USENIX Association

Proceedings of the FAST 2002 Conference on File and Storage Technologies

Monterey, California, USA
January 28-30, 2002

© 2002 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

SnapMirror®: File System Based Asynchronous Mirroring for Disaster Recovery

Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, Shane Owara

Network Appliance Inc.
Sunnyvale, CA

{hugo, stephen, mikef, hitz, srk, owara}@netapp.com

Abstract

Computerized data has become critical to the survival of an enterprise. Companies must have a strategy for recovering their data should a disaster such as a fire destroy the primary data center. Current mechanisms offer data managers a stark choice: rely on affordable tape but risk the loss of a full day of data and face many hours or even days to recover, or have the benefits of a fully synchronized on-line remote mirror, but pay steep costs in both write latency and network bandwidth to maintain the mirror. In this paper, we argue that asynchronous mirroring, in which batches of updates are periodically sent to the remote mirror, can let data managers find a balance between these extremes. First, by eliminating the write latency issue, asynchrony greatly reduces the performance cost of a remote mirror. Second, by storing up batches of writes, asynchronous mirroring can avoid sending deleted or overwritten data and thereby reduce network bandwidth requirements. Data managers can tune the update frequency to trade network bandwidth against the potential loss of more data. We present SnapMirror, an asynchronous mirroring technology that leverages file system snapshots to ensure the consistency of the remote mirror and optimize data transfer. We use traces of production filers to show that even updating an asynchronous mirror every 15 minutes can reduce data transferred by 30% to 80%. We find that exploiting file system knowledge of deletions is critical to achieving any reduction for no-overwrite file systems such as WAFL and LFS. Experiments on a running system show that using file system metadata can reduce the time to identify changed blocks from minutes to seconds compared to purely logical approaches. Finally, we show that using SnapMirror to update every 30 minutes increases the response time of a heavily loaded system only 22%.

1 Introduction

As reliance on computerized data storage has grown, so too has the cost of data unavailability. A few hours downtime can cost from thousands to millions of dollars depending on the size of the enterprise and the role of the data. With increasing frequency, companies are instituting disaster recovery plans to ensure appropriate data availability in the event of a catastrophic failure or disaster that destroys a site (e.g. flood, fire, or earthquake). It is relatively easy to provide redundant server and storage hardware to protect against the loss of physical resources. Without the data, however, the redundant hardware is of little use.

The problem is that current strategies for data protection and recovery offer either inadequate protection, or are too expensive in performance and/or network bandwidth. Tape backup and restore is the traditional approach. Although favored for its low cost, restoring from a nightly backup is too slow and the restored data is up to a day old. Remote synchronous and semi-synchronous mirroring are more recent alternatives. Mirrors keep backup data on-line and fully synchronized with the primary store, but they do so at a high cost in performance (write latency) and network bandwidth. Semi-synchronous mirrors can reduce the write-latency penalty, but can result in inconsistent, unusable data unless write ordering across the entire data set, not just within one storage device, is guaranteed. Data managers are forced to choose between two extremes: synchronized with great expense or affordable with a day of data loss.

In this paper, we show that by letting a mirror volume lag behind the primary volume it is possible to reduce substantially the performance and network costs of maintaining a mirror while bounding the amount of data loss. The greater the lag, the greater the data loss, but the cheaper the cost of maintaining the mirror. Such asynchronous mirrors let data managers tune their systems to strike the right balance between potential data loss and cost.

We present SnapMirror, a technology which implements asynchronous mirrors on Network Appliance filers. SnapMirror periodically transfers self-consistent snapshots of the data from a source volume to the destination volume. The mirror is on-line, so disaster recov-

SnapMirror, NetApp, and WAFL are registered trademarks of Network Appliance, Inc.

ery can be instantaneous. Users set the update frequency. If the update frequency is high, the mirror will be nearly current with the source and very little data will be lost when disaster strikes. But, by lowering the update frequency, data managers can reduce the performance and network cost of maintaining the mirror at the risk of increased data loss.

There are three main problems in maintaining an asynchronous mirror. First, for each periodic transfer, the system must determine which blocks need to be transferred to the mirror. To obtain the bandwidth reduction benefits of asynchrony, the system must avoid transferring data which is overwritten or deleted. Second, if the source volume fails at any time, the destination must be ready to come on line. In particular, a half-completed transfer can't leave the destination in an unusable state. Effectively, this means that the destination must be in, or at least recoverable to, a self-consistent, state at all times. Finally, for performance, disk reads on the source and writes on the destination must be efficient.

In this paper, we show how SnapMirror leverages the internal data structures of NetApp's WAFL® file system [Hitz94] to solve these problems. SnapMirror leverages the active block maps in WAFL's snapshots to quickly identify changed blocks and avoid transferring deleted blocks. Because SnapMirror transfers self-consistent snapshots of the file system, the remote mirror is always guaranteed to be in a consistent state. New updates appear atomically. Finally, because it operates at the block level, SnapMirror is able to optimize its data reads and writes.

We show that SnapMirror's periodic updates transfer much less data than synchronous block-level mirrors. Update intervals as short as 1 minute are sufficient to reduce data transfers by 30% to 80%. The longer the period between updates, the less data needs to be transferred. SnapMirror allows data managers to optimize the tradeoff of data currency against cost for each volume.

In this paper, we explore the interaction between asynchronous mirroring and no-overwrite file systems such as LFS [Rosenblum92] and WAFL. We find that asynchronous block-level mirroring of these file systems does not transfer less data synchronous mirroring. Because these file systems do not update in place, logical overwrites become writes to new storage blocks. To gain the data reduction benefits of asynchrony for these file systems, it is necessary to have knowledge of which blocks are active and which have been deallocated and are no longer needed. This is an important observation since many commercial mirroring products are implemented at the block level.

1.1 Outline for remainder of paper

We start, in Section 1.2, with a discussion of the requirements for disaster recovery. We go on in Sections 1.3 and 1.4 to discuss the shortcomings of tape-based recovery and synchronous remote mirroring. In Section 2, we review related work. We present the design and implementation of SnapMirror in Section 3. In Section 4, we use system traces to study the data reduction benefits of asynchronous mirroring with file system knowledge. Then, in Section 5, we compare SnapMirror to asynchronous mirroring at the logical file level. Section 6, presents experiments measuring the performance of our SnapMirror implementation running on a loaded system. Conclusion, acknowledgments, and references are in Sections 7, 8, and 9.

1.2 Requirements for Disaster Recovery

Disaster recovery is the process of restoring access to a data set after the original was destroyed or became unavailable. Disasters should be rare, but data unavailability must be minimized. Large enterprises are asking for disaster recovery techniques that meet the following requirements:

Recover quickly. The data should be accessible within a few minutes after a failure.

