The Quick Migration of File Servers

Keiichi Matsuzawa
Hitachi, Ltd.
Yokohama, Japan
keiichi.matsuzawa.kd@hitachi.com

Mitsuo Hayasaka
Hitachi, Ltd.
Yokohama, Japan
mitsuo.hayasaka.hu@hitachi.com

Takahiro Shinagawa
The University of Tokyo
Tokyo, Japan
shina@ecc.u-tokyo.ac.jp

ABSTRACT

Upgrading file servers is indispensable for improving the performance, reducing the possibility of failures, and reducing the power consumption. To upgrade file servers, files must be migrated from the old to new servers, which poses three challenges: reducing the downtime during migration, reducing the migration overhead, and supporting the migration between heterogeneous servers. Existing technologies are difficult to achieve all of the three challenges. We propose a quick file migration scheme for heterogeneous servers. To reduce the downtime, we exploit the post-copy approach and introduce on-demand migration that allows file access before completing the migration. To reduce the overhead, we introduce background migration that migrates files as soon as possible without affecting the performance and incurs no overhead after the migration. To support heterogeneity, we introduce stub-based file management that requires no internal states of the old server. We implemented our scheme for Linux and supported the NFS and SMB protocols. The experimental results depict that the downtime was a maximum of 23 s in a 4-level 1000-file directory and the migration time was 70 min in NFS and 204 min in SMB with 242 GiB of data.

CCS CONCEPTS

• **Information systems → Storage replication**;

KEYWORDS

File server, migration, post-copy

1 INTRODUCTION

Computer systems tend to become obsolete rapidly. Therefore, upgrading them in 3-5 years is desirable to improve the performance, reduce the possibility of failures, and reduce the energy consumption [15, 26]. This is also applicable in case of file servers. Upgrading file servers is one of the most popular motivations for purchasing new file servers [30]. Therefore, smooth migration of file servers is one of the major concerns for system administrators [29].

Upgrading file servers poses three challenges. The first is to reduce the downtime during migration from the old to new servers. The file servers could approximately store 400 TiB of data on an average. Therefore, the migration of such massive amounts of data may require a few hours to several days [30]. Terminating file services throughout the file migration process significantly impairs the user experience. Since the file access operation typically times out in 30-120 s [16, 22], the downtime should be shorter than that. The second is to reduce the overhead during file migration. Since file migration consumes the server resources, it affects the user's file access performance. Therefore, file migration processes should be kept as short as possible. Additionally, there should be no overhead after the completion of the migration process because performance improvement is the primary objective of upgrading file servers.

The third is to support migration between heterogeneous file servers. Although production file servers will provide migration tools between homogeneous servers, there are various cases where products and vendors of the old and new file servers are different. For example, a vendor may provide a more attractive product than the vendor of the old server may. The vendor of the old server may have withdrawn from the file server market. The old server may be a personal file server that does not have a corresponding enterprise file server. Therefore, upgrading file servers may introduce a new server with an internal architecture different from that of the old one. Furthermore, since access to the internal of production servers may be limited, modification or running hand-made tools in the old server may be impossible. Therefore, the file migration scheme must be independent of the internal architecture and functions of the old file server.

A classical but extensively used method is pre-copy migration with a file synchronization tool like rsync [31]. In this method, the file tree is repeatedly copied from the old to new servers until the number of updated files becomes sufficiently small. Therefore, it will take a long time to complete the overall file migration process on busy servers and incur

SYSTOR '18, June 4–7, 2018, HAIFA, Israel
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5849-1/18/06...$15.00
https://doi.org/10.1145/3211890.3211894

continuous overhead on file server access due to the iterative copy. Moreover, at the final stage, the access to the old server must be terminated, and the remaining updated files must be copied to the new server, which requires a considerably long downtime, e.g., up to several tens of hours in case of busy servers. This downtime could have a critical impact on the business in a company. Post-copy migration is used in volume-based storage systems [18, 24, 35]. However, they assume that the file systems of the old and new storage are identical. Systems with global namespace (GNS) could bundle up multiple servers into a single namespace and support transparent file migration between the servers [1, 34, 36]. However, they require a virtualization layer to convert the logical file name into the physical location, which incurs a constant overhead during file access.

In this paper, we propose a file migration scheme between heterogeneous servers for the quick upgrading of the file servers. To reduce the downtime during file migration, we exploit the post-copy approach. In this approach, the file server that was accessed by the clients is immediately switched from the old server to the new one. To allow access to files in the new server, we introduce *on-demand migration* that only migrates the existence of a file in advance and copies the remaining data and metadata at a later instance when they are initially accessed. This technique allows the clients to access files before completing the overall migration process.

To reduce the migration overhead, we introduce *background migration* that periodically traverses the directory tree and migrates files that have not yet been migrated in the background. By conducting migration during idle time, we minimize the performance impact on client file access while reducing the duration of the overall file migration process. After the completion of the migration process, the clients can directly access the new file server without overhead.

To support the heterogeneous servers, we introduce *stub-based file management* that manages the intermediate state of file migration only on the new file server. Since file migration only requires the standard file access interface to the old server, it is not dependent on the internal structure of the old server. Although each of the introduced techniques may not be very novel, a combination of these techniques, which is effective to achieve the three challenges in upgrading file servers, has not been previously proposed.

We have implemented our scheme using a Linux kernel and user-space programs. Our implementation supports both NFS and SMB and can migrate files from any servers that support these protocols. The implementation is mature and actually used in a production system. The experimental evaluation illustrates that the maximum downtime was 23 s in NFS and 7 s in SMB in case of a 4-level 1000-file directory and that the migration time was 70 min using the NFS protocol and 204 min using the SMB protocol with 242 GiB data.

The contributions of this paper are as follows:

- This paper identifies three challenges to upgrade the file servers quickly and proposes a combination of techniques that can complete these challenges.
- This paper presents a solid implementation that supports the NFS and SMB protocols in a Linux kernel, which is mature enough to be commercialized.
- This paper demonstrates, by experimentation, that the implementation satisfies the three challenges.

