Reusing migration to simply and efficiently implement multi-server operations in transparently scalable storage systems

Shafeeq Sinnamohideen

May 2010
CMU-CS-10-141

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Gregory R. Ganger, chair
Garth Gibson
Priya Narasimhan
Jiri Schindler, NetApp

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

This material is based on research sponsored in part by the National Science Foundation, via grant #CNS-0326453, by the Air Force Research Laboratory, under agreement number F49620–01–1–0433, and by CyLab at Carnegie Mellon University under grant DAAD19–02–1–0389 from the Army Research Office. Support was also provided by the companies of the PDL Consortium (including APC, EMC, Facebook, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Seagate, Symantec, VMware, and Yahoo!)

Keywords: Object-based storage, object ID assignment algorithms, namespace flattening, OSD, metadata scalability, multi-server operations, cross-server operations, cross-directory operations, transparent scalabilily

Abstract

Distributed file systems that scale by partitioning files and directories among a collection of servers inevitably encounter multi-server operations. A common example is a RENAME that moves a file from a directory managed by one server to a directory managed by another. Transparently scalable systems (those that provide the same semantics for multi-server operations as they do for single-server operations) traditionally implement dedicated protocols for these rare operations. This thesis explores an alternate approach, with simplicity as a goal, that exploits the existence of migration functionality normally used for load balancing. When a client request would involve files on multiple servers, the system can migrate responsibility for those files onto one server and have it service the request. Although migration may be more expensive than a dedicated cross-server protocol, trace analysis of deployed file systems indicates that such operations are extremely rare in file system workloads. A prototype system that uses this approach to supporting multi-server operations scales linearly and performs well even when multi-server operations are 100X more common than the worst-case trace. Thus, when migration functionality exists in the system, multi-server operations can be efficiently handled with very little additional implementation complexity.

For my parents.

Acknowledgments

I would like to thank the members and alumni of the PDL and other residents of D-level and CIC for their friendship and collaboration, including Michael Abd-El-Malek, Rajesh Balan, Chris Colohan, Jason Flinn, Charlie Garrod, James Hendricks, Benoit Hudson, Andy Klosterman, Michael Mesnier, Dushyanth Naryanan, Brandon Salmon, Raja Sambasivan, Matthew Wachs, and Theodore Wong. Also Chuck Cranor, Bill Courtwright, Michael Stroucken, and Charlene Zang, and other members of the Self-$*$ project. I'd like to thank Greg for his support and advice, Satya for encouraging me to begin this path, and Garth, Jiri, and Priya for their feedback, insight and support.

I also thank the members and companies of the PDL Consortium (including APC, EMC, Facebook, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Seagate, Symantec, VMware, and Yahoo!) for their interest, insights, feedback, and support. I also thank Intel, IBM, NetApp, Seagate and Sun for hardware donations that enabled this work. This material is based on research sponsored in part by the National Science Foundation, via grant #CNS-0326453, by the Air Force Research Laboratory, under agreement number F49620–01–1–0433, and by CyLab at Carnegie Mellon University under grant DAAD19–02–1–0389 from the Army Research Office.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Transparent scalability is a desired feature for many distributed systems. This means that it should be possible to increase both capacity and performance by adding servers and spreading data and work among them. It also means that client applications and users are presented a consistent set of semantics, regardless of which servers are hosting which data.

In the case of file systems, many designs scale by partitioning the set of files across the set of servers. Each file is managed by a particular server, and accesses to files managed by different servers are completely independent. The vast majority of operations affect only a single file, so this approach works well in the common case. A few operations, such as a cross-directory rename or a snapshot, affect more than one file or directory and so may involve files managed by two distinct servers. A transparently scalable system would provide the same semantics in this case as in the case where all files are on the same server. Providing strong semantics on a single server is relatively easily done using techniques like local locking and write-ahead logging, but doing so is more difficult when multiple servers are involved.

Many existing systems do not provide identical semantics in this case. Applications, on the other hand, frequently rely on specific consistency semantics (e.g., atomicity of rename) and have no convenient way of knowing which semantics (same-server or multi-server) any given operation will get. Additionally, systems that aim to scalably support legacy protocols, such as NFSv3 [9] or CIFS [35], must maintain the semantics defined by the protocol, regardless of whether any given application relies on them.

Systems that do provide transparent scalability use some sort of distributed protocol to handle operations that involve multiple servers. In the case where all objects are accessible from any server, this is often done by having one server acquire locks on all objects involved, update the objects and any logs as necessary, and then release the locks. The underlying distributed lock manager will trigger the appropriate cache invalidations to maintain consistency between servers. GPFS uses such an approach [47]. An alternative, frequently used when servers do not share a common storage pool, is to have each server execute only the portion of the operation that pertains to it. All objects are then updated to their final state using a multi-phase commit protocol to ensure atomicity.

These solutions, while effective, are also complex to implement, debug, and verify, particularly for the cases involving failures. Furthermore, in many workloads, multi-server operations occur very rarely. For example, in Farsite's experimental workload of 2.1 million operations, only 8 involved multiple servers [13]. Thus, the programmer effort traditionally involved in supporting transparent multi-server operations is out of proportion with its utilization in most workloads.

1.1 Thesis statement

This dissertation develops an alternate approach to handling multi-server operations. Specifically:

Reusing migration to convert multi-server operations into single-server operations is a simple and efficient method of enabling transparent scalability.

Most scalable systems have mechanisms to migrate objects from one server to another. This capability is used to ensure that server resources are utilized efficiently. For example, when a server is overloaded, some of the objects it's responsible for should be moved to other servers, moving the load associated with them. Similarly, when a server's disks are nearly full, some of its objects should be moved to other servers with more free space. There has been much work on load balancing policies, and the underlying mechanisms used for transferring objects are fairly straightforward.

This same migration mechanism can be used to support multi-server operations, instead of a dedicated multi-server operation protocol. If servers can only process requests for objects they are responsible for, all the objects involved must be the responsibility of the same server. If this precondition is not satisfied, the system will migrate the required objects until it is satisfied, and only then execute the operation. Objects that were moved may be returned to their original servers immediately, or they can be left until load-balancing policy dictates that they be moved again. All of the inter-server communication is encapsulated in the migration mechanism, which already exists, saving implementation and debugging effort.

To support the thesis statement, this dissertation takes the following steps. First, it characterizes several existing distributed file system workloads, showing that the expected frequency of multi-server operations is low. Second, it introduces a technique for assigning files to servers in a way that reduces the occurrence of multi-server operations. Third, it describes a transparently scalable prototype storage system that uses migration to support multi-server operations, demonstrating the feasibility of the approach. Fourth, it evaluates the prototype to show that performance is reasonable (i.e, the approach is efficient) for a range of workloads and system parameters and that the implementation is relatively simple.

This thesis depends on the following primary assumptions:

- A migration mechanism already exists in the system.
- Implementing a migration-only mechanism is simpler than general-purpose distributed transactions.
- Multi-server operations are rare.

The next section briefly explains why each of these assumptions are valid for scalable distributed file systems, and identifes further sections where they are discussed in more detail. If this approach is applied to a system that does not meet these assumptions, then it will either not be the simplest solution or its performance will be worse than the alternative approaches. However, the resulting system will still operate correctly.

1.2 Argument

The efficacy of using migration to implement cross-server operations depends on two factors :

- There is little performance cost from using migration instead of a dedicated protocol.
- Supporting multi-server operations using migration requires less implementation effort than a dedicated protocol for the same purpose.

Some performance penalty is to be expected, but, in many cases, we expect this penalty will be small enough to be outweighed by the savings in implementation cost. System architects make many such trade-offs in the course of designing a system. If the architect's goal is the best possible performance at any cost, then any performance penalty is unacceptable. Frequently, however, the goal is to maximize "performance for a fixed cost" (e.g., a deadline) or "performance per unit of effort". System builders often have a list of potential performance enhancements that they have not yet had time to pursue. The performance penalty will be acceptable as long as it is inconsequential, or if the effort saved in supporting multi-server operations can then be used to optimize other aspects of the system to gain more performance than was lost. Amdahl's law suggests that this will be likely — the vast majority of operations are single-server, so a small improvement on each can overcome a large slowdown on the rare multi-server operation.

1.2.1 On the performance penalty

The performance penalty is expected to be small in most environments. The size of the penalty is governed by how frequent cross-server operations are and how much more expensive each one is compared to the alternate approach. In most file system scenarios, the frequency of cross-server operations is very low. Thus, even though each cross-server operation may be more expensive, because the latency of migration will be added directly to the latency of the multi-server operation, the effect on overall throughput will be negligible.

On the frequency of multi-file and multi-server operations

The frequency of multi-file operations is a property of a given workload. These operations exhibit the well-known properties of spatial and temporal locality, just as single-file operations do. Thus, traditional techniques to improve overall performance, such as grouping files by directory, subtree, or access pattern and placing an entire group on the same server, result in most multi-file operations affecting multiple files on the same server. The only times an actual multi-server operation would occur in practice are in the rare case of a multi-file operation that affects files far enough apart in the directory tree to be on separate servers. Chapter 3 describes the potential multi-file operations and their frequency in traces of deployed large-scale storage systems. Chapter 4 describes techniques for assigning files to servers in a manner that increases the likelihood that all files in a multi-file operation will already reside on a single server, thus reducing the frequency of multi-server operations.

On the cost of multi-server operations

The cost of a multi-server operation depends on the cost of performing a migration, which in turn depends on how much data needs to be moved and the mechanisms for moving it.

For the purposes of supporting multi-server operations, usually only a single item needs to be migrated. Many systems, however, cannot migrate at the granularity of a single item. Single item migration requires that the mechanism used to map items to servers must handle the case where every item is potentially on a different server, and the servers must store items such that individual ones can be transferred separately. Additionally, because each server could be responsible for a sparse set of items, the simple task of determining what the "next" item is is complicated by the fact the next item could always be on another server. All of these add to the basic complexity of the system by requiring more complex data structures and increasing overheads even when no migrations are in progress.

Depending on the system architecture, migrating files either involves copying them over the network from one server to another or relies on storing all files in a common storage pool, in which case migration merely involves a logical hand-off of responsibility. If shared storage is used, the latency for the logical hand-off can be expected to be on the order of that for a distributed transaction (since the hand-off is a simple distributed transaction itself) and, thus, to have negligible effect on throughput. If shared storage is not available, then the latency will be heavily dependent on the size of the unit of migration. If only metadata needs to be transferred, and the unit is reasonably sized (a directory or a small number of directories), the transfer latency can also be small enough. Fortunately, it seems likely that one or both of these conditions will be true in most future storage systems.

Chapter 5 describes the prototype system, relying on shared storage, that we built, and Chapter 6 evaluates the performance of the system both when using shared storage, and when emulating the migration cost of a system with private storage.

1.2.2 On the implementation effort

Both migration and a dedicated multi-server operation mechanism require distributed protocols. Migration, however, is simpler to implement than generalized distributed transactions, particularly if the system was not designed from the ground up to support either. Furthermore, most scalable systems naturally include redistribution as a part of their basic functionality. Section 2.4 discusses the complexity of correctly implementing a distributed transaction protocol, and Section 2.5 describes migration protocols. Section 6.6 contrasts the implementation complexity of supporting multi-server operations by reusing migration and by a dedicated protocol.

1.3 Outline

The remainder of this dissertation is organized as follows. Chapter 2 discusses background and related work. Chapter 3 presents an analysis of workloads to determine the occurrence of multi-file operations. Chapter 4 describes our approach to assigning files to servers. Chapter 5 describes a prototype storage and its support for transparent multi-server operations. Chapter 6 evaluates the efficacy of this approach.

Chapter 2

Background

Among others, challenges in scaling the number of servers in a system include handling the infrequent operations that involve multiple servers and managing the distribution of files across servers. This chapter discusses the types of operations that could involve multiple servers, how close existing systems come to being transparently scalable, how systems that handle multi-server operations transparently do so, and the importance of migration in a multi-server file system

2.1 Multi-item operations

There are a variety of file system operations that manipulate multiple files, creating a consistency challenge when the files are not all on the same server. Naturally, every CREATE and DELETE involves two files: the parent directory and the file being created or deleted. Most systems, however, assign a file to the server that owns its parent directory. At some points in the namespace, of course, a directory must be assigned somewhere other than the home of its parent; or else all metadata will be managed by a single metadata server. Therefore, the CREATE and DELETE of that directory will involve more than one server, but none of the other operations on it will do so. This section describes other significant sources of multi-item operations.

The most commonly noted multi-item operation is RENAME, which changes the name of a file. The new name can be in a different directory, which would make the RENAME operation involve both the source and destination parent directories. Also, a RENAME operation can involve additional files if the destination name exists (and thus should be deleted) or if the file being renamed is a directory (in which case, the '..'

entry must be modified and the path between source and destination traversed to ensure a directory will not become a child of itself). Application programming is simplest when the RENAME operation is atomic, and both the POSIX and the NFSv3 specifications call for atomicity. [1]

Many applications rely on the specified atomicity of RENAME as a building-block to provide application-level guarantees. For example, many document editing programs implement atomic updates by writing the new document version into a temporary file and then using RENAME to move it to the user-assigned name. Similarly, many email systems write incoming messages to files in temporary directory and then RENAME them into a user's mailbox directory. Without atomicity, applications and users can see strange intermediate states, such as two identical files (one with each name) existing or one file with both names as hard links.

Creation and deletion of hard links (LINK and UNLINK) are also multi-item operations in the same way that CREATE is. However, the directory the link is to be created in may not be the parent of the the file being linked to, making it more likely that the two are on different servers than for a CREATE and UNLINK.

The previous examples assume that each directory is indivisible and could only be assigned to one server at a time. But a single heavily used directory might have more traffic than a single server can support. Some systems resolve this issue by splitting directories and assigning each part of the directory to a different server [43, 52]. In that case, simply listing the entire directory requires an operation on every server across which it is split, and renaming a file within a directory might require two servers if the source name is in one part of the directory and the destination is in a different part.

Transactions are a very useful building block. Some modern file systems, such as NTFS [41] and Reiser4 [45], are adding support for multi-request transactions. For example, an application could update a set of files atomically, rather than one at a time, and thereby preclude others seeing intermediate forms of the set. This is particularly useful for program installation and upgrade. The files involved in such a transaction could very easily be spread across servers.

Point-in-time snapshots [10, 29, 39, 44] have become a mandatory feature of most storage systems, as a tool for consistent backups. Snapshots also offer a building block for on-line integrity checking [39] and remote mirroring of data [44]. Snapshot is usually supported only for entire file system volumes, but

[1] Each specification indicates one or more corner cases where atomicity is not necessarily required. For example, POSIX requires that, if the destination currently exists, the destination name must continue to exist and point to either the old file or the new file throughout the operation.

some systems allow snapshots of particular subtrees of the directory hierarchy. In any case, it is clearly a substantial multi-item operation, with the expectation that the snapshot captures all covered files at a single point in time.

2.2 Transparent scalability

We categorize existing systems into three groups based on how fully they provide transparent scalability as the number of servers increases. Transparent scaling implies scaling without client applications having to be aware of how data is spread across servers; a distributed file system is not transparently scalable if client applications must be aware of capacity exhaustion of a single server or different semantics depending upon which servers hold accessed files.

No transparent scalability: Many distributed file systems, including those most widely deployed, do not scale transparently. NFS, CIFS, and AFS all have the property that file servers can be added, but each serves independent file systems (called *volumes*, in the case of AFS). A client can mount file systems from multiple file servers, but must cope with each server's limited capacity and the fact that multi-file operations (e.g., RENAME) are not atomic across servers.

We focus in this paper on distributed file systems that provide fairly strong consistency semantics. But, file systems that provide weaker consistency, such as eventual consistency with post-hoc conflict detection and application or user-assisted resolution (e.g., Bayou [50], Pangaea [46], and Ivy [40]), could also fit into this "no transparent scalability" category.

Transparent data scalability: An increasingly popular design principle is to separate metadata management (e.g., directories, quotas, data locations) from data storage [7, 23, 24, 51, 53]. The latter can be transparently scaled relatively easily, assuming all multi-object operations are handled by the metadata servers, since each data access is independent of the others. Clients interact with the metadata server for metadata activity and to discover the locations of data. They then access data directly at the appropriate data servers. Metadata semantics and policy management stay with the metadata server, permitting simple, centralized solutions. The metadata server can limit throughput, of course, but off-loading data accesses pushes the overall system's limit much higher [25]. To go beyond this point, the metadata service must also be scalable.

Most modern storage systems designed to be scalable fall into this category. Most are implemented initially with a single metadata server, for simplicity. Examples include Google FS [23], NASD [24], Panasas [53], Lustre [38], the original version of Ursa Minor [2], and most SAN file systems. These systems are frequently extended to support multiple metadata servers, each exporting a distinct portion of the namespace, and the ability to dynamically migrate files from one metadata server to another. Such a solution, however, is not transparently scalable because clients see different semantics for operations that cross metadata server boundaries.

Full transparent scalability: A few distributed file systems offer full transparent scalability, including Farsite [3], GPFS [47], Frangipani [51], and recent versions of Ursa Minor [48] . Most use the data scaling architecture above, separating data storage from metadata management. Then, they add protocols for handling metadata operations that span metadata servers. Section 2.3 discusses these further.

Another way to achieve transparent scalability is to use a virtualization appliance with a collection of independent NFS or CIFS file servers. Examples of such "file switches" include Anypoint [54], Mirage [8], Cuckoo [33], and Katsurashima et al.'s "NAS switch" [31]. The file switch aggregates an ensemble of file servers into a single virtual server by interposing on and redirecting client requests appropriately. In the case of multi-server operations, the file switch serves as a central point for serialized processing and consistency maintenance, much as a disk array controller does for a collection of disks. Thus, the virtual server remains a centralized, but much more capable, file system.

2.3 Multi-server operations

Existing systems that support multi-item operations that span server boundaries use one of two approaches:

- One server acquires a lock on all items and executes the entire operation.
- Execute a portion of the operation on each server, using a distributed transaction protocol.

GPFS [47] and Frangipani [51] are examples of a systems that uses the first approach. All of the servers in a cluster are attached to all the underlying storage through a common SAN. When an operation executing on a particular server requires multiple objects (or parts of an object), it acquires a lock on all of those objects through a core distributed lock management service. Once the necessary locks have been acquired, the operation can proceed, knowing that that no other server will try to modify the locked state.

The action of acquiring a lock may require the current lock-holder to perform certain activities before relinquishing the lock. For example, servers are allowed to buffer updates in their in-memory caches and private transaction logs. Because the next lock-holder expects the state of an object on the underlying storage to be consistent, the server relinquishing the lock will have to first flush any relevant dirty state to the shared disk.

Farsite [13] and Slice [6], on the other hand, use a distributed transaction protocol. Each of the servers holding items affected by a transaction performs the portion of the transaction that pertains to the items it is responsible for. One of these servers functions as a leader and determines whether the transaction should commit on all servers or must be rolled back on all servers, as described in section 2.4. The inter-server coordination required for a RENAME operation in Farsite requires on average 12 server-server messages in the worst case.

2.4 Distributed transactions

Traditionally, cross-server operations are implemented using a distributed transaction protocol, such as a two-phase commit [26]. Since each server already must implement atomic single-server operations, usually by using write-ahead logging and rollback, the distributed transaction system can be built on top of the local transaction system. A transaction affecting two servers would first add a “prepare” entry to both of the logs, covering the update of the respective data items. If both servers successfully “prepared”, the transaction is finalized with a “commit” entry to both logs. If the “prepare” did not succeed on all servers, each server must examine its log and roll back its state to that of the beginning of the transaction. Crash recovery, however, is now much more complicated: with single-server transactions, it is sufficient to examine the log and undo any incomplete transactions. With more than one server, it is possible for some servers to crash and others survive. If one crashed while “preparing”, all servers must rollback that transaction, as mentioned before. If one crashed between the “prepare” and the “commit”, the recovering server does not know if the “prepare” succeeded on all other servers. By examining the logs of the other servers, the recovering server can determine if the ”commit” appears in any log, in which case it should be added to the recovering log, or if it did not commit and should be undone locally. Any step involving communication with other servers may fail, and if other servers have crashed, it may not be possible to proceed until they are online as well.

Distributed transactions may complicate other aspects of the system as well. Consider a simple operation that reads and updates two items, each on different servers. In order to prevent a single-server operation on either server from modifying one of the items between the read and write phase of the cross-server operation, the server executing the transaction must lock both items for the duration of the transaction. This must also be done for a single-server transaction, but all potential contention is local to a single server. A lock held by a different server introduces the potential for the lock holder to crash independently of the server managing the lock. While there are many existing techniques, such as leases, to handle this situation, a lock recovery scheme is simply not needed when locks can only be local to a server. Handling non-crash faults, such as intermittent networks or Byzantine servers, adds far more complexity to any distributed protocol [13].