Recover consistently. The data must be in a consistent state so that the application does not fail during the recovery attempt because of a corrupt data set.

Minimal impact on normal operations. The performance impact of a disaster recovery technique should be minimal during normal operations.

Up to date. If a disaster occurs, the recovered data should reflect the state of the original system as closely as possible. Loss of a day or more worth of updates is not acceptable in many applications.

Unlimited distance. The physical separation between the original and recovered data should not be limited. Companies may have widely separated sites and the scope of disasters such as earthquakes or hurricanes may require hundreds of miles of separation.

Reasonable cost. The solution should not require excessive cost, such as many high-speed, long-distance links (e.g. direct fiber optic cable). Preferably, the link should be compatible with WAN technology.

1.3 Recovering from Off-line Data

Traditional disaster recovery strategies involve loading a saved copy of the data from tape onto a new server in a different location. After a disaster, the most recent full backup tapes are loaded onto the new server. A series of nightly incremental backups may follow the

full backup to bring the recovered volume as up-to-date as possible. This worked well when file systems were of moderate size and when the cost of a few hours of downtime was acceptable, provided such events were rare.

Today, companies are taking advantage of the 60% compound annual growth rate in disk drive capacity [Growchowski96] and file system size is growing rapidly. Terabyte storage systems are becoming commonplace. Even with the latest image dump technologies [Hutchinson99], data can only be restored at a rate of 100-200 GB/hour. If disaster strikes a terabyte file system, it will be off line for at least 5-10 hours if tape-based recovery technologies are used. This is unacceptable in many environments.

Will technology trends solve this problem over time? Unfortunately, the trends are against us. Although disk capacities are growing 60% per year, disk transfer rates are growing at only 40% per year [Grochowski96]. It is taking more, not less, time to fill a disk drive even in the best case of a purely sequential data stream. In practice, even image restores are not purely sequential and achieved disk bandwidth is less than the sequential ideal. To achieve timely disaster recovery, data must be kept on-line and ready to go.

1.4 Remote Mirroring

Synchronous remote mirroring immediately copies all writes to the primary volume to a remote mirror volume. The original transfer is not acknowledged until the data is written to both volumes. The mirror gives the user a second identical copy of the data to fall back on if the primary file system fails. In many cases, both copies of the data are also locally protected by RAID.

The down side of synchronous remote mirroring is that it can add a lot of latency to I/O write operations. Slower I/O writes slow down the server writing the data. The extra latency results first from serialization and transmission delays in the network link to the remote mirror. Longer distances can bloat response time to unacceptable levels. Second, unless there is a dedicated high-speed line to the remote mirror, network congestion and bandwidth limitations will further reduce performance. For these reasons, most synchronous mirroring implementations limit the distance to the remote mirror to 40 kilometers or less.

Because of its performance limitations, synchronous mirroring implementations sometimes slightly relax strict synchrony, to allow a limited number of source I/O operations to proceed before waiting for acknowledgment of receipt from the remote site[1]. Although this approach can reduce I/O latency, it does not reduce the link bandwidth needed to keep up with the writes. Further, the improved performance comes at the cost of some potential data loss in the event of a disaster.

A major challenge for non-synchronous mirroring is ensuring the consistency of the remote data. If writes arrive out-of-order at the remote site, the remote copy of the data may appear corrupted to an application trying to use the data after a disaster. If this occurs, the remote mirroring will have been useless since a full restore from tape will probably be required to bring the application back on line. The problem is especially difficult when a single data set is spread over multiple devices and the mirroring is done at the device level. Although each device guarantees in-order delivery of its the data, there may be no ordering guarantees among the devices. In a rolling disaster, one in which devices fail over a period of time (imagine fire spreading from one side of the data center to the other), the remote site may receive data from some devices but not others. Therefore, whenever synchrony is relaxed, it is important that it be coordinated at a high enough level to ensure data consistency at the remote site.

Another important issue is keeping track of the updates required on the remote mirror should it or the link between the two systems become unavailable. Once the modification log on the primary system is filled, the primary system usually abandons keeping track of individual modifications and instead keeps track of updated regions. When the destination again becomes available, the regions are transferred. Of course, the destination file system may be inconsistent while this transfer is taking place, since file system ordering rules may be violated, but it's better than starting from scratch.

2 Related Work

There are other ways to provide disaster recovery besides restore from tape and synchronous mirroring. One is server replication.

Server replication is another approach to providing high availability. Coda is one example of a replicated file system [Kistler93]. In Coda, the clients of a file server are responsible for writing to multiple servers. This approach is essentially synchronous logical-level mirroring. By putting the responsibility for replication on the clients, Coda effectively off-loads the servers. And, because clients are aware of the multiple servers, recovery from the loss of a server is essentially instantaneous. However, Coda is not designed for replication over a WAN. If the WAN connecting a client to a remote server

1. EMC's SRDF™ in semi-synchronous mode or Storage Computer's Omniforce® in log synchronous mode.

is slow or congested, the client will feel a significant performance impact. Another difference is that where Coda leverages client-side software, SnapMirror's goal is to provide disaster recovery for the file servers without client side modifications.

Earlier, we mentioned that SnapMirror leverages file system metadata to detect new data since the last update of the mirror. But, there are many other approaches.

At the logical file system level, the most common approach is to walk the directory structure checking the time that files were last updated. For example, the UNIX dump utility compares the file modify times to the time of the last dump to determines which files it should write to an incremental dump tape. Other examples of detecting new data at the logical level include programs like rdist and rsync [Tridgell96]. These programs traverse both the source and destination file systems, looking for files that have been more recently modified on the source than the destination. The rdist program will only transfer whole files. If one byte is changed in a large database file, the entire file will be transferred. The rsync program works to compute a minimal range of bytes that need be transferred by comparing checksums of byte ranges. It uses CPU resources on the source server to reduce network traffic. Compared to these programs SnapMirror does not need to traverse the entire file system or do checksums to determine the block differences between the source and destination. On the other hand, SnapMirror needs to be tightly integrated with the file system whereas approaches which operate at the logical level are more general.

Another approach to mirroring, adopted by databases such as Oracle, is to write a time-stamp in a header in each on-disk data block. The time-stamp enables Oracle to determine if a block needs to be backed up by looking only at the relatively small header. This can save a lot of time compared to approaches which must perform check sums on the contents of each block. But, it still requires each block to be scanned. In contrast, SnapMirror uses file system data structures as an index to detect updates. The total amount of data examined is similar in the two cases, but the file system structures are stored more densely and consequently the number of blocks that must be read from disk is much smaller.

