The following paper was originally published in the
Proceedings of the USENIX Annual Technical Conference (NO 98)
New Orleans, Louisiana, June 1998

Fast Consistency Checking for the Solaris File System

J. Kent Peacock, Ashvin Kamaraju, and Sanjay Agrawal
Sun Microsystems Computer Company

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

Fast Consistency Checking for the Solaris File System

J. Kent Peacock, Ashvin Kamaraju, Sanjay Agrawal
{kent,akk,sanj}@eng.sun.com
Sun Microsystems Computer Company

ABSTRACT

Our Netra NFS group at Sun set out to solve the challenging problem of providing remote Network File System (NFS) service with high performance and availability. An NFS server must guarantee the permanence of changes to the file system before acknowledging an NFS request. Thus, the server's underlying local file system must perform update operations synchronously to stable storage with potentially high latency. Our solution to this problem involves using the Solaris Unix File System (UFS), derived from the Berkeley Fast File System (FFS), in conjunction with nonvolatile RAM (NVRAM) as fast stable storage. We evaluated the system using the LADDIS benchmark and as a result, developed a cacheing technique for block-mapping information that gave us a 23% increase in measured server throughput in our standard RAID-5 server configuration. With recent increases in disk capacity and RAID technology, file-system sizes have reached a point not imagined by the FFS designers, requiring an approach to checking file-system consistency that does not grow proportionately with file-system size. We examined several log-based solutions to providing fast crash recovery, but none could use the NVRAM effectively and meet our performance requirements. As an alternative, we developed an approach that uses UFS but maintains file-system working-set information, so that the consistency checker needs to examine only the active portions of a file system. This approach met our performance goals and also reduced file-system consistency-checking times to between 3% and 25% of those in the original UFS implementation.

1 Introduction

The goal of the Netra NFS project at Sun was to produce a dedicated NFS file server with performance and availability that would satisfy the increasing requirements of client−server networks.

Since the work was to be performed in the context of the Solaris Operating System, the natural file system to use was the native UFS file system. UFS was derived from the Berkeley UNIX Fast File System developed during the 1980s [McKusick84]. At that time, disks and file systems were small and slow relative to those found in today's computing systems, and conglomerations of drives were not in general use. Thus, UFS was not designed to handle gracefully file systems that can be hundreds of gigabytes.

One of the features of UFS is that a large volume of data is maintained in a set of caches in main memory, and is flushed back to disk in the background. This feature improves file-system performance significantly, though at the expense of possible loss of recently written data when the system crashes. In addition, it is necessary to perform a file-system consistency-checking operation on reboot after a crash in order to ensure reliable operation after the next mount of the file system. Without this consistency check, files and data could be lost or corrupted. The program that does this task, *fsck*(1M) [McKusick94], is required to find and correct inconsistencies; it does so by reading all the control information contained in the file system and correcting that information when it discovers errors. The amount of information is proportional to the size of the file system. Thus, as file systems have grown huge, the time to perform the consistency check has become unacceptably large.

A widely used solution to this problem has been the application of traditional database-logging techniques to allow fast recovery. The log contains information about the most recent transactions on the file-system control data, and so the log replay can restore the file system to a consistent state after a crash. This approach is much faster than *fsck* because only those files referenced in the log need to be processed.

Logging solutions are less desirable in the context of an NFS file server, however. To reduce disk writes, typical implementations batch many transactions into a single log write. This batching is especially effective when there are many operations that are asynchronous, since the originator of an asynchronous operation does not require that the transaction be completed before it continues. As noted, this behavior is typical for a local

UFS file system. However, when a UFS file system is accessed remotely through NFS, the NFS protocol definition requires that the transactions for most operations be complete before acknowledgement is returned to the client. With few or no asynchronous operations, the expected batching does not occur, and the log can become a significant bottleneck because each transaction generates a separate write to the log.

As an alternative approach, we considered creating a faster file-system checking program. At any given time, there is a limited set of files and auxiliary control information active on a file system. If it were possible to keep track of the active portions of the file system, the scope of the file-system check could be reduced dramatically to only the ‘‘working set’’ of the file system. The ‘‘fast fsck’’ provided with Netra NFS records the working set by adding a small amount of state information to the file system and performing ‘‘lightweight logging’’ of certain transactions in the file system, such as directory operations. The file system accomplishes this lightweight logging efficiently by adding transaction state information to a reserved portion of disk blocks containing file inodes. The mini-log can be thought of as a log that is distributed throughout the file system. The net result is a file-system checking program that takes a fraction of the time used by the original UFS *fsck* program.

2 Previous Work

The current Solaris UFS file system follows its predecessor, the Berkeley Fast File System (FFS) [McKusick84], in its approach to providing file-system robustness. It uses synchronous writes of metadata in a sequence such that the *fsck*(1M) program can restore the file system to a consistent state after a system crash. Without ancillary information, *fsck* must assume that all file-system metadata may be inconsistent, and must therefore scan all of the metadata to find any inconsistencies present. This scan takes unacceptable time in enormous file systems.

The most commonly used solutions to the problem of providing fast recovery are borrowed from database design — namely transaction logging (e.g., [Hagmann87, Chutani92]) or shadow paging (e.g., [Seltzer93, Hitz94]) of file-system metadata and, in some cases, of actual file data. The Veritas File System (VxFS), although not described in the literature, is prevalent in commercial UNIX systems; it uses transaction logging for file-system data and metadata. An interesting approach to avoiding the necessity of synchronous writes, called *soft updates*, is presented in [Ganger94]. This approach shows performance gains of a factor of between 2 and 15 on metadata-intensive benchmarks, and reduces the consistency-checking time from 5 to 7 minutes to 3 to 5 seconds, but only in a local file system environment where file I/O is asynchronous. In an NFS context [Sandberg85, NFS94], almost all remote file operations are synchronous with respect to the client. Thus, it is doubtful that soft updates would provide as great a performance benefit in an NFS environment, although the fast-reboot benefit would still be obtained.

Vahalia and colleagues [Vahalia95] dealt with the same problem that we faced. In fact, the first two sections of their paper give an excellent summary of what is required to provide good NFS server performance. Their approach uses metadata logging at the UFS level as a *redo-only* log that records only the new value of each modified object and can roll completed transactions only in the forward direction. They solve the problem of NFS requests requiring synchronous commitment to stable storage by batching writes to the log, which is set up as a separate disk device. The goal is to keep the log device busy at all times when under heavy load, so that log requests are batched while the disk is busy doing one write and these batched requests are sent immediately to the disk when the write finishes.