As can be seen, most of the additional complexity is in the recovery paths. Not only must the recovery path handle recovery from a wide variety errors or crashes, it must also handle errors during recovery. This leads to a large number of cases that must be detected and handled correctly. Since errors in general are rare, any particular error is even rarer, which means bugs in the fault-handling path may be triggered rarely and be even harder to reproduce. This places more reliance on test harnesses, which must be crafted to exercise each of the many error conditions and combinations thereof.

2.5 Migration

In any system with many servers, the question arises as to which files should be assigned to which servers. Some systems, such as AFS, NFS, Panasas, and Lustre, split the file system namespace into several *volumes* and assign each metadata server one or more volumes whose boundaries cannot be changed after creation. Others, such as xFS, Ceph, and OntapGX, are able to assign individual files to distinct servers. In general, supporting finer granularities requires more complexity in the mechanism that maps files to metadata servers.

Managing large-scale storage systems would be very difficult without migration — at the very least, hardware replacement and growth must be accounted for. Additionally, migration is a useful tool for addressing load or capacity imbalances. Almost every storage system has some way of performing migration, in the worst case by backing up data on the original server, deleting it, and restoring on the destination server.

Such offline migration, however, is obtrusive to clients, which will notice periods of data unavailability. If the need for migration is rare, it can be scheduled to happen during announced maintenance periods. As a system gets larger, the need for migration increases, while the tolerance for outages decreases. To address this issue, many modern systems [14, 30, 52, 53] can perform migration dynamically, while serving client requests, leaving clients unaffected except for very brief periods of unavailability. Dynamic migration involves 3 main steps :

- The source server must stop serving the items to be moved
- The responsibility for the items must be moved to the destination
- The destination server must start serving the items

The complete migration process must be atomic—in the face of any crash failure, either the source or destination must be responsible for and able to serve the items in question, and, at all times, all relevant parties agree on which server is responsible for which item. A non-atomic migration raises the possibility of items disappearing forever or different versions of the same item being available to clients. As this is clearly undesirable, most systems implement migration using a simplified form of distributed transaction [13, 52, 30]. The presence of a authoritative mechanism to maps items to servers simplifies the task because it provides a central point of coordination, instead of having to rely on distributed logs.

The middle step, that of transferring responsibilty for the items from source to destination servers, can take different amounts of time depending on what state must be moved between servers. In the simplest case, where all hard state resides on shared storage, no state needs to be transferred between servers other than a token indicating responsibility. Ursa Minor [48] and Slice [6] are examples of such systems. In other systems that use shared storage, such as GPFS [47], Frangipani [51], and Ceph [52], servers may still maintain some local state, such as locks, local transaction logs, and client callbacks and capabilities. In GPFS and Frangipani, local logs are truncated, ensuring the state in shared storage is up-to-date. Ceph, on the other hand, tranfers the source server's logs and client capabilities directly to the destination server. GPFS uses both techniques, shipping small items of state directly to the destination and sending large state transfers through the underlying shared storage.

In systems that utilize private storage, the transfer of responsibility must also transfer the actual state being migrated. The common approach, taken by AFS [30] and OntapGX [14], is to take a snapshot of the state being migrated and transfer the snapshot to the destination. As an optimization, the source server is permitted to continue servicing operations—when the initial snapshot has been transferred, a new snapshot, encompassing changes since the initial snapshot, is taken and transferred to the destination. This cycle repeats until the Nth snapshot is small enough to be transferred while the source server is not serving client requests. At this point, the destination is completely up-to-date and can begin serving clients.

Any system that provides dynamic migration would be able to utilize migration to provide transparent scalability. The latency of migration would, however, be directly added to the latency of every multi-server operation that required a migration. Thus, the faster a system can perform migration, the better its performance will be when using migration to support multi-server operations. Since the major factor influencing the latency of migration is the amount of state to be migrated, systems with small amounts of local state are most amenable to using migration for transparent scalability.

The process of assigning files to servers can be thought of as analogous to lock managment. A server assigned responsibility for a file (or collection of files) has effectively been granted an exclusive lock on that file and migration changes the ownership of that lock. Given the relative rarity and granularity of migration, the centralized migration managers used in AFS [30] and OntapGX [14] need not be as efficient or complex as the distributed lock managers used for fine-grained locking in GPFS [47], Slice [5], and Frangipani [51].

Chapter 3

Trace analysis

To understand how often multi-file operations occur in practice, two sets of well-studied distributed file system traces were examined. These traces are those of three NFS file servers at Harvard collected by Ellard et al. [15] and discussed in Section 3.1, and two CIFS filers at NetApp collected by Leung et al. [37] and discussed in Section 3.2. For comparison, the NFS workload generated by an industry-standard benchmark for evaluating the performance of NFS file servers, SPECsfs97, is also discussed in Section 3.1.

Because of the different network protocols, each set included slightly different information. For the purposes of determining how common multi-server operations were, two properties of each trace are relevant: the percentage of multi-file operations in the trace and the "distance" between the files affected by each multi-file operation. Only multi-file operations can potentially involve multiple servers—single-file operations will always involve only one server. The most common scheme for partitioning files across servers is to assign one or more subtrees of the file system namespace to each server. If the files affected by a multi-file operation both lie close to each other in the namespace, they are likely to fall within the same subtree and, thus, both be assigned to the same server. Files further apart in the namespace are correspondingly more likely to be in subtrees served by different servers.

A metric that captures distance in namespace is the number of links in the directory tree that must be traversed on the shortest path between the two files of interest. For example, `/a/b` and `/a/c` would be have a distance of two. The distance between `/a/b` and `/c/d` would be four, because of the two hops from `/a/b` to the root and two hops from the root to `/c/d`. Multi-file operations with small distances are likely to be single-server, whereas operations with large distances are more likely to be multi-server.

3.1 NFS traces

The NFS traces examined are of three departmental NFS servers at Harvard University. The workloads of these servers varied significantly and are described below. Also considered is the workload generated by the SPECsfs97 benchmark, which is the industry standard for measuring the performance of NFS file servers.

EECS03: The EECS03 trace captures NFS traffic observed at a Network Appliance filer between February 2^{nd}–28^{th}, 2003. This filer served home directories for the Electrical Engineering and Computer Science Department. It served an engineering workload of research, software development, course work, and WWW traffic. Detailed characterization of the EECS03 trace can be found in [18].

DEAS03: The DEAS03 trace captures NFS traffic observed at another Network Appliance filer between February 1^{st}–28^{th}, 2003. This filer served the home directories of the Department of Engineering and Applied Sciences. It served a heterogenous workload of research and development *combined with* e-mail and a small amount of WWW traffic. The workload seen in the DEAS03 trace can be best described as a combination of that seen in the EECS03 trace and e-mail traffic. Detailed characterization of the DEAS03 trace can be found in [17] and [18].

Campus: The Campus trace captures a subset of the NFS traffic observed by the Campus storage system between October 1^{st}–31^{st}, 2001. The Campus storage system provided storage for the e-mail, web, and computing activities of 10,000 students, staff, and faculty via fourteen 53 GB storage disk arrays. The subset of activity captured in the Campus trace includes only the traffic between one of the disk arrays (home02) and the general e-mail and login servers. NFS traffic generated by serving web pages or by students working on CS assignments is not included. Despite the these exclusions, the Campus trace contains more operations per day (on average) than either the EECS03 or DEAS03 trace. Detailed characterization of the Campus trace can be found in [16] and [17].

SPECsfs: The SPECsfs97 benchmark is based on a survey of workloads seen by the “typical” NFS server [49]. It consists of a number of client threads, each of which emits NFS requests for file and directory operations according to an internal operation probability model. The benchmark programmatically generates an initial filesystem state during its setup phase. This initial namespace is both simple and uniform, with only 3 levels of directories, each identical.

3.1.1 Operations

In the NFS protocol, the potentially multi-file operations are:

- CREATE: creates a new file, affecting the new file and its parent directory.
- LINK: links an existing file to a new name, affecting the existing file and its new (additional) parent directory.
- MKDIR: creates a new directory, affecting the new directory and its parent directory.
- SYMLINK: creates a new symlink, affecting the new link and its parent directory.
- READDIRPLUS: reads the attributes for all files in a directory, requiring both the directory and all its children.
- REMOVE: deletes an existing file and its entry in its parent directory.
- RENAME: moves an existing file or directory to a new name, possibly in a different directory. It affects both source and destination directories, any file that may already exist with the destination name and must be deleted, and, if a directory is being renamed, the renamed directory itself.
- RMDIR: deletes an existing directory and its entry in its parent directory.

For all of these operations, except RENAME and LINK, the file(s) and directory involved are in a parent-child relationship and are thus almost always going to be on the same server in the same way that single-file operations are. In a RENAME, the destination directory can be anywhere in the namespace. The RENAME RPC includes enough information to determine whether it involves more than one directory, and cross-directory RENAMEs are counted separately from RENAMEs of a file to a different name in the same directory. Although the RENAME RPC does not contain enough information by itself to determine how far apart the source and destination directories are, this information can often be recovered by the method described in Section 3.1.2.

Similarly, for a LINK operation, the directory the new name is being inserted in can be any directory, not just the one the file was originally created in. Assuming that the file was assigned, when created, to the same server as its original parent, the destination directory may not be assigned to the same server. The contents of a LINK RPC often include enough information to infer whether the file was originally created in a different directory, but it is not always possible to determine the distance between the new name and the original name.

3.1.2 Reconstruction

Simply counting the number of single-file operations in a NFS trace is straightforward. Determining whether a RENAME is single-directory or cross-directory is similarly straightforward. Determining the distance between the source and destination directories of a RENAME or LINK, and if a LINK is cross-directory, are much more difficult.

The reason for this difficulty is that most NFS operations address a file (or directory) by its *filehandle* and not by its pathname. A filehandle uniquely and persistently identifies a file in the exported file system. A RENAME operation, for instance, will include the source and destination filehandles, which can be compared to determine if they are the same or not. But, these filehandles provide no information about where in the directory tree the source and destination directories lie.

The NFS protocol assumes that the NFS client will perform all pathname traversals. Thus the NFS requests, as in the RENAME example, only need to include the filehandle of the last element in the path. The LOOKUP operation is used to translate a parent filehandle and a child name into the child's filehandle. Every time a client attempts to access a path that it has not seen before, it will issue a sequence of LOOKUPs beginning from the root and progressing all the way down the path. The client may cache the results of LOOKUPs, and the trace may not not include every packet, so the chain of LOOKUPs seen in the trace may not be complete. Other operations, such as MKDIR and CREATE return the filehandle of the newly created object. Still other operations, such as READDIR, return a list of names that exist, but do not specify their corresponding filehandles. Although the simple listing of names would be useful, the trace does not include the responses to READDIRs.

Nevertheless, by observing the known name, filehandles, and parent-child relationships during the course of the trace, it is possible to reconstruct most of the elements of the directory tree that *must* have existed in order for the trace to be valid. At any point in the trace, this reconstructed file system will include all the files referenced (but not deleted) in the trace up to that point. Any files that existed in the original file system, but have not yet been referenced or listed, will remain unseen. Because of this, it is quite likely that some (busy) subtree could be completely known, but that subtree's connection to the root cannot be determined yet. As subsequent operations reveal more information, it may become possible to connect it to its proper place in the namespace. Additionally, if a directory is referenced in the trace, but none of its children were accessed, none of the children will be visible. Because the READDIR responses do not provide any information on the number of entries in a directory, the number of unreferenced children cannot be determined. Similarly, because the responses to FSSTAT are not included in the trace, the total size of the original filesystem (and, thus, the number of unreferenced files) cannot be determined.

Given the reconstructed namespace hierarchy, the method used for determining the distance between the two files in a multi-file operation (the source and destination directories of a RENAME or the original and new directories of a LINK) works as follows: When a multi-file operation is encountered, the path is recursively followed upwards from each of the files in question. When the paths intersect a common ancestor, the sum of the numbers of links followed is counted as the distance between the files. If one traversal ends in an ancestor that has not yet been seen, then the distance is unknown. If subsequent operations reveal the common ancestor, previously unknown distances are not reevaluated, but future operations involving the newly discovered ancestor will have known distances.

3.1.3 Results

Figure 3.1 shows the distribution of NFS operations for each of the workloads. Single-object operations accounted for at least 98% of every workload. Of the remaining (multi-object) operations, the vast majority (e.g. CREATE) involved a parent directory and one of its children and are, thus, likely to involve the same server. Only a small fraction (between 0.004% and 0.010%) of NFS operations involved a RENAME between two directories or a LINK of a file to a new directory, and, for all of these, one of the two directories was the parent, child, or sibling of the other. Siblings (e.g. `mv /a/foo /b/foo`) were more common than the other two cases (`mv /a/foo /a/b/foo` or `mv /a/b/foo /a/foo`). These cases are also likely to involve

	EECS		DEAS		Campus		SpecSFS
Total operations	163,755,512	100.0%	789,104,790	100.0%	585,730,355	100.0%	100.0%
Single file or dir	160,811,696	98.2%	782,347,892	99.1%	579,906,161	99.0%	98.0%
GETATTR	16,836,393	10.3%	197,103,711	25.0%	13,475,241	2.30%	11.0%
SETATTR	5,385,233	3.29%	4,429,365	0.561%	7,649,003	1.31%	1.0%
ACCESS	36,554,813	22.3%	12,050,877	1.53%	16,517,405	2.28%	7.0%
READ	31,580,501	19.3%	409,879,410	51.9%	383,368,210	65.5%	18.0%
WRITE	18,497,209	11.3%	124,319,586	15.8%	123,753,745	21.1%	9.0%
READDIR	1,529,783	0.934%	2,641,479	0.335%	2,257,374	0.385%	18.0%
LOOKUP	50,407,746	30.7%	31,838,772	4.04%	34,876,085	5.95%	27.0%
Other	18,822	0.011%	84,692	0.011%	266,472	0.045%	7.0%
Parent & child	2,930,861	1.79%	7,543,510	0.956%	5,779,208	0.986%	2.0%
CREATE	1,359,889	0.830%	1,931,159	0.245%	1,496,054	0.255%	1.0%
REMOVE	1,092,608	0.667%	3,280,382	0.416%	2,897,888	0.495%	1.0%
MKDIR	38,835	0.024%	15,408	0.002%	616	$<0.001\%$	0.0%
RMDIR	39,197	0.024%	112,824	0.014%	716	$<0.001\%$	0.0%
LINK	309,589	0.189%	965,300	0.122%	1,422,781	0.243%	0.0%
RENAME	90,743	0.055%	174,406	0.022%	6,132	$<0.001\%$	0.0%
File & 2 nearby dirs	7,110	0.004%	75,122	0.010%	44,946	0.008%	0.0%
LINK	5,547	0.003%	39,235	0.005%	$\star$ 44,684	0.008%	0.0%
RENAME	2,176	0.001%	35,887	0.005%	262	$<0.001\%$	0.0%
File & 2 distant dirs	619	$<0.001\%$	616	$<0.001\%$	33	$<0.001\%$	0.0%
LINK	6	$<0.001\%$	63	$<0.001\%$	0	0.000%	0.0%
RENAME	613	$<0.001\%$	553	$<0.001\%$	33	$<0.001\%$	0.0%

Table 3.1: NFS operation breakdowns for the Harvard traces. The number and percentage of each type of operation in each NFS trace are shown, along with the percentage of such operations in the SpecSFS97 benchmark. Operations are divided into four categories: those that affect only a single file or directory, those that affect a parent directory and one of its children, those that affect a file and two nearby directories, and those that affect a file and two distant directories. Two directories are considered to be ”nearby” if one is a parent, child, or sibling of the other. Additionally, LINKS for which the original directory of file being LINKed is unknown are classified as involving a nearby directory—this the source of all nearby LINKS in the Campus trace. Two directories are considered to be distant if they are not adjacent or if the distance between them cannot be determined. Related NFS operations are grouped together under one label: “READDIR” also includes READDIRPLUS and READLINK, and “GETATTR” also includes FSSTAT. “Other” includes MKNOD, NULL, PATHCONF, ROOT, STATFS, and WRITECACHE.

the same server. Multi-object operations involving directories further apart in the namespace accounted for just 4 out of every 1,000,000 operations in the EECS trace and less than 1 in 1,000,000 operations in every other trace.

It follows that a transparently scalable storage system faced with any of these NFS workloads would experience at most a 0.010% rate of multi-server operations (if every multi-object RENAME or LINK was multi-server), and more likely an order of magnitude or two lower. The relative infrequency of these operations suggests that a system would only suffer a small performance penalty from a high overhead approach to handling multi-server operations.

3.2 CIFS traces

The second set of traces are of CIFS traffic to two enterprise-class filers in NetApp's corporate data center [37]. One was used by their engineering department and the other by the their marketing, sales, and finance departments.

The operations in the CIFS protocol are similar in relevant aspects to those in the NFS protocol, except that there is no LINK operation. Fortunately, the equivalent of the RENAME RPC includes the path of the source and destination directories, so it is always possible to determine not only that the the directories are different, but also how far apart the source and destination directories are in the directory tree. Additionally, the traces include part of the server's response to READDIR requests. This information means that the number of children (at the point in time the READDIR RPC was executed) will always be known accurately, and that the names of some of the never-referenced children will also be known.

3.2.1 Reconstruction

Reconstructing the possible file system state at the end of a CIFS trace is much easier than for NFS. All relevant CIFS operations include the full path of the file or directory being operated on. Thus, reconstruction is actually unnecessary for purpose of counting operations, although it will be necessary for the analysis in Section 4.5.

We built a reconstruction tool that takes a CIFS trace and produces a directory tree, on a local file system, that includes the files and directories known to exist in the trace. When a file or directory is created or deleted in the trace, it is created or deleted in the local file system. If a file is opened, read, or written in the trace, it is assumed to exist and is created at that point. When a READDIR returns (part of) the contents of a directory, those files are also assumed to exist and is created. For names only seen in a directory listing, it is unknown whether they are files or directories; for names learned from other RPCs, it can be inferred from the RPC whether the object in question is a file or a directory. Although READDIR provides the child count of a directory, because subsequent CREATE and DELETE operation may add or remove files from the directory, the child may not be accurate in the future. Rather than trying to account for each operation's effect on the child count, the child count returned by READDIR is ignored, which makes the CIFS reconstruction consistent with NFS reconstruction.

3.2.2 Results

Figure 3.2 shows the operation distribution provided by Leung et al. [37] for the Engineering and Corporate traces. Additionally, we were able to obtain a segment of the Corporate trace; its distribution is also shown and differs somewhat from that of the overall Corporate trace. In particular, it has a higher percentage of READs and WRITEs than either complete trace and has more CLOSEes than OPENS. Some of this discrepancy may be due to differences in how CREATEs and OPENs are accounted for. CREATEs in CIFS may happen explicitly, as a result of a CREATE RPC, or implicitly, as a result of a OPEN of a non-existent file. Conversely, files can be OPENed explicitly by RPC and implicitly as a result of a CREATE. Our results for the CIFS traces consider all OPENs to be CREATES, which is a worst-case assumption.

Like the NFS traces, the vast majority of operations are single-object. Except for CREATEs, which may really be OPENs, the percentages of potentially cross-directory RENAMEs (0.04%-0.08%) are similar to those of the NFS traces.

We were able to calculate RENAME distances for operations within the short segment of the Corporate trace. Of the 193 cross-directory RENAMEs we found out of the 12.5 million CIFS operations in the segment, 84% had a destination directory that was either the immediate parent, child, or sibling of the source directory. The remaining 31 "distant" RENAMES correspond to a multi-server operation rate of 2 per 1,000,000, which is within the range of those observed in the NFS traces and the CIFS workload used to evaluate Farsite [12]. The anonymization scheme used for these traces intentionally preserved filename extensions; examining a sample of cross-directory RENAMEs suggests many of them are caused by programs that create a file in a

	Engineering		Corporate	
Total operations	120,077,019	100.0%	144,751,276	100.0%
Single file or dir	103,226,846	86.0%	137,456,558	87.0%
GETATTR	49,882,755	41.5%	43,551,680	38.7%
SETATTR	919,664	0.77%	2,910,332	2.1%
READ	20,523,242	17.1%	28,482,816	19.7%
WRITE	8,912,434	7.4%	8,656,251	6.0%
READDIR	11,584,693	9.6%	16,022,330	11.1%
OPEN∗	0	0.0%	0	0.0%
CLOSE	8,422,985	7.0%	6,777,623	4.7%
Other	2,981,073	2.5%	7,177,940	5.0%
Parent & child	16,850,173	14.0%	18,770,453	13.0%
CREATE∗	16,806,651	14.0%	18,653,751	12.9%
DELETE	11,483	0.021%	54,161	0.037%
RENAME	31,601	0.027%	87,269	0.043%
File & 2 nearby dirs	667	<0.001%	3,562	0.001%
RENAME	667	< 0.001%	3,562	0.001%
File & 2 distant dirs	177	<0.001%	1,956	<0.001%
RENAME	177	<0.001%	1,956	<0.001%

Table 3.2: CIFS operation breakdowns for the NetApp traces. The number and percentage of each type of operation in each trace are shown. Operations are divided into four categories: those that affect only a singly file or directory, those that affect a parent directory and one of its children, those that affect a file and two nearby directories, and those that affect a file and two distant directories. Two directories are considered to be nearby if one is a parent, child, or sibling of the other. Otherwise they are considered to be distant. CREATE and OPEN operations cannot easily be distinguished from each other, therefore they are counted as CREATES. Related CIFS operations are grouped together under one label: "READDIR" consists of FINDFIRST2 and FINDNEXT2, "GETATTR" consists of QUERYFILEINFO and QUERYPATHINFO, and "Other" consists of FLUSH, LOCK and all session and pipe management operations.

temporary directory and move it to the final location.