3 SnapMirror Design and Implementation

SnapMirror is an asynchronous mirroring package currently available on Network Appliance file servers. Its design goal was to meet the data protection needs of large-scale systems. It provides a read-only, on-line, replica of a source file system. In the event of disaster, the replica can be made writable, replacing the original source file system.

Periodically, SnapMirror reflects changes in the source volume to the destination volume. It replicates the source at a block-level, but uses file system knowledge to limit transfers to blocks that are new or modified and that are still allocated in the file system. SnapMirror does not transfer blocks which were written but have since been overwritten or deallocated.

Each time SnapMirror updates the destination, it takes a new snapshot of the source volume. To determine which blocks need to be sent to the destination, it compares the new snapshot to the snapshot from the previous update. The destination jumps forward from one snapshot to the next when each transfer is completed. Effectively, the entire update is atomically applied to the destination volume. Because the source snapshots always contain a self-consistent, point-in-time image of the entire volume or file system, and these snapshots are applied atomically to the destination, the destination always contains a self-consistent, point-in-time image of the volume. SnapMirror solves the problem of ensuring destination data consistency even when updates are asynchronous and not all writes are transferred so ordering among individual writes cannot be maintained.

The system administrator sets SnapMirror's update frequency to balance the impact on system performance against the lag time of the mirror.

3.1 Snapshots and the Active Map File

SnapMirror's advantages lie in its knowledge of the Write Anywhere File Layout (WAFL) file system and its snapshot feature [Hitz94], which runs on top of Network Appliance's file servers. WAFL is designed to have many of the same advantages as the Log Structured File System (LFS) [Rosenblum92]. It collects file system block modification requests and then writes them to an unused group of blocks. WAFL's block allocation policy is able to fit new writes in among previously allocated blocks, and thus it avoids the need for segment-cleaning. WAFL also stores all metadata in files, like the Episode file system [Chutani92]. This allows updates to write metadata anywhere on disk, in the same manner as regular file blocks.

WAFL's on-disk data structure is a tree that points to all data and metadata. The root of the tree is called the *fsinfo block*. A complete and consistent version of the file system can be reached from the information in this block. The fsinfo block is the only exception to the no-overwrite policy. Its update protocol is essentially a database-like transaction; the rest of the file system image must be consistent whenever a new fsinfo block overwrites the old. This insures that partial writes will never corrupt the

file system.

It is easy to preserve a consistent image of a file system, called a *snapshot*, at any point in time, by simply saving a copy of the information in the fsinfo block and then making sure the blocks that comprise the file system image are not reallocated. Snapshots will share the block data that remains unmodified with the active file system; modified data are written out to unallocated blocks. A snapshot image can be accessed through a pointer to the saved fsinfo block.

WAFL maintains the block allocations for each snapshot in its own *active map file*. The active map file is an array with one allocation bit for every block in the volume. When a snapshot is taken, the current state of the active file system's active map file is frozen in the snapshot just like any other file. WAFL will not reallocate a block unless the allocation bit for the block is cleared in every snapshot's active map file. To speed block allocations, a summary active map file maintains for each block, the logical-OR of the allocation bits in all the snapshot active map files.

3.2 SnapMirror Implementation

Snapshots and the active map file provide a natural way to find out block-level differences between two instances of a file system image. SnapMirror also uses such block-level information to perform efficient block-level transfers. Because the mirror is a block-by-block replica of the source, it is easy to turn it into a primary file server for users, should disaster befall the source.

3.2.1 Initializing the Mirror

The destination triggers SnapMirror updates. The destination initiates the mirror relationship by requesting an initial transfer from the source. The source responds by taking a *base reference snapshot* and then transferring all the blocks that are allocated in that or any earlier snapshot, as specified in the snapshots' active map files. Thus, after initialization, the destination will have the same set of snapshots as the source. The base snapshot serves two purposes: first, it provides a reference point for the first update; second, it provides a static, self-consistent image which is unaffected by writes to the active file system during the transfer.

The destination system writes the blocks to the same logical location in its storage array. All the blocks in the array are logically numbered from 1 to N on both the source and the destination, so the source and destination array geometries need not be identical. However, because WAFL optimizes block layout for the underlying array geometry, SnapMirror performance is best when the source and destination geometries match and the optimizations apply equally well to both systems. When the block transfers complete, the destination writes its new fsinfo block.

3.2.2 Block-Level Differences and Update Transfers

Part of the work involved in any asynchronous mirroring technique is to find the changes that have occurred in the primary file system and make the same changes in another file system. Not surprisingly, SnapMirror uses WAFL's active map file and reference snapshots to do this as shown in Figure 1.

When a mirror has an update scheduled, it sends a message to the source. The source takes an *incremental reference snapshot* and compares the allocation bits in the active map files of the base and incremental reference snapshots. This active map file comparison follows the following rules:

If the block is not allocated in either active map, it is **unused**. The block is not transferred. It did not exist in the old file system image, and is not in use in the new one. Note that it could have been allocated and deallocated between the last update and the current one.

If the block is allocated in both active maps, it is **unchanged**. The block is not transferred. By the file system's no-overwrite policy, this block's data has not changed. It could not have been overwritten, since the old reference snapshot keeps the block from being re-allocated.

If the block is only allocated in the base active map, it has been **deleted**. The block is not transferred. The data it contained has either been deleted or changed.

If the block is only allocated in the incremental active map, it has been **added**. The block is transferred. This means that the data in this block is either new or an updated version of an old block.

Note that SnapMirror does not need to understand whether a transferred block is user data or file system metadata. All it has to know is that the block is new to the file system since the last transfer and therefore it should be transferred. In particular, block de-allocations automatically get propagated to the mirror, because the updated blocks of the active map file are transferred along with all the other blocks.

In practice, SnapMirror transfers the blocks for all existing snapshots that were created between the base and incremental reference snapshots. If a block is newly allocated in the active maps of any of these snapshots, then it is transferred. Otherwise, it is not. Thus, the destination has a copy of all of the source's snapshots.