Hitz and associates [Hitz94] provide another point of reference that is perhaps closer to our approach. They included RAID-4 parity striping and NVRAM in their design. Their implementation uses shadow paging and makes use of a snapshot facility in combination with NVRAM logging of requests at the NFS level. Recovery consists of backing up to the most recent consistency snapshot and replaying the NFS log from that point. The only apparent drawback to this approach is that creating a snapshot involves modifying the entire block-map file and performing the housekeeping necessary to flush dirty data to the disk; the authors estimate that this task takes on the order of 1 second. Presumably, the recovery time when a system crashes at saturation could be at least as long as the time since the previous snapshot. We would not expect the NFS log to be replayed and processed faster after the reboot than when the original requests were seen, given that the server was in a saturated mode.

3 Netra NFS Design Features

The Netra NFS project started with the primary aim of producing an NFS server with very high performance, reliability and availability with the following design goals:

- High throughput with low latency, as measured by LADDIS [Whittle93]
- Tolerance of single disk failures without loss of file-system data
- Fast reboot after a power outage or system failure
- Fast disk and file-system initialization
- Minimal on-disk format changes
- Simplified browser-based administration

We give details of the features we implemented to satisfy these goals in the following sections: in section 3.1, we describe our NVRAM solution to provide low latency; in section 3.2, our use of Solstice Disk Suite to provide RAID-5 fault-tolerance; in section 3.3, our development of a cacheing technique to improve write performance; and in section 3.4, a modification to the UFS allocator to force locality of allocation for performance and to aid in fast recovery.

3.1 NVRAM Acceleration

It was clear from the beginning of the project that we required some form of NVRAM-based stable storage to satisfy the goals of high throughput and low response time [Moran90, Hitz94]. Originally, the entire memory of the machine was made nonvolatile by using an integrated uninterruptible power supply (UPS) that had sufficient backup power to shut down the system cleanly. Because we wanted to guarantee truly stable memory, we decided to use the Prestoserve [Presto93] accelerator to cache data. This approach provided us with a clean, debugged solution for fast stable storage. In recent versions, the UPS was been replaced by NVRAM boards.

3.2 Solstice Disk Suite

To satisfy the goal of tolerance for single disk failures, the Netra NFS server also includes RAID-5 and mirrored solutions based on the Solstice Disk Suite (SDS) product. In SDS, the update of a single arbitrary disk block within an industrial-strength RAID-5 device incurs 6 I/O operations: (1) read old data, (2) read old parity, (3) write new data to prewrite log, (4) write new parity to prewrite log, (5) write new data, and (6) write new parity. The prewrite log operations are necessary to preserve the atomicity of the new data and of the parity write operations, and to avoid the loss of data due to a system crash involving a disk failure. In the standard SDS implementation, the prewrite area is a reserved portion of the RAID-5 aggregate. To reduce the number of I/O operations needed, we decided to configure a second Prestoserve cache in addition to the one layered above the RAID metadevice, called Presto Upper. The second cache, Presto Lower, is configured below the RAID driver and completely covers the prewrite area of the RAID device. This cache eliminates two disk operations per random data-block write.

3.3 Bmap Cache

Our performance analysis of server systems under SPECsfs1.1 (SPECnfs_A93) LADDIS load revealed that write operations accounted for 35 to 50 percent of the time spent waiting for completion of NFS requests. We made traces while running LADDIS that showed that an average NFS write operation took 2.58 disk operations when NVRAM was not configured. We found that when a file grows beyond a certain size, the UFS file system has to write at least two disjoint pieces of metadata information: the inode and an indirect mapping block. Thus, when the write of the data is included, some NFS writes require three disk operations. We decided to look for a way to reduce the number of writes required. (The more recent SPECsfs2.0 version of LADDIS does not have this emphasis on writes to relatively large files; hence, improvements in this area are less critical now. However, in the context of SPECsfs1.1, our technique yields significant performance gains.)

Our solution, *bmap cache*, combines the disjoint metadata into one disk block, saving one I/O operation per write. The solution involved extending the file inode to cache that part of the indirect block that pointed to blocks at the end of the file. The inode and block-mapping information could then be updated atomically in one disk operation most of the time, rather than in two [Peacock96]. The simplest implementation would have been just to add the *bmap cache* fields to the inode itself, and to increase the size of the inode appropriately. However, since some utilities understand these structures and the layout of the file system, we decide not to change the size, layout, or location of inodes on disk. Instead, we stored the *bmap cache* in an inode extension that was separate from the main inode structure. The inode extension structure fits in a regular inode slot and looks like an unallocated inode. In the current implementation, file inodes are allocated from the even inodes, and an inode extension is allocated from the corresponding odd inode, in effect doubling the size of the inode.

With this allocation strategy, a *bmap cache* is stored on disk in the disk block that contains its inode. When a block is written at the end of a file to extend the file, the pointer to the new block is placed into the cache, rather than into the actual indirect block. Updating the metadata then involves writing the inode and its

extension to the disk in the same block; that requires only one I/O operation. When a block is allocated beyond the end of the cache, the cache is flushed to its actual indirect block, resulting in additional I/O, and the cache is set up to contain the mapping for the new block. In addition, although the indirect block is allocated when there are blocks that would be mapped by it, it is not initialized until the *bmap cache* is flushed.

3.4 Hot-Spot Allocator

In UFS, the file-system partition is divided into *cylinder groups*. Each cylinder group has a controlling structure residing in a single block that contains the resource bitmaps and is followed by an array of inodes. The size of each cylinder group is bounded by the size of the bitmaps that can be contained in a single block.

Part of the original purpose of cylinder groups was to allow locality of allocation for directory subtrees and to provide a mechanism to spread allocation across the disk. Each cylinder group has two allocation rotors that mark starting points to search circularly through the free-list bitmaps to allocate the next inode or block within the cylinder group. In addition, there is a single cylinder-group rotor that is used as the starting point to search for a lightly used cylinder group from which to allocate. Even with these rotors, it is possible for allocations to be distributed across the entire file system during any given short time period.

We have modified the rotor behavior to produce a *hot-spot allocator*, so-called because, at any given time, there is a single hot-spot cylinder group from which all allocations are done. Each individual allocation rotor starts at the beginning of its cylinder group when it is entered as the hot spot; when an allocation cannot be satisfied without wrapping either of the rotors back to the beginning of the cylinder group, the cylinder-group rotor is stepped to the next cylinder group, which becomes the new hot spot. We thus force temporal locality of reference in terms of inode and block allocation.

There are several good reasons to have these rotors move strictly sequentially through the file-system cylinder groups. Much recent work [Rosenblum92, Seltzer93, Seltzer95, Hitz94] has focused on high locality for file-system writes to relieve the perceived write bottleneck [Ousterhout89]. By forcing locality, the hot-spot allocator increases the chance for full stripe writes in RAID-5 disk arrays. Another advantage is that this locality of allocation makes it easier to keep track of the working set of the file system. We describe the relationship between the hot-spot allocator and consistency checking in section 5.2.