3.3 Conclusion

Examining traces of real-world workloads reveals that the worst-case occurrence of potentially multi-server operations is low (2%). If files are assigned to servers in a manner that keeps subtrees together, the worst-case is far lower (0.01%) and the expected case is one in a million. As long as subtrees can be preserved most of the time, a system that uses migration to implement multi-server operations would only incur the penalty of migration extremely rarely. Thus, even a substantial migration penalty would only result in a small overall slowdown.

Chapter 4

Object-ID assignment

In many distributed file systems [5, 9, 13, 14, 30, 47], including the prototype described in Chapter 5, every object in the system is assigned a unique identifier and an object's identifier has a role in determining which server hosts that object. If files that were likely to be involved in multi-object operations together tended to be hosted by the same server, then the number of multi-server operations would be much smaller than the number of multi-object operations. This chapter discusses methods for assigning object identifiers so that files likely to be involved in multi-object operations usually are hosted by the same server.

4.1 Overview

In any system with multiple servers, the question arises as to which files should be assigned to which servers. If every file was independent and had an identical workload, the dominant concern would be to ensure every server ended up with an equal number of files—with a uniform workload, the load and capacity utilization of each server would be the same. Most deployed systems, however, must handle workloads that do not exhibit these ideal properties.

First, actual file system workloads exhibit both spatial and temporal locality. Accesses to files in one directory are frequently correlated with accesses to other files in that same directory, and a recently accessed file is likely to be accessed again in the near future. Second, the initial access to a file requires an access to its parent directory; an access to a directory makes an access of its children more likely. Furthermore, many directory operations (CREATE, LINK) operate on both a parent directory and a child file's inode at the same time. If the parent and child are assigned to different servers, the operation would involve both servers.

Since multi-server operations are more difficult than single-server ones, it would be advantageous to minimize the number of multi-server operations by ensuring that a file was always assigned to the same server as its parent. Of course, such a scheme would result in every file residing on just one server. To avoid this, systems such as Farsite [13], Ontap GX [14] and Ceph [52] will partition the directory tree into multiple subtrees and assign each server one or more subtrees. All operations within a contiguous subtree will be local to one server, but operations that cross the boundary between subtrees will involve multiple servers.

A danger with a subtree partitioning scheme is that determining which server is responsible for a file requires knowing that file's place in the directory hierarchy, which may require traversing the directory path between the root and the file in question. Yet, file systems generally have a unique identifier for each file (called a FileID, filehandle, inode number, or Object-ID depending on the system; *OID* will be used here), and once the file name has been resolved to an OID, subsequent accesses use only the OID instead of the full pathname. But, the OID by itself does not convey any information about which server is responsible for it; only the initial path traversal does. If the operation includes the OID but not the full path name, as most NFS operations do, the client must cache the OID-to-name and name-to-server mappings revealed by the original lookup in order to know which server to issue the request to. If responsibility for a subtree is migrated to a different server, the clients must repeat the (expensive) path traversal to determine the new correct server. Some systems speed this process up by using an additional pathname-to-server mapping service [52].

Other systems, such as AFS [30], use the most significant bits of their OID to signify which *volume* a file is in. Entire volumes are assigned to servers as a unit. Given just the OID, a client can determine which volume that file resides in, check whether it already knows which server is responsible for that volume, and if not, fetch that information from a volume location service. The volume location service is a relatively simple key-value store, where they key is the fixed-size volume ID. and the value is the identity of the server responsible for that volume. Directory operations within a volume will all be single-server, but those crossing a volume boundary may by multi-server (and, in AFS, not performed atomically). Once a file has been created in a volume, it cannot be migrated to a different volume, and a volume cannot generally be split into smaller sub-volumes, which limits the flexibility the system has for performing load-balancing.

It would be desirable to combine the locality and variable sizes supported by subtree partitioning schemes with the simplicity and one-step OID-to-server translation of volume-based schemes. Our approach to accomplishing this is with a *namespace flattening* policy that encodes the hierarchy of the file system namespace into the value of the OID. Like a volume-based scheme, the responsible server can be identified by examining a prefix of the OID, though the length of the prefix can now be variable instead of fixed. Varying the length of the prefix (and thus how many bits remain to identify a file under that prefix) allows for differently sized subtrees. If the OID is generated appropriately, once an OID has been assigned, changing the length of the prefix will result in splitting one “volume” into two smaller units, each with locality within it.

Many namespace flattening algorithms are possible; this chapter presents two such algorithms, *child-closest* and *cousin-closest*, discusses some difficulties a namespace flattening algorithm may encounter and their solutions, and evaluates the effectiveness of namespace flattening at representing the traced workloads described in Chapter 3.

4.2 Child-closest policy

The child-closest policy, as the name suggests, aims to assign OIDs to the children of a directory that are similar to the parent directory’s OID. The children will also have OIDs that are similar to each other. This policy functions as follows:

First, the OID is divided bitwise into a *directory segment* and a *file segment*. The directory segment is further subdivided into a number of *directory slots*. Each slot corresponds to a level in the directory hierarchy, and the value in a slot identifies that directory within its parent. The root directory uses the most significant slot, each of its children the next most significant, and so on. When creating a new directory, the child’s directory segment is copied from its parent, with a new value chosen for the most significant empty directory slot. Figure 4.1 shows the correspondence between elements of example hierarchy and slots within an OID.

The file segment is a simple sequential counter for files created in that directory. A directory itself has a file segment of all 0s. The first child file of that directory has the same directory segment, but file segment of 1. The second has file segment of 2 and so on. Figure 4.2 shows an example directory tree and the OID that the child-closest policy assigns to each file or directory in the tree, using a dot notation (akin to that used for IP addresses) to represent the directory and file slots.

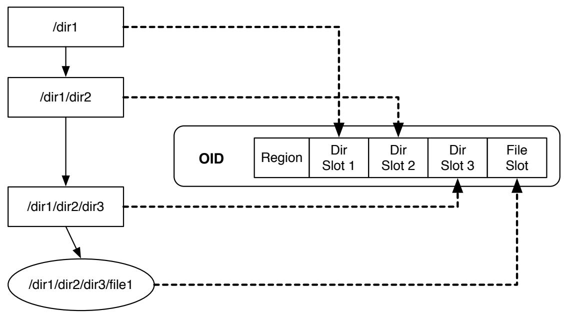


Figure 4.1: File and directory slots. An example file system path and OID are shown. The OID is segmented into a number of directory slots and a file slot. The least significant bits of the OID are to the right and the most significant are to the left. The correspondence between elements in the path and slots in the OID are shown by dotted lines for the child-closest policy.

The child-closest policy has the convenient property that the most significant bits of the OID identify a directory and the subtree below it. In the example in Figure 4.2, an OID mask of 1.2.x.x would specify the subtree beginning with `/dir1/dir2`. Additionally, `/dir1/dir2/file1` and `/dir1/dir2/file2` and so on will have consecutive OIDs, making it likely that their entries will be close to each other in whatever OID index structures the system uses.

This policy is very similar to that used in Farsite, except that the Farsite FileID is variable-length and grows with with directory depth [13]. Supporting variable-length object identifiers significantly complicates the the protocols and system components that must handle OIDs, making a fixed-sized OID preferable. But, with a fixed-size OID, the namespace may have both more levels than there are directory slots and more files in a directory than can be represented in the file segment bits. This condition is termed *overflow*.

To accommodate overflow, 2 prefix bits are used to further split the OID into 4 regions. The first, *primary*, region uses the assignment policy above. If the hierarchy grows too deep, the too-deep child directory is assigned a new top-level directory slot with a different prefix (the *too-deep* region). Its children grow downwards from there, as before.

If there are too many files in a directory, and the next directory slot value is unused, the large directory takes over the OIDs reserved for its nonexistent sibling; the new file is assigned an OID that would be used by its nonexistent cousin. If cousins already exist, the new file is assigned an OID from the *too-wide* region. Within this region, fewer bits are allocated to the directory segment, and more to the file segment, so

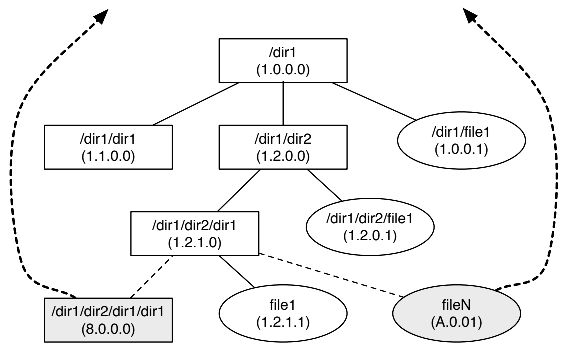


Figure 4.2: Child-closest OID assignment policy. The OID chosen for each element of this simple directory tree is shown. For clarity the example uses a 16 bit OID and a "." is used to separate the value of each directory slot and file segment. The straight dashed lines show a too-deep directory overflowing to a new "root" and a file in a large directory overflowing to the *too-wide* region. The curved dashed lines point to the logical "parent" that the overflowed objects are assigned OIDs under.

more files per directory can be handled. Finally, if either of these additional regions overflow, the *catch-all* prefix is used, and OIDs are assigned sequentially from this region. No namespace flattening is done in the catch-all region—it is a fall-back only.

In the case of any overflow, the additional children are effectively created under new "roots" and thus have very different OIDs from their parents. However, those children will still have locality with their own children (the parent's grandchildren). Thus, one large subtree will be split into two widely separated subtrees, each with locality within itself. If the two subtrees are both large enough, the loss of locality at the boundary between subtrees should not have a significant effect, because most operations will be local to one subtree or the other.

4.3 Cousin-closest

The child-closest policy suffers from the problem that, while child file OIDs are numerically consecutive with their parent directory's OID, the OIDs of child directories are not consecutive with the parent directory or other child files. Instead, they fall within an N-bit bitmask of the parent directory, where N is the number of bits in a directory slot. In the example in Figure 4.2, directories 1.1.0.0 and 1.2.0.0 differ only in 1.x.0.0 but are not consecutive—the 1.1.x.x OIDs lie between them.

These intervening OIDs may either be unused, in the case of empty directories, or be occupied by

1.1.0.0's descendents. If the system maintained indices sorted by OID, these descendents would occupy the intervening entries in the index. Since common operations, such as an NFS3READDIRPLUS involve accessing every child, both file and directory, it would be convenient if all the children of a directory were numbered closer together. Additionally, eliminating sparseness in the OID space is useful for features such as bulk capabilities and metadata prefetching [27].

The cousin-closest policy aims to address these issues. It uses the same file and directory slot partitioning as the child-closest policy, but it constructs the directory slot values in a different manner. The root directory uses the *least* significant directory slot. A child directory's OID is constructed by shifting its parent's OID one slot to the left and inserting a new value into the least significant directory slot. File numbers are assigned within a directory the same as with the child-closest policy and overflow is handled the same as well.

The application of this policy to the same simple subtree example is shown in Figure 4.3. The net effect is that directories at the same level of the hierarchy (cousins) will be numerically close to each other—the only intervening OIDs will be those of their child files. Child directories will not be close to their parent directory's OID either numerically or in bitmask. Thus, while a prefix of the OID still identifies a directory, it does not capture that directory's children. For this reason, if the cousin-closest policy were used in a system that partitions responsibility between servers based on OID, the number of multi-server operations would be much higher than with the child-closest policy. Therefore, the rest of this chapter focuses on the child-closest policy.

4.4 Optimizations

The previous examples assumed that the bit-widths of the directory and file slots were constant within each region, but this is only a simplification. By analogy to IP subnetting, the the too-wide, normal, and too-deep regions correspond to Class A, B and C networks. Similarly to how CIDR allows variable-size IP subnetworks [20], the namespace flattening polices could use variable-size directory and file slots.

The bit-widths of the directory slot in one OID need not be the same as those in another OID, nor do the widths need to be the same even within a single OID. If a directory is expected to have more subdirectories than average, it could use a wider directory slot for its children. If it is expected to have fewer than average, it could use fewer bits. Such a scheme would require the flattening policy to decide, when the parent

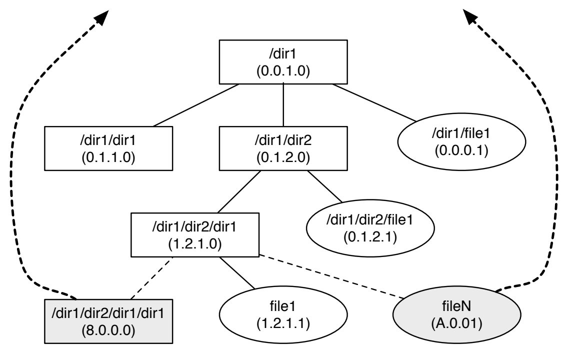


Figure 4.3: Cousin-closest OID assignment policy. The OID chosen for each element of this simple directory tree is shown. For clarity the example uses a 16 bit OID and a "." is used to separate the value of each directory slot and file segment. The straight dashed lines show a too-deep directory overflowing to a new "root" and a file in a large directory overflowing to the *too-wide* region. The curved dashed lines point to the logical "parent" that the overflowed objects are assigned OIDs under.

directory is created, the number of bits to allocate for its child directories. For every subsequent creation of a child directory, the flattening policy would then have to know how many bits it was allowed to use. Thus, the number of bits allocated to the children must be recorded in the parent's metadata along with the total number of children (which must be recorded anyway). Storing and retrieving this extra information does not add much overhead, because the other fields in the parent directory's metadata must be read or modified when creating a new child.

The handling of overflow, as described previously, results in the overflowed children receiving OIDs with different most significant bits from those of their parents, which makes it likely that overflowed children will be served by a different server than their parents. Obviously, minimizing the occurrence of overflow will reduce the number of affected OIDs. But, when overflow does occur, it is possible to assign OIDs to the overflowed children that are closer to those of their parents, making it more likely both will be served by the same server. In addition to using the two global overflow regions, a number of local overflow regions could also be used. The overflow region "closest" to the original OID would be used first, and the more distant ones used only if the more suitable ones were full. One approach would be for a child that encounters overflow to try to allocate a new OID from its grandparent's OID space. Farsite uses such a policy when restricted to using fixed-size FileIDs. If overflow is rare, and only by a few children per directory, the overflowed children will have OIDs similar to their grandparent, which is still close to the OID of their

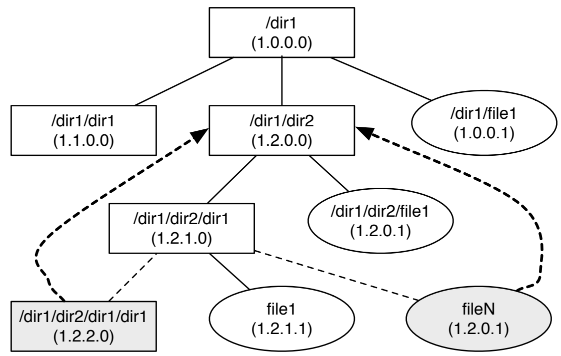


Figure 4.4: Overflow into parent rather than overflow region. This example shows the same tree from Figure 4.2, but with the too-deep subtree is overflowing into its parent directory's OID space. The curved dashed lines point to the logical "parent" that the overflowed objects are assigned OIDs under.

parent. If overflow is common, or one directory overflows by a large amount, the overflows will cascade back up the directory tree, resulting in the contents of many directories being jumbled up into the same OID space. This is particularly aggravated for very deep subtrees, which will be "squashed" into the OID space normally used by a single directory. Depending on how much of a subtree is assigned to each server, this may not affect the number of multi-server operations, but may affect the efficiency of each server. Figure 4.4 shows an example of this strategy.

It is also the case that not all levels of the directory hierarchy have the same meaning. Looking at AFS and NFS installations at CMU, the top few levels, by convention, identify which administrative unit and volume a particular file belongs to and may have far more entries than the average directory. For example, `/afs/ece/user/shafeeq` corresponds to a volume assigned to a single user or purpose, and `/afs/ece/user` has one entry for each of the thousands of users. AFS imposes the requirement that a volume fits entirely on one server and most volumes are small enough that a transparently scalable system would also assign all the files in a volume to the same server. Logically, multi-object operations within that volume are far more likely than ones involving a directory in that volume and a directory in some other volume. Therefore, as long as an overflow for a file within that volume would remain within that volume's OID space, it would not involve a multi-server operation.

One way to accomplish this, without the weaknesses of the Farsite policy, is to consider the volume-identifying portion of the namespace separately from the namespace within each volume. Some number

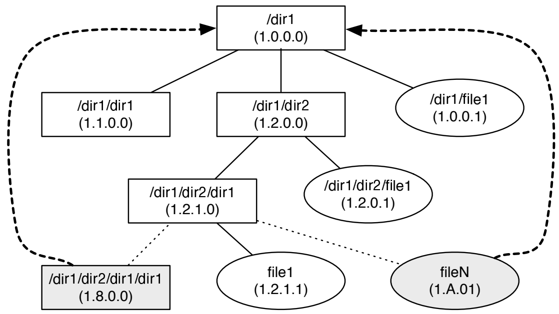


Figure 4.5: Overflow into volume-local overflow region. This example shows the same tree from Figure 4.2 but with a too-deep directory overflowing into the volume-local overflow OID space for `/dir1`. The curved dashed lines point to the logical "parent" that the overflowed objects are assigned OIDs under.

of the most significant bits of the OID can be reserved for a "volume identifier" and the remaining bits allocated according to the child-closest or cousin-closest policies. Each volume's OID space would have its own overflow region for handling any overflowed children; only when a volume's local overflow region was full would the global overflow region be used. Figure 4.5 shows an example of this optimization in action. As previously mentioned, the number of bits allocated for the "volume identifier" need not remain constant.

4.5 Evaluation

The goal of namespace flattening is to select OIDs that have locality corresponding to the directory hierarchy, thereby enabling the effective use of a simple OID-to-server mapping service. If the child-closest OID assignment policy successfully preserves the directory structure, then assigning a range of OIDs to each server will be equivalent to assigning a subtree to each server. If multi-object operations are concentrated within a subtree, as Section 3.1.3 shows, multi-server operations that cross subtrees, and thus servers, should be rare.

In evaluating how well namespace flattening captures a directory hierarchy, the metrics of interest are:

- How many OID bits are required to represent the entire namespace without overflow?
- With a fixed number of OID bits, how often does overflow occur?

The first metric corresponds to the maximum length of a Farsite-style variable-length OID, while the second metric corresponds that best that a system with fixed-length OIDs could do. With fixed-length OIDs, the exact choice of directory slot and file slot widths influences how often overflow occurs. The NFS and CIFS traces described in Chapter 3 were analyzed to determine the OID lengths required and overflow rates for the child-closest policy. Additionally, Douceur et al. performed a similar analysis on Windows local-disk file-systems [13]; it is summarized in Section 4.5.5.

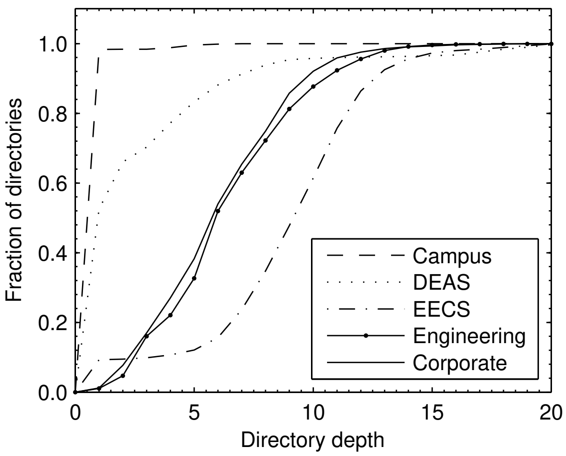
4.5.1 Methodology

For each of the NFS and CIFS traces, the file-system namespace was reconstructed from the trace as described in Sections 3.1.2 and 3.2.1. Once the original namespace was reconstructed, it was then analyzed to determine the following properties, which are plotted in Figure 4.6 and summarized in Table 4.1:

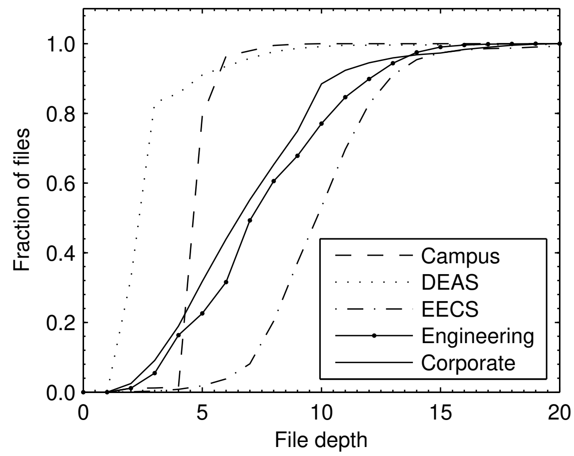
- The number of files in each directory.
- The number of directories in each directory.
- The depth of each file and directory.