2 RELATED WORK

2.1 Pre-Copy Migration

Pre-copy migration is a popular method that copies files from the old to new servers beforehand iteratively and then switches the file servers after copying all the files. Most vendors suggest the usage of the pre-copy migration with a tool such as rsync [31] and Robocopy [21]. However, this method depicts two drawbacks. The first is the long downtime. This method has to terminate both the old and new servers while switching the servers to copy the files updated after performing the last copy operation. In busy servers, the sizes of the updated files are large, and copying all of them takes a long time. Furthermore, detecting updated files from outside of the file server takes a long time. In our estimation, traversing the file tree to check time-stamps via NFS in an average file server with 400 TiB data will take approximately 28 hours. This could be unacceptable in a company with time-critical business. The second is the long migration time. Since the file copy process is repeatedly performed, the busy files will be copied multiple times. This will increase the total migration time and continuously incur file access overhead.

2.2 Post-Copy Migration

In contrast to pre-copy migration, post-copy migration defers the data copy operation and switches the access servers first. When the new server receives an access request, the data are copied from the old to new servers. This concept is extensively used in memory replication such as the copy-on-write mapping between processes in UNIX-like OSs [28] and the memory page copies in a virtual machine (VM) live migration [13]. ImageStreaming [24] applies the post-copy concept to storage volumes of VMs during live migration. However, to the best of our knowledge, the application of the post-copy approach to perform file server migration has not yet been proposed. Unfortunately, simply applying the post-copy approach to file server migration compromises the performance of the file access operation because migrating files after receiving each access request increases the response time, and the total access efficiency could be further decreased due to the small random I/O events.

A natural extension of post-copy is to combine with pre-copy. For example, some file systems provide point-in-time snapshot [2]. In this scheme, by transferring the snapshot data in pre-copy basis and retrieving differential data after the transfer in post-copy basis, both better transfer throughput and low downtime are achieved. However, since we assume a migration scheme that is independent of the internal architecture of the old server, integrating the pre-copy method will incorporate the cost of detecting updated files in the old server, as mentioned in Section 2.1. As a result, the downtime and total migration time could be long.

2.3 Replication Using the GNS

GNS is a virtualization layer that aggregates multiple servers into a single namespace [1]. This layer manages the mapping between a file in the virtualized namespace and the actual file in a file server. This layer accepts the requests from clients and redirects the requests to the actual file server; therefore, the clients do not need to be aware of the file location.

X-NAS [36] and NAS Switch [34] support the file replication based on GNS. Some commercial products also provide similar solutions [5, 8, 10]. In these systems, the virtualization layer replicates the files in the background and redirects the access requests to the appropriate server depending on the progress of the replication. Clients can keep accessing the files transparently during the replication process.

Parallel NFS (pNFS) [27] is an extension of the NFS protocol that supports out-of-band data transfer. A metadata server provides the layout of file data that includes information about the file location, and the clients directly send the access requests to the target server based on the location information. pNFS can reduce the overhead of data access; however, the overhead of the metadata access remains.

GNS-based solutions require the old servers to operate under the GNS management in advance. Furthermore, all the access requests must be directed through the virtualization layer, which increases the overhead of access redirection.

2.4 Replication Using Archive Data

DAB [17], Cumulus [33], and Panache [9] encapsulate data and metadata into an archive file and share the file among various file servers for replication. DAB and Cumulus aim to create a backup and restoration point among the homogeneous servers. Panache shares the updated data among the servers to cache remote servers in widely distributed environment. Windows Server Migration Tool [20] is a feature provided by the Windows server that allows for the migration of server roles, features, and data. It exports the server configurations and data into a single file, and the target server imports the file to inherit the configurations and data. These methods assume that servers can recognize the archived file format. Therefore, they are not applicable to perform heterogeneous file server migration.

DAB and Panache support on-demand metadata restoration in a manner that is similar to that used in our technique. However, our technique obtains the metadata per file, whereas DAB obtains the metadata per directory. Panache supports orphan inodes to defer the replication of metadata. However, Panache is essentially a global caching system and does not assume that the cached data is invariant at the remote site. Our technique assumes that the replicated data and file hierarchy may be updated during migration.

2.5 Volume-Level Replication

SnapMirror [35] replicates volume data and transfers incremental updates. The source server keeps accepting requests from clients and redirects them to the target server during the replication process. Storage vMotion [18] and ImageStreaming [24] provide live VM migration with volumes. They replicate the volume images in a transparent manner between storage devices. Unfortunately, volume-level replication is not appropriate for upgrading the file servers because the old and new file servers may be heterogeneous.

3 DESIGN

We exploit the post-copy approach to migrate file servers. In this approach, file servers are switched from the old to new servers before completing the migration of all files. To achieve this, we use three techniques: on-demand migration, background migration, and stub-based file management. We will explain each of them. Note that on-demand migration is dependent on stub-based file management and background migration is dependent on on-demand migration.

3.1 On-Demand Migration

On-demand migration is a technique to migrate the files when they are accessed for the very first time. When a file is accessed, the new file server copies only a part of the file information, i.e., the part of metadata that is required to respond to the access request. This allows the new server to accept requests before completing the file migration and to respond to the requests in a short time. When the file data are accessed, the new server copies the data from the old server. By copying the data on demand, we can significantly reduce the downtime during file server migration.

To manage the process of on-demand migration, we exploit the concept of a stub that was used in hierarchical storage management. A stub is an intermediate state of files or directories. A stub file is similar to a regular file; however, it does not contain the entire file data. Instead, a stub file contains the location information of the file data, i.e., the path in the old file server. When a client attempts to access the file data, the new file server retrieves the data from the old server, returns the data to the client, and stores the data. When the entire data of a file is copied, it indicates that the migration of the file has completed successfully. Directories also take intermediate states. A directory in that state does not stores any child directory entries, and access to the directory invokes the retrieval of directory entries and creation of corresponding intermediate-state inodes. Further details of stub management are presented in Section 3.3.

3.2 Background Migration

With on-demand migration, only the files accessed from the clients are migrated. Hence, file data that have not been accessed are never migrated. To reduce the response time on the first access and the total migration time, we introduce background migration that actively migrates the un-accessed files. Background migration is based on on-demand migration. We use a crawling program that traverses the directory tree and accesses the data of the files in the new server. By accessing the file data, the on-demand migration mechanism copies the file data from the old server. This process consumes both the CPU and storage resources. Therefore, to avoid the influence on the user's file access performance, we assume that the crawling program runs when the file server is not busy, e.g., at midnight or during weekends.

When the total amount of file data is large, the migration of all the files will require a long time. In this case, the crawling program may not be completed in one night or weekend. Therefore, we will have to run the crawling program intermittently. However, the clients can still access the files and directories while the crawling program is suspended. Thus, the directory tree may be different before and after the suspension of the crawling program. To maintain consistency, the crawling program must handle this situation.