The hot-spot allocator also has a policy of slow reuse, since its allocation pointer increases relentlessly. This policy avoids certain race conditions that can cause loss of data when blocks are reused before pointers to them have been cleared. Although UFS is designed to avoid this situation, there have been observed instances (now fixed) where such races have occurred.

4 Reasons Not to Use Logging

We had already decided that NVRAM would be a key component of the system when we turned to solving the problem of fast recovery. We considered several log-based solutions before settling on fast consistency checking and found that existing log-based approaches did not give us sufficient performance, even with NVRAM enabled.

The first solution that we tried was the Solstice Disk Suite transaction logging for UFS. The LADDIS performance was so bad that it was difficult even to generate a valid LADDIS run, except at loads of less than 100 NFS ops. That is when we first observed how the synchronous nature of NFS requests fits poorly with a log-based approach. Hoping that perhaps the Veritas File System would perform faster when aided with NVRAM, we replaced UFS with it and analyzed its performance. Although it was not as bad as SDS logging, it still fell short of our performance goals. We considered implementing an NVRAM-based log for NFS requests, similar to [Hitz94], but decided that such a solution would not by itself solve the fast-recovery problem. We would still have to restore the underlying UFS file system to a consistent state after a reboot.

The key insight here is that *given enough CPU power, providing NFS service is a fundamentally disk-bound operation at saturation.* Each NFS request requires a certain number of I/O operations, so any technique that reduces or eliminates disk operations — such as the use of NVRAM and the *bmap cache* — will result in higher NFS throughput. Log-based solutions simply require more disk I/O, because every datum is written twice: once in the log and once in its actual place (except in log-based file systems, where the log *is* the actual place, as in the [Hitz94] approach). Adding NVRAM to a disk-based logging solution helps, but only by reducing effective disk response times. Such logs are typically written in a circular fashion — the worst case access pattern for a write cache. We have observed in our Veritas testing with NVRAM that dirty-write hit rates in the cache are only about 7%, whereas our chosen approach exhibits write hit rates between 60% and 70%.

Logging solutions are typically designed with disks, rather than NVRAM, as the target logging device. As such, there is an implicit requirement to batch logging information into a separate disk area so that the write cost to the disk can be amortized over a number of requests, and so that atomicity of transactions can be provided. When NVRAM is considered as an integral part of the logging design, the use of batching becomes unnecessary, because access time is independent of location. Thus, it seemed that an elegant approach to logging is simply to think of the Presto cache on top of the disk metadevice as the log for writes to the file system.

With no separate log area, it becomes helpful to include a small amount of additional information in certain file-system data structures. This extra state information allows us to undo partially written metadata transactions without having to do a full *fsck*. Since NFS operations are synchronous, the only partial transactions that should be found in the file system during recovery are those that were in progress. Correct operation of the server is ensured because the NFS client should retry the undone operation, since it has not received a completion acknowledgment. An additional benefit is that it also works in a degraded mode when the NVRAM is disabled due to a fault or to a low-battery indication. Data are simply written through to the actual disks with enough information to do fast recovery, since we do not insist on atomicity in the updating of the disjoint metadata involved in a transaction.

5 Fast Consistency Checking

The normal *fsck* program restores consistency of a UFS file system after an unclean shutdown by completely examining and fixing the following structures:

- Free block and inode bitmaps
- Directory consistency and inode link counts
- Allocation summary information

The resource-use bitmaps are reconstructed from the state of files at the time of reboot. A disk block is marked as allocated if it is referenced by an inode, which is in turn also referenced. Inodes that are referenced by directory entries or by shadow inode pointers are also marked as allocated, and their link counts are set to the number of references. (Shadow inode pointers are inode pointers within an inode that point to another inode; they are currently used to implement access control lists (ACLs)). All unreferenced blocks and inodes can then be marked free in the respective bitmaps. Inodes must also be reachable from the root directory of the file system to be marked as allocated. The summary counts of allocated and free resources are generated from the reconstructed resource bitmaps.

There are four changes to UFS that we implemented to facilitate fast recovery:

- Busy block and inode Bitmaps
- Inode linktags
- Cylinder-group flushing
- Last-inode counters

Figure 1 shows the layout of a single cylinder group showing the new elements we added and their locations, highlighted in italics.

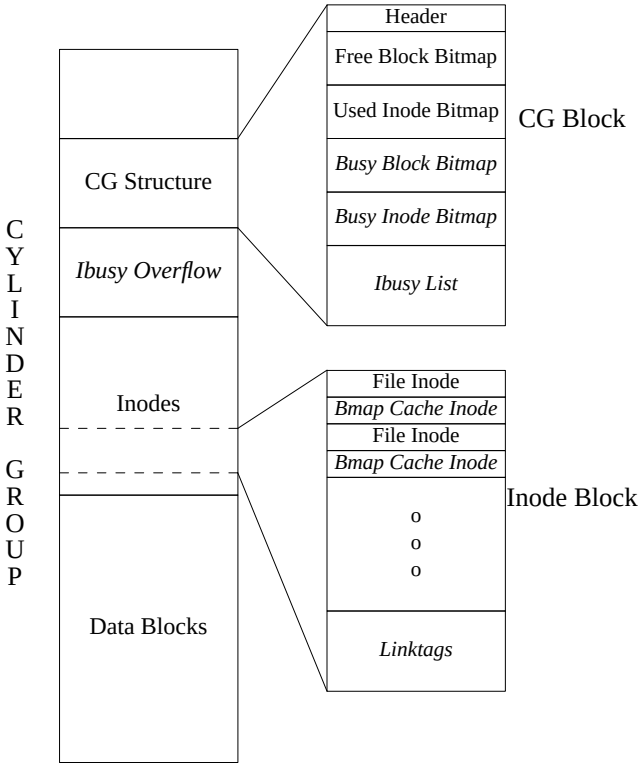

Figure 1. Cylinder Group Layout.

The busy block and inode bitmaps aid in recovering the free-block bitmaps, the inode linktags are used to do incremental repair of inode link counts, and cylinder-group flushing is used to keep low the quantity of dirty file-system metadata. The last-inode counter in each cylinder group is used to limit the number of inode slots that need to be initialized when the file system is created and looked at during a full *fsck*. In sections 5.1 to 5.4, we describe each of these components.

5.1 Busy Bitmaps

The first basic concept behind recovering the block-allocation bitmap without scanning the entire file system is to maintain a list of all blocks that are busy — that is, possibly in transition between the free and

allocated state — with reference to inodes that may or may not have been committed to stable storage. The second requirement is thus that a list be maintained of all inodes that are busy. The condition that must hold between these lists is that any block in the busy-block list either belongs to a file in the busy-inode list or is free. This condition allows the recovery algorithm to look at only those blocks owned by the busy inodes and to consider only those blocks that are in the busy-block list.

