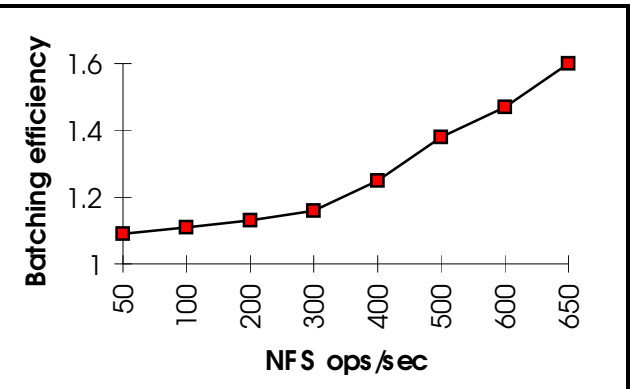

Figure 5: Batching efficiency as a function of server load

The primary motivation for logging was quick crash recovery, and to measure that, we powered off the system when running at peak load. At that time, the file system had about 3 Gbytes of data over 10 (logging case) or 11 (no-logging case) disks. In absence of logging, the file system was recovered by *fsck*. In the best case, this took about 450 seconds. Whenever *fsck* ran into a serious error, it had to be run interactively; this took much longer.

By comparison, the log recovery took between 3 and 14 seconds for the entire file system, depending on the size of the log at the time of the crash. This gain is somewhat exaggerated due to the fact that on our server, *fsck* recovers one disk at a time, while the log recovers all disks simultaneously. Even accounting for that, the log recovery would outperform *fsck* by a large factor.

Figure 4 describes the results of the LADDIS benchmark. The maximum throughput we could achieve was about 650 NFS operations per second, both with and without logging. Beyond that, the CPU became a bottleneck. At that peak load, the peak active log size was 10 Mbytes, and the batching efficiency was 1.6 (40% reduction in log writes).

The NoBatch curve describes the results of the logging implementation without batched writes. Besides uniformly high latency, the throughput peaks out at 400 NFS operations per second, since the log disk becomes a bottleneck.

The NoLogging and Logging curves are not too different. The average latency is a little higher with logging at low loads, but lower at high loads. This reflects the effect of the batching efficiency increasing with load. Figure 5 describes the variation of batching efficiency with incoming load.

A close look at the LADDIS test suite explains the similarity between the logging and no-logging results. 80% of the LADDIS requests are non-intrusive (*lookup*, *getattr*, *read*, *readlink*, and *readdir*), which are completely unaffected by logging. Another 16% are *write* and *setattr* requests, which typically modify only one metadata object (the inode), and are also relatively unaffected by logging

Test	CD	RD	CF	RF	Total
No Logging	370	151	339	121	981
Logging	123	66	264	89	542
Gain	3.01	2.28	1.28	1.36	1.81

Table 1: Small file benchmark results. The numbers are the elapsed time in seconds

(an in-place write is replaced by a log write). Only 4% of the load consists of requests that affect multiple metadata objects (*create*, *remove*, *rename*, *link*, *mkdir*, and *rmdir*); these are the ones that benefit most from logging.

To concentrate on these, we ran a "small file benchmark", in which we ran four types of tests on a server with eight disks. The tests were:

- **CD:** create 16000 directories; each disk has 20 directories with 100 sub-directories each.
- **RD:** remove the above tree (thus, 16,000 *rmdirs*).
- **CF:** copy a 1-Kbyte file 16000 times (same distribution as for the CD test).
- **RF:** remove the above files.

Table 1 shows the results of the benchmark. Logging is faster by a factor of 1.28 to 3.01 in the four tests. The actual improvement of server performance is even better, since the tests take into account client-side processing as well as other requests such as lookup and getattr that are necessary for these operations.

Table 2 shows measurements made on the server of the average elapsed time (in milliseconds) for the four metadata requests tested by the small file benchmark. The improvements are substantial; *mkdir*, in particular, is improved by more than a factor of six.

It has been difficult to compare these results with other NFS servers, for lack of clear criteria to base comparisons on. While many researchers [Hagm 87, Selt 93] have published performance measurements on log-structured or log-enhanced file systems, they have concentrated on benchmarks that deal with local access. Conversely, many others [Hitz 94, Jusz 94] have published measurements of NFS performance using LADDIS and other tests. Some of these do not use logging at all, and the others do not have any measurements that explicitly isolate the effect of logging.