One method to achieve this is to perform progress management that saves the current state of the crawling. However, maintaining consistency by saving the progress while allowing the clients to update is complicated because the clients could significantly alter the directory structure using the delete and rename operations. To avoid the risk of migration leakage, the crawling program restarts the directory traversal from the root every time it resumes operation. The file system has a counter that counts the number of files that were not migrated yet. This counter is incremented when a new stub file is created and is decremented when the file is completely migrated. If a client deletes a stub file, the counter is also decremented. The crawling program verifies this counter every time it starts; if the counter is zero, it indicates that the migration has completed; otherwise, it restarts the traversal of the directory from the root directory.

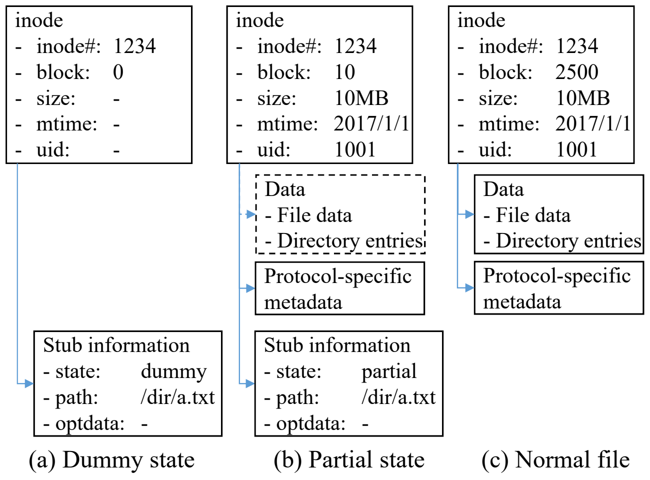


Figure 1: Internal structures of a file.

This approach may consume server resources to access the same files repeatedly. Alternatively, we can always track the changes in the directory tree and notify the crawling program about it. However, it will incur an overhead on file access by the clients, which will degrade the performance during file server migration. The approach we adopted depicts the benefit of incurring no overhead on the client's file access request if the file has already been migrated.

This approach tolerates power interruption and network failures in the migration process. The state transition of files is one way, and the counter is incremented and decremented only once for each file. Therefore, even if an unexpected reboot occurs, traversing the file tree and verifying the file statuses can restore the counter value.

3.3 Stub-Based File Management

To achieve on-demand migration, we use stub-based file management. When the file servers are switched, the new server creates a stub root directory, which corresponds to the root directory of the old server. A stub file is a file whose existence can be observed but the file data and metadata may not be completely copied from the old server yet. When a client accesses the stub file, the new server retrieves only the data or metadata that are necessary to respond to the request, reducing the response time during file server migration.

A stub file is in of the two states: either a dummy state or partial state. Figure 1 depicts the differences between these two states in addition to the state of a normal file. Each stub file has an inode similar to that in a normal file. A stub file in a dummy state (Figure 1 (a)) has an inode in which most of the attributes would not be filled yet and which contains no file data or additional metadata. However, in a stub file in a partial state (Figure 1 (b)), the inode attributes are completely filled, and the file data may be partially retrieved from the old server. Additionally, the stub file may contain additional protocol-dependent metadata such as access control lists (ACL), alternative streams, or various time-stamps.

A file created during the process of on-demand migration is initially in a dummy state. Then, when a client accesses the metadata or the data of the file, the file transitions to a partial state. We use the dummy state to reduce the response time while performing the first access to a directory. If a directory contains thousands of files, retrieving the metadata of all files will require a long time because the metadata must be retrieved one-by-one by sending requests to the old server. However, in case of normal access to a directory, only a part of the metadata, i.e. directory entries, are sufficient for responding to the access request. Therefore, we initially copy a part of the metadata and later retrieve the remaining data and metadata when they are accessed for the first time.

To maintain the state and the relation to its corresponding file in the old server, each stub file holds the stub information as metadata (Figure 1 (a) and (b)). The stub information contains a file state ("state"), the file path in the old server ("path"), and optional data ("optdata"). The optional data are implementation-dependent and explained in Section 4. When all the data and metadata are completed, the stub file transitions to a normal file and the stub information is deleted (Figure 1 (c)) . Note that a file created on the new server by a client's request becomes a normal file and not a stub file.

To maintain consistency between a stub file and the corresponding file in the old server, the file path in the stub information is set using the stub information of the parent directory and not the pathname of the parent directory in the new server. For example, when a directory "/A/B" is a stub file and when the parent directory "/A" is renamed to "/X", its pathname in the new server becomes "/X/B". However, when a stub file "C" is created in the directory "B", the pathname in the stub information of "C" is set as "/A/B/C" which is the effective path in the old server and not the path "/X/B/C" in the new server.

For background migration, the file system must maintain a counter that contains the number of files that have not been migrated yet, as described in Section 3.2. If we count the number of files in the old server by traversing the directory, it will take a long time. Instead, the counter is initially set to be one, which indicates the stub root directory. When a stub directory is accessed, we retrieve the directory entry of the corresponding directory in the old server and increment the counter by the number of files. At this time, stub files corresponding to the directory entry are created in the directory. When a stub file is fulfilled and transitioned to a normal file, the counter is decremented. If the counter becomes zero, it indicates that the file migration has completed. This mechanism reduces the downtime in switching file servers.

This stub-based file management does not depend on the internal structure of the old server. It uses a standard file access protocol only to retrieve the metadata and data. Therefore, migration between heterogeneous servers is possible.

3.4 Migration Flow

The overall flow of file server migration is as follows. At first the clients still access the old server. The administrator initially installs and configures a new server. Further, the administrator creates a special account on the old server so that the new server can access all the files in the old server. Subsequently, the administrator creates a dummy root directory in the new server that corresponds to the root directory in the old server and enables on-demand migration. Thus, the clients can start accessing the new server.

To switch file servers, the administrator initially makes the old file server to be read-only and then directs all the clients to remount the file servers from the old server to the new one. Since on-demand migration is in effect, the clients can still access all the files using the new server. To reduce the response time and the total migration time, the administrator periodically runs a crawling program in the background when the server is not busy. Typically, the crawling program will be run during midnights or on weekends. However, the administrator can run it more frequently to reduce the migration time or to reduce the overhead. The access speed of the crawling program is also a configurable parameter. This program traverses all the directories in the new server and retrieves all the metadata and data of every file with on-demand migration.