The busy-block list is implemented as a bitmap that is parallel to the free-block bitmap in each cylinder group. In a similar fashion, there is a busy inode bitmap parallel to the allocated-inode bitmap in each cylinder group. If each file could be contained in a single cylinder group, then the consistency of each cylinder group could be done independently with only these bitmaps. However, each cylinder group is limited to about 16 MBytes, so large files will span multiple cylinder groups. To handle this situation, and to enable each cylinder group to be checked individually, we gave each cylinder group an additional *ibusy* list containing inodes not in the cylinder group that have allocated or freed currently busy blocks in the cylinder group. The initial segment of the *ibusy* list is inside the cylinder-group block itself, with the adjacent block containing any overflow from that list. This is shown in Figure 1.

For this busy-block and inode approach to be workable, it is necessary to mark blocks and inodes busy before any actual state changes occur, and the marking must be done synchronously. Here, the hot-spot allocation strategy is useful. When a cylinder group is entered as the new hot spot, every free block in the cylinder group is marked busy, so this operation must be performed only once. Inodes are marked busy when they have blocks added or removed or during operations that modify their link counts. An inode number is added to the *ibusy* list of a cylinder group only when the first block from that cylinder group is allocated to it, so only one synchronous write of the cylinder-group information is required for all the allocations that the inode performs in the cylinder group.

5.2 Cylinder-Group Flushing

UNIX normally does periodic full flushing of the file system's modified metadata to get the file system into a ''clean'' state. The salient feature of the clean state is that the file-system metadata are consistent, so no *fsck* is required before a clean file system is mounted. One of our goals was to perform incremental cache flushing of the working set — particularly of the inode cache — because it is very difficult to get a file system into the clean state while under heavy load, and the clean state cannot be maintained for very long. Indeed, we found that LADDIS benchmark runs performed with the usual periodic cache flushing disabled performed significantly better.

Even though the clean state is infrequent and fleeting, we should take advantage of it. Reaching it essentially means that all metadata in the file system are consistent, and therefore all the busy bitmaps throughout the file system can be cleared. Rather than clear them by rewriting all the cylinder groups, we maintain in the file-system superblock a generation counter that is incremented whenever the clean state is reached. There is a corresponding counter in each cylinder group that is set to the value prevailing in the superblock whenever an operation affecting the busy bitmaps is performed. When the cylinder group and superblock counters are found not to match, the busy-bitmap information is known to be stale, and so is reset to a cleared state before the operation is performed. Similarly, on recovery with *fsck*, any cylinder group whose counter does not match the superblock's is considered to have stale busy information and thus does not need to be checked.

The recovery time for the busy bitmaps is proportional to the number of busy inodes, so a mechanism that tries to flush inodes and mark cylinder groups as clean is desirable. The busy bitmaps in combination with the busy-inode list on each cylinder group provide the ability to clean an individual cylinder group. We do so simply by flushing all the inodes in both the busy-inode bitmap and the *ibusy* list individually. The cylinder group is put into a *cleaning* state at the start, and, if an operation affecting the busy lists has not occurred after flushing of all the inodes, the cylinder group can be marked clean, which we do by setting its generation counter to be one less than the counter in the superblock.

The cylinder group flusher works like a two-handed paging daemon [Leffler89]. That is, the cylinder groups directly in front of the hot-spot cylinder group are flushed in a circular fashion. This approach provides the LRU properties that we would expect in such a clock-based pager. The flushing is conducted from a separate kernel thread that runs periodically and is set up to consume a fixed percentage of real time each time that it runs. With this approach, we can set the amount of time the flusher takes to a reasonably low value (10% of the flushing interval), and thus affect server performance only slightly.

5.3 Inode Linktags

A nonshadow inode's link count is simply the count of the number of directory entries that refer to the inode. It would be wonderful if the busy inode lists were sufficient to manage and recover inode link counts. However, they are not, because the link count on an inode before an operation is performed does not contain sufficient information to allow us to find the references to the inode without scanning the entire file system directory structure. To solve this problem, we incorporate some additional logical state, which we call a *linktag*, into the inode to enable undoing of a link-count increment or decrement when the corresponding directory update was not written to disk before a reboot. The *linktag* structure has the following form:

```
struct linktag {
      ino_t   dl_target;
      ino_t   dl_direct;
      off_t   dl_off;
      int     dl_cnt;
};
```

where the *dl_target* is the inode number of the inode whose link count has changed, *dl_direct* is the inode number of the directory that has had an entry added or removed pointing to *dl_target*, *dl_off* is the offset within the directory to a 512-byte directory block where the directory operation should occur, and *dl_cnt* is the expected number of entries pointing to *dl_target* within that 512-byte block (the 512-byte block is chosen because that is the size of the directory unit that UFS manipulates atomically). The count is necessary because a given directory block could contain multiple entries pointing to a given inode.

We use the *dl_cnt* and *dl_off* fields to make the creation and use of a *linktag* reasonably efficient. A search confined to a single directory block is much faster than a scan of a large directory for references to a given file. Setting *dl_cnt* to the expected count after the directory operation makes the recovery action simple. When *fast fsck* processes a *linktag* entry, it finds the actual count of references to *dl_target* at *dl_off* in *dl_direct*. *Fast fsck* then adjusts the link count of *dl_target* by adding the difference between the actual count and *dl_cnt*.

Since the *linktag* structure must be written to the file system synchronously with the updated link count in *dl_target*, we made use of the *bmap-cache* mechanism of reserving inode slots within the same inode block. In particular, the final one-eighth of each inode block is reserved to contain *linktag* entries. The *linktags* are kept as a list bound to the in-core *dl_target* inode and are placed into the reserved inode slots any time the *dl_target* inode is flushed to stable storage. The normal sequencing of operations requires that link-count increments are written before the creation of the directory entry and that decrements are written after the deletion of a directory entry. With the addition of the *linktag* state, the inode and its *linktag* must be written before the directory entry is changed. We thus require an additional write before a directory-entry deletion that was not done before, although this write is typically absorbed by the Presto Upper cache. The directory operations themselves are still done synchronously, and the *linktag* can be removed from the target inode after the directory block has been written. It is not necessary to rewrite the inode block with the *linktag* entry removed, because its *dl_cnt* reflects the true state of the directory. It will be cleared on the next flush of the inode to stable storage.

It is possible to have a single *linktag* entry involved in more than one operation. For example, suppose that there are two simultaneous link operations on a given file to the same block of the same directory. We handle this situation by associating a reference count with the *linktag* that prevents the *linktag* from being discarded until after all the operations using it have completed. The semantics of *dl_cnt* ensure that the link count will be recovered correctly, no matter in what state the multiple operations are when a crash occurs.

There is another subtle case that involves file renaming or linking: the link count is incremented before the *dl_off* value for the new directory entry is known. In this case, *dl_off* is set to an invalid offset of DL_NOSLOT that denotes that this *linktag* is a place holder. When a free slot in the directory has been obtained, the *linktag* is updated and is written to reflect the location of the new entry.