These analyses are affected by the limitations of trace reconstruction. Only files and directories referenced in the trace are known, and, for directories that are seen, not all of the directory's children may be seen. Additionally, in NFS traces, it is possible for disconnected subtrees to exist if their parent was never referenced in the trace. These disconnected subtrees are treated as children of the root directory. The Campus trace, in particular, is distorted because almost half the directories in the trace have parents that are never seen.

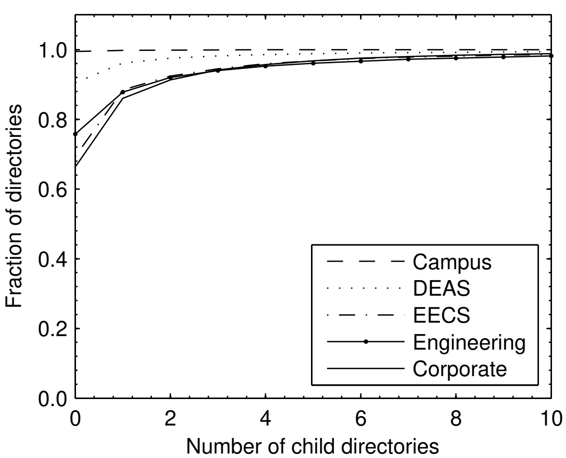
These issues will result in a reconstructed file system that is smaller than that of the original trace. Any subtree seen in the reconstructed file system will also exist in the original file system (though it may be misplaced in the case of NFS traces), but subtrees not referenced in the trace will simply be omitted. The effect of these omissions that the file and directory counts shown in this chapter represent a *lower bound* for the original file system. Since these parameters influence the number of bits required and the incidence of overflow, those values are also lower bounds.



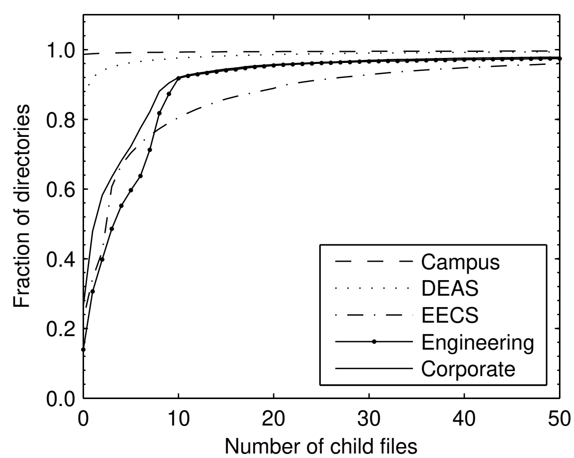
(a) Depth of directories.



(b) Depth of files.



(c) Number of directory children.



(d) Number of file children.

Figure 4.6: File and directory depths and sizes. CDFs are shown for the number of files or directories found at each level of the directory hierarchy reconstructed from each trace. For directory objects, the number of children of that directory are counted, and CDFs shown separatly for the number of child directories and child files.

Because the traces provide no information the size of the original file system, it is not possible to determine what fraction of the original file system is reconstructed. For this CIFS traces, comparing the total

OID slot	EECS	DEAS	Campus	Engineering	Corporate
Total objects	9558617	2871810	720265	2471650	2098232
Files	8856685	2490899	389633	1550067	1507386
Directories	700659	380911	330632	226438	253960
Roots	67215	204093	326504	1	1
Max depth	29	25	12	33	46
Max files in dir	36730	142513	11283	1162	4682
Max dirs in dir	822	3401	215	4501	2163
Max OID bits	109	91	56	153	113

Table 4.1: Trace properties. Properties of the directory hierarchy reconstructed from each trace are shown. The number of total objects is the number of directory entries known to exist in the trace. The number of directories is the number of objects known to be directories, and the number of files is the number of objects seen in the trace that are known to be files. These may sum to less than the total number of objects because of directory entries that are known to exist (from the size of the directory) but never seen in the trace (becuse of truncated or dropped packets). Almost al of these "missing" objects are children of directories whose childen are never accessed. The number of "roots" represents the number of directories that had a parent that was not seen in the trace. Thus, those directories cannot be connected to their proper place in the hierarchy and are instead considered to be isolated subtrees. Each of the roots is placed under a special top-level directory, but this top-level directory is not included in the totals. The "Max OID bits" represents the number of OID bits required to represent the hierarchy with ideally sized and numbered slots and no overflow.

number of directory entries in Figure 4.1 to the number of known files and directories shows that the reconstructed file system includes at least 72% to 84% of the directory entries that are returned during the trace. Since the distribution of file and directories among the "missing" entries is not known, they cannot be included in the analysis without assuming a distribution for them. Even if an appropriate distribution could be determined, the children of any of these missing, but assumed to exist, directories will also be unseen. Thus, including these directories in the analyses would artificially increase the number of empty directories. To avoid introducing these artificial distortions, the "missing" entries are not included in the analysis.

Analyzing the original file system, rather than a reconstructed one, would avoid these problems, but the original tree is not available for any of the traced systems or other similarly-sized systems. Statistical summaries, such as those produced by the `fsstats` tool [11] contain almost all the information necessary to evaluate namespace flattening, but unfortunately do not separate counts of files and directories — which is crucial. Simply modifying `fsstats` to separate these categories would allow for the correct calculation of the worst-case maximum number of directory slots and OID bits required. Calculating the number of overflows would require knowing the number of entries in every directory along every path, and not just the maximum number of entries. A tool that collected this data would allow for the reconstruction the original file system, but without the original file names. While many sites have been convinced to release statistics [11], enough data to reconstruct the entire file system, even with anonymized names, may be considered too sensitive by some sites. The repository of `fsstats` statistics includes sites with up to 20 M file and 1.3 M directories in their original file systems, which is about double the 9.6 M files and 0.7 M directories of the largest reconstructed file system, so the size of the hierarchies considered in the following analyses are within the range of sizes seen in surveys of complete file systems.

Knowing that the reconstructed file system has fewer objects than the original file system, analyses based on reconstructed file systems can be expected to differ from the original in the following ways. First, since the children of never-read directories are omitted from the reconstruction, the reconstructed filesystem may be shallower than the original, and thus require fewer directory slots, and have fewer depth overflows. Second, some files in each directory may be omitted, requiring fewer file bits and causing fewer width overflows. Third, some subdirectories in each directory may be omitted, requiring fewer bits in each directory slot and thus reducing the number of width overflows, and reducing the depth of the reconstructed file system, reducing the number of depth overflows. All of these effects reduce the number of OID bits required, thus all the analyses of reconstructed filesystems are lower bounds for similar analyses of the original file system.

4.5.2 Variable-length OIDs

Each of the reconstructed namespaces was traversed, and an OID was assigned for each file and directory using the variable-length Farsite format. Each directory uses the minimum file and directory slot widths needed to represent its children. Thus, each OID uses the minimum number of bits to represent the directory structure without overflow. The distribution of OID lengths is plotted in Figure 4.7. Collisions between OIDs that have different slot widths, but map to the same binary representation, are not accounted for in this analysis—for example, OIDs `0x10.0x1` and `0x1.0x10` would have the minimum representation of `0x101`. If such a collision occurs, it can be resolved by either choosing the next available OID, or adding an extra bit to distinguish the colliding OIDs.

4.5.3 Fixed-length OIDs, variable-size slots

The same reconstruction was performed for OIDs limited to a fixed size. Each directory still chose its optimum file and directory slot widths. This represents the best use of OID bits, given that the OID allocator

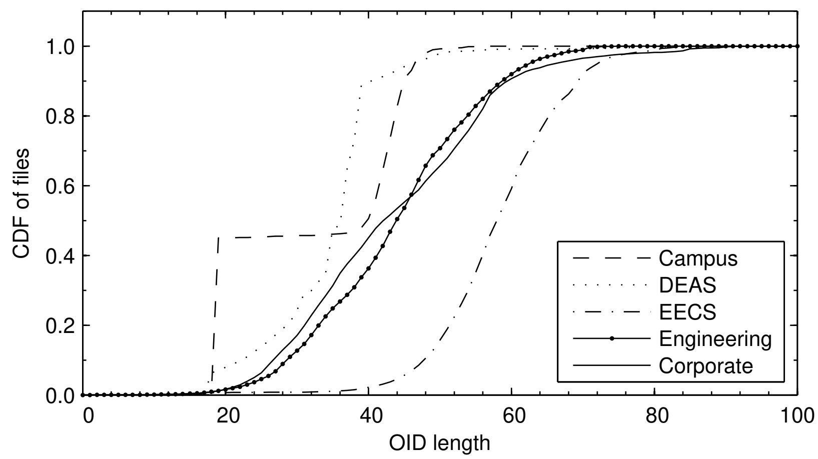


Figure 4.7: Number of variable-length OID bits necessary. For each trace, CDF of OID lengths required to uniquely represent each object are shown. Each file or directory slot was the minimum size necessary to represent that directory. Collisions between OIDs with the same binary representation are not accounted for—these values represent a lower bound on the OID size necessary to represent the hierarchy seen in the trace.

OID size	EECS	DEAS	Campus	Engineering	Corporate
96 bits	3	0	0	0	0
64 bits	2	0	0	2	0
32 bits	7	3	1	8	1

Table 4.2: Overflow in OID length. For each trace, the number of objects that could not be assigned an OID under their parent are shown. Each directory and file slot was the minumum width required to represent that directory. Objects for which this required more bits than available are considered to have experienced overflow and are assigned an OID from a single overflow region. Collisions between OIDs are not accounted for—doing so will increase the number of bits required. Therefore, these values represent a lower bound.

knows what the final directory structure will be. Table 4.2 shows the number of directories and files that cannot be assigned their desired OID because the fixed OID length is not large enough to represent the desired OID. The children of overflowed parents are not counted as overflowed, because they have the same locality with their parent that they would have had without overflow. Because of this, overflows are rarer than the distribution of OID lengths in Figure 4.7 suggests—many of the long OIDs are from the same subtree and, thus, experience overflow once for the entire subtree.

4.5.4 Fixed-length OIDs, fixed-size slots

The same reconstruction was performed with fixed-length OIDs in which the directory slot and file slot widths were fixed across the entire namespace. This corresponds to the behavior of the namespace flattening

OID slot bits			**EECS**		**DEAS**		**Campus**		**Engineering**		**Corporate**	
Tot	Dir	File		.		.		.		.		.
96	5	10	11478	0.12%	778	0.03%	0	0.00%	1061	0.07%	708	0.05%
96	4	11	3491	0.04%	2327	0.09%	0	0.00%	44	<0.01%	115	0.01%
96	3	14	6	< 0.01%	1	0.00%	0	0.00%	12	<0.01%	2	<0.01%
96	10	12	103727	1.09%	22051	0.83%	730	0.18%	51113	3.30%	44344	2.94%
96	10	15	103727	1.09%	22051	0.83%	730	0.18%	51113	3.30%	44344	2.94%
96	7	18	93254	1.01%	8714	0.32%	74	0.02%	20596	1.33%	27998	1.86%
64	5	8	96288	1.01%	6688	0.25%	74	0.02%	20862	1.35%	28339	1.86%
64	4	11	103820	1.09%	3021	0.11%	1	<0.01%	10566	0.68%	0	0.00%
64	3	12	11400	0.12%	779	0.03%	0	0.49%	1064	0.07%	9848	0.65%
64	10	12	209762	2.21%	44297	1.70%	862	0.09%	98238	6.35%	709	0.05%
32	5	6	634588	6.69%	129446	4.85%	5381	1.36%	215852	13.94%	234522	15.57%
32	4	7	631496	6.65%	113127	4.24%	5323	1.35%	190223	12.29%	210816	13.99%
32	3	10	626006	6.59%	86686	3.25%	3770	0.95%	176435	11.40%	185215	12.29%

Table 4.3: Overflow in depth. For each trace, the number of objects that could not be assigned an OID under their parent, because all directory slots were used, are shown for several choices of OID, directory slot, and file slot widths. The number of overflows depends mainly on the number of directory slots, since only that many levels of the directory hierarchy can be represented without overflow. Objects that experienced overflow were assigned an OID in the overflow region, which used the same slot widths as the main region but with 20 bits reserved for disambiguating overflowed subtrees. This results in the overflow region having fewer directory slots than the main region. If the object in question was a directory, its children are not also counted as overflowed because they will be local to their parent.

policies without any of the optimizations in Section 4.4. For simplicity, instead of the three overflow regions described in Section 4.2, only a single overflow region was used. This overflow region used the same directory slot widths as the main region, but had four fewer directory slots. The bits that would have been allocated to the missing directory slots were instead used to distinguish overflows from different parents.

A number of OID lengths and slot widths were tried; the OID lengths reflect those used in existing systems, and the slot widths were chosen based on the file and directory sizes in Figure 4.6. The number of "too-deep" and "too-wide" overflows encountered in each case are shown in Table 4.3 and Table 4.4, respectively.

If 96 bits were available for the OID, as is the case in Ursa Minor, all of the traces can be represented nearly perfectly in terms of depth by more than one choice of slot widths. Even with fewer bits, the best choice of slot widths is still nearly perfect, except for the Campus trace. Most of the overflows were from a small number of relatively deep subtrees—if the number of directory levels representable was greater than the "knee" in Figure 4.6(b), the exact choice of directory slot widths did not matter significantly.

Overflows in width, however, were much more common even with large OID sizes. The major source

OID slot bits			EECS		DEAS		Campus		Engineering		Corporate	
Tot	Dir	File		.		.		.		.		.
96	5	10	1656024	12.11%	1766902	66.13%	30914	7.82%	32191	2.08%	58895	3.91%
96	4	11	1454319	10.63%	1649454	61.74%	15813	4.00%	52013	3.36%	56792	3.77%
96	3	14	522694	3.82%	1208560	45.24%	1888	0.48%	82998	5.36%	69526	4.62%
96	10	12	976613	7.14%	1426045	53.38%	7196	1.82%	545	0.04%	2790	0.19%
96	10	15	99326	0.73%	822294	30.78%	0	0.00%	139	0.01%	2203	0.15%
96	7	18	26551	0.19%	43065	1.61%	198	0.05%	4735	0.31%	11590	0.77%
64	5	8	2817873	20.60%	1978617	74.06%	135531	34.30%	61672	3.34%	108849	7.23%
64	4	11	1454319	10.60%	1649454	61.74%	15813	4.00%	52013	3.36%	56729	3.77%
64	3	12	1225390	8.96%	1512818	56.62%	9084	2.30%	83404	5.39%	70113	4.65%
64	10	12	976613	7.14%	1426045	53.38%	7196	1.82%	545	0.04%	2790	0.19%
32	5	6	4898888	35.82%	2147035	80.36%	272846	69.05%	140854	9.10%	227091	15.07%
32	4	7	3763574	27.52%	2080858	77.88%	209553	53.03%	107932	6.97%	172993	11.48%
32	3	10	1829711	13.39%	1803038	67.49%	31866	8.07%	89429	5.78%	96758	6.42%

Table 4.4: Overflow in width. For each trace, the number of objects that could not be assigned an OID under their parent are shown. Each parent directory can hold as many files as fit in its file slot bits, and as many child directories as fit in its directory slot bits; any additional files or directories experience overflow. Objects that experienced overflow are assigned an OID from the overflow region, which used the same slot widths as the main region but with 20 fewer bits available for directory slots. The total number of directory slots in the overflow region is correspondingly reduced, and the bits that would have been used for the missing directory slots were instead used to disambiguate overflowed subtrees.

of width overflows were small numbers of very large directories, which argues for using wider file and directory slots in the "too-wide" overflow region. Because overflows for files and directories are summed together, increasing the number of file slot bits at the expense of directory slot bits my decrease the number of files that overflow but increase the number of child directories that overflow.

Although locality with the parent is lost with an overflow in width, the children still have some locality among themselves. When the parent's file slot fills up, a new file slot is allocated in the overflow region, and subsequent children are inserted in that slot until the new slot, in turn, fills up and the process repeats. Thus, while the number of files that overflow in width may be large, they fall into a much smaller number of distinct groups. While a very large directory may have 10,000 children across 10 overflow slots, and all 10,000 children are counted as overflowed, scanning every file in the directory would only experience 10 switches of OID space. Creating each of those 10,000 files would, however, have a higher likelihood of incurring a multi-server operation than if there were no overflow.

4.5.5 Farsite results

In the course of evaluating the OID assignment scheme used in Farsite, Douceur et al. examined the file-system of a departmental file server with 2.3 million data files and 168,000 directories [13]. Using variable-length OIDs, they found that the the mean identifier length was 52 bits, the maximum was 107 bits, and the 99th percentile was 80 bits. When using fixed 107 bits OIDs, 5% of files encountered overflow; with 48 bits OIDs, 46% encountered overflow. They also observed that OID length grew logarithmically with the size of the file-system, so much larger file systems would only require slightly longer OIDs. Their results are consistent with ours.

4.6 Conclusion

The child-closest OID assignment policy can, even with fixed slot widths, assign children OIDs that fall under their parent's OID more than 99% of the time, except when faced with very large directories, when it can assign at most 70% of children OIDS local to their parent. The OID allocation policy does not need to be perfect—not all overflows will lead to multi-server operations, and any system with multiple servers will need to cross a server boundary somewhere.

The version of the policy evaluated was relatively simple, and either of two optimizations would reduce the number of multi-server operations caused by imperfect OID assignments. First, if the overflow regions used different slot widths, the instances of repeat overflow for large directories would be reduced. Second, if several local overflow regions were used instead of a single global one, overflow would reduce OID locality, but would not result in multi-server operations.

Chapter 5

Prototype

In order to explore the feasibility of using migration to support multi-server operations, I have implemented this approach in the metadata path of the Ursa Minor distributed storage system. This chapter describes the high-level organization of Ursa Minor and its Metadata and Namespace services, and provides more detail on the internal components that enable its transparent scalability.

5.1 Ursa Minor

Ursa Minor [2] is a scalable storage system that, like other direct-access storage systems [24], is structured in two primary parts: a data path for handling data access and a metadata path for handling metadata access. This separation allows each path to be optimized for its purpose. Modern scalable storage systems are expected to scale to thousands of storage nodes and tens or hundreds of metadata nodes, so each part is a large distributed system in its own right.

As a whole, an Ursa Minor *constellation*, consisting of data nodes and metadata nodes, is expected to be highly available and durable, like other distributed storage systems. Ursa Minor is also intended to be incrementally scalable, allowing nodes to be added to or removed from the system as storage requirements change or hardware replacement becomes necessary. To provide for this, Ursa Minor must include a mechanism for *migrating* data from one data node to another and metadata from one metadata node to another.

The data path of Ursa Minor consists of a number of storage nodes, termed *workers*, which store byte streams identified by a unique 128 bit Self-$\star$ Object ID (abbreviated as *SOID*). There are no restrictions on which objects can reside on which worker, and an object's data can be replicated or erasure-coded across multiple workers, allowing the flexibility to tune an individual object's level of fault-tolerance and performance to its particular needs. Because data operations only affect one a single object at a time, they affect only a single worker (or group of workers identically) at a time. Thus, data path can scale transparently simply by adding more storage nodes.

Accessing a particular file's data requires two steps: first, the file name must be translated to a SOID, and second, the workers(s) responsible for the file data must be identified so that they can be contacted to retrieve the data. In Ursa Minor, these functions are performed by the Namespace Service (NSS) and the Metadata Service (MDS), respectively, which together make up the metadata path.

In Ursa Minor, unlike most contemporary distributed file systems, the responsibility for managing consistency between clients that concurrently operate on the same state is left to the clients themselves. Thus, the MDS and NSS do not ensure the consistency of client caches, although they do protect the consistency of internal MDS and NSS structures. Shifting responsibility to the client simplifies the implementation of the the MDS and the NSS in ways that are highlighted below. As no current Ursa Minor client implements cache consistency, the semantics provided by Ursa Minor are weaker than those provided by strongly-consistent file systems. Because of this difference, care was taken to ensure that strong consistency could be retrofitted into the metadata path at a later date, and the performance analyses in Chapter 6 were structured so that Ursa Minor did not unduly benefit from its lack of cache consistency.