When the counter in the new file server becomes zero, it indicates that the file server migration has completed. The administrator terminates the crawling program and shuts down the old server. Although the on-demand migration is not disabled, its code is never executed and the clients can directly access files without any overhead.

4 IMPLEMENTATION

Our implementation contains three programs: a file server program, a retriever program for on-demand migration, and a crawling program for background migration. Additionally, the stub management layer is located on the file system.

4.1 File Server Program

The file server program is an ordinary one. We reused the existing file server programs even though we have slightly modified them so that they can handle the protocol-specific issues described in Section 4.5. The stub-based file management is implemented in the kernel layer and is not dependent on the file server programs. We used the nfsd subsystem of a Linux kernel 2.6.30.1 for NFS and Samba 4.1.0 for SMB.

4.2 Retriever Program

The retriever program is an agent that retrieves the file data and metadata from the old server based on the requests from the stub management layer. In our implementation, the retriever program is a user-space application written in the C language, which communicates with the stub management layer in a UNIX domain socket. The stub management layer sends a request packet that contains the pathname in the old server and the type of request. The retriever program converts the request into a request for the old file server and returns the result data to the stub management layer.

The retriever program usually has to wait for a certain period of time to obtain a reply from the server. It should also handle multiple requests in parallel. Therefore, it maintains a thread pool to handle requests. In addition, to avoid inefficiency by sending multiple small requests, the retriever program retrieves the file data collectively with a specified block size from an aligned offset. Currently, the block and alignment size is 10 MiB. To access the file server, the retriever program uses the nfs client in the Linux kernel for NFS and libsmbclient bundled with Samba for SMB. The NFS client is configured to maximize the copy speed; the buffer size for read and write requests is 1 MiB, and the client metadata cache period is one week. In our architecture, the contents of the old server never vary during migration; therefore, a long cache expiration time does not cause any problem.

4.3 Crawling Program

The crawling program is a background process to migrate the files proactively. In our implementation, the crawling program is a single-threaded user-space application written in the C language and runs per a file share. Because it contains a single thread, the directory traversal is performed sequentially. The reason for not performing this operation in parallel is the complexity of synchronization among multiple crawlers. As described in Section 3.2, the clients could significantly alter the structure of the directory tree. Therefore, parallel crawling could cause directory traversal leakage or duplication. Parallel crawling is our future work.

4.4 Stub Manager

The stub manager is implemented in the Linux kernel virtual file system (VFS) layer. It does not depend on a specific file sharing protocol and supports both NFS and SMB. It also does not depend on a specific file system and does not modify the data structure on storage. To store the stub information, we use the extended attributes of files. Several modern file systems support extended attributes, and we can add additional information to the files without modifying the file systems. The stub information currently stores one-byte state information, a path string, and a 64-bit protocol-specific field that is described in Section 4.5.2.

To allow on-demand retrieval of file data and metadata, we installed a hook in the VFS layer. When a client accesses a stub directory, the stub manager first creates stub files in a dummy state in the directory. A stub file in the dummy state contains an inode with all-null attributes. When a client tries to access the metadata for the first time, the stub manager sends a request to the retriever program for retrieving metadata from the old server and temporarily suspends access. When the retriever program returns a response, the stub manager updates the metadata of the file and returns the metadata to the client. Access to the file data is handled in a similar manner except that the file data may be only partially filled using the function of sparse files.

Instead of modifying the kernel, user-space implementation is also worth considering. Common file server programs such as NFS-Ganesha [23] and Samba [25] accept additional plugin modules. Therefore, developing a module to communicate with the retriever program will achieve a similar functionality. However, we chose to implement our scheme in the kernel space for two reasons. The first is performance. Since our scheme requires multiple access to file metadata to implement stub files, user-space implementation will incur the overhead of multiple system calls. The second is the ease of supporting multiple protocols. If our scheme is implemented in a NFS server, it will not work for access via SMB. In addition, even if a server program support multiple protocols, implementing exclusive file access among multiple access is difficult and error-prone. Moreover, since various processes to achieve additional functionalities, such as virus detection and data compression, will run on the file server, direct access to the locale file system must also be supported. The kernel-space implementation can easily support all of local, NFS, and SMB access.

4.5 Protocol-Specific Issues

Although file migration may seem to be straightforward, there are several protocol-specific issues. We further explain some of the issues we support in our implementation.

4.5.1 Access control and user ID mapping. Each file sharing protocol contains an individual access control scheme and user ID space [14]. If a file server supports multiple protocols, it needs to manage the internal mapping between the ACLs of each protocol. However, we cannot obtain the internal mapping information of the old server from the new server. Further, there is no standard rule to determine the mapping [3, 11]. Fortunately, the differences between the mapping rules are not significant and are negligible in practice. In our current implementation, we retrieve ACLs via SMB and remap the ACLs for NFS access permissions because the SMB protocol provides more fine-grained ACLs than NFS. Similarly, mappings between the uids / gids of NFS and sids of SMB are internally managed in the server when there is no directory server. Therefore, we remap the uids / gids to the sids using a user-defined mapping table. Note that enterprise servers are usually operated with directory servers and remapping IDs is rarely used in actual situations.

4.5.2 Truncate. In the stub-based file management system, file data regions that are not retrieved yet are managed as holes of sparse files. If a client once truncates and then expands a stub file using the NFS truncate operation, the newly created hole at the end of the stub file cannot be distinguished from the data regions that were not retrieved. Therefore, the stub manager attempts to fill the hole with the data of the old server even though it should have been left as a hole. To distinguish a hole from a region that was not copied, we use the "minsize" attribute from the stub information. This attribute indicates the minimum size of the stub file in the past. The "minsize" attribute is initially set as the file size in the old server. If a client truncates a stub file and if its size becomes smaller than the "minsize", the stub manager updates the "minsize" attribute with the new file size. If the size of the hole region is beyond this minsize, it is considered to be a newly created hole; therefore, the retriever program does not fill that region with the old data.

Currently, the hole punching operation is under discussion as a feature of NFS 4.2 [12]. This operation can create a hole anywhere in a file, which makes it difficult to distinguish the holes from regions that were not replicated. Further investigation is required to support the hole punching operation in an accurate manner.

4.5.3 Hard link. The stub manager maintains the relation between a stub file in the new server and the corresponding file in the old server using a pathname. If the replication process is not aware of the hard links, a hard link file in the old server is replicated into multiple separate normal (not hard link) files in the new server. To maintain the hard links, we obtained the hard link relation information (i.e., the inode number and link count) using the NFS protocol.