Directories also present a more complicated situation because each one has a ''..'' entry that points to its parent directory, and this link is reflected in the link count. Since there is only one ''..'' entry, we deal with this link also by setting an invalid offset value of DL_DOTDOT into the *dl_off* field.

Rename operations are complex — especially a move of a directory from its current owner to another directory already containing a directory with the same name. We handle such operations by breaking them down into individual *linktags* for each of the links that is created or removed. We did not implement an atomic rename operation because the *linktag* paradigm is not strong enough to support one. The biggest problem is that removing the old name of a renamed entity cannot be accomplished with only the *dl_cnt* and *dl_off* model of the *linktag*. It could be done if there were some

means of uniquely identifying each entry pointing to a given target. We could add a unique 1-byte tag to each directory entry pointing to a given target; we could then use the tag to identify which entry should be removed by *fsck* to complete the final stage of a rename.

5.4 Last Inode

One of the problems with enormous file systems is that a many empty inodes are typically configured when the file system is initialized. The standard UFS implementation requires that all these inodes be initialized during *mkfs*(1M), then examined during any full *fsck* operation. Dealing with such a large number of inodes dominates the time taken to create and *fsck* a file system. To reduce this time, a field giving the highest-numbered inode in use in each cylinder group, *lastino*, was added to each cylinder group. During a full *fsck* operation, this field allows the inodes examined to be limited to only those that might be used in each cylinder group. During file-system creation, this field allows us to bypass the initialization of all but the inode block containing the root inode, saving considerable time. The file system then initializes inode blocks as needed while the file system is in use. In particular, we found that making a new file system on a 40-GByte RAID-5 metadevice took 97 seconds with this feature enabled versus 1378 seconds with the standard UFS — a 14-fold reduction.

6 Performance Results

One of our goals was to increase the overall performance of NFS service provided by the Netra NFS server, as measured by the SPECsfs1.1 LADDIS benchmark [Whittle93]. LADDIS provides a client-independent characterization of a server's response time and throughput under a load consisting of direct remote calls to NFS service procedures on the server under test.

We performed a set of LADDIS benchmark runs to detail the effects on performance of enabling different aspects of our implementation. In each run, the server was loaded past the saturation point in steps. In addition, we crashed the server while it was under heavy load, and measured file-system recovery times under different configurations. The following elements were varied:

- RAID-5 (parity striped) versus RAID-0 (striped)
- Presto versus no Presto
- Bmap cache on versus off
- Fast fsck on versus off
- Veritas File System versus UFS

In addition, we performed a number of crash tests under various configurations and system loads to characterize the differences in recovery times.

6.1 Testing Configuration

The Netra NFS server used for benchmarking was an Ultra-2 with 2 300-MHz UltraSPARC(TM)-II processors with 1 GByte of DRAM and 32 MByte of NVRAM. The storage that we used for benchmarking was a SPARCStorage(TM) MultiPack configured with 12 4.2-GByte disks. The network controller used was the Sun Fast Ethernet (100Base-T). LADDIS load was generated by 2 Ultra-I machines, each configured with 64 MByte of DRAM and 167-MHz UltraSPARC processors.

The server was installed with the Netra NFS 1.2 release, which is essentially Solaris 2.5.1 with the UFS kernel module and utilities replaced by the Netra NFS-specific ones. The Sun Enterprise Volume Manager (Solstice Disk Suite) Version 2.4 provided RAID-0 and RAID-5 support. For the Veritas tests, the VxFS 3.2.2 File System and the VxLD 1.0.1 NFS Accelerator for Solaris were used. VxFS is normally configured with the transaction log contained at the beginning of the file system. An NFS load causes a lot of seeking between the log area and the actual file data, so Veritas developed VxLD to allow the log to reside on a device separate from the file system, thus avoiding the seek overhead. The VxFS plus VxLD configuration is thus similar to the approach described by Vahalia [Vahalia95].

For the RAID-0 and RAID-5 UFS tests, 11 drives were used. In the RAID-0 Veritas tests, 11 drives were set up as the file system, and a twelfth drive was designated as the VxLD log device. We set up all metadevices using an 8-KByte interleave size; in all cases, a single file system was created on the metadevice.

The Veritas setups would not strictly satisfy our design constraints for RAID-5 operation because the log disk is a single point of failure, and it would have to be mirrored to avoid loss of data should a log disk fail. This mirroring would decrease performance, but these results provide an approximate comparison. Also, we gave the Veritas RAID-5 setup the benefit of an extra disk drive in the tests.

6.2 LADDIS Results

The two factors that most influence performance are the choice of RAID-0 over RAID-5 and the presence or absence of Presto NVRAM cacheing. Figure 2 shows a set of LADDIS response-versus-throughput curves for these four combinations, with both *bmap*

cache and *fast fsck* enabled. In addition, there is a plot with Presto Lower disabled, (Presto Upper only), to show that component's effect on the RAID-5 results. No benefit accrues from configuring Presto Lower in a RAID-0 configuration. Presto increased the LADDIS *figure-of-merit number* (maximum throughput, and response time at that throughput) for RAID-5 from 303 NFS ops. at 61.7 msec to 1397 NFS ops at 13.8 msec and for RAID-0 from 859 NFS ops. at 44.4 msec to 3046 NFS ops at 20.0 msec.

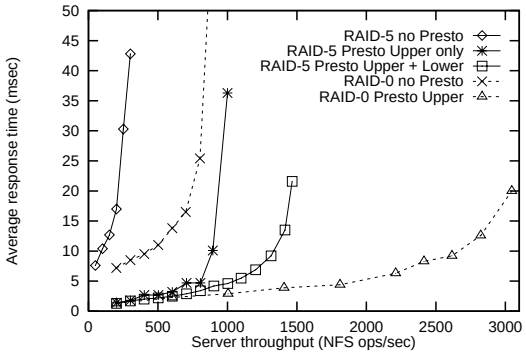

Figure 2. Comparison of RAID and Presto combinations.

Figures 3 through 6 show more detailed sets of LADDIS response-versus-throughput curves for benchmark runs done with various combinations of features enabled. In general, the *fast fsck* feature caused an increase in disk activity and a decrease in throughput, whereas the *bmap cache* decreased the disk activity and increased the throughput. Enabling Presto reduced the size of these differences.

The Veritas results illustrate why we chose not to use an existing disk-based logging solution with Presto added. In Figure 3, from the curves obtained with Presto disabled, we see that the VxFS/VxLD combination tracks the Solaris baseline plot closely up to 400 NFS ops., although it is not as good as the *bmap-cache* plot up to 500 NFS ops.; but then is better above those points, and actually has the highest throughput of 587 NFS ops. at 58.2 msec. The plot for *fast fsck* alone shows relatively poor performance, reaching only 250 NFS ops. at 43.4 msec, and saturating at 298 NFS ops at 94.2 msec.