[Hitz 94] provides the closest point of reference. Its FAServer runs on the Intel 486 platform, and uses a log-structured file system in conjunction with NV-RAM. The configuration, therefore, is not too different from ours. Their published measurements, for an eight-system cluster, extrapolate to about 400 NFS ops/sec per server, at an average latency of under 15 miliseconds. We compare that to our benchmark on the i486, which peaks at 597 NFS ops/sec. with a latency of 29 miliseconds. The low latency of the FAServer is almost entirely due to their use of NV-RAM, which allows them to complete operations without waiting for disk I/O.

8. Conclusion

The logging enhancements met all their goals. There was a tremendous improvement in crash recovery time. Under a heavy, mixed, load, logging yielded a small gain in latency and equal throughput. For small file and directory operations, the gains were substantial (a factor of 1.50 to 6.30 improvement in server latency).

All this was achieved at a low cost. The entire implementation and testing took about twelve man-weeks. We wanted to keep code changes to a minimum, to make it easy to integrate enhancements to the baseline code. We were able to do that; the logging enhancements modify the ufs code in five regards:

1. The update descriptor is passed along as an additional argument to a number of vnode operations and ufs functions.
2. Intrusive operations acquire logging locks on

Operation	*mkdir*	*rmdir*	*create*	*remove*
No Logging	106.4	50.1	30.4	36.5
Logging	16.9	17.3	17.1	24.4
Gain	6.30	2.89	1.78	1.50

Table 2: Average turnaround times for metadata operations on server, in msec.

metadata objects they access.

3. The functions that modify the metadata objects do not release the dirty buffers; instead, they add entries to the update descriptor's item list.

4. Intrusive requests allocate an update descriptor at the beginning and call *processUpdateDescriptor()* at the end.

5. The *sync()* routine releases log holds and update descriptors.

Besides these changes, we had to write the logging module. This consisted of the functions and data structures that implement logging and recovery. The module has limited interaction with the baseline code, and is accessed through a narrow, well-defined, interface.

We left the FFS on-disk structures unchanged. This allows users to migrate existing disks to our server without backing up and restoring all files. The log needs to be kept on a separate disk, but that can be a small, inexpensive disk. The automatic batching technique gives greater savings under heavy load, and ensures that the performance of the log disk does not impact overall system throughput.

Some work remains to be done. We need to extend logging to include indirect blocks. We need to integrate performance improvements such as NFS write gathering [Jusz 94] and file system clustering [McVo 91] with our logging framework. We must add a checksum to log entries to guard against a crash leaving behind a partially written log entry. We also need to provide a background *fsck*-like function to recover from hard disk errors that corrupt one or more sectors. Finally, we must evaluate the suitability of the implementation for an NFS v3.0 server [Pawl 94]. The new version of the protocol, with its inherent ability to batch writes, might change the assumptions made in our design.

9. Acknowledgments

Several people have contributed to this effort. In particular, we would like to thank the rest of the Calaveras team – Percy Tzelnic, Steve Glaser, Lev Vaitzblit, Wayne Duso, and K. K. Ramakrishnan for their help and discussions throughout this project, and Chet Juszczak for his insightful discussions during the early presentations of this work.

References

[Chut 92] Chutani, S., Anderson, O.T., Kazar, M.L., Mason, W.A., and Sidebotham, R.N., "The Episode File System", Proceedings of the Winter 1992 Usenix Technical Conference, Jan. 1992, pp. 43-59.

[Finl 87] Finlayson, R.S., and Cheriton, D.R., "Log Files: An Extended File Service Exploiting Write-Once Storage", Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, Nov. 1987, pp. 139-148.

[Hagm 87] Hagman, R.B., "Reimplementing the Cedar File System Using Logging and Group Commit", Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, Nov. 1987, pp. 155-162.

[Hitz 94] Hitz, D., Lau, J., and Malcolm, M., "File System Design for an NFS File Server Appliance", Proceedings of the Winter 1994 Usenix Technical Conference, Jan. 1994, pp. 235-245.

[Jusz 89] Juszczak, C., "Improving the Performance and Correctness of an NFS Server", Proceedings of the Winter 1989 Usenix Technical Conference, Jan. 1989, pp. 53-63.