When the retriever program creates a stub file, it checks whether the link count is more than one. If so, the retriever program creates a provisional file under a hidden directory whose path is uniquely generated using the inode number. When the retriever program creates another stub file and its link count is more than one, it checks whether the provisional file already exists. If it exists, the retriever program creates a hard link to the provisional file instead of creating a new stub file. This procedure can maintain the hard links of the old server on the new server. The provisional file and directory are eliminated after completing the migration.

4.5.4 Access delay. In a situation where thousands of files exist in a directory, the access to a file may require a long time to create many stub files in the directory. If the file server does not respond to a request for a long time, the client may hang up for a while or the request times out, neither of which is desirable. In NFSv3, the NFS3ERR_JUKEBOX return code is introduced to indicate that the target file is temporarily unavailable [4]. If a client receives this error code, the client will wait for a while and retry later. Therefore, if we take more than 500 ms to respond to a request, we will return this error code. To perform this, we slightly modify the NFS server. This configuration could extend the downtime during file server migration. However, we assume that this configuration is the accurate approach to handle access delays. This issue is discussed further in Section 5.2.

4.5.5 Thumbnail. Windows Explorer attempted to create thumbnails of files in a directory. This caused several read requests to retrieve the file contents, which degraded the performance of directory access if the directory contained several stub files. To suppress this, we exploited the offline attribute of the Windows file attributes. This attribute is defined to indicate that the file data has been physically moved to an offline storage, and therefore, Explorer does not create thumbnails for the offline files. Therefore, we slightly modified the Samba server so that it sets the offline attribute of a file when the file is a stub file.

4.6 Limitations

Our method copies data and metadata using common file sharing protocols to support heterogeneous file servers. The limitation of this approach is that we cannot migrate the states of functions or configurations that are not supported by the common file sharing protocols. For example, some file servers support the reduction of data by compression and deduplication. NFS and SMB do not provide the internal expression for such reduced data. Thus, we cannot reproduce the states of reduced data. However, data compression and deduplication can be achieved using a post process. A post process allows the new file server to use arbitrary internal data structures without depending on the old server. Another limitation is the disk quota. It is implemented outside the file sharing protocol. We cannot handle such configurations; therefore, the administrator must manually reconfigure the quota settings after file server migration. Note that these are not significant problems in actual production environments.

5 EVALUATION

This section depicts the results of the performance experiments. We evaluated the downtime, total migration time, and overhead after migration.

Table 1: Server and client setup

	Physical Machine	Virtual Machine
CPU	Xeon E5-2603 x 2	vCPUs x 8
Memory	64GB	16GB
Storage	NVMe SSD (x 3)	LSI Logic SAS
NIC	10GbE Copper	VMXNET3
OS	Ubuntu Server 17.10 Windows Server 2016	Linux 2.6.30.1
VMM	-	VMWare ESXi 6.5

5.1 Setup

Table 1 describes the setup of the file servers and client. We used the new server on a VM because the kernel does not support the latest storage device. We used an Ubuntu Server for the NFS experiments and a Windows Server for the SMB experiments on the old server and client, respectively. We used the NFSv3 and SMB3.0 protocols. NFSv3 is a stateless protocol and supports only POSIX-compatible metadata. In contrast, SMB is a stateful protocol and its semantics and metadata are relatively rich (e.g. ACLs and DOS attributes). Therefore, copying files via SMB requires more operations to open files and obtain metadata compared to NFS [11].

5.2 Downtime

We measured the downtime for our file server migration. A client mounts the old server and opens a 1 GiB test file. After the client reads the first half of the file, it switches the connection to the new server. We measured the durations of the three steps; *remount* closes the test file, unmounts the old server, and mounts the new server; *open* opens the file with the same file path; *read* reads the first byte. We regarded the total time of the three steps to be the downtime.

To open and read the test file, the directory entries in the test file path must be retrieved, and the dummy files in each directory must be created; therefore, the depth of the test file and width (the number of files in each directory) affect the downtime. We performed the experiments by varying the depth to 1, 4, and 16 and by varying the width to 10, 100, and 1000. We tested the following four configurations to clarify the impact of our on-demand migration and performed experiments thrice to calculate the average time.

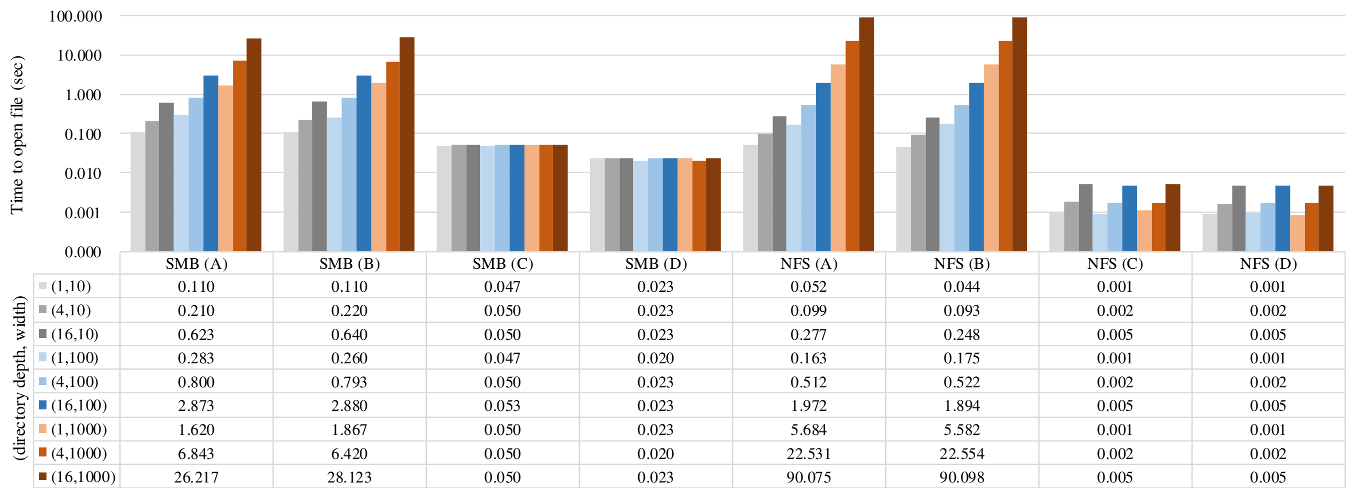
(A) Data and metadata are post-copy with a dummy state.
(B) Data and metadata are post-copy without a dummy state; the new server replicates all the metadata immediately when a parent dummy directory is accessed.
(C) Data are post-copy and metadata are pre-copy; the new server copies the entire directory tree in advance.
(D) Data and metadata are pre-copy; a client runs rsync or robocopy in advance to copy all the files.