5.2 Metadata Service (MDS)

The Metadata Service in Ursa Minor maintains information on each object, similar to that maintained by the inodes of a local-disk file system. For each object, the MDS maintains a record that includes the object's size, link count, attributes, permissions, and the list of worker(s) storing its data. Clients communicate with the MDS via RPCs. Since clients are untrusted, the MDS must verify that each request will result in a valid state and that the client is permitted to perform that action. Some requests, such as creating or deleting an object, require the MDS to coordinate with workers. Others, such as updating an attribute or timestamp, reside wholly within the MDS. The semantics defined for the MDS imply that individual requests are atomic (they either complete or they don't), consistent (the metadata transitions from one consistent state to another), isolated (simultaneous requests are equivalent to some sequential order), and durable (once completed, the operation's results will never be rolled back). The transaction mechanism used to ensure this is discussed in detail in Section 5.8.

The MDS is responsible for all object metadata in Ursa Minor. Individual object metadata records are stored in metadata tables. Each table includes all records within a defined range of SOIDs. The tables are internally structured as B-trees indexed by SOID and are stored as individual objects within Ursa Minor. The ranges can be altered dynamically, with a minimum size of one SOID and a maximum of all possible SOIDS. Within those limits, the MDS may use any number of tables, and, collectively, the set of tables contains the metadata for all objects. Storing the tables as objects in Ursa Minor allows the MDS to benefit from the reliability and flexibility provided by Ursa Minor's data path, resulting in the metadata path holding no hard system state.

Because each table is itself an object, adding a new record to table T may increase the size of T's data, requiring an update of T's metadata, which may be in a different table, S. These recursive updates are an internal source of multi-object operations since both table T and table S are involved — though in most cases they do not require full atomicity. These recursive transactions are described in more detail in Section 5.8.2.

Each Ursa Minor cluster includes one or more metadata servers. Each metadata server is assigned a number of metadata tables, and each table is assigned to at most one server at a time. Thus, accessing the metadata of any particular object will only involve one server at a time. Because the metadata tables are themselves objects, they can be accessed by any metadata server using Ursa Minor's normal data I/O facilities.

The assignment of tables to servers is recorded in a *Delegation Map* that is persistently maintained by a *Delegation Coordinator*. The delegation coordinator is co-located with one metadata server, termed the *Root Metadata Server*. The root server is just like any other metadata server, except it happens to host the metadata for the objects used by the metadata service. Clients request the delegation map when they want to access an object for which they do not know which metadata server to contact. They cache the delegation map locally and invalidate their cached copy when following a stale cached delegation map results in contacting the wrong metadata server. Tables can be reassigned from one server to another dynamically by the delegation coordinator, and this process is discussed in Section 5.5.

5.3 Namespace Service (NSS)

The Namespace Service manages directory contents. Directories are optional in Ursa Minor — applications satisfied with the MDS's flat SOID namespace (e.g., databases, mail servers, scientific applications) need not use directories at all. Other applications expect a traditional hierarchical directory tree, which the Namespace Service provides.

Similarly to a local-disk file system, a directory entry is a record that maps a filename to a SOID. Directories are B-tree structured, indexed by name, and stored as ordinary objects, with their own SOIDs. At present, each directory object contains all of the directory entries for that directory, though there is no obstacle to splitting a directory across multiple objects.

One design alternative, used by Ceph [52] and C-FFS [22], is to locate the inodes of a directory's children in the directory file itself. This would provide the benefits of increased locality (the child's directory entry and inode are frequently updated together) and reduced contention for access to the inode tables (each directory effectively becomes a small table). This approach, however, complicates implementing hard links and renames, as well as support for applications that do not require directories at all, so we did not choose it.

Namespace servers are tightly coupled with metadata servers (in our implementation, a single server process exports both RPC interfaces). Each namespace server is responsible for directories whose SOIDs are within the range exported by its coupled metadata server. This ensures that a directory's "inode" (the attributes stored by the MDS) and its contents will always be served by the same process. When a metadata table is reassigned to another server, the responsibility for those directories goes with it. For the rest of this paper, we do not distinguish between the MDS and the NSS, and use the term "metadata server" to refer to the combined server.

The NSS aims to support directory semantics sufficient to implement an overlying file system with POSIX, NFS, CIFS, or AFS semantics. As such, it provides the POSIX notions of hard links, including decoupling of unlink and deletion, and the ability to select how already-existing names are handled. Typical operations include creating a file with a given name, linking an existing object under a new name, unlinking an file, looking up the SOID corresponding to a file name, and enumerating the contents of directories.

5.4 SOID assignment

In Ursa Minor, the SOID of an object determines which table, and thus which metadata server, that object is assigned to. It follows that there may be advantages in choosing to use particular SOIDs for particular files. For instance, the **ls -al** command will result in a series of requests, in sequential order, for the attributes of every file in a given directory. If those files all had numerically similar SOIDs, their metadata would reside in the same (or nearby) B-tree pages, making efficient use of the server's page cache. Similarly, most file systems exhibit spatial locality, so an access to a file in one directory means an access to another file in that same directory is likely. Secondly, many directory operations (CREATE, LINK) operate on both a parent directory and a child inode at the same time. If the parent and child had nearby SOID numbers, they would likely reside in the same table, simplifying the transaction as discussed in Section 5.6.

For these reasons, it would be useful to assign SOIDs such that children of a directory receive SOIDs similar to those of the directory itself. Fortunately, the child-closest policy described in 4.2 does just this. The net effect of combining a child-closest SOID assignment policy with SOID-range tables is that each table usually ends up containing a subtree. This is somewhat analogous to the volume abstraction offered by systems like AFS, but without the predefined, rigid mapping of subtree to volume. Unlike these systems, a too-large or too-deep subtree will overflow into another table, quite possibly not one served by the same server. One can think of these overflowed subtrees as being split off into separate sub-volumes, as is done in Ontap GX [14] and Ceph [52]. The "local-overflow" optimization described in Section 4.4, if implemented, would assist in keeping the overflow within a single table.

Of the 128 bits in the SOID, 96 bits are available for use by the SOID assignment policy. By tuning the bit widths of the directory segment, file segment, and directory slots to match the system's workload, instances of overflow can be made extremely rare [27]. Namespace manipulations, such as linking or renaming files across directories, however, will result in the renamed file having a SOID that is not similar to the SOID of its new parent or siblings. An analogous situation happens in local disk files systems: a renamed file's inode still resides in its original cylinder group after a rename. The MDS includes an operation to atomically change an object's SOID which could be used to renumber a renamed file into the SOID range of its new parent. We do not, however, implement this yet.

The SOID of a deleted file is available for re-use as soon as the file's storage has been reclaimed from the relevant workers (this step is performed lazily in most cases). Thus, as long as a directory's size does not change over time, changing it's contents does not affect the chance of overflow. In fact, reusing a SOID as soon as possible should provide for a slight efficiency gain, by keeping the metadata B-tree compact.

In all of these cases, outside of the SOID selection policy, MDS treats the SOID as an opaque integer and will operate correctly regardless of how much or little locality the SOIDs preserve. Performance will be better with higher locality, however. The segment sizes do not need to remain constant over the life of a constellation, or even across the SOID namespace, so there is the potential to adaptively tune them based on the observed workloads. We have not yet implemented this functionality.

5.5 Metadata migration

Ursa Minor includes the ability to dynamically migrate objects from one metadata server to another. It does so by reassigning responsibility for a metadata table from one server to another. Because the metadata table (and associated directories) are Ursa Minor data objects accessible to all metadata servers, the contents of the metadata table never need to be copied. The responsibility for serving it is simply transferred to a different server. This section describes the process for doing so in more detail.

Each metadata server exposes an RPC interface via which the delegation coordinator can instruct it to ADD or DROP a table. In order to migrate table T from server A to server B, the coordinator first instructs server A to DROP responsibility for the table. When that is complete, the coordinator updates the delegation map to state that B is responsible and instructs server B to ADD T. At all times, at most one server is responsible.

When server A is instructed to DROP T, it may be in the process of executing operations that use T. Those operations will be allowed to complete. Operations waiting for T will be aborted with an error code of "wrong server", as will any new requests that arrive. Clients that receive such a response will contact the coordinator for a new delegation map. Once the table is idle, server A sets a bit in the table header to indicate that the table was cleanly shut down, flushes the table from its in-memory cache, and responds to the coordinator that the table has been dropped.

Adding a table to server B is also simple. When instructed to ADD responsibility, server B first reads the header page of table T. Since T's header page indicates it was shut down cleanly, no recovery or consistency check procedure is necessary, so server B simply adds an entry for T to its in-memory mapping of SOID to table. Any subsequent client requests for SOIDs within T will fault in the appropriate pages of T. Before its first write to T, server B will clear the "clean" bit in the header, so any subsequent crash will cause the recovery procedure to run.

5.6 Multi object operations

For a server, performing a transaction on a single object is simple: acquire a local lock on the SOID in question and on the SOID's table, perform the operation, and then release all locks.

Performing a transaction with multiple objects or tables within a single server is similar, but complicated by the need to avoid deadlocks between operations that try to acquire the same locks in opposite orders. Each server's local lock manager avoids deadlock by tracking all locks that are desired or in use. When all locks required for an operation are available, the lock manager acquires all of them simultaneously and allows the operation to proceed.

In the more complicated case (shown in Figure 5.1) of a multi-object and multi-server operation, the server's local lock manager will discover that all the required resources are not local to the server. The lock manager blocks the operation and sets out to acquire responsibility for the required additional tables. To do so, it sends a BORROW request to the Delegation Coordinator. The BORROW request includes the complete list of tables required by the operation; the coordinator's lock manager will serialize conflicting BORROWs. When none of the tables required by a BORROW request are in conflict, the coordinator issues a series of ADD and DROP requests to move all the required tables to the requesting server and returns control to it. Those tables will not be moved again while the transaction is executing.

When the transaction completes, the requesting server sends a RETURN message to the coordinator, indicating that it no longer requires exclusive access to that combination of tables. The coordinator determines whether it can now satisfy any other pending BORROW requests. If so, the coordinator will migrate a RETURNed table directly to the next server that needs it, otherwise that table will be migrated back to its original server. Note that, while waiting for a BORROW, a server can continue executing other operations on any tables it already has; only the operation that required the BORROW is delayed.

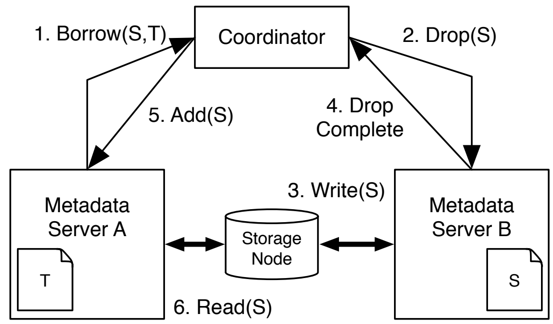


Figure 5.1: Borrowing a table. The sequence of operations required for Server A to handle an operation requiring tables S and T, when table T is initially assigned to server A and table S to server B. Returning to the original state is similar.

5.7 Root metadata server

Relying on a single root metadata server simplifies coordination, but has implications for the scalability of the metadata service as a whole. Fortunately, the functions of the root metadata server can be distributed across a cluster of servers. The root metadata server's first function is to serve metadata on the objects used by the metadata service. In an extremely large constellation, these internal objects may be numerous and busy enough to exceed the capacity of a single root metadata server. The solution is simply to distribute internal metadata across a tier of "internal" metadata servers instead of a single one. The client-facing metadata servers will only ever need to issue single-object operations to the internal metadata servers, and thus the internal tier should parallelize well.

The second function of the root metadata server is to serve the delegation map. While this is a simple operation, every client will request a new copy of the delegation map when that client discovers that its cached copy is stale. This load could be mitigated by allowing clients to fetch the delegation map from any metadata server. The root, however, has the only completely authoritative copy; the copy at any other metadata server S is authoritative for the objects assigned to S but may be stale for objects delegated to other metadata servers. Therefore, a client that updated its delegation map from that of a non-root server may find that the updated copy is inaccurate and need to retry to obtain a fresher delegation map. The same situation can occur even if clients contact the root directly — the root may migrate a table immediately after the client updated, so clients are already prepared to retry repeatedly.

The root metadata server's third function is to serve as a coordinator for multi-server operations. This task could instead be delegated to the internal tier of metadata servers. Each internal server would be assigned a range of SOIDs, and that internal server would be responsible for coordinating all multi-server operations that fall entirely into that range of SOIDs. Since each internal server is responsible for disjoint ranges of objects, each can proceed independently of the others. Only in the case where a multi-server operation involves tables coordinated by two different coordinators would the root server need to be invoked to coordinate the operation.

While each of these techniques is relatively straightforward, we have not yet implemented them in Ursa Minor because the root metadata server was only a bottleneck in a single experiment. However, we expect it may be an issue in the future.

5.8 Transactions

Underlying the Metadata and Namespace Services is a transactional layer that manages updates to the B-tree structures used for storing inodes and directories. The B-tree implementation is based on Berkeley DB 1.8.5 [42] that we have modified such that it uses Ursa Minor objects, instead of files, for storing B-tree contents. This version of Berkeley DB includes no transaction support at all. However, the Ursa Minor storage nodes and their access protocols guarantee that individual B-tree pages are written atomically to the storage nodes and that data accepted by the storage nodes will be stored durably. Ursa Minor's transaction system extends these guarantees to transactions involving multiple B-tree pages spread across multiple B-trees.

Atomicity is provided using a simple shadow-paging scheme. All updates to the B-tree data object are deferred until commit time. The data object includes two storage locations for each page, and the location written alternates on each write of that page. Thus, one location will contain the most recent version of that page, and the other location will contain the next most recent version. Each page includes a header that links it to all the other pages written in the same transaction, which will be used by the recovery mechanism to determine whether the transaction committed or needs to be rolled back. Reading a page requires reading both locations and examining both headers to identify the latest version. The server may cache this information, so subsequent re-reads only need the location with the latest page contents.

Isolation is guaranteed by allowing only a single transaction to execute and commit on each B-tree at a time. Every transaction must specify, when it begins, the set of B-trees it will operate on. It acquires locks for all of those B-trees from the local lock manager, and holds them until it either commits or aborts. If, during execution, the transaction discovers it needs to operate on a B-tree it does not hold a lock for, it aborts and restarts with the new B-tree added to the set. This strategy is similar to that used by Sinfonia [4] mini-transactions, which share the limitation of specifying their read and write sets up front. Most transactions require only a single execution. The main sources of repeated executions are operations that traverse a file system path: at each step, the SOID of the next directory to read is determined by reading the current directory.

Consistency is only enforced for the key field of the B-tree records; maintaining the consistency of the data fields is the responsibility of the higher level code that modifies them.

Durability is provided by synchronously writing all modified pages to the storage nodes at commit time. The storage nodes may either have battery-backed RAM or themselves synchronously write to their internal disks.

5.8.1 Recovery

If the metadata server crashes while committing a transaction, it is possible for the B-tree to be in an inconsistent state: for example, only 2 of the 3 pages in the last transaction may have been written to the storage nodes before the crash. To resolve this condition, the metadata server performs a recovery process when it restarts after an unclean shutdown. First, it queries each storage node to determine the location of the last write to the B-tree object (the storage node must maintain this information as part of the PASIS protocol [1]). The location of the last write corresponds to the last page written. Reading that page's header will reveal the identity of all other pages that were part of the same transaction.

If all the other pages have transaction numbers that match that of the last written page, then we know that the transaction completed successfully. If any of them has an earlier transaction number, we know that not all page writes were completed, and a rollback phase is performed: any page with the latest transaction number is marked invalid, and its alternate location is marked as the valid one. At the end of rollback, the latest valid version of every page is the same as it was before the start of the rolled-back transaction. The recovery process can proceed in parallel for B-trees with independent updates, whereas two B-trees involved in the same transaction must be recovered together. Because there is at most one transaction committing at a time on a given B-tree, at most one rollback on a given B-tree will be necessary.

5.8.2 Recursive transactions

As described in 5.2, a transaction that expands a metadata table T will need to update the metadata table's metadata, stored in table S. For example, a CREATE of a new object (with SOID 64.01) will add a metadata record to the relevant metadata table (with SOID 4.10). The length of the table may change as a result, requiring an update to the record for SOID 4.10 (which is stored in table 4.1). In most cases, it is sufficient if the update to S happens before the update to T. Ursa Minor accomplishes this by ensuring that no server is responsible for the metadata of any table it is exporting. Thus, S and T will be served by different servers, and, in the event of a crash, T's length may have increased without any new data having been written to T, which is a condition the recovery process can handle.

The potential for recursion ends at the *bootstrap table*, which contains, only one metadata record. Thus the bootstrap table's metadata is static and recorded in the constellation configuration (similarly to the superblock of a filesystem). The only object in the bootstrap table is the *internal table*, which contains the metadata for the other metadata tables. Both the bootstrap and internal tables are served by the root metadata server, in violation of the previous rule. The root metadata server always assumes that any transaction it performs will involve both tables, avoiding the need to support nested transactions.

5.9 Caching

All of the components in the Ursa Minor metadata path include some form of caching, as shown in Figure 5.2. This section describes the behavior of each level of cache and the interactions between them.

5.9.1 Delegation Cache

The delegation map, described in Section 5.5, is required in order for a client to decide which server is responsible for operations on a particular SOID. The delegation map requires approximately 100 bytes per table (10 kB to 76 kB for the experiments in Chapter 6), and clients fetch it in its entirety from the root metadata server on startup. When the delegation map changes, as a result of a migration, the client will continue to use its stale copy of the delegation map until the stale map causes it to send a request to a server that is no longer responsible for an object that the client thinks it is. The server will respond with a "wrong

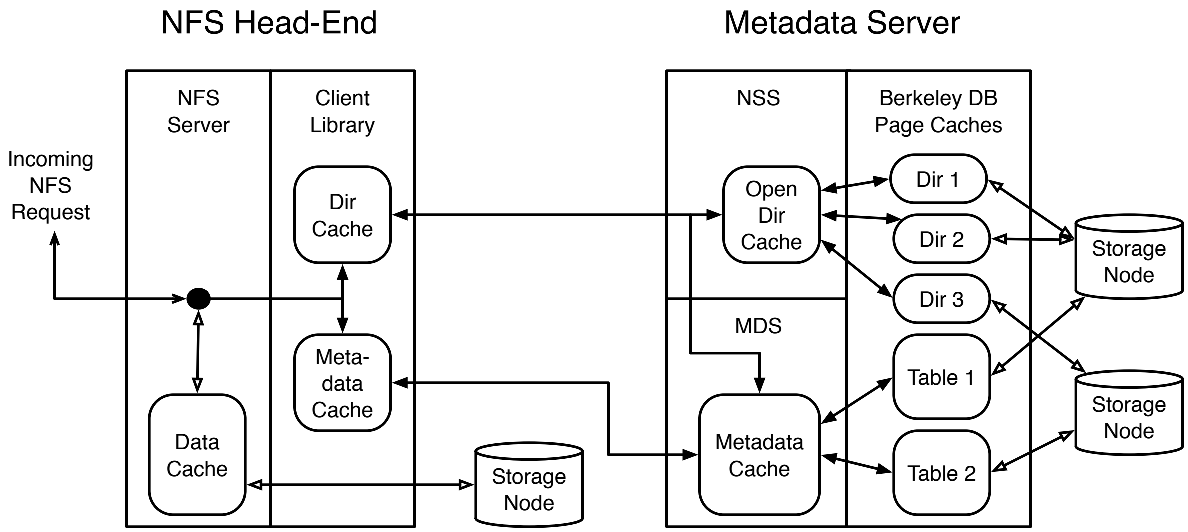


Figure 5.2: Caches along the Ursa Minor metadata path. An example client (a NFS head-end server) is shown on the left, and an Ursa Minor metadata server is shown on the right. Traffic using the metadata protocols is shown with dark arrows, while traffic using the data protocol is shown with white arrows.

server" error code, causing the client to realize that it's cached copy is stale. The client will then attempt to fetch a new copy of the delegation map from the root metadata server before retrying the operation.

Since it is likely that a number of operations will simultaneously notice a stale delegation map, the following steps are taken to reduce the number of spurious retries and delegation update requests: First, the client will only request one new copy of the map at a time — further operations that receive a "wrong server" response will be blocked until the new map is received, and only then retried. Second, the server increments a version number on each update to the delegation map. When a client requests an updated map, it includes its cached version number. If the server's version number is greater than the clients, it will immediately respond with the new map. If the server's version is the same as the client's, then the client already has the latest version and the server will wait to respond until the server's version of the map changes. This avoids a burst of failure, update, and retry cycles when a migration is in progress and the new server is not yet ready to serve requests on the table in question. To avoid encountering RPC timeouts or blocking a client indefinitely, the server will also respond before the RPC timeout expires even if it thinks the client already has the latest version.