Figure 4 shows the normal mode of operation with RAID-5 and Presto enabled; here, there is a complete reversal. The Veritas plot shows the lowest performance, reaching only 747 NFS ops. at 49.3 msec. This is still better better than the no-Presto case, but it is far below the case of *bmap cache* plus *fast fsck* (1466 NFS ops. at 21.6 msec). The benefit of the *bmap cache* can be seen clearly in Figure 4. The maximum throughput increases by 23% when the *bmap cache* is added to the *fast fsck* configuration.

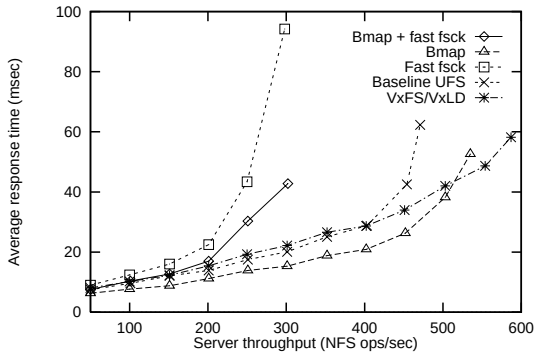

Figure 3. RAID-5 LADDIS performance without Presto.

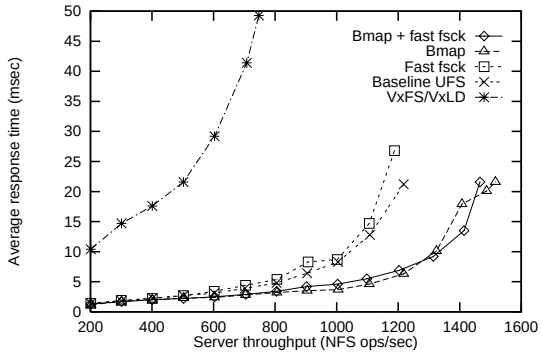

Figure 4. RAID-5 LADDIS performance with Presto.

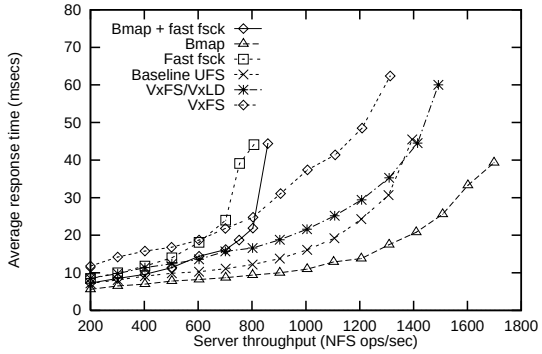

Figure 5. RAID-0 LADDIS performance without Presto.

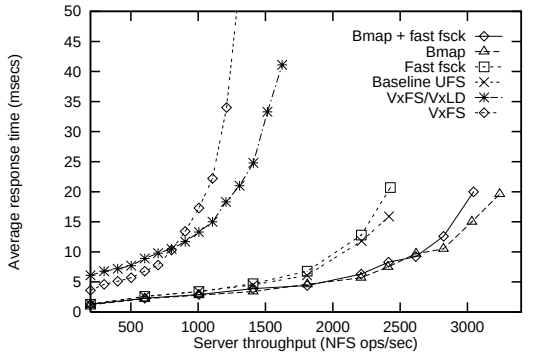

Figure 6. RAID-0 LADDIS performance with Presto.

The RAID-0 results, shown in Figures 5 and 6, are similar, although of course the throughput scales are larger. We included plots of VxFS with and without VxLD to show the effect of moving the VxFS log to a separate disk.

Most of the runs in the results shown in the graphs were repeated multiple times; the results were stable. To get an estimate of the sampling error, we did 10 runs of an 11-disk RAID-0 configuration at 3000 NFS ops. The average response time at that load was 10.92 msec, with a standard deviation over the 10 runs of 1.05 msec.

6.3 Crash-Test Results

To obtain a characterization of the recovery performance, we deliberately and repeatedly crashed the NFS server under test 5 minutes into the second of two 10-minute LADDIS runs, at varying client loads. We used the first run to measure throughput and response time, so that we could determine the point at which saturation occurred. We collected data in the 11-disk RAID-0 configuration using standard UFS as a baseline, *bmap cache* alone, *bmap cache* plus *fast fsck* with some variations on its tuning parameters, and VxFS without VxLD. All tests were run with Presto enabled. The results are shown in Figure 7, where we note many interesting points.

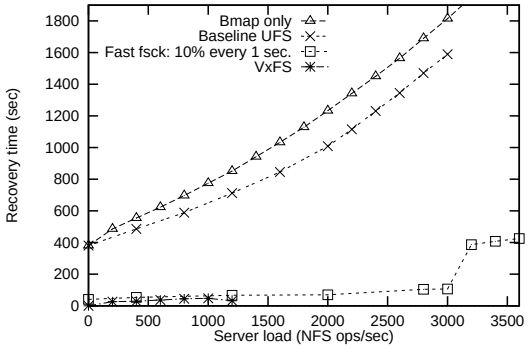

Figure 7. Recovery time versus server load .

The baseline UFS plot is almost a straight line, increasing as a function of load. This almost linear rise occurs because the *fsck* time is proportional to the number of files and allocated blocks in the file system, which in turn is directly proportional to the LADDIS load. The constant portion — where the load is 0 and the file system is empty — reflects that *fsck* has to read all the cylinder groups and inodes at least once. This takes 382 seconds in our 44-GByte file system. The *bmap* plot shows higher recover times than baseline UFS and this is probably because *fsck* writes *bmap-cache* indirect-block information out to the correct place in the indirect block. We made the VxFS plot without using VxLD because we encountered difficulty with VxLD. The VxLD enhancement moves the log off the file system proper onto a separate device, but on reboot copies the log data back into the file system. The VxFS recovery then proceeds as though the data had been logged into the file system originally. This copying typically took between 45 and 50 seconds, so it should be added to the VxFS recovery times if VxLD is used. The VxFS recovery times are otherwise good, with no overhead on an empty file system.

The *fast fsck* result is between the baseline UFS and VxFS numbers, as might be expected. There is a constant overhead portion of about 42 seconds that occurs because *fast fsck* has to read all the cylinder groups at least once. The main reason that it does so is to maintain consistency of allocation counts in the file-system superblock and in the cylinder-group summary area just after the superblock. It would be possible to maintain a busy-cylinder-group bitmap within the superblock and only visit those cylinder groups marked busy. Doing that would reduce the empty or clean file-system overhead to 0, and probably would lower the *fast fsck* recovery time below that of VxFS, although it would complicate maintaining consistency of the summary area.