The *remount* time depends only on protocols. The average times for NFS and SMB are 0.162 s and 0.305 s, and the standard deviations are 0.006 s and 0.028 s, respectively.

Figure 2 depicts the *open* time. Each bar presents a directory structure with a different depth and width. The results of (A) and (B) are approximately proportional to the product of the depth and width. For example, in case of SMB (A), the time to open a file in the root directory with 10 files (the "(1,10)" case) was 0.11 s, while opening a file having 16 levels with 1000 files in each directory (the "(16,1000)" case) took 26.217 s. Accessing a file in a wide directory via NFS took much longer; the "(16,1000)" case took ten times as long as the "(16, 100)" case. The reason was that the file server returned NFS3ERR_JUKEBOX to notify the client that the retrieval program required more than 500 ms to look up a directory because creating 1000 stub files took approximately one second. When the Linux NFS client received this error code, it retried after 5 s; therefore, retrieving one directory took approximately 5.5 s. Although tuning parameters can reduce this time, even if we do not tune it, the client still continued to access the files without causing any errors.

Compared to (A), (B) requires additional accesses to the old server to fill the metadata of all the files in the directories. However, (B) did not increase the open time as much. This was because the NFS client used the READDIRPLUS operations to look up the directory entries. The response of READDIRPLUS already included the metadata; therefore, additional access requests were not necessary. The SMB client used QUERY_DIRECTORY requests to retrieve the directory entries. Their responses contained basic metadata; however, their responses did not contain ACLs. Thus, if several files depicted large ACLs, the open time would increase in (B). We confirmed that, if a file contained a large ACL, additional access requests were sent to the server. In (C) and (D), the open time was less than 0.05 s. This confirmed that most of the open time in our approach was required to copy the directory entries and create stub files. The results of the SMB (C) were approximately slower than that of SMB (D) by 0.02 s even though both the conditions did not retrieve metadata. This was because the Windows SMB client issued a few read requests internally during the file open operation.

The *read* time depends on whether the file data was retrieved. The time in (A), (B), and (C) was 0.058 s in NFS and 0.217 s in SMB. In contrast, the time in (D) was 0.017 s in NFS and 0.022 s in SMB. The on-demand data retrieval increased by 0.031 s in NFS and 0.249 s in SMB, which were not significant. In SMB, data retrieval caused multiple operations due to its statefulness; therefore, the time was longer than that of NFS. However, the times were much shorter than that required for reading the entire 1 GiB of data. Therefore, on-demand migration was effective to reduce the downtime.



	SMB (A)	SMB (B)	SMB (C)	SMB (D)	NFS (A)	NFS (B)	NFS (C)	NFS (D)
(1,10)	0.110	0.110	0.047	0.023	0.052	0.044	0.001	0.001
(4,10)	0.210	0.220	0.050	0.023	0.099	0.093	0.002	0.002
(16,10)	0.623	0.640	0.050	0.023	0.277	0.248	0.005	0.005
(1,100)	0.283	0.260	0.047	0.020	0.163	0.175	0.001	0.001
(4,100)	0.800	0.793	0.050	0.023	0.512	0.522	0.002	0.002
(16,100)	2.873	2.880	0.053	0.023	1.972	1.894	0.005	0.005
(1,1000)	1.620	1.867	0.050	0.023	5.684	5.582	0.001	0.001
(4,1000)	6.843	6.420	0.050	0.020	22.531	22.554	0.002	0.002
(16,1000)	26.217	28.123	0.050	0.023	90.075	90.098	0.005	0.005

Figure 2: Time required to open a file.

In total, the server migration took less than 30 s except for the case when the files were in wide directories. However, in normal cases, directories have at most tens of files [19]; therefore, it is a rare case.

The downtime in the pre-copy approach mostly depends on the replication time of the last copy and not the open time. Therefore, we evaluate the downtime of the pre-copy approach in the succeeding section.

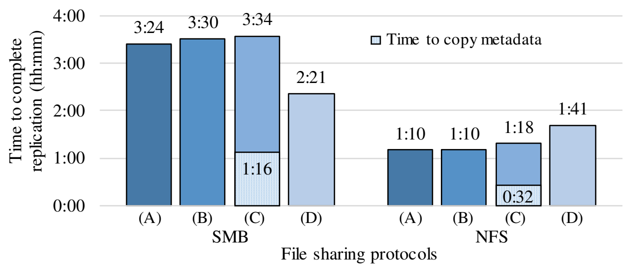


Figure 3: Total migration time.

5.3 Total Migration Time

We evaluated the total migration time using a dataset obtained from our office. In this dataset, more than 100 users had stored files for over 10 years. The size and directory depth distribution of the files were similar to that of previous studies [6, 7, 19]. The file share contained 30 K directories, 484 K files, and 242 GiB of data in total. The average file size was 500 KiB, and its distribution followed the Pareto model. The depth of 90% of the files was less than 12, and the average depth of all the files was 8.84. These two distributions indicated that our dataset reflected the common file server usages. We measured the replication time using the four configurations in Section 5.2. We measured the replication time of the data and metadata of (C) separately.

Figure 3 depicts the results. The replication times of (A), (B), and (C) were almost identical. This was reasonable because the differences were mostly in the order and timing. (A) was a little faster than (B) and (C) because of the effects of metadata caching in VFS. Since (A) replicated data right after replicating the metadata, the inodes and directory entries were cached. In contrast, (C) replicated the metadata in advance and the data later. Due to heavy access of the file data, the metadata could be evicted from the cache. Therefore, the metadata could be accessed again, which led to a longer total replication time. In case of (B), the retriever program traversed the directory tree by performing a depth-first search. Therefore, the file metadata in a shallow directory were evicted when the program walked its large sub-tree.

The results of (D) were observed to vary with protocols. Since the pre-copy was performed by a client, the replication time was expected to be longer due to the network transfers. In case of NFS, the replication time was slower than that of other configurations. In contrast, in case of SMB, the replication time was faster. This was due to the statefulness of the SMB protocol and the behavior of the retrieval program. In this case, the client transferred the file data in bulk. The retriever program was designed for on-demand access; therefore, it opened a file each time it accessed a file segment. This behavior increased the total migration time in the SMB protocol. We can optimize this behavior by keeping the file handles for bulk data transfer during background migration.