5.9.2 Client metadata cache

In addition to the delegation map, clients can cache "inode" contents (object attributes and layout information) returned by metadata servers. This cache is write-through — metadata modifying operations update the cache after the operation succeeds at the server. The total size of the cache is fixed, and entries are replaced in LRU order.

Clients that share objects must manage cache consistency among themselves; the MDS does not provide a mechanism for maintaining strong metadata cache consistency. It does, however, provide two methods of detecting consistency hazards. Stale layout information and capabilities will be detected when the client contacts the wrong storage nodes or the correct storage nodes with the wrong capabilities. Generally, clients first read the metadata for an object before modifying it. The server includes a version number in every metadata response. When issuing a metadata modifying request, the client includes the cached version number for that object. If the server has a version number higher than the client's cached version, then the server knows that the client has missed an update, and will reject the request, in a form of optimistic concurrency control.

Stronger cache consistency, using mechanisms such as callbacks, would be straightforward to add to the MDS client and server. But since no current application requires strong consistency, this feature was left until it is required.

5.9.3 Client directory cache

Clients also cache the directory contents returned by RPCs such as READDIR, which returns the entire contents of the directory, and LOOKUP which provides information about a single directory entry. Entries from both sources are cacheable—if an entry is in the cache, then it definitely exists. If an entry is not in the cache, the server must be contacted to determine if it exists but is not cached or does not exist at all. Negative-entry caching would alleviate this problem, but it has not yet been implemented. The total size of the directory cache is fixed, but the number of directories in it will vary depending on the size of the cached directories. Like the client metadata cache, clients that concurrently modify the same directories must manage the consistency of their directory caches on their own.

This cache is also write through — operations like CREATE and UNLINK modify cached directories to match effect at the server. For the sake of simplicity, although the NSS protocol and server support partial directory reads, this capability is not used and directories are fetched and cached in their entirety. This greatly simplifies the implementation of the head-ends's NFS3READDIR, which must support partial reads. As with client metadata cache, the MDS does not ensure the consistency of the client directory cache, although it could if required.

5.9.4 Server B-tree page cache

The metadata service uses the Berkeley DB B-tree package to access all persistent metadata tables and directory contents. Berkeley DB includes a buffer pool that caches B-tree pages. Each open table and directory is an individual B-tree object and has its own individual, fixed size, buffer pool. Each buffer pool maintains its own LRU order. Because the buffer pool management is a part of the Berkeley DB code base, using a more efficient unified buffer pool would be difficult. For every page in Berkeley DB's cache, the Ursa Minor transaction layer must maintain a shadow copy and an additional working copy, tripling the memory required to represent a page. Ideally, only pages that were modified in a transaction would need to have additional copies, but because of layering, the transaction layer does not know which pages will be modified until after the modification occurs. Therefore, it must pessimistically assume that any page will be modified. Because Berkeley DB is not threadsafe, accesses to a B-tree object must take place within a transaction, which ensures only a single thread accesses a given B-tree object at a time.

Each metadata table ADDed to a server is opened in Berkeley DB as part of the ADD operation, creating that table's buffer pool. Memory for the buffer pool is allocated on demand; since each table's pool is relatively small (1 MB by default), however, it quickly reaches its full size. Directory B-trees, on the other hand, are cached in their entirety. Since directories are usually only a few pages in size and READDIRs will read the entire directory, caching the full directory is not a large increase in memory usage. When a table is DROPped, it, and any open directories whose SOIDS lie in that table, are closed and the memory allocated to their buffer pools reclaimed.

5.9.5 Server directory cache

Since each open directory is cached in its entirety by the B-tree layer, the size of the directory cache is controlled by regulating the number of open directories. A directory must be open in order for an operation to read or modify its contents. After the operation that opened it, a directory is kept open and used for other operations until it must be closed and evicted to make room for a different directory to be opened. The default size is 1,000 directories per metadata server — managing the cache based on the amount of memory used, as done for the client cache, would be preferable, but doing so is difficult because the actual amount of memory used for each directory is not exposed at the layer that makes eviction decisions.

5.9.6 Server metadata cache

In addition to the per-table B-tree page caches, the MDS also maintains a unified cache of object metadata records. In contrast to the page caches, which contain pages with entries in the marshalled on-disk record format, each entry in the in-memory metadata cache is a complex linked structure. Again, due to layering, determining the exact size of each entry is difficult, so this cache is also managed by entry count (10,000 objects by default).

Also unlike the B-tree page cache, entries in this cache can be locked individually and accessed outside of a transaction. Thus, the common case of a LOOKUP that hits in this cache only has to contend with another operation on the same object. A miss, on the other hand, must begin a transaction and unmarshall the appropriate record from the B-tree (which may be in the page cache), which will contend with all other transactions using the same table.

Since servicing a cache miss may be relatively slow, placeholder entries are used for objects that are in the process of being created or read from a lower layer. Negative cache entries, for SOIDs known not to exist in the B-tree, are supported, but are not enabled by default[1]. When a table is DROPped, any cache entries for SOIDs in that table are removed.

5.10 Handling failures

Any of the components of the metadata path can fail at any time, but all failures should be handled quickly and without data loss. In general, our design philosophy considers servers trustworthy; we are primarily concerned with crashes or permanent failure and not with arbitrarily faulty computations or malicious servers.

[1]Many of the misses in the metadata cache are due to operations that test whether an object does exist, and then create it, so the the miss has the effect of causing the B-tree layer to page in (if necessary) the same page that would be needed during the create. Thus, these mandatory misses usually don't change the total work the B-tree layer must perform.

5.10.1 Failure of a metadata server

The most obvious components to consider for failure are the metadata server software and the hardware that it runs on. A constellation monitoring component polls all metadata servers (as well as other components) periodically. If the server does not respond within a time-out interval, that metadata server instance is considered failed. The monitoring component will then attempt to start a replacement metadata server instance, either on the same hardware or on a different node. The new instance queries the delegation coordinator to determine the tables for which it is responsible and runs the recovery process. After recovery completes, the new instance is in exactly the same state as the previous instance. While the new instance is starting and recovering, client requests sent to the old instance will time-out and be retried.

Not only does a failed metadata server affect clients, but it may also affect another server if it failed in the middle of a migration. The delegation coordinator will see its ADD or DROP request time out and propagate this error to any operation that depended on the migration. The metadata being migrated will be unavailable until the metadata server restarts, just like any other metadata served by the failed server. It is reasonable for a multi-server operation to fail because one of the servers it needs is unavailable.

When the failed metadata server restarts, the delegation map it receives from the coordinator will be unchanged from when the server began its last ADD or DROP: a failed ADD will be completed at this time, and a failed DROP effectively never happened. Instead of waiting for a server to restart, the tables assigned to the failed server could simply be reassigned to other working servers. Doing so, however, complicates the process of recovering a table that was involved in a multi-table (but same server) transaction: As described in Section 5.8.1, both tables must be recovered together, which poses a problem if the two tables have been reassigned to different servers for recovery. Although it is possible to detect and handle this case, in the interest of simplicity, we avoid it by always trying to recover all the tables assigned to a failed server as one unit. A system that used per-server transaction logs that covered multiple tables would face a similar problem—it would be much simpler to replay the entire log rather than partition it and replay it in pieces.

5.10.2 Failure of the delegation coordinator

A failed delegation coordinator will prevent the system from performing any more delegation changes, although all metadata servers and clients will continue to operate. As the delegation map is stored in an object and synchronously updated by the coordinator, the coordinator is stateless and can simply be restarted.

If the failure happened during a migration, the metadata table(s) being migrated will be in one of two states: the delegation map says server A is responsible for table T but server A does not think it is, or the delegation map says no server is responsible for T. The delegation map is always updated in an order such that a server will never be responsible for a metadata table that is not recorded in the delegation map. To handle the first case, a newly started coordinator will contact all metadata servers to determine which tables they are serving and issue the appropriate ADD requests to make the server state match the delegation map. In the second case, an appropriate server is chosen for tables that have no assigned server, and an ADD request is issued.

Additionally, since the delegation coordinator is usually also the root metadata server responsible for the metadata objects, failure of this machine means that the other metadata servers will not be able to perform any operations that cause metadata tables to expand. The criticality of the root metadata server makes it a good candidate for replication, perhaps using a replicated state machine protocol such as Zzyzx [28].

5.10.3 Network partitions

A data-center scale distributed system, while being physically in one room, is interconnected by a complex network that may include several layers of switches and links. A failure of one link or switch may leave functioning subsets of Ursa Minor nodes and clients on both sides of the failed component.

Correct operation of the constellation requires that there be at most one instance of each Ursa Minor component at a time. If the constellation monitor is on the one side of a network partition, it may incorrectly declare a properly operating metadata server on the other side of the partition to have failed.

Restarting a new instance would result in two servers trying to serve the same objects, violating the consistency assumptions. To avoid this, the delegation coordinator revokes the capabilities used by the old instance to access its storage nodes before granting capabilities to the new instance to do the same. Once capabilities have been successfully revoked on a quorum of storage nodes, not enough storage nodes remain that might still honor that capability. Thus, while the old instance may still be running, it will not be able to access its backing store, preserving consistency. Also clients will not be able to use capabilities granted by the old instance to access client data. If the revocation attempt fails to reach a quorum of storage nodes, perhaps because they are also on the other side of the network partition, the coordinator will not reassign the partitioned server's tables until the partition heals. The old instance, which can still contact the storage nodes and clients on its side of the partition, continues uninterrupted until then.

Similarly, there must only be one delegation coordinator across the now partitioned system. One method to ensure this, which we have not yet implemented, is to use a quorum protocol amongst metadata servers to elect a new coordinator [34].

5.10.4 Failure of a storage node

For storage node failures, we rely on the Ursa Minor data storage protocol to provide fault tolerance by replicating or erasure-coding object data across multiple storage nodes. Since the contents of the metadata tables cannot be reconstructed from any other source, they must be configured with appropriately high fault tolerance.

5.11 NFS head-end

While the server-side of the metadata path has been described in detail, Ursa Minor also provides client libraries that implement the client side of the MDS and NSS RPC protocols. Applications may be linked against these libraries and access Ursa Minor directly or linked against a POSIX shim library that implements POSIX file system calls on top of Ursa Minor. Both of these approaches involve modifying user applications. The alternative, of implementing an installable filesystem through the VFS layer [32] or FUSE [21] would be preferable, but require interacting with the client's OS kernel, which is more difficult to implement than a purely user-level solution.

The solution employed in Ursa Minor is to use a translating *head-end* to translate client NFSv3 requests into the corresponding Ursa Minor data and metadata requests. The head-end is an Ursa Minor client application that implements the server side of the NFSv3 protocol. Exporting NFSv3 allows for running standard NFS benchmarks, such as SPECsfs97, against Ursa Minor without involving the host OS's NFS implementation. The same approach of using a translating layer between a standard filesystem protocol and a special internal protocol has been used in many systems, such as OntapGX [14] and Slice [5].

The NFS head-end is responsible for issuing NFS filehandles and maintaining the mapping between a filehandle and the SOID of the file to which it corresponds. It does so by including the SOID is the filehandle.

The NFS head-end serves data-only NFS requests, such as NFS3READ, by reading the file data from the storage nodes and returning it to the client. Once read from the storage nodes, file data is cached by the head-end for future use.

Reading a file's data from the storage nodes requires metadata (the storage node list) for that file. Similarly, returning any file attributes, such as length or mode, requires retrieving those attributes from the MDS. The NFS head-end accesses metadata through the Ursa Minor client library interface. Since the attribute read and the data read may occur at different points in the NFS head-end code-path, a single NFS request may cause multiple metadata calls to the Ursa Minor client library. The client library implements a metadata cache, described in Section 5.9 to speed up such repeated requests.

Data modifying operations, such as NFS3WRITE, update the data cache and are propagated to the storage nodes lazily in accordance with NFS semantics. Any metadata updates required to perform the data write (such as updating the "number of blocks used" attribute) are performed when the data is flushed to the storage node.

Metadata modifying operations, such as NFS3SETATTR or NFS3CREATE, cause an immediate corresponding Ursa Minor operation. Directory operations, such as NFS3CREATE, may cause the NFS head-end to first perform an Ursa Minor LOOKUP in order to verify that the new file's name doesn't already exist before performing a CREATE to create the file. The CREATE will perform the same existence check again at the metadata server before actually creating the object. This partially duplicated work is an artifact of how the head-end was developed; the head-end predates the NSS and therefore had to implement directories itself. Given the requirement to maintain both modes of operation, restructuring the head-end's code paths to avoid duplicating work when using the NSS would be difficult.

A list of NFS operations and the corresponding MDS and NSS operations they induce is given in Appendix A.

A constellation may include any number of NFS head-ends. Each NFS head-end exports a separate NFS filesystem, using distinct ranges of SOIDs. Of the 128 bit SOID, 32 bits are reserved to identify the head-end, and the remaining 96 bits are available to be used by the SOID assignment policy. This is done to avoid requiring the coordination between NFS head-ends that would be necessary to maintain cache consistency if more than one NFS head-end was exporting the same NFS filesystem. As far as the metadata servers are concerned, the workload they receive from the NFS head-ends is similar to what they would receive if the head-end's NFS clients used Ursa Minor directly instead of NFS.

Chapter 6

Evaluation

To demonstrate and enable evaluation, we constructed a transparently scalable Metadata Service for Ursa Minor that uses migration to support multi-server operations. We evaluate the performance of Ursa Minor with a standard benchmark (SpecSFS97) and a range of modified workloads to reveal the sensitivity our implementation to characteristics of the workload and explore the influence of system configuration options likely to be encountered in systems like Ursa Minor. Section 6.1 describes the benchmarks we used. Section 6.2 describes the hardware and software configurations. Section 6.3 discusses Ursa Minor's performance in scalability benchmarks. Section 6.4 discusses the influence of workload properties, Section 6.5 discusses the influence of system parameters, Section 6.6 discusses the difficulty of implementing multi-server operations, and Section 6.7 discusses additional observations.

6.1 Benchmark

The SPECsfs97 [49] benchmark is widely used for comparing the performance of NFS servers. It is based on a survey of workloads seen by the typical NFS server and consists of a number of client threads, each of which emits NFS requests for file and directory operations according to an internal access probability model. Each thread creates its own subdirectory and operates entirely within it. Since each thread accesses a set of files independent from all other threads, and each thread only has a single outstanding operation, this workload is highly parallelizable and contention-free.

In fact, using the namespace flattening policy described in Section 5.4, Ursa Minor is trivially able to assign each thread's files to a distinct SOID range. Thus, each metadata table consists of all the files belonging to a number of client threads, and all multi-object operations will only involve objects in the same table. While this is very good for capturing spatial locality, it means that multi-table and multi-server operations will never occur for the default SPECsfs97 workload. To force multi-table operations, we adjusted the namespace flattening policy to perform more poorly and intermix each client thread's directories with those of other threads using the same NFS mountpoint.

Because the SPECsfs97 benchmark directly emits NFS requests, these requests must be translated into the Ursa Minor protocol by an *NFS head-end* as described in Section 5.11. Each head-end is an NFS server and an Ursa Minor client, and it issues a sequence of Ursa Minor metadata and/or data requests in order to satisfy each NFS request it receives. In the default SPECsfs97 workload, 73% of NFS requests will result in one or more Ursa Minor metadata operations, and the remaining 27% are NFS data requests that may also require an Ursa Minor metadata operation. Like any Ursa Minor client, the head-end can cache metadata, so some metadata operations can be served from head-end's client cache, resulting in a lower rate of outgoing Ursa Minor metadata requests than incoming NFS requests. Each head-end is allocated a distinct range of SOIDs for its use, and it exports a single NFS file-system. Thus, different head-ends will never contend for the same objects, but the client threads connected to a head-end may access distinct objects that happen to be in the same metadata table.

Each run of SPECsfs97 produces a latency measurement at several target throughput levels. As specified by SPEC, the single throughput metric we report from a run is the highest throughput achieved in that run with an average latency of less than 40 ms. All graphs in this section show the throughput metric from at least 3 runs averaged together. Except as noted below, we comply with the SPECsfs97 run reporting rules in all regards, including uniform access.

6.1.1 Modifications to SPECsfs97

In order to use SPECsfs97 to benchmark Ursa Minor, we found it necessary to make a number of practical modifications to the benchmark parameters and methodology specified by SPEC. First, we modified the configuration file format to allow specifying operation percentages in floating point to better match the percentages revealed by trace analysis. Second, we doubled the warmup time for each run to 10 minutes to ensure the measured portion of the run did not benefit from startup effects. Neither of these changes should affect the workload presented during the timed portion of the run.

The potentially multi-object NFS operations are CREATE, which always involves a parent and child, and LINK, RENAME, and DELETE. The default SpecSFS97 workload contains no LINKs or RENAMEs. The SPECsfs97 load generator can be configured to generate those operations, but the source and destination directory will always be the same in both cases. Similarly, all generated DELETEs only involve files still in their original parent directory. To induce cross-directory operations, we modified the load generator to randomly select the target directory of a LINK. Doing the same for RENAME was not feasible given the implementation of the load generator. This modification should have no effect on the experiments in Section 6.3 using the SPECsfs97 operation mix, but will affect those using other operation mixes.

While SPECsfs97 is intended to stress all components of a storage system, we are most interested in isolating the performance of the metadata service—for our purposes, the NFS head-ends can thus be considered part of the load-generating system and not part of the system under test. In order to stress MDS performance, we must provision the Ursa Minor constellation so that the MDS is always the bottleneck. Doing so requires enough storage nodes to collectively hold the metadata objects in their caches—otherwise, the storage nodes become the bottleneck. The number of files used by SPECsfs97 is a function of the target throughput and, at high load levels, would require more storage nodes than we have available. Thus we reduce the number of files by half, correspondingly reducing the total size of metadata by half. To avoid confusion, we refer to this benchmark as *SFS-half*. With the reduction, SFS-half uses up to 14 million files and directories, requiring 44 GB of metadata storage. Finally, we configure the head-end NFS servers to discard any file data written to them and to substitute zeroes for any file data reads. The Ursa Minor metadata operations associated with file reads and writes are still performed, but the Ursa Minor data operations are not, so we can omit storage nodes for holding file data.

Although SFS-half has fewer files than SPECsfs97, the number of files is still a function of target load. If the head-end cache sizes are held constant, runs of SFS-half with different target load levels will experience different hit rates in the head-end cache. Because misses in the head-end cache must be served by the MDS, changing the head-end hit rate changes the operation mix seen by the MDS. To eliminate this confounding effect, we configure SPEC-half to use a constant number of files (8 million, requiring 26 GB of metadata or 4 million, requiring 13 GB) regardless of target load and refer to this benchmark as *SFS-fixed*.

6.1.2 Inducing multi-server operations

Since multi-server operations do cause additional overhead, it is important to consider the effect they have on overall performance. To examine this, so we modified the SFS-fixed workload so that LINK operations would always involve a directory other than the original parent. The default SPECsfs97 workload contains no LINKs; if $X\%$ of LINK operations are added, the resulting workload is referred to as *SFS-Xpct*.

To keep the total number of operations constant, we reduce the number of CREATE operations by one for every LINK operation we add. Thus, the sum of LINK and CREATE is a constant 1% of the NFS workload. The resulting MDS workload contains a higher fraction of both, because the head-end cache absorbs many of the LOOKUP requests. Both LINK and CREATE modify one directory and one inode, so any performance difference between the two can be attributed to the overhead of performing a multi-server operation instead of a single-table, and thus single-server, one. A RENAME, however, modifies two directories and their inodes and is slower than a CREATE even on a single server, which is why we use LINK as the source of cross-directory operations in this experiment.

Since each SPECsfs thread's directories are distributed across all the H tables used by that thread's head-end, and tables are distributed uniformly across servers, $(H-1)/H$ of LINKs will be multi-table. All of those will be multi-server as long as $H <= S$, where S is the number of metadata servers. When $H > S$, up to $(H-S)/H$ multi-table operations may hit a different table on the same server. Additionally, files that require a multi-server LINK will cause any READDIRPLUS requests on their new parent directory to also involve both servers.

6.2 Experimental setup

Table 6.1(a) lists the hardware used for all experiments, and Table 6.1(b) lists the assignment of Ursa Minor components to physical machines. This particular assignment was chosen to ensure as uniform hardware and access paths as possible for each instance of a component, given the constraints of available hardware and network configuration described in Section 6.2.1.