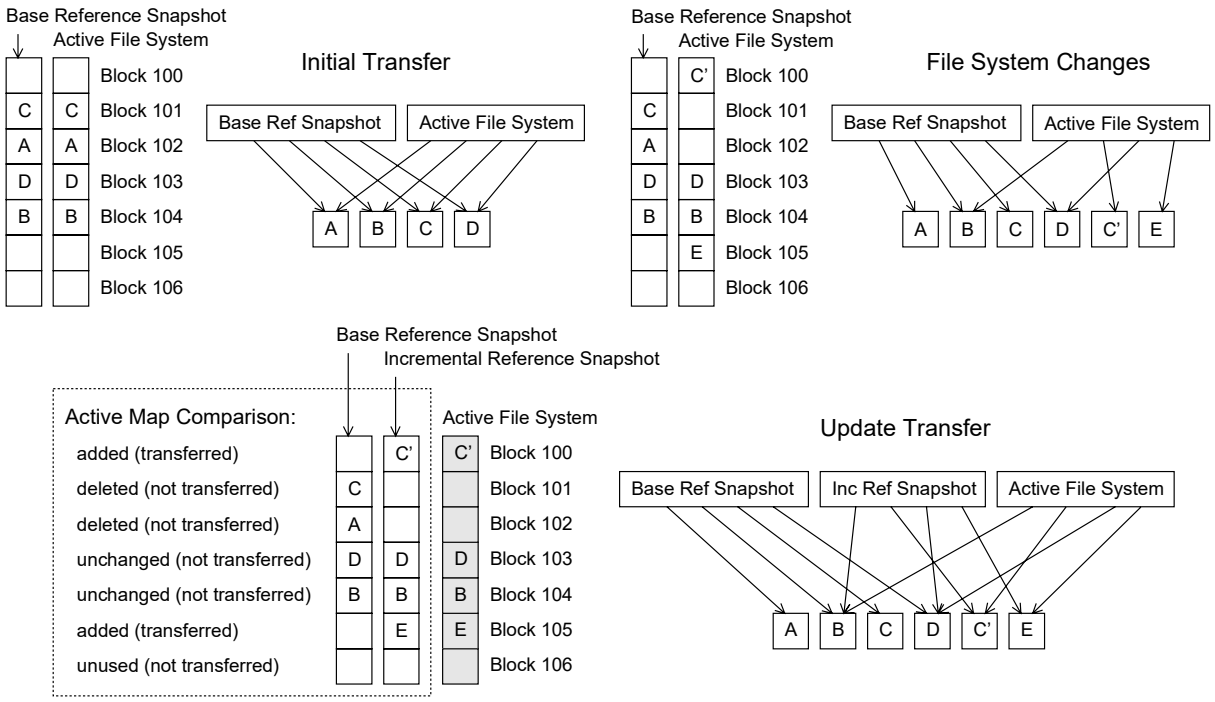

Figure 1. SnapMirror's use of snapshots to identify blocks for transfer. SnapMirror uses a base reference snapshot as point of comparison on the source and destination filers. The first such snapshot is used for the Initial Transfer. File System Changes cause the base snapshot and the active file system to diverge (C is overwritten with C', A is deleted, E is added). Snapshots and the active file system share unchanged blocks. When it is time for an Update Transfer, SnapMirror takes a new incremental reference snapshot and then compares the snapshot active maps according to the rules in the text to determine which blocks need to be transferred to the destination. After a successful update, SnapMirror deletes the old base snapshot and the incremental becomes the new base.

At the end of each transfer the fsinfo block is updated, which brings the user's view of the file system up to date with the latest transfer. The base reference snapshot is deleted from the source, and the incremental reference snapshot becomes the new base. Essentially, the file system updates are written into unused blocks on the destination and then the fsinfo block is updated to refer to this new version of the file system with is already in place.

3.2.3 Disaster Recovery and Aborted Transfers

Because a new fsinfo block (the root of the file system tree structure) is not written until all blocks are transferred, SnapMirror guarantees a consistent file system on the mirror at any time. The destination file system is accessible in a read-only state throughout the whole SnapMirror process. At any point, its active file system replicates the active map and fsinfo block of the last reference snapshot generated by the source. Should a disaster occur, the destination can be brought immediately into a writable state.

The destination can abandon any transfer in progress in response to a failure at the source end or a network partition. The mirror is left in the same state as it was before the transfer started, since the new fsinfo block is never written. Because all data is consistent with the last completed round of transfers, the mirror can be reestablished when both systems are available again by finding the most recent common SnapMirror snapshot on both systems, and using that as the base reference snapshot.

3.2.4 Update Scheduling and Transfer Rate Throttling

The destination file server controls the frequency of update through how often it requests a transfer from the source. System administrators set the frequency through a cron-like schedule. If a transfer is in progress when another scheduled time has been reached, the next transfer will start when the current transfer is complete. SnapMirror also allows the system administrator to throttle the rate at which a transfer is done. This prevents a flood of data transfers from overwhelming the disks, CPU, or network during an update.

3.3 SnapMirror Advantages and Limitations

SnapMirror meets the emerging requirements for data recovery by using asynchrony and combining file system knowledge with block-level transfers.

Because the mirror is on-line and in a consistent state at all phases of the relationship, the data is available during the mirrored relationship in a read-only capacity. Clients of the destination file system will see new updates atomically appear. If they prefer to access a stable image of the data, they can access one of the snapshots on the destination. The mirror can be brought into a writable state immediately, making disaster recovery extremely quick.

The schedule-based updates mean that SnapMirror has as much or as little impact on operations as the system administrator allows. The tunable lag also means that the administrator controls how up to date the mirror is. Under most loads, SnapMirror can reasonably transmit to the mirror many times in one hour.

SnapMirror works over a TCP/IP connection that uses standard network links. Thus, it allows for maximum flexibility in locating the source and destination filers and in the network connecting them.

The nature of SnapMirror gives it advantages over traditional mirroring approaches. With respect to synchronous mirroring, SnapMirror reduces the amount of data transferred, since blocks that have been allocated and de-allocated between updates are not transferred. And because SnapMirror uses snapshots to preserve image data, the source can service requests during a transfer. Further, updates at the source never block waiting for a transfer to the remote mirror.

The time required for a SnapMirror update is largely dependent on the amount of new data since the last update and, to some extent, on file system size. The worst-case scenario is where all data is read from and re-written to the file system between updates. In that case, SnapMirror will have to transfer all file blocks. File system size plays a part in SnapMirror performance due to the time it takes to read through the active map files (which increases as the number of total blocks increase).

Another drawback of SnapMirror is that its snapshots reduce the amount of free space in the file system. On systems with a low rate of change, this is fine, since unchanged blocks are shared between the active file system and the snapshot. Higher rates of change mean that SnapMirror reference snapshots tie up more blocks.

By design, SnapMirror only works for whole volumes as it is dependent on active map files for updates. Smaller mirror granularity could only be achieved through modifications to the file system, or through a slower, logical-level approach.