In the pre-copy approach, files are repeatedly copied until the size of the updated files become small. Therefore, the downtime and migration time depend on the number of this iteration and the size of the updated files. To estimate the

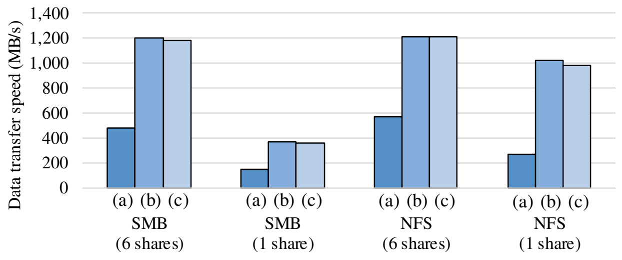


Figure 4: Data transfer speed of the client file read.

downtime, we updated a part of this dataset and measured the replication time. In the NFS case, we randomly updated 43K 2.4 GiB files and replicated them using rsync. The result was that the operation took 328 s. In case of rsync, traversing the directory tree took more than two min because rsync checks the update time of every file. Therefore, the downtime is at least two min, which is longer than our post-copy approach. In the SMB case, we randomly updated 47K 520 MiB files and replicated them using robocopy. The result was that the operation took 27 s. In case of robocopy, it did not access each file; instead, it accessed the directory entries to determine the files to be copied. However, if the clients updated more files than that observed in this situation, the downtime becomes longer than that in our post-copy approach.

As for the migration time, the pre-copy approach took 69% of the time required by our approach using SMB and was already longer than our approach using NFS (Figure 3). If clients updated any files since the last replication was performed, it would take more time. The time was dependent on the update frequency and amount of files. However, in busy servers, it is likely that the total migration time will be longer than that in our approach even in case of SMB.

5.4 Overhead After Migration

We measured the overhead of on-demand migration in the following three situations: (a) before migration, (b) after migration, and (c) normal case. We first prepared six file shares filled with large files to evaluate the bulk data transfer performance. A client sequentially reads one file per share in parallel. We measured the transfer speed of the file read. Figure 4 shows the results. In case of 6 shares, (b) and (c) depicted approximately 1200 MB/s, and the 10-GbE network bandwidth was saturated. As expected, (a) depicted lower performance than that depicted by (b) and (c); however, (b) and (c) depicted approximately the same performance.

To evaluate the performance in a mixed workload, we used filebench [32] v1.5.0-alpha3. We chose the *fileserver-new* workload that deployed 10,000 files, a 1.2 GiB dataset, and 50 client processes to repeatedly open, close, read, write, append, and delete files. We measured the operations per second for

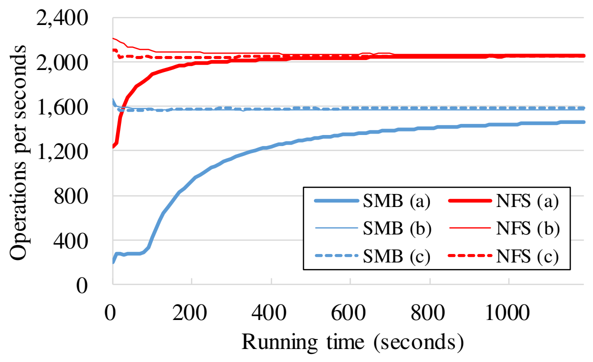


Figure 5: Performance of the mixed workload.

20 min. This workload generated repeated accesses to the same files; therefore, the client cache was disabled during the course of this experiment. The current filebench does not support the Windows platform; therefore, we used a Linux client for both NFS and SMB.

Figure 5 depicts the results. The results of (b) and (c) were approximately identical. (a) initially depicted a lower performance due to the on-demand migration. However, as the files are replicated onto the new server, the performance soon increased to become close to the performance of (c). SMB (a) depicted slower movement due to the statefulness of the protocol. In the first tens of seconds, the file access produced metadata retrieval; therefore, SMB required more operations than that required by NFS. Still, retrieving a file generated multiple SMB requests, and the filebench randomly accessed the files. Thus, the performance recovery was slower than that in NFS. Finally, the performance became almost the same as that of (c) after the migration of all the files.

6 CONCLUSION

We proposed a quick file server migration scheme. In this scheme, file data and metadata are migrated from the old to new servers in the post-copy approach using two mechanisms. The first is on-demand migration that retrieves the file data and metadata only when they are accessed by the clients for the first time. The second is background migration that traverses the directory tree and proactively retrieves the file data and metadata. These two mechanisms are based on a stub-based file management that allows intermediate states of file migration. This scheme can reduce the downtime during file migration, reduce the migration overhead including the total replication time and file access overhead after migration, and support heterogeneous file servers.

In the current implementation, there is still room for optimization of replication performance. This is because the total replication time can be further reduced by the parallelized crawling of directory trees and the bulk transfer of files.

REFERENCES

[1] Ted Anderson, Leo Luan, Craig Everhart, Manuel Pereira, Ronnie Sarkar, and Jane Xu. 2004. Global namespace for files. *IBM Systems Journal* 43, 4 (2004), 702–722.

[2] Alain Azagury, Michael E. Factor, Julian Satran, and William Micka. 2002. Point-in-Time Copy: Yesterday, Today and Tomorrow. In *Proceedings of 19th IEEE Mass Storage Systems and Technologies*. 259–270.

[3] Ellie Berriman and Binguxe Cai. 2011. *NetApp Storage System Multiprotocol users guide*. Technical Report 3490. NetApp. http://www.netapp.com/us/media/tr-3490.pdf

[4] Brent Callaghan, Brian Pawlowski, and Peter Staubach. 1995. Network File System (NFS) Version 3 Protocol Specification. Internet Requests for Comments. (1995). https://www.rfc-editor.org/info/rfc1813

[5] Data Dynamics Inc. 2017. StorageX 8.0. (2017). Retrieved 2018-04-28 from https://www.datadynamicsinc.com/launch/

[6] John R. Douceur and William J. Bolosky. 1999. A Large-scale Study of File-system Contents. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS ’99)*. ACM, New York, NY, USA, 59–70. https://doi.org/10.1145/301453.301480

[7] Allen B. Downey. 2001. The structural cause of file size distributions. In *Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2001)*. 361–370.