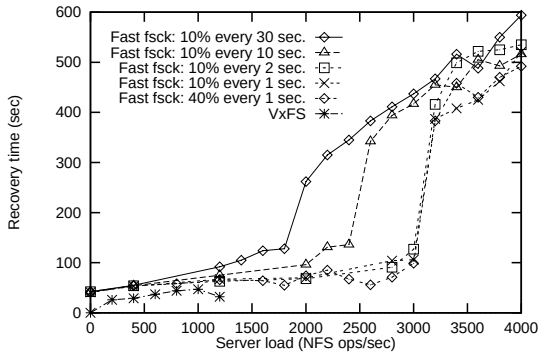

Figure 8. Flushing-rate effect on *fast fsck* times.

The rate and duty cycle of cylinder-group flushing have an effect on the shape of the recovery-time curves as a function of the tuning parameters, as shown in Figure 8. There are two parameters: the percent of real time that the flushing thread is active, and the interval between activations. The default values were set such that the flusher runs every 30 seconds for 10% of the interval, or for 3.0 sec. We see that, as the load increases, the recovery time rises, because the higher load on the server allows more inodes and cylinder groups to become busy between flushes. Decreasing the interval between flushes yields a significant reduction in recovery time at intermediate loads.

Running the flusher every 1.0 sec allows the recovery time to be flat up to the saturation point at

3000 NFS ops., indicating that it is doing a good job of keeping the file system mostly clean. Once the load is beyond saturation, however, the flusher seems unable to clean the file system sufficiently fast, probably because it does not get enough disk bandwidth to clean effectively. The flusher does only one disk operation at a time, so that it slows dramatically when the disk queues become large. The recovery time curve for using a 40% duty cycle every second was nearly identical to the 10% every second curve, so increasing the flusher's active time did not give any benefit.

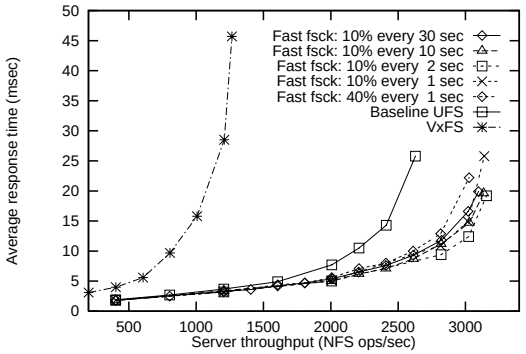

Figure 9. Flushing-rate effect on throughput and response time.

Figure 9 shows the corresponding LADDIS throughput curves for the tuning values shown in Figure 8, with the VxFS response curve shown for comparison. Using VxFS, for about a 60% reduction in maximum throughput, we can achieve a 25% to 50% reduction in recovery time between *fast fsck* and VxFS. Notice that the *fast fsck* response curves are virtually identical across the range of tuning parameters, although the 40% duty-cycle plot is above the others.

6.4 Comparison with Other Approaches

The LADDIS numbers reported by Vahalia [Vahalia95] reach a peak of 597 NFS ops. at around 29 msec of latency on a 60-MHz Pentium platform with 11 disks. In comparison, their nonbatching logging implementation peaks at around 400 NFS ops. Vahalia's approach is clearly viable, and might exhibit better performance using an NVRAM-based logging device, although a Presto-based solution would likely behave similarly to VxFS/VxLD. The authors measured reported recovery times by crashing the system at peak LADDIS load of around 600 NFS ops., at which time their file system had about 3 GByte of data over 10 (logging case) or 11 (no-logging case) disks. The recovery times were 450 sec for the standard *fsck* case, and between 3 and 14 sec for log-based recovery. The authors mention that the standard *fsck* checking was done serially, implying that their disks contained separate file systems.

Bearing in mind that the architectures of our system and Vahalia's are very different in terms of CPU, bus, memory, and controller configuration, we still attempted to obtain an approximate comparision with their results. We created a single file system of about 5 GByte on a RAID-0 device with 11 disk drives. The runs used for comparison had *bmap cache* and *fast fsck* enabled, and timings were done both with and without Presto enabled. We measured an average response time at 600 NFS ops. of 12.4 msec; saturation occurred at about 874 NFS ops. and 38.1 msec. Our higher throughput and lower latency is probably due to the combined benefits of using the *bmap cache* and more powerful processors. With Presto enabled and a necessarily larger file system, the response time at 600 NFS ops. was 2.2 msec, and saturation occurred at about 3000 NFS ops at 16 msec.

On a file system populated by a 600 NFS ops. LADDIS run and unmounted cleanly, *fast fsck* ran in 4 sec, which would be the best case on a lightly loaded system after a crash. Table 1 compares recovery times under several scenarios when the server was crashed at 600 NFS ops.

File System	Recovery Time (sec)	Standard Deviation
Vahalia	3—14	—
fast fsck + Presto	15.6	4.4
VxFS + Presto	34.3	3.1
VxFS	51.9	6.5
fast fsck	54.0	10.8
UFS	219.0	8.9

Table 1. Comparison of Recovery Times

As we might expect, the *fast fsck* recovery times without Presto are longer than are those achieved with Vahalia's logging approach, because the information maintained to do the recovery is less precise than a transaction log. On the other hand, our throughput and latency numbers appear to be better even without Prestoserve enabled and the recovery times are competitive with Presto on. It may be surprising that *fast fsck* with Presto takes less time to do recovery than does VxFS on a small file system, but consider that the relative overhead for *fast fsck* to read all the cylinder groups is lower with fewer cylinder groups.

To get a comparison with Hitz's results, [Hitz94], we used benchmark results from the SPEC web page at *http://www.spec.org/osg/sfs93/results/results.html* for the Network Appliance Corporation F520 and F540. These systems use a 275 MHz Alpha processor with 14 disk drives in a RAID-4 configuration. The maximum

throughputs are 2361 NFS ops at 8.3 msec for the F520 and 2230 NFS ops. at 7.7 msec for the F540. If we multiply our RAID-5 throughput number of 1397 by 13/10 to scale for the larger number of data drives, we get 1905 NFS ops. as a comparable extrapolation at 13.8 msec. It is likely that the difference between these results is due to lower RAID-4 parity overhead afforded by the log-structured WAFL file system, which is designed to do writes in full stripe widths.

7 Conclusions and Additional Work

The benchmarks for crash-recovery times as well as those for the server performance under LADDIS loads indicate that UFS with the fast consistency checker is a highly competitive local file system for use in an NFS server. Since our approach was designed with NVRAM as a key component, we were able to achieve performance that was significantly better than that obtained with logging approaches with NVRAM added as an afterthought, and we maintained acceptable crash-recovery times. The *fast fsck* consistency checking was 4 to 30 times faster than the original UFS *fsck*, depending on the configuration and on the load when the system went down.