4 Data Reduction through Asynchrony

An important premise of asynchronous mirroring is that periodic updates will transfer less data than synchronous updates. Over time, many file operations become moot either because the data is overwritten or deleted. Periodic updates don't need to transfer any deleted data and only need to transfer the most recent version of an overwritten block. Essentially, periodic updates use the primary volume as a giant write cache and it has long been known that write caches can reduce I/O traffic [Ousterhout85, Baker91, Kistler93]. Still at question, though, is how much asynchrony can reduce mirror data traffic for modern file server workloads over the extended intervals of interest to asynchronous mirroring.

To answer these questions, we traced a number of file servers at Network Appliance and analyzed the traces to determine how much asynchronous mirroring would reduce data transfers as a function of update period. We also analyzed the traces to determine the importance of using the file system's active map to avoid transferring deleted blocks for WAFL as an example of no-overwrite file systems.

4.1 Tracing environment

We gathered 24 hours of traces from twelve separate file systems or volumes on four different NetApp file servers. As shown in Table 1, these file systems varied in size from 16 GB to 580 GB, and the data written over the day ranged from 1 GB to 140 GB. The blocks counted in the table are each 4 KB in size. The systems stored data from: internal web pages, engineers' home directories, kernel builds, a bug database, the source repository, core dumps, and technical publications.

In synchronous or semi-synchronous mirroring all disk writes must go to both the local and remote mirror. To determine how many blocks asynchronous mirroring would need to transfer at the end of any particular update interval, we examined the trace records and recorded in a large bit map which blocks were written (allocated) during the interval. We cleared the dirty bit whenever the block was deallocated. In an asynchronous mirroring system, this is equivalent to computing the logical-AND of the dirty map with the file system's active map and only transferring those blocks which are both dirty and still part of the active file system.

4.2 Results

Figure 2 plots the blocks that would be transferred by SnapMirror as a percentage of the blocks that would

File System Name	Filer	Description	Size (GB)	Used (GB)	Blocks Written (1000's)	Written Deleted (%)
Build1	Ecco	Source tree build space	100	68	7757	69
Cores1		Core dump storage	100	72	319	85
Bench		Benchmark scratch space and results repository	87	56	512	91
Pubs		Technical Publications	32	16	262	59
Users1		Engineering home directories	350	292	10803	78
Bug	Maglite	Bug tracking database	16	11	1465	98
Cores2		Core dump storage	550	400	11956	76
Source		Source control repository	50	36	3288	70
Cores3	Makita	Core dump storage	255	151	1582	77
Users2		Engineering home directories and corporate intranet site	580	470	13752	53
Build2	Ronco	Source tree build space	320	271	34779	80
Users3		Engineering home directories	380	323	15103	85

Table 1. Summary data for the traced file systems. We collected 24 hours of traces of block allocations (which in WAFL are the equivalent of disk writes) and de-allocations in the 12 file systems listed in the table. The 'Blocks Written' is the total number of blocks written and indicates the number of blocks that a synchronous block-level mirror would have to transfer. The 'Written Deleted' column shows the percentage of the written blocks which were overwritten or deleted. This represents the potential reduction in blocks transferred to an asynchronous mirror which is updated only once at the end of the 24-hour period. The reduction ranges from 52% to 98% and averages about 78%.

be transferred by a synchronous mirror as a function of the update period: 1 minute, 5 minutes, 15 minutes, 30 minutes, 1 hour, 6 hours, 12 hours, and 24 hours. We found that even an update interval of only 1 minute reduces the data transferred by at least 10% and by over 20% on all but one of the file systems. These results are consistent with those reported for a 30 second write-caching interval in earlier tracing studies [Ousterhout85, Baker91]. Moving to 15 minute intervals enabled asynchronous mirroring to reduce data transfers by 30% to 80% or over 50% on average. The marginal benefit of increasing the update period diminishes beyond 60 minutes. Nevertheless, extending the update period all the way to 24 hours reduces the data transferred to between 53% and 98% – over 75% on average. This represents a 50% reduction compared to an update interval of 15 minutes. Clearly, the benefits of asynchronous mirroring can be substantial.

As mentioned above, we performed the equivalent of a logical-AND of the dirty map with the file system's active map to avoid replicating deleted data. How important is this step? In conventional write-in-place file systems such as the Berkeley FFS [McKusick84], we do not expect this last step to be critical. File overwrites would repeatedly dirty the same block which would eventually only need to be transferred once. Further, because the file allocation policies of these file system often result to the reallocation of blocks recently freed, even file deletions and creations end up reusing the same set of blocks.

The situation is very different for no-overwrite file systems such as LFS and WAFL. These systems tend to avoid reusing blocks for either overwrites or new creates. Figure 3 plots the blocks transferred by SnapMirror, which takes advantage of the file system's active map to avoid transferring deallocated blocks, and an asynchronous block-level mirror, which does not, as a percentage of the blocks transferred by the synchronous mirror for a selection of the file systems. Because, most of the file systems in the study had enough free space in them to absorb all of the data writes during the day, there were essentially no block reallocations during the course of the day. For these file systems, the data reduction benefits of asynchrony would be completely lost if SnapMirror were not able to take advantage of the active maps. In the figure, the 'all other, include deallocated' line represents these results. There were two exceptions, however. Build2 wrote about 135 GB of data while the volume had only about 50 GB of free space and Source wrote about