6.2.1 Hardware configuration

Given the physical location of the machines used for these experiments, ensuring adequate network bandwidth required careful attention to the network topology and node assignment. After several iterations, we

Type	Type A	Type B
Count	38	75
RAM	2 GB	1 GB
CPU	3.0 Ghz Xeon	2.8 Ghz Pentium 4
Disk	4× ST3250823AS	1× WD800J
NIC	Intel Pro/1000 MT	Intel Pro/1000 XT
OS	Linux 2.6.26	
Switch	3× HP ProCurve 2848	

(a) Hardware configuration.

Component	HW type	Quantity
Storage nodes	A	24
Metadata servers	B	32
NFS head-ends	A & B	48
Load generators	A	5
Root metadata server Root storage node Constellation manager	A	1

(b) Ursa Minor configuration.

Table 6.1: Hardware and software configuration used for all experiments. The number of metadata servers used varied; all other components remained constant. The root metadata server and its storage node only stored metadata for objects used by the metadata service. Metadata accessible by clients was spread across the remaining storage nodes and metadata servers.

arrived at the setup shown in Figure 6.1. All storage nodes, due to their physical location were attached to Switch 1. All metadata servers and most of the NFS head-ends were equally distributed over the remaining two switches. The root MDS and the remaining NFS head-ends were also attached to Switch 1. The inter-switch network consisted of 8 trunked 1 Gbps links between Switch 1 and each of the other two switches. All of the traffic between the metadata servers and the storage nodes, as well as half of traffic between head-ends and metadata servers, flowed over these links.

Of the theoretical 8 Gbps trunk capacity, we observed actual throughput of 5 Gbps in each direction. Due to limitations in the the link-aggregation policy used by the switches, traffic is not perfectly balanced across all links, and the degree of imbalance varied with the number of metadata servers used. The highest observed throughput on a single link was 900 Mbps, suggesting that the busiest link is limiting the total bandwidth across the trunk. Thus, the network may be a bottleneck for configurations with 32 or more metadata servers. To reduce bandwidth contention, the SFS load generators were attached to Switch 2 and Switch 3 and loaded only the NFS head-ends on the same switch. The NFS head-ends, however, accessed metadata servers on both switches equally.

To improve utilization for small experiments, the cluster could be split in two independent halves, one using half of Switch 1 and all of Switch 2, the other using the other half of Switch 1 and all of Switch 3. In this configuration, the inter-switch links carried all of the traffic between metadata servers and storage nodes, plus at most 10% of the head-end traffic.

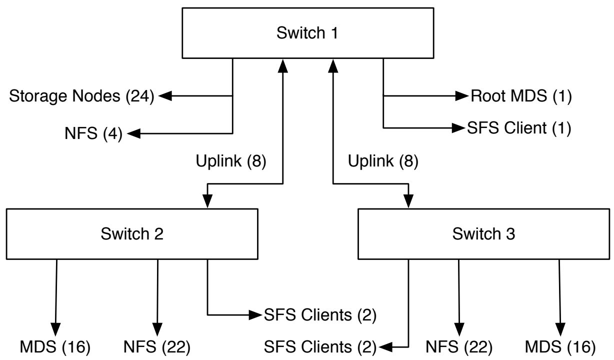


Figure 6.1: Network configuration. The network topology of the cluster used for experiments. The large constellation used all machines, and the two small constellations used all the machines on either Switch 2 or Switch 3 and half of those on Switch 1, with separate root metadata servers for each constellation.

6.2.2 Software configuration

We configured the test constellations with the goal of ensuring that the MDS was always the bottleneck. The root metadata server was only responsible for objects internal to the MDS (i.e., the metadata for the metadata table objects themselves) and the delegation map. Depending on the experiment, we used either a large constellation, with 48 NFS head-ends, or a small constellation, with 24 NFS head-ends. Two small constellations could operate simultaneously without interference. In either case, each head-end served 20 SFS client threads (960 or 480 in total), and the SOID range assigned to each head-end was split across 8 or 16 tables. The resulting 384 tables were assigned in round-robin fashion across metadata servers, so that every head-end used some object on each metadata server. Similarly, tables were stored on storage nodes (24 in the large constellation and 12 in the small) such that each metadata server used every storage node. These choices increase the likelihood of multi-table operations and contention, and they are intended to be pessimistic. The SOID assignment policy was configured to support a maximum of 4095 files per directory and 1023 subdirectories per directory, which was sufficient to avoid overflow in all cases. Each storage node used 1.6 GB of battery-backed memory as cache. In addition, each metadata server used 1 MB per table for its B-tree page cache and used as much memory as necessary to cache the metadata of 10,000 files and the contents of 1,000 directories. The head-ends had 256 MB each for their client-side metadata caches.

6.3 Scalability

This section evaluates how well Ursa Minor scales, in number of metadata servers, for various benchmark workloads, with and without multi-server operations.

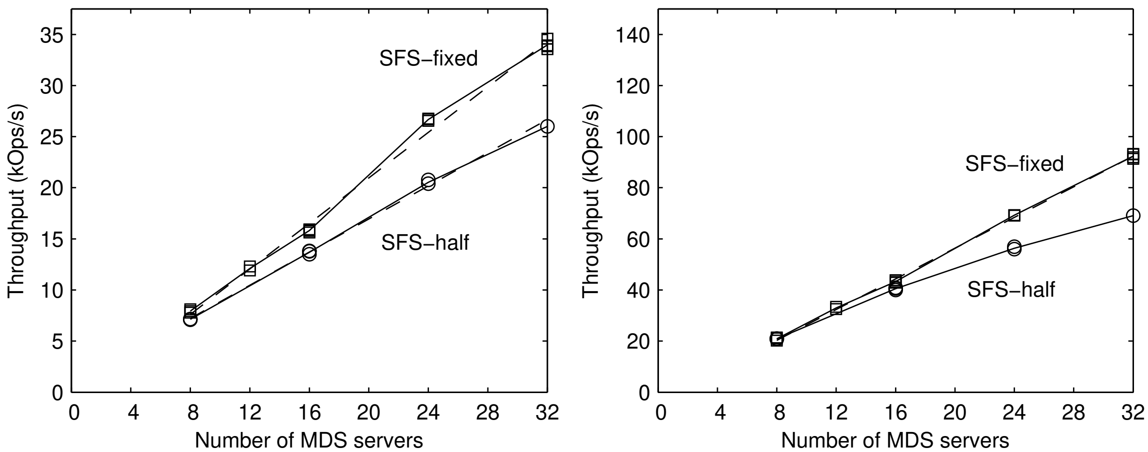
6.3.1 Without multi-server operations

Figure 6.2 shows that the Ursa Minor MDS is transparently scalable for the SFS-fixed workload. Specifically, Figure 6.2(a) shows that the throughput of the MDS increases linearly as the number of metadata servers increases. The NFS throughput of the head-ends also increases linearly, as shown in Figure 6.2(b). This is as expected, because the basic SPECsfs97 and SFS-fixed workloads cause no multi-server operations. Thus, adding additional servers evenly divides the total load across servers. Because the head-ends include caches and because the SFS operation rate includes NFS data requests, the number of requests that reach the metadata servers is lower than that seen by the head-ends. However, the workload presented to the MDS is much more write-heavy—26% of requests received by the MDS modify metadata, compared to 7% of NFS requests that definitely will modify metadata and 9% that possibly will. 63% of MDS requests involve multiple objects, but those objects are always in a parent-child relationship and assigned to the same table by the namespace flattening policy.

For the SFS-half workload, however, the MDS throughput shown in Figure 6.2(a) increases linearly, while the NFS throughput shown in Figure 6.2(b) increases sub-linearly. The reason for this is that the SFS working set size increases with throughput, whereas the head-end cache size remains constant. Thus, as the number of metadata servers increases, the miss rate of the head-end caches also increases, which both decreases NFS throughput and causes the operation mix seen by the MDS to contain an increasing proportion of reads.

6.3.2 With multi-server operations

The SFS operation mix is not representative of all workloads—the trace analyses in Chapter 3 reveals some traces to have very different operation mixes. To see how well Ursa Minor performs for such workloads, we configured SFS-fixed to use the operation mix from the DEAS [16] trace since it had a more read-dominated workload in contrast to SFS. The known cross-directory RENAMEs (0.005%) and potentially cross-directory LINKs (0.12%) are sources of potentially multi-server operations in this workload. As a result, 0.92%-1.02%



(a) MDS throughput

(b) NFS throughput

Figure 6.2: Throughput vs. number of metadata servers for workloads with no multi-server operations. The number of MDS operations and NFS operations completed per second are shown separately for the SFS-fixed and SFS-half workloads. Note the difference in scales between MDS and NFS operations, which is due to caching at the NFS head-ends. Each data point is the throughput reported by a single benchmark run and the averages of at least 3 runs with the same number servers are shown in solid lines. If a linear fit with correlation coefficient of > 0.995 was possible, it is shown in dashed lines. All runs used the large constellation with 8 million files across 48 head-ends with 8 tables per head-end (384 tables).

of the MDS operations caused by the DEAS workload (using the pathologically bad OID assignment from Section 6.1.2) would involve multiple servers. Figure 6.3 shows that Ursa Minor still scales linearly up to 24 metadata servers for such a DEAS-like workload. NFS performance for this DEAS-like workload is increased in comparison the the SFS-fixed workload, and is due to the difference in operation mixes, but MDS performance remains the same.

The benchmark chooses a random target directory for LINKS, with no regard to locality. Table 3.1 shows that LINKS that actually involve more than one directory in the DEAS trace are only 0.005% of the workload, and of those, the vast majority involve a nearby target directory. Thus, this benchmark represents a worst-case for a DEAS-like workload.

To separate the influence of the multi-server MDS operation rate from that of the rest of the workload, Figure 6.3 also shows performance for SFS-fixed modified as described in Section 6.1.2 to cause an MDS multi-server operation rate of approximately 1% while preserving the SFS read:write ratio. This is also linear, though lower in throughput than the unmodified SFS-fixed by 15%-25%. This is logical, since multi-server operations incur the additional overhead of migration. The magnitude of this penalty is heavily dependent on the Ursa Minor configuration—the same experiment scaled down and run on the small constellation had a slowdown of 0%-10%—and this dependency is discussed further in Section 6.5. Note that these slowdowns are relative to a workload with no multi-server operations; a system that exhibits 0% slowdown can execute multi-server operations as fast as single-server ones and is therefore optimal.

6.3.3 Root metadata server

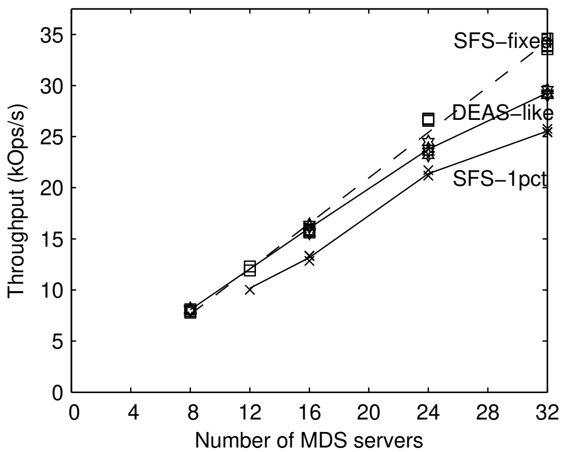
Both of the workloads with multi-server operations exhibit sub-linear throughput at 32 metadata servers. The reason is that the single root metadata server is saturated. In the absence of multi-server operations, the only requests that the root metadata server sees are UPDATE requests from other metadata servers whenever the size of a metadata table grows, which amounts to nearly zero load. Whenever there is a multi-server operation, in addition to the BORROW and RETURN, the root metadata server sees LOOKUP requests from the destination server as it tries to access a newly migrated table. Additionally, any NFS head-ends that request metadata in the migrated table will contact the original server, discover that their cached delegation map is stale, and request an updated table from the root metadata server. This imposes a significant workload on the root metadata server, as shown in Figure 6.4. Section 5.7 discusses ways to reduce the load on the root metadata server.

6.4 Sensitivity to workload

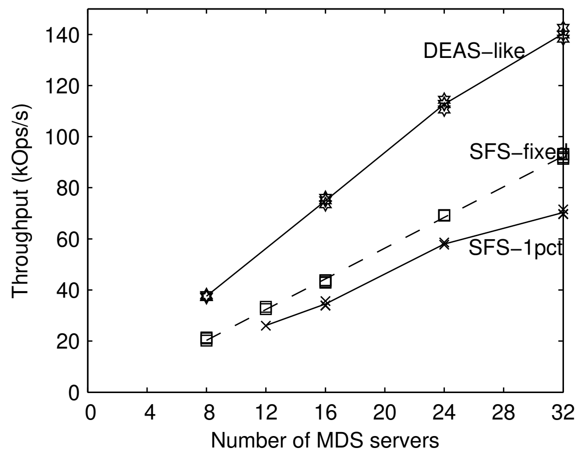
As seen in Section 6.3.2, the overheads incurred by multi-server operations depend on the workload. Specifically, they are expected to depend on the frequency of multi-server operations in the workload, on the number of files the workload involves, and on the mix and number of operations in the workload. This sections examines the influence of each of these properties.

6.4.1 Percentage of multi-server operations

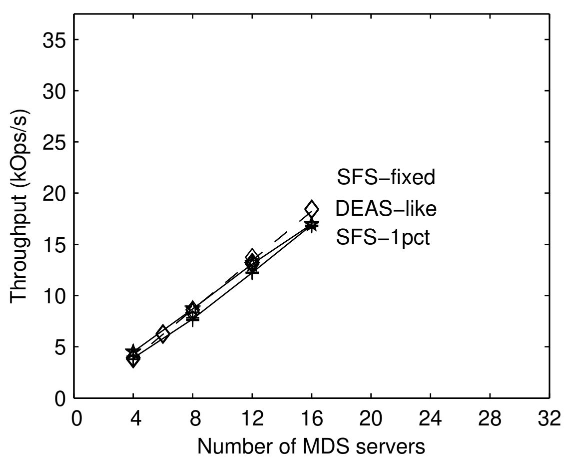
Figure 6.5 shows the MDS throughput for SFS-fixed as a function of the percentage of MDS operations that involve multiple servers. NFS cross-directory operation percentages of 0.05% to 1.00% resulted in MDS multi-server operation percentages of 0.07% to 4.75%. Because of the indirect control of the multi-server operation rate, repeated runs at the same cross-directory operation rate varied in the rate of multi-



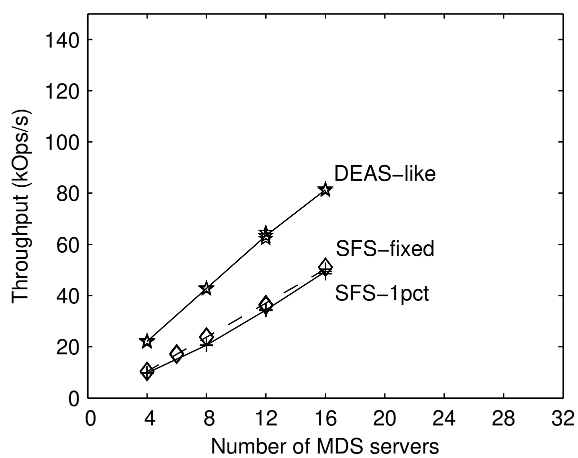
(a) Large constellation MDS throughput



(b) Large constellation NFS throughput

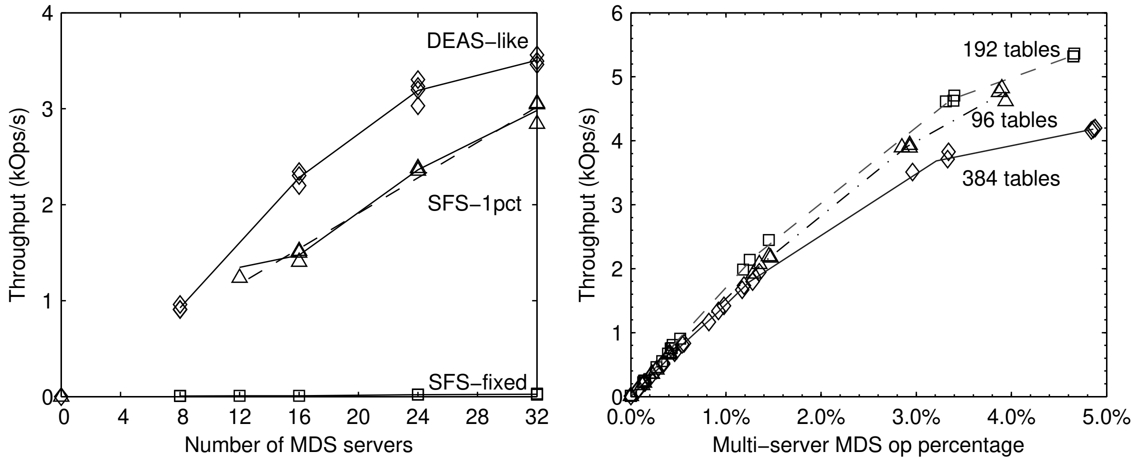


(c) Small constellation MDS throughput



(d) Small constellation NFS throughput

Figure 6.3: Throughput vs. number of metadata servers for workloads with multi-server operations. The number of NFS operations and MDS operations completed per second are shown separately for SFS-fixed using both the default SFS and DEAS operation mixes. The SFS-fixed workload causes no multi-server operations, whereas the DEAS-like workload causes approximately 1% of MDS operations to actually involve multiple metadata servers (21% involve multiple objects). NFS throughput is higher for DEAS because many of its requests are for file data, which can be served from the head-end cache. MDS throughput for both workloads is similar except at 32 servers, when the DEAS-like workload is bottle-necked by the root metadata server. For comparison, the SFS-1pct line shows the SFS-fixed workload (63% multi-object) modified to cause the same 1% multi-server operation rate as the DEAS-like workload. Thus the difference between SFS-fixed and SFS-1pct illustrates the overhead of multi-server operations. The top row shows runs using the large constellation with 8 million files across 48 head-ends with 8 tables per-head end (384 tables). The bottom row shows runs usisg the small constellation with 4 million files across 24 head-ends with 16 tables per-head end (384 tables).



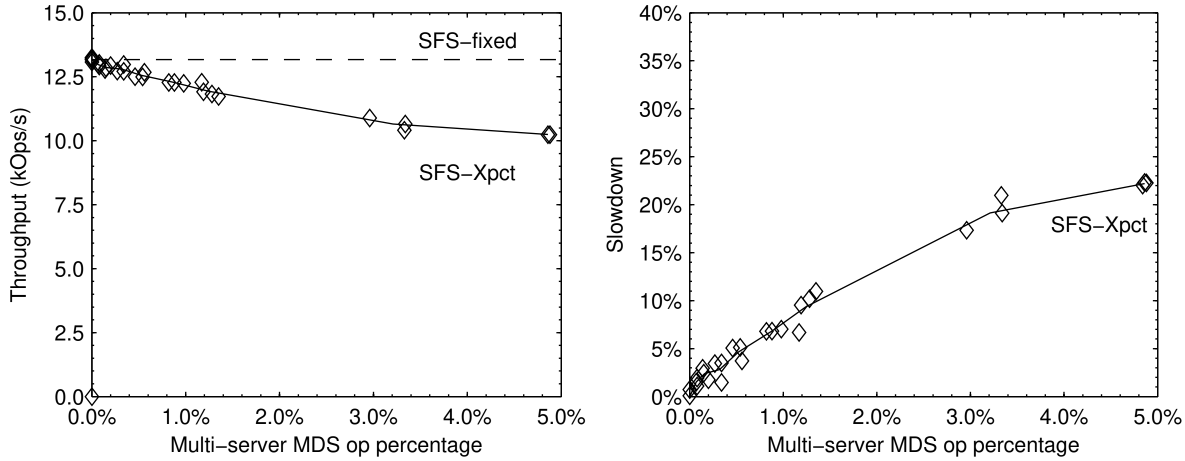
(a) Root throughput vs. number of servers

(b) Root throughput vs. percentage of multi-server ops

Figure 6.4: Root metadata server load. The number of MDS operations per second handled by the root metadata server for the DEAS operation mix is shown on the left. On the right is the root throughput required as the percentage of multi-server ops varies. In addition to those required to coordinate multi-server ops, these requests are either metadata servers requesting metadata for the metadata table objects or clients trying to update their delegation map. Note that, while the root throughput increases with the multi-server operation percentage, the overall throughput of the system, as shown in Figure 6.9(a), decreases.

server operations they caused. Because of this, we could not reliably obtain multi-server operation rates of less than 0.05%. Although 1.00% may seem like a small fraction of the total operations in the workload, for comparison, the common *data* WRITE operation makes up only 9% of the SpecSFS97 NFS workload. Indeed, these multi-server operation percentages far exceed those found in the trace analyses of Chapter 3, but are analyzed for completeness.