An additional avenue of exploration is to determine the effectiveness of the hot-spot allocator when a parity-block cache is added to the RAID-5 implementation. We expect this cache to improve the single-writer write throughput of the system, and to increase the maximum LADDIS throughput. We are now characterizing the system's behavior with SPECsfs2.0.

8 Acknowledgments

A few people who worked with us on developing the ideas and implementation described here deserve mention: John Corbin was involved early in the architectural discussions surrounding how we should use the NVRAM effectively, and Rob Gittens worked on the changes to SDS that supported Presto Lower. Also, the reviewers comments were very helpful in improving the paper, and Pei Cao's shepherding of the paper in particular was most appreciated. Finally, we would like to thank Lyn Dupre´ for her expert and efficient help in editing the paper.

9 References

[Chutani92] S. Chutani, O. Anderson, M. Kazar, B. Leverett, W. Mason, R. Sidebotham, ''The Episode File System,'' *Proceedings of the USENIX Winter 1992 Conference*, pages 43-59, January 1992.

[Ganger94] G. Ganger, Y. Patt, ''Metadata Update Performance in File Systems,'' *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 49-60, November 1994.

[Hagmann87] R. Hagmann, ''Reimplementing the Cedar File System Using Logging and Group Commit,'' *ACM Symposium on Operating Systems Principles*, pages 155-162, November 1987.

[Hitz94] D. Hitz, J. Lau, M. Malcolm, ''File System Design for an NFS File Server Appliance,'' *Proceedings of the USENIX Winter 1994 Conference*, pages 235-245, January 1994.

[Juszczak94] C. Juszczak, ''Improving the Write Performance of an NFS Server,'' *Proceedings of the USENIX 1994 Winter Conference*, pages 247-259, January 1994.

[Leffler89] S.J. Leffler, M. McKusick, M.J. Karels, J.S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System* Reading, MA: Addison-Wesley, 1989.

[McKusick84] M. McKusick, W. Joy, S. Leffler, R. Fabry, ''A Fast File System for UNIX,'' *ACM Transactions on Computer Systems*, **2**(3):181-197, August 1984.

[McKusick94] M. McKusick, T.J. Kowalski, ''Fsck — The UNIX File System Check Program,'' *4.4 BSD System Manager's Manual* Sebastopol, CA: O'Reilly & Associates, 1994.

[Moran90] J. Moran, R. Sandberg, D. Coleman, J. Kepecs, B. Lyon, ''Breaking Through the NFS Performance Barrier,'' *Proceedings of the 1990 Spring European UNIX Users Group*, April, 1990.

[NFS94] B. Pawlowski, C. Juszczak, P Staubach, C. Smith, D. Level, D. Hitz, ''NFS Version 3: Design and Implementation,'' *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 137-151, June 1994.

[Ousterhout89] J.K. Ousterhout, F. Douglis, ''Beating the I/O Bottleneck: A Case for Log-Structured File Systems,'' *Operating Systems Review*, **23**(1):11-27, January 1989.

[Ousterhout90] J.K. Ousterhout, ''Why aren't Operating Systems Getting Faster as Fast as Hardware?'' *Proceedings of the USENIX Summer 1990 Conference*, June 1990.

[Patterson88] D. Patterson, G. Gibson, R. Katz, ''A Case for Redundant Arrays of Inexpensive Disks

(RAID),'' *ACM SIGMOD Conference*, June 1988.

[Peacock96] J.K. Peacock, "Method and Apparatus for Caching File Control Information", U. S. Patent Application Serial No. 08/673,958.

[Presto93] Digital Equipment Corporation (DEC), *Guide to Prestoserve*, DEC OSF/1 Prestoserve Product Documentation, (Order number AA-PQTOA-TE), March 1993.

[Rosenblum92] M. Rosenblum, J. Ousterhout, ''The Design and Implementation of a Log-Structured File System,'' *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 1-15, Association for Computing Machinery SIGOPS, October 1991.

[Sandberg85] R.D. Sandberg, S. Goldberg, S. Kleiman, D. Walsh, B. Lyon, ''Design and Implementation of the Sun Network Filesystem,'' *Proceedings of the USENIX Summer 1985 Conference*, June 1985.

[Seltzer93] M. Seltzer, K. Bostic, M. McKusick, C. Staelin, ''An Implementation of a Log-Structured File System for UNIX,'' *Proceedings of the Winter 1993 USENIX Conference*, pages 307-326, January 1993.

[Seltzer95] M. Seltzer, K. Smith, H. Balarkishnan, J. Chang, S. McMains, V. Padmanabhan, ''File System Logging Versus Clustering: A Performance Comparison,'' *Proceedings of the USENIX 1995 Conference*, pages 249-264, January 1995.

[Vahalia95] U. Vahalia, C.G. Gray, D. Ting, ''Metadata Logging in an NFS Server,'' *Proceedings of the USENIX 1995 Conference*, pages 265-276, January 1995.

[Wittle93] M. Wittle, B. Keith, ''LADDIS: The Next Generation in NFS File Server Benchmarking,'' *Proceedings of the USENIX Summer 1993 Conference*, pages 111-128, June 1993.

10 Author Information

J. Kent Peacock, Ph.D., has worked as an architect and implementor on the Netra NFS project since 1995. Between 1990 and 1997, he worked as a consultant for SMCC, SunLabs, SunSoft, Veritas, Hewlett-Packard, and Intel, generally doing multiprocessor implementations and file-system improvements. While at Intel, he worked on the multiprocessor version of SVR4 that was released by USL as SVR4/MP. Prior to becoming a consultant, Dr. Peacock rolled the dice at two startup companies: Dialogic Systems and Counterpoint Computers. He earned his B.Sc. in electrical engineering from the University of Manitoba, Canada, and then Masters and Ph.D. degrees in computer science from the University of Waterloo, Canada, where he met and fell in love with UNIX in 1978. When he is not trying to make computers do things faster, he spends time trying to lower his golf handicap of 11 or playing in the Long Day Flute Quartet and the HP Symphony.

Ashvin Kamaraju, M.A., M.S., is the engineering manager for Netra file-server and Proxy-cache server products at Sun. He has been at Sun for the past 7 years, where he has worked on the stable memory allocator and almost all the subsystems of the Netra NFS server. Prior to joining the file-server project, he worked in the desktop-systems group at Sun, doing operating-system porting, memory-management software and virtual-memory drivers for graphics acclerators. He earned his M.A. in mathematics and computer science and his M.S. in chemical engineering from the University of Cincinnati.

Sanjay Agrawal, M.S., is an engineer in the Netra NFS server group at Sun, where he has been working on NFS server performance and benchmarks. He holds an M.S. in computer science and a B.S. in electrical engineering from Indian Institute of Technology at Kanpur, India.

All the authors can be reached by U.S. Mail at Sun Microsystems, 901 San Antonio Rd., MS UMPK03-119, Palo Alto, California 94303, or by electronic mail at {kent,akk,sanj}@eng.sun.com.