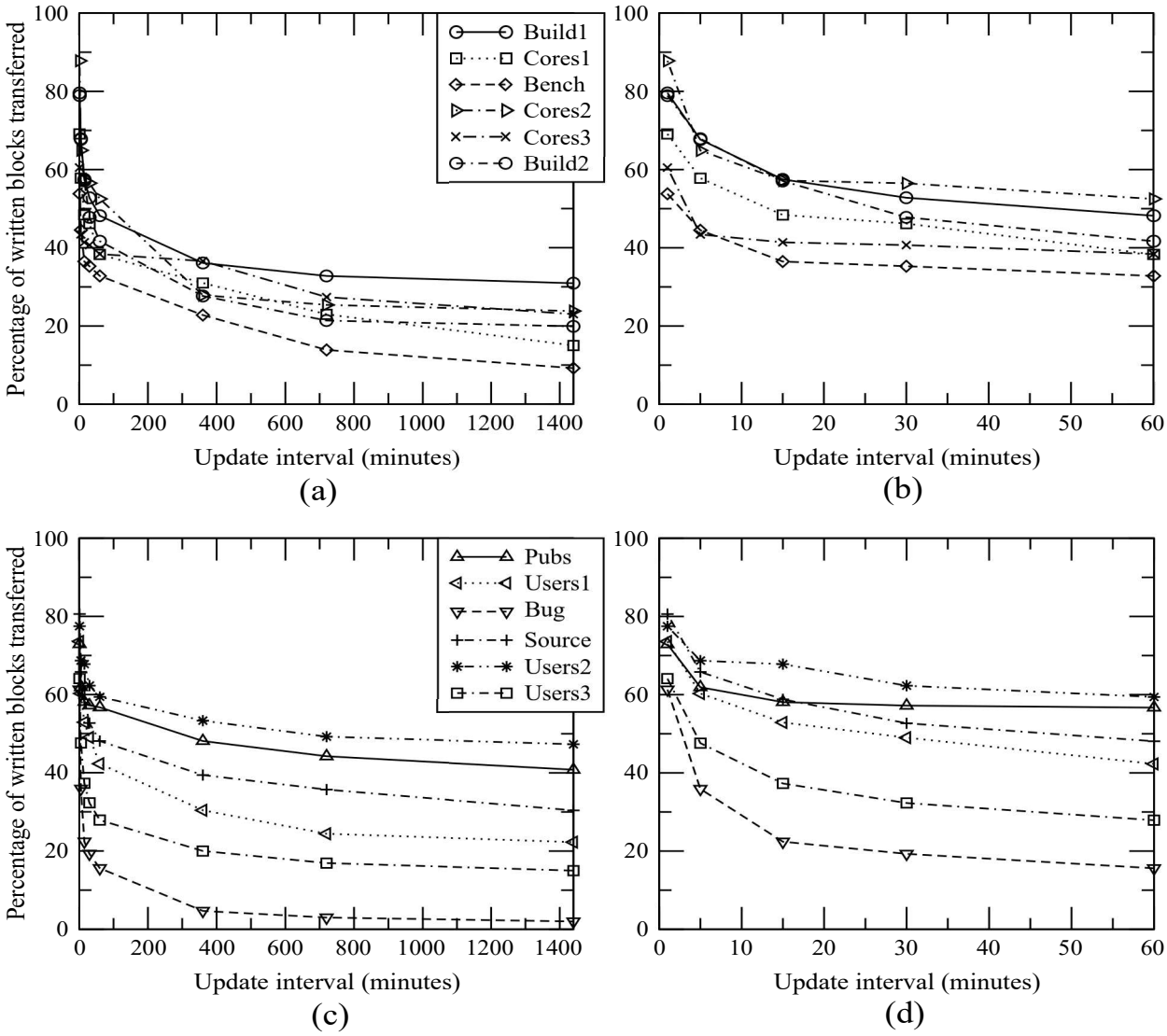

Figure 2. Percentage of written blocks transferred by SnapMirror vs. update interval. These graphs show, for each of the 12 traced systems, the percentage of written blocks that SnapMirror would transfer to the destination mirror as a function of mirror update period. Because the number of traces is large, the results are split into upper and lower pairs of graphs. The left graph in each pair (a and c) show the full range of intervals from 1 minute to 1440 minutes (24 hours). The right graphs in each pair (b and d) expand the region from 1 to 60 minutes. The graphs show that most of the reduction in data transferred occurs with an update period of as little as 15 minutes, although substantial additional reductions are possible as the interval is increased to an hour or more.

13 GB of data with only 14 GB of free space. Inevitably, in these file systems, there was some block reuse as shown in the figure. Even in these two cases, however, the use of the active map was highly beneficial. Successful asynchronous mirroring of no-overwrite file systems requires the use of the file system's active map or equivalent information.

An alternative to the block-level mirroring (with or without the active map) discussed in this section is logical or file-system level mirroring. This is the topic of the next section.

5 SnapMirror vs. Asynchronous Logical Mirroring

The UNIX dump and restore utilities can be used to implement an asynchronous logical mirror. Dump works above the operating system to identify files which need to be backed up. When performing an incremental, the utility only writes to tape the files which have been created or modified since the last incremental dump. Re-

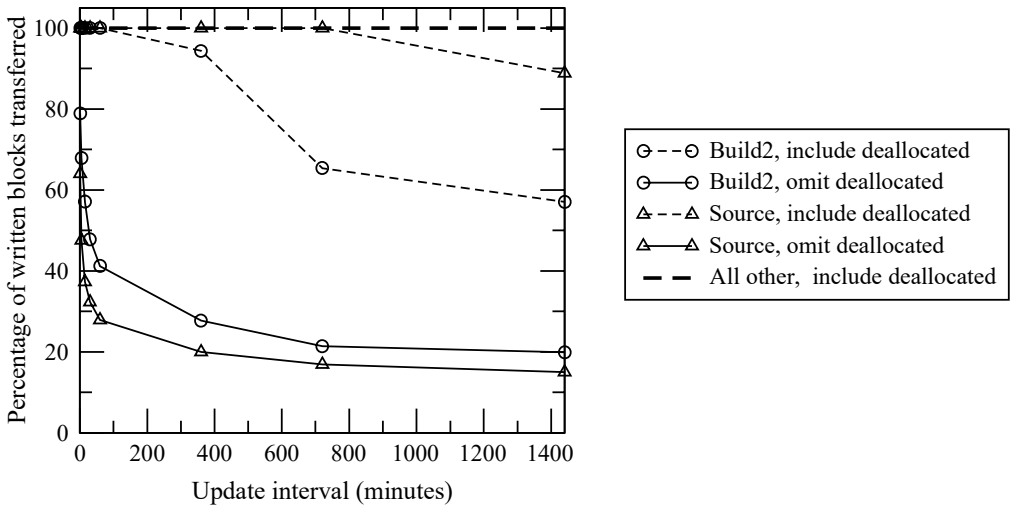

Figure 3. Percentage of written blocks transferred with and without use of the active map to filter out deallocated blocks. Successful asynchronous mirroring of a no-overwrite file system such as LFS or WAFL depends on the file system's active map to filter out deallocated blocks and achieve reductions in block transfers. Without the use of the active map, only 2 of the 12 measured systems, would see any transfer reductions.

File System Name	Size (GB)	Used (GB)		Files		System	Data transferred (GB)	Time (sec.)	Rate (MB/s)
		Base	End	Base	End				
Users4	96	63	65	1001131	1054917	SnapMirror	2.1	140	15.4
						logical	4.0	493	8.3
Users5	192	135	150	5297016	6423984	SnapMirror	15.3	797	19.7
						logical	25.2	7200	3.6

Table 2. Logical replication vs. SnapMirror incremental update performance. We measured incremental performance of SnapMirror and logical replication on two separate data sets. Since SnapMirror sends only changed blocks, it transfers at least 39% less data than logical mirroring.

store reads such incremental dumps and recreates the dumped file system. If dump's data stream is piped directly to a restore instead of a tape, the utilities effectively copy the contents of one file system to another. An asynchronous mirroring facility could periodically run an incremental dump and pipe the output to a restore running on the destination. The following set of experiments compares this approach to SnapMirror.