[Jusz 94] Juszczak, C., "Improving the Write Performance of an NFS Server", Proceedings of the Winter 1994 Usenix Technical Conference, Jan. 1994, pp. 247-259.

[Kaza 90] Kazar, M., Leverett, B., Anderson, O., Vasilis, A., Bottos, B., Chutani, S., Everhart, C., Mason, W.A., Tu, S., and Zayas, E., "Decorum File System Architectural Overview", Proceedings of the Summer 1990 Usenix Technical Conference, Jun 1990, pp. 151-164.

[Klei 86] Kleiman, S.R., "Vnodes: an Architecture for Multiple File System Types in Sun UNIX", Proceedings of the Summer 1986 Usenix Technical Conference, Jun 1986, pp.

[Kow 78] Kowalski, T., "*FSCK*: The UNIX System Check Program", Bell Laboratory, Murray Hill, NJ, Mar. 1978.

[McKu 84] McKusick, M.K., Joy, W.N., Leffler, S.J., and Fabry, R.S., "A Fast File System for UNIX", Transactions on Computer Systems, Volume 2, No. 3, Aug. 1984, pp. 181-197.

[McVo 91] McVoy, L.W., and Kleiman, S.R., "Extent-like Performance from a UNIX File System", Proceedings of the Winter 1991 Usenix Technical Conference, Jan. 1991, pp. 1-11.

[Moha 92] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwartz, P., "ARIES: A Transaction Recovery Method Supporting Fine-Grain Locking and Partial Rollbacks Using Write-Ahead Logging", ACM Transactions on Database Systems, Vol. 17, No. 1, Mar. 1992, pp. 96-162.

[Mora 90] Moran, J., Sandberg, R., Coleman, D., Kepecs, J., and Lyon, B., "Breaking Through the NFS Performance Barrier", Proceedings of the Spring 1990 European UNIX Users Group Conference, Apr 1990, pp. 199-206.

[Oust 89] Ousterhout, J.K., and Douglis, F., "Beating the I/O Bottleneck: A Case for Log-Structured File Systems", Operating Systems Review, 23(1), Jan. 1989, pp. 11-27.

[Oust 90] Ousterhout, J.K., "Why Aren't Operating Systems Getting Faster as Fast as Hardware?", Proceedings of the Summer 1990 Usenix Technical Conference, Jun 1990, pp.

[Pawl 94] Pawlowski, B., Juszczak, C., Staubach, P., Smith, C., Lebel, D., and Hitz, D., NFS Version 3 Design and Implementation", Proceedings of the Summer 1994 Usenix Technical Conference, Jun 1994, pp. 137-151.

[Rama 94] Ramakrishnan, K.K., Vaitzblit, L., Gray, C., Vahalia, U., Ting, D., Tzelnic, P., Glaser, S., and

Duso, W., "Operating System Support for a Video-On-Demand File Service", ACM Multimedia Journal, *to appear*.

[Rose 91] Rosenblum, M., and Ousterhout, J.K., "The Design and Implementation of a Log-Structured File System", Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles, Oct. 1991, pp. 1-15.

[Sand 85] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, R., "Design and Implementation of the Sun Network File System", Proceedings of the Summer 1985 Usenix Technical Conference, Jun 1985, pp. 119-130.

[Selt 93] Seltzer, M., Bostic, K., McKusick, M.K., and Staelin, C., "An Implementation of a Log-Structured File System for UNIX", Proceedings of the Winter 1993 Usenix Technical Conference, Jan. 1993, pp. 307-326.

[Thom 78] Thompson, K., "UNIX Implementation", Bell System Technical Journal, 57(6), Jul.-Aug. 1978.

[Witt 93] Wittle, M., and Keith, B., "LADDIS: The Next Generation in NFS File Server Benchmarking", Proceedings of the Summer 1993 Usenix Technical Conference, Jun 1993, pp. 111-128.

Uresh Vahalia received an MS in Computer Science from Syracuse University in 1985. He has since been working on operating systems and network protocols. He is currently at EMC Corporation, wehre he is developing file and continuous media servers. Uresh can be reached by email at `vahalia@emc.com`.

Dennis Ting received a BS in Computer Science from the University of Utah in 1971. He is currently a principal engineer at EMC Corporation. He has been involved in the development of real-time operating systems, network protocols, and high-performance network file servers.