[8] EMC Corporation. 2009. EMC Rainfinity File Management Appliance Getting Started Guide. (2009). Retrieved 2018-04-28 from https://www.emc.com/collateral/TechnicalDocument/docu8513.pdf

[9] Marc Eshel, Roger Haskin, Dean Hildebrand, Manoj Naik, Frank Schmuck, and Renu Tewari. 2010. Panache: A Parallel File System Cache for Global File Access. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*. 155–168.

[10] F5 Networks, Inc. 2013. ARX Series Datasheet. (2013). Retrieved 2018-04-28 from https://www.f5.com/pdf/products/arx-series-ds.pdf

[11] Steven M. French. 2007. A New Network File System is Born: Comparison of SMB2, CIFS, and NFS. In *Proceedings of Linux Symposium*, Vol. 1. 131–140.

[12] Thomas Haynes. 2016. Network File System (NFS) Version 4 Minor Version 2 Protocol. Internet Requests for Comments. (2016). https://www.rfc-editor.org/info/rfc7862

[13] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. 2009. Post-copy Live Migration of Virtual Machines. *SIGOPS Operating System Review* 43, 3 (July 2009), 14–26. https://doi.org/10.1145/1618525.1618528

[14] Dave Hitz, Bridget Allison, Andrea Borr, Rob Hawley, and Mark Muhlestein. 1998. Merging NT and UNIX Filesystem Permissions. In *Proceedings of the 2nd USENIX Windows NT Symposium*. 10.

[15] Intel, Inc. 2013. Planning Guide: Updating IT Infrastructure. (2013). http://www.intel.com/content/dam/www/public/us/en/documents/guides/server-refresh-planning-guide.pdf

[16] JM Project. 1993. Linux Programmer’s Manual: NFS. (1993). Retrieved 2018-04-28 from https://linuxjm.osdn.jp/html/util-linux/man5/nfs.5.html

[17] Nemoto Jun, Sutoh Atsushi, and Iwasaki Masaaki. 2017. Directory-Aware File System Backup to Object Storage for Fast On-Demand Restore. *International Journal of Smart Computing and Artificial Intelligence* 1, 1 (2017), 1–19.

[18] Ali Mashtizadeh, Emré Celebi, Tal Garfinkel, and Min Cai. 2011. The Design and Evolution of Live Storage Migration in VMware ESX. In *Proceedings of the 2011 USENIX Annual Technical Conference*. 1–14.

[19] Dutch T. Meyer and William J. Bolosky. 2011. A Study of Practical Deduplication. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*. 1–14.

[20] Microsoft. 2009. Windows Server Migration Tools and Guides. (2009). Retrieved 2018-04-28 from https://technet.microsoft.com/en-us/library/dd759159(v=ws.11).aspx

[21] Microsoft TechNet. 2016. Command-Line Reference Robocopy. (2016). Retrieved 2018-04-28 from https://technet.microsoft.com/en-us/library/cc733145(v=ws.11).aspx

[22] Edgar A. Olougouna. 2013. SMB 2.x and SMB 3.0 Timeouts in Windows. (2013). Retrieved 2018-04-28 from https://blogs.msdn.microsoft.com/openspecification/2013/03/27/smb-2-x-and-smb-3-0-timeouts-in-windows/

[23] NFS-Ganesha project. 2017. NFS-Ganesha Wiki : Fsalsupport. (2017). Retrieved 2018-04-28 from https://github.com/nfs-ganesha/nfs-ganesha/wiki/Fsalsupport

[24] QEMU Wiki. 2011. Image Streaming API. (2011). Retrieved 2018-04-28 from https://wiki.qemu.org/Features/ImageStreamingAPI

[25] Samba Team. 2017. Samba Wiki : Writing a Samba VFS Module. (2017). Retrieved 2018-04-28 from https://wiki.samba.org/index.php/Writing_a_Samba_VFS_Module

[26] Bianca Schroeder and Garth A. Gibson. 2007. Understanding Disk Failure Rates: What Does an MTTF of 1,000,000 Hours Mean to You? *ACM Transactions on Storage* 3, 3, Article 8 (Oct. 2007). https://doi.org/10.1145/1288783.1288785

[27] Spencer Shepler, Mike Eisler, and David Noveck. 2010. Network File System (NFS) Version 4 Minor Version 1 Protocol. Internet Requests for Comments. (2010). http://www.rfc-editor.org/rfc/rfc5661.txt

[28] Andrew S. Tanenbaum. 2007. *Modern Operating Systems* (3rd ed.). Prentice Hall Press.

[29] TechTarget. 2017. NAS trifecta: Price, features and performance. *Storage Magazine* 16, 8 (2017), 14.

[30] TechTarget. 2017. Snapshot 1: New NAS buys motivated by performance and outdated hardware. *Storage Magazine* 16, 2 (2017), 12.

[31] Andrew Tridgell and Paul Mackerras. 1996. *The rsync algorithm*. Technical Report TR-CS-96-05. ANU Research Publications.

[32] Tarasov Vasily, Zadok Erez, and Shepler Spencer. 2016. Filebench: A Flexible Framework for File System Benchmarking. *;login: The Usenix Magazine* 41, 1 (2016), 6–12.

[33] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. 2009. Cumulus: Filesystem Backup to the Cloud. *ACM Transactions on Storage (TOS)* 5, 4, Article 14 (Dec. 2009), 28 pages. https://doi.org/10.1145/1629080.1629084

[34] Katsurashima Wataru, Yamakawa Satoshi, Torii Takashi, Ishikawa Jun, Kikuchi Yoshihide, Yamaguti Kouji, Fujii Kazuaki, and Nakashima Toshihiro. 2003. NAS switch: a novel CIFS server virtualization. In *Proceedings of 20th IEEE Mass Storage Systems and Technologies*. 82–86.

[35] Mike Worthon. 2012. *SnapMirror Configuration and Best Practices Guide for Clustered Data ONTAP*. Technical Report 4015. NetApp. https://www.netapp.com/us/media/tr-4015.pdf

[36] Yasuda Yoshiko, Kawamoto Shinichi, Ebata Atsushi, Okitsu Jun, and Higuchi Tatsuo. 2003. Concept and evaluation of X-NAS: a highly scalable NAS system. In *Proceedings of 20th IEEE Mass Storage Systems and Technologies*. 219–227.