5.1 Experimental Setup

To implement the logical mirroring mechanism, we took advantage of the fact that Network Appliance filers include dump and restore utilities to support backup and the Network Data Management Protocol (NDMP) copy command. The command enables direct data copies from one filer to another without going through the issuing workstation. For these experiments, we configured dump to send its data over the network to a restore process on another filer. Because this code and data path are included in a shipping product, they are reasonably well tuned and the comparison to SnapMirror is fair.

To compare logical mirroring to SnapMirror, we first established and populated a mirror between two filers in the lab. We then added data to the source side of the mirror and measured the performance of the two mechanisms as they transferred the new data to the destination file system. We did this twice with two sets of data on two different sized volumes. For data, we used production full and incremental dumps of some home directory volumes. Table 2 shows the volumes and their sizes. The full dump provided the base file system. The incremental provided the new data.

We used a modified version of restore to load the incremental data into the source volume. The standard restore utility always completely overwrites files which have been updated; it never updates only the changed blocks. Had we used the standard restore, SnapMirror and the logical mirroring would both have transferred whole files. Instead, when a file on the incremental tape matched an existing file in both name and inode number, the modified restore did a block by block comparison of the new and existing files and only wrote changed blocks into the source volume. The logical mirroring mechanism, which was essentially the standard dump utility, still transferred whole files, but SnapMirror was able to take advantage of the fact that it could detect which blocks had been rewritten and thus transfer less data.

For hardware, we used two Network Appliance F760 filers directly connected via Intel GbE. Each utilized an Alpha 21164 processor running at 600 MHz, with 1024 MB of RAM plus 32 MB non-volatile write cache. For the tests run on Users4, each filer was configured with 7 FibreChannel-attached disks (18 GB, 10k rpm) on one arbitrated loop. For the tests run on Users5, each filer was configured with 14 FibreChannel-attached disks on one arbitrated loop. Each group of 7 disks was set up with 6 data disks and 1 RAID4 parity disk. All tests were run in a lab with no external load.

5.2 Results

The results for the two runs are summarized in Table 2 and Figure 4. Note that in the figure, the two sets of runs are not rendered to the same scale. The 'data scan' value for logical mirroring represents the time spent walking the directory structure to find new data. For SnapMirror, 'data scan' represents the time spent scanning the active map files. This time is essentially independent of the number of files or the amount new data but is instead a function of volume size. The number was determined by performing a null transfer on a volume of this size.

The most obvious result is that logical mirroring takes respectively 3.5 and 9.0 times longer than SnapMirror to update the remote mirror. This difference is due both to the time to scan for new data and the efficiency of the data transfers themselves. When scanning for changes, it is much more efficient to scan the active map files than to walk the directory structure. When transferring data, it is much more efficient to read and write blocks sequentially than to go through the file system code reading and writing logical blocks.

Beyond data transfer efficiency, SnapMirror is able to transfer respectively 48% and 39% fewer blocks than the logical mirror. These results show that savings from transferring only changed blocks can be substantial compared to whole file transfer.

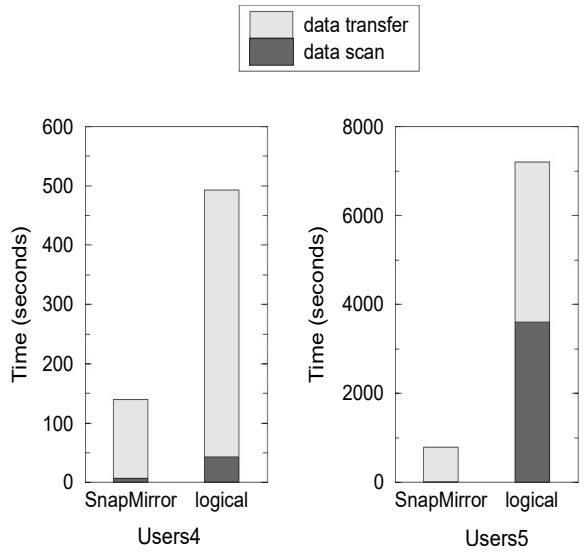

Figure 4. Logical replication vs. SnapMirror incremental update times. By avoiding directory and inode scans, SnapMirror's data scan scales much better than that of logical replication. Note: tests are not rendered on the same scale)

6 SnapMirror on a loaded system

To assess the performance impact on a loaded system of running SnapMirror, we ran some tests very much like the SPEC SFS97 [SPEC97] benchmark for NFS file servers.

In the tests, data was loaded onto the server and a number of clients submitted NFS requests at a specified aggregate rate or offered load. For these experiments, there were 48 client processes running on 6 client machines. The client machines were 167 MHz Ultra-1 Sun workstations running Solaris 2.5.1, connected to the server via switched 100bT ethernet to an ethernet NIC on the server. The server was a Network Appliance F760 filer with the same characteristics as the filers in Section 5.1. The filer had 21 disks configured in a 320 GB volume. The data was being replicated to a remote filer.

6.1 Results

After loading data onto the filer and synchronizing the mirrors, we set the SnapMirror update period to the desired value and measured the request response time over an interval of 60 minutes. Table 3 and Figure 5 report the results for an offered load of 4500 and 6000 NFS operations per second. In the table, SnapMirror data is the total data transferred to the mirror over the 60 minute

Load (ops/s)	Update Interval	CPU busy	Disk busy	SnapMirror data (MB)
4500	base	66%	34%	0
	1 min.	93%	50%	12817
	15 min.	74%	43%	6338
	30 min.	69%	40%	2505
6000	base	87%	54%	0
	1 min.	99%	67%	13965
	15 min.	94%	62%	8071
	30 min.	91%	60%	3266

Table 3. SnapMirror Update Interval Impact on System Resources. During SFS-like loads, resource consumption diminishes dramatically when SnapMirror update intervals increase. Note: base represents performance when SnapMirror is turned off.

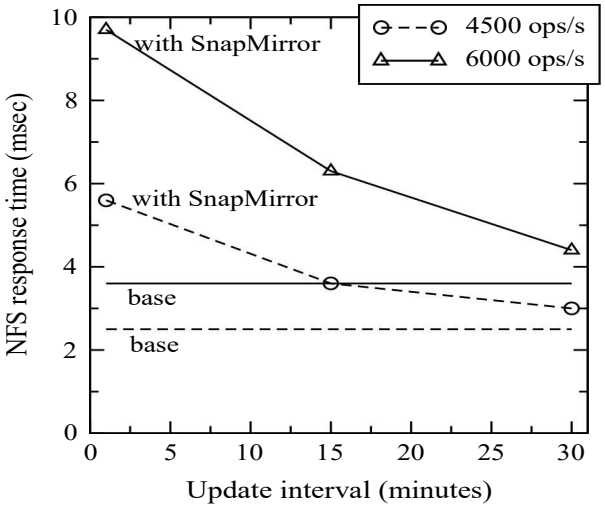

Figure 5. SnapMirror Update Interval vs. NFS response time. We measured the effect of SnapMirror on the NFS response time of SFS-like loads. By increasing SnapMirror update intervals, the penalty approaches a mere 22%.

run.

Even with the SnapMirror update period set to only one minute, the filer is able to sustain a high throughput of NFS operations. However, the extra CPU and disk load increases response time by a factor of two to over three depending on load.

Increasing the SnapMirror update period to 30 minutes decreases the impact on response time to only about 22% even when the system is heavily loaded with 6000 ops/sec. This reduction comes from two major effects. First, each SnapMirror update requires a new snapshot and a scan of the active map files. With less frequent updates, the impact of these fixed costs is spread over a much greater period. Second, as the update period increases, the amount of data that needs to be transferred to the destination per unit time decreases. Consequently SnapMirror reads as a percentage of the total load decreases.

7 Conclusion

Current techniques for disaster recovery offer data managers a stark choice. Waiting for a recovery from tape can cost time, millions of dollars, and, due to the age of the backup, can result in the loss of hours of data. Failover to a remote synchronous mirror solves these problems, but does so at a high cost in both server performance and networking infrastructure.

In this paper, we presented SnapMirror, an asynchronous mirroring package available on Network Appliance filers. SnapMirror periodically updates an online mirror. It provides the rapid recovery of synchronous remote mirroring but with greater flexibility and control in maintaining the mirror. With SnapMirror, data managers can choose to update the mirror at an interval of their choice. SnapMirror allows the user to strike the proper balance between data currency on one hand and performance and cost on the other.

By updating the mirror periodically, SnapMirror can transfer much less data than would a synchronous mirror. In this paper, we used traces of 12 production file systems to show that by updating the mirror every 15 minutes, instead of synchronously, SnapMirror can reduce data transfers by 30% to 80%, or 50% on average. Updating every hour reduces transfers an average of 58%. Daily updates reduce transfers by over 75%.

SnapMirror benefits from the WAFL file system's ability to take consistent snapshots both to ensure the consistency of the remote mirror and to identify changed blocks. It also uses the file system's active map to avoid transferring deallocated blocks. Trace analysis showed that this last optimization is critically important for no-overwrite file systems such as WAFL and LFS. Of the 12 traces analyzed, 10 would have seen no transfer reductions even with only update after 24 hours.

SnapMirror also leverages block level behavior to solve performance problems that challenge logical-level mirrors. In experiments comparing SnapMirror to dump-based logical-level asynchronous mirroring, we found that using block-level file system knowledge reduced the time to identify new or changed blocks by as much as two orders of magnitude. By avoiding a walk of directory and inode structures, SnapMirror was able to detect changed data significantly more quickly than the logical

schemes. Furthermore, transferring only changed blocks, rather than full files, reduced the data transfers by over 40%. Asynchronous mirror updates can run much more frequently when it takes a short time to identify blocks for transfer, and only the necessary blocks are updated. Thus, SnapMirror's use of file system knowledge at a block level greatly expands its utility.

SnapMirror fills the void between tape-based disaster recovery and synchronous remote mirroring. It demonstrates the benefit of combining block-level and logical-level mirroring techniques. It gives system administrators the flexibility they need to meet their varied data protection requirements at a reasonable cost.

8 Acknowledgments

The authors wish to thank Steve Gold, Norm Hutchinson, Guy Harris, Sean O'Malley and Lara Izlan for their generous contributions. We also wish to thank the reviewers and our shepherd, Roger Haskin, for their helpful suggestions.

9 References

[Chutani92] S. Chutani, et. al. *The Episode File System*. Proceedings of the Winter 1992 USENIX Conference, San Francisco, CA, January 1992. pp. 43-60.

[Baker91] M. Baker, J. Hartman, M. Kupfer, L. Shirriff, J. Ousterhout. *Measurements of a Distributed File System*. Proceedings of the 13th Symposium on Operating System Principles. October 1991. pp. 198-212.

[EMC] EMC Symmetrix® Remote Data Facility. http://www.emc.com/.

[Growchowski96] E.G. Grochowski, R.F. Hoyt. *Future Trends in Hard Disk Drives*. IEEE Transactions on Magnetics, V32. May 1996. pp. 1850-1854.

[Hitz94] D. Hitz, J. Lau, M.A. Malcolm. *File System Design* for *an NFS File Server Appliance*. Proceedings USENIX Winter 1994 Conference. pp. 235-246. http://www.netapp.com/tech_library/3002.html

[Hutchinson99] N.C. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, S. O'Malley. *Logical vs. Physical File System Backup*. Proceedings Third Symposium on Operating System Design and Implementation. February 1999.

[Kistler92] J.J. Kistler, M. Satyanarayanan. *Disconnected Operation in the Coda File System*. ACM Transactions on Computer Systems, 10(1). February 1992.

[Kistler93] J.J. Kistler, *Disconnected Operation in a Distributed File System*. Technical Report CMU-CS-93-156. School of Computer Science, Carnegie Mellon University, 1993. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/coda/Web/docs-coda.html

[McKusick84] M.K. McKusick, W.J. Joy, S.J. Leffler, R.S. Fabry. *A Fast File System for UNIX*. Transactions on Computer Systems 2,3. August 1984. pp. 181-197

[Ousterhout85] J.K. Ousterhout, H. Da Costa, D. Harrison, J.A. Kunze, M. Kupfer, J.G. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," *Proceedings of the 10th Symposium on Operating Systems Principles* (SOSP), Orcas Island, WA, December, 1985, pp. 15-24.

[Rosenblum92] M. Rosenblum, J.K. Osterhout. *The Design and Implementation of a Log-structured File System*. ACM Transactions on Computer Systems, Vol.10, No.1 (Feb. 1992), pp. 26-52.

[SPEC97] The Standard Performance Evaluation Corporation. SPEC SFS97 Benchmark. http://www.spec.org/osg/sfs97/.

[Storage] Storage Computer Corporation Omniforce© software. http://www.storage.com/

[Tridgell96] A. Tridgell, P. Mackerras. *The rsync algorithm*. Department of Computer Science Australian National University. TR-CS-96-05.