As expected, throughput decreases as the percentage of multi-server operations increases, since each multi-server op requires a table migration. Accordingly, the latency of LINKs are up to 4 $\times$ that of CREATEs, as shown in Figure 6.6(a). The influence of this on overall latency is shown in Figure 6.6(b): the migrations necessary for multi-server LINKs and READDIRs account for most of the increase in latency, while the remainder can be be attributed to increased I/O between the MDS and storage nodes and to increased time spent waiting to acquire locks inside the MDS.



(a) MDS throughput.

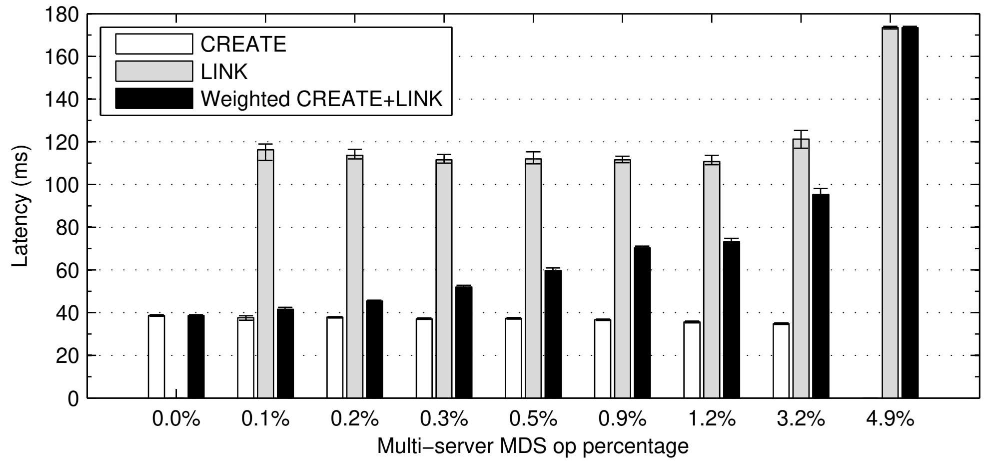
(b) MDS slowdown.

Figure 6.5: Throughput vs. percentage of multi-server operations. The numbers of MDS operations completed per second are shown in Figure 6.5(a) for the SFS-Xpct workload where the percentage of multi-server LINK operations (X) varies on the X-axis. The slowdown (relative to the case when no multi-server ops are present) is shown in Figure 6.5(b). The actual multi-server operation percentage achieved in a given run varies from the target percentage; the actual percentage is plotted for each run, and the lines connect the average of all runs with the same target percentage. All runs used the small constellation with 4 million files, 384 tables, and 12 metadata servers.

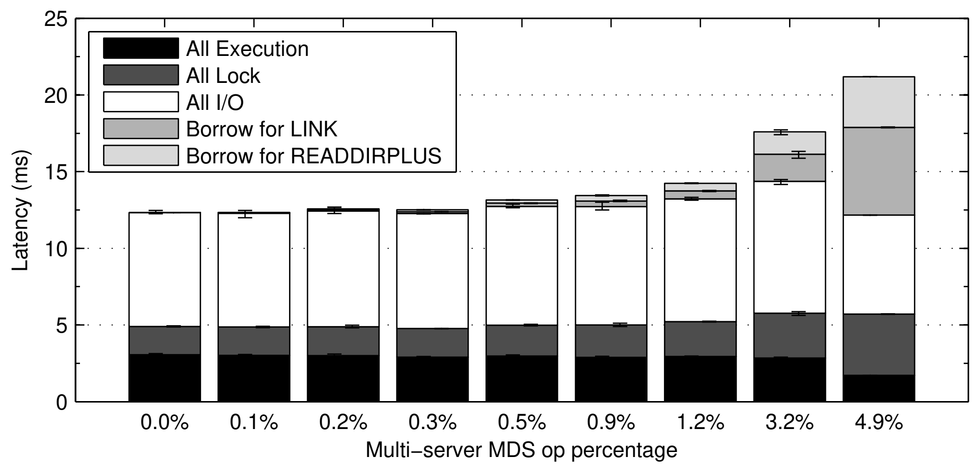
6.4.2 Workload size

The total number of files in the workload and the distribution into directories can affect performance it at least two ways. First, the number of files per directory and the depth of that directory in the hierarchy influence how often the SOID assignment policy encounters overflow. The effect of increasing overflow is to increase the rate of multi-server operations. Thus, for two workloads that differed only in the number of files, the workload with the larger number of files should have a larger fraction of its multi-object operations involve multiple servers, shifting its performance to the right along the curves in Figure 6.5. Since the directory hierarchy created by SpecSFS97 is simple, overflow never occurs, and this effect is not seen.

The second effect of varying the number of files in the workload will be to change the hit-rates of all of the metadata caches in the system; a smaller workload should see higher hit rates and thus perform better. Figure 6.7 shows the experiment from Section 6.4.1, but with additional SFS-fixed configurations using 2 million and 1 million files. When there are no multi-server operations, workloads with fewer files clearly perform better. When there are multi-server ops, with 2 million files, the slowdown is the same as with

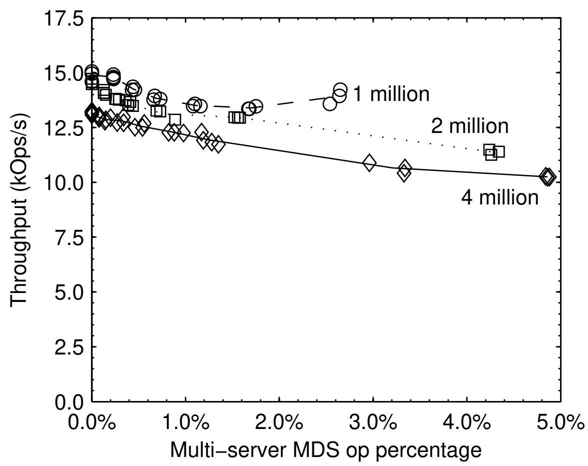


(a) CREATE and LINK alone

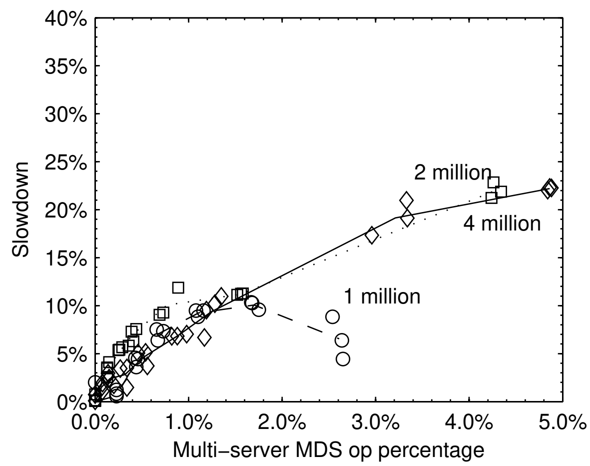


(b) Average over all ops

Figure 6.6: Effect on latency. The graph on the top shows latency of CREATE and LINK operations individually, as well as the weighted average of those two operations (CREATE and LINK represent a constant 1% of the NFS workload). On the bottom is the average latency over all operations, broken down into its major components. LINK operations require a borrow when the target file of the link is assigned to a different server than the new parent directory; a subsequent READDIRPLUS of the new parent directory will trigger a BORROW to read target file's inode. This experiment used the small constellation with 12 metadata servers and 384 tables.



(a) MDS throughput.



(b) MDS slowdown.

Figure 6.7: Influence of workload size. The numbers of MDS operations completed per second are shown in Figure 6.7(a) for SFS-fixed workloads with varying percentages of multi-server LINK operations. The slowdown (relative to the case when no multi-server ops are present) is shown in Figure 6.7(b). Separate curves are shown for workloads with 1 million, 2 million, 4 million and 8 million files. The actual multi-server operation percentage achieved in a given run varies from the target percentage; the actual percentage is plotted for each run, and the lines connect the average of all runs with the same target percentage. The runs with 8 million files used the large constellation with 384 tables and 12 metadata servers, while all others used the small constellation with 384 tables, and 12 metadata servers.

4 million files—although with fewer total files, fewer files are affected by each migration, the fraction of files affected by migration remains the same. With 1 million files, the slowdown is initially the same as for the larger workloads, but throughput actually increases when the percentage of multi-server operations is high. The reason is that with 1 million files and high LINK rates, the NFS head-end experiences more metadata client cache misses when performing NFS *data* operations than it does at low LINK rates. While the underlying reason for this behavior has not been determined, the extra metadata misses can easily be served from the metadata server's metadata cache and, thus, the MDS throughput increases because of the "easier" workload.

6.4.3 Operation mix

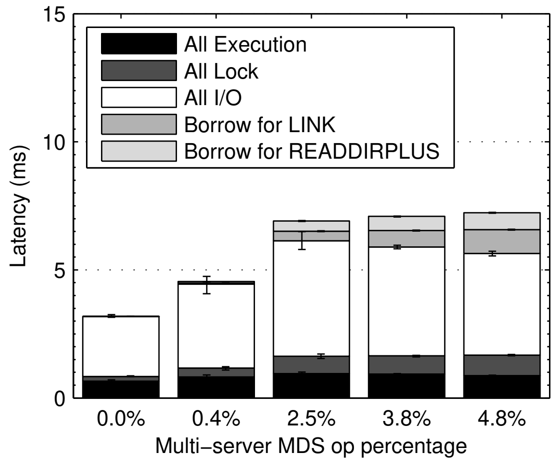
The exact distribution of operations and operation types in the workload will affect how much work a server must perform in order to service them. For instance, a workload that features mostly metadata modifying operations will require the server to perform a write for each one of them, while a workload that mostly reads from a small set of objects should be able to benefit from the server's cache and require little work to satisfy. These workloads will clearly perform differently even before multi-server operations are involved.

Figure 6.3, however, shows that the MDS performs similarly for both the DEAS-like and SFS-1pct workloads, despite the two workloads having very different operation mixes. While they cause similar rates of multi-server operations, SFS-1pct contains $> 10\times$ the number of metadata modifying operations as DEAS-like yet it sees only 17% lower throughput. Even though the head-end's client cache absorbs some metadata reads, as shown by the difference in NFS performance, the MDS sees 73% metadata reads for SFS-1pct compared to 31% for the DEAS-like workload. Examining the interaction between the metadata server and the storage nodes reveals that the latency seen by the metadata server when reading a B-tree page from the storage node is nearly the same as that for writing a page. Thus, a metadata modifying operation should have the same latency as metadata read that misses in the server's cache—only metadata reads that hit in the server cache will be faster. This behavior is common to both single-server and multi-server MDS operations.

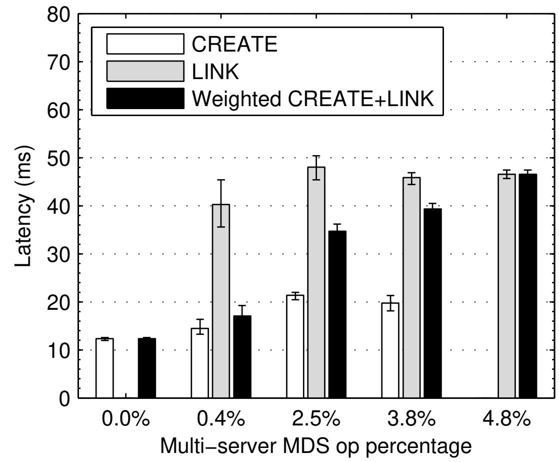
6.4.4 Operation Rate

The previous experiments considered Ursa Minor's performance at maximum load. Most deployed systems, however, operate at less than peak load most of the time [16, 37]. To explore the overhead of multi-server operations under such partial-load conditions, we configured the load generators to generate a constant 20 k NFS operations per second (approximately 55% of maximum load), while varying the multi-server operation rate.

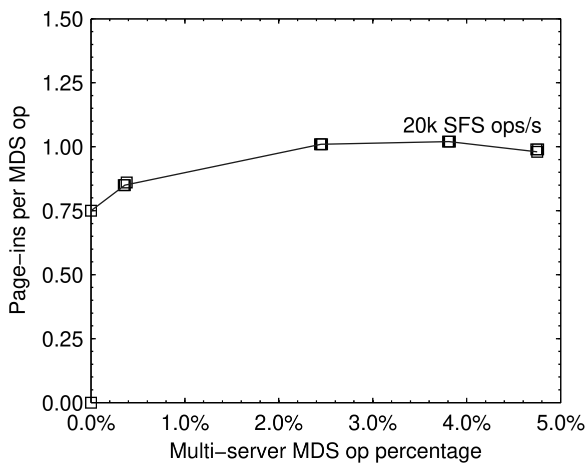
Figure 6.8(a) shows the average latency for all MDS operations required to service that workload. Compared to the maximum-load case shown in Figure 6.6(b), overall latencies, and particularly I/O latency, are lower due to the reduced contention. In both cases, the overall latency is almost doubled at high rates of multi-server operations but the increase is more rapid at partial load. Similarly, latencies for single-server CREATEs and multi-server LINKs in Figure 6.8(b) are both lower than in the full-load case in Figure 6.6(a) although the difference between LINK and CREATE latency is smaller at partial load. I/O latency increases with increasing percentages of multi-server operations because more B-tree page reads are required, as shown in Figure 6.8(c) and discussed further in Section 6.5.1. The end result is that, while both single-



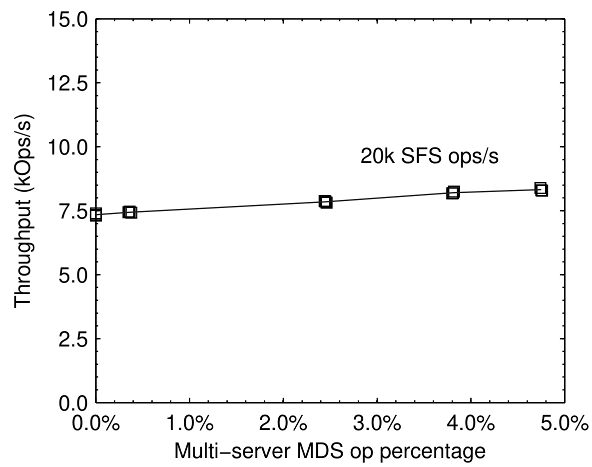
(a) Latency of all operations.



(b) Latency of CREATE and LINK.



(c) B-tree page-ins per MDS op.



(d) MDS throughput.

Figure 6.8: MDS under partial load. The performance of the MDS is shown for SFS-*X*pct workloads that present a constant 20,000 ops/sec workload to the NFS head-ends. The resulting MDS throughput is shown in Figure 6.8(d) and increases slightly with the rate of multi-server operations, due to the behavior of the head-end. Figure 6.8(a) shows the average latency for all MDS operations and Figure 6.8(b) shows the latency of CREATE, LINK and the average latency just those two operations. Figure 6.8(c) shows the average number of B-tree pages read during each MDS operation, and the increased reads correspond to the increased I/O latency in Figure 6.8(a).

server and multi-server operations are faster at partial load, the relative penalty of a multi-server operation is greater than it is at full-load.

Curiously, while the NFS load presented to the NFS head-ends remains constant, the number of MDS operations issued by the head-ends increases slightly as the percentage of multi-server operations increases. The increase primarily consists of extra single-object metadata reading operations.

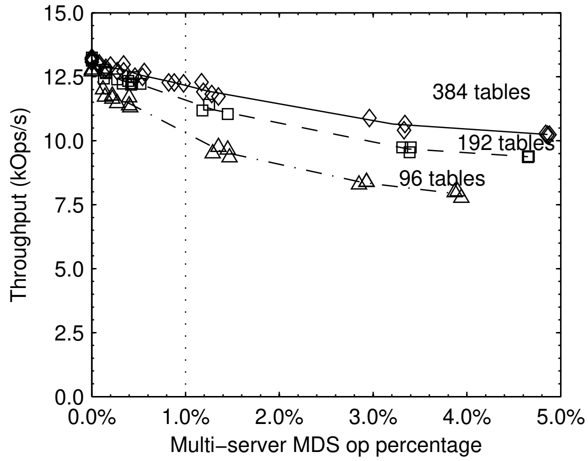
6.5 Sensitivity to system parameters

The overhead of migration, and thus the overhead of performing multi-server operations, is affected by the various parameters and features the system was designed and configured with. This section explores the parameters that directly affect migration overhead, the origins of those overheads, and the influence of common system tuning parameters.

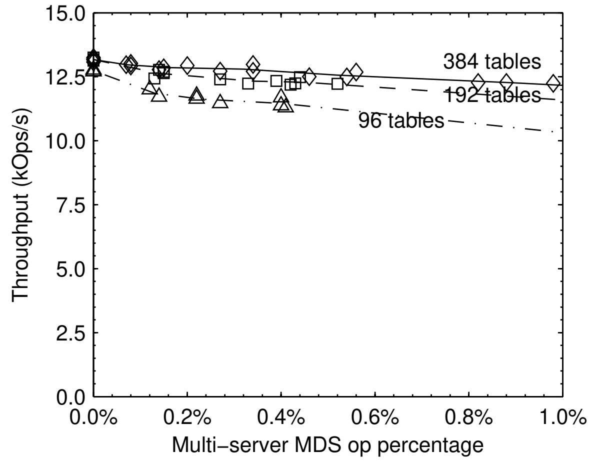
6.5.1 Migration granularity

One of the main factors that affects migration overhead is the granularity at which migration is performed. Since Ursa Minor can only migrate entire tables as a single unit, each table can only be involved in one multi-server operation at a time. Furthermore, a table is unavailable for serving any operations while the migration is in progress. Thus the higher the number of tables in the system, the lower the overhead of multi-server operations should be.

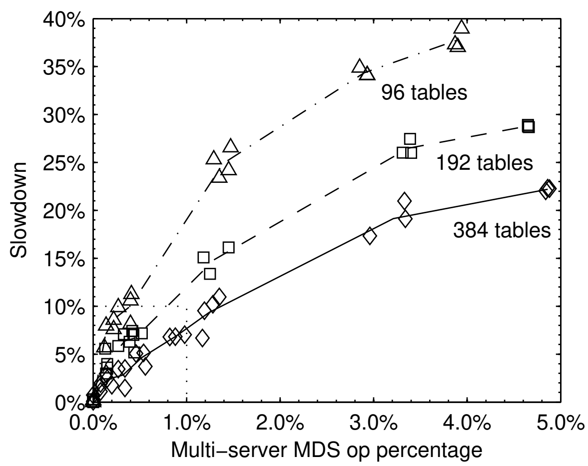
Figure 6.9(a) shows the MDS throughput for SFS-fixed as a function of the percentage of MDS operations that involve multiple servers, as in Section 6.4.1. Separate curves are shown for Ursa Minor configurations with 12 metadata servers, 24 head-ends, and each head-end's metadata split across 16, 8, or 4 tables for a total of 384, 192, or 96 tables in the system. As expected, configurations with coarser granularity suffer a greater slowdown for the same workload, when multi-server operations are present. Some of the decrease in throughput can be attributed to the decrease in total B-tree cache size (a constant 1 MB per table), but the slowdown for each configuration is computed against that configuration's throughput with no multi-server operations and, thus, should account for the degradation due to cache size. The remainder of the degradation is due to both the increased number of multi-server operations, which are themselves slower, and to side-effects of migration that slow down other single-server operations.



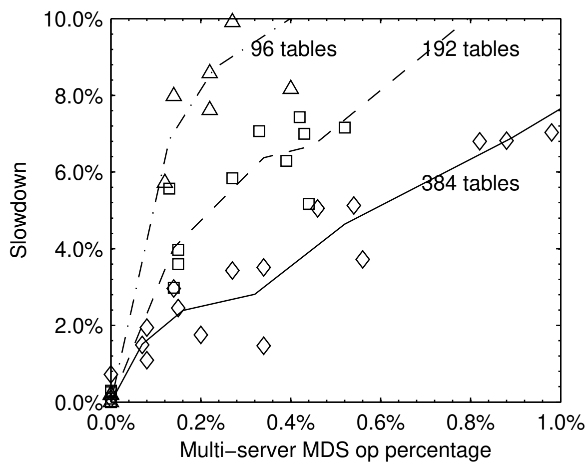
(a) MDS throughput.



(b) MDS throughput magnified.



(c) MDS slowdown.



(d) MDS slowdown magnified.

Figure 6.9: Influence of migration granularity. The numbers of MDS operations completed per second are shown in Figure 6.9(a) for SFS-fixed workloads with varying percentages of multi-server LINK operations. The slowdown (relative to the case when no multi-server ops are present) is shown in Figure 6.9(c). For clarity, the right column contains magnified plots corresponding to the dotted regions with low percentages of multi-server ops. The actual multi-server operation percentage achieved in a given run varies from the target percentage; the actual percentage is plotted for each run, and the lines connect the average of all runs with the same target percentage. All runs used the small constellation with 4 million files and 12 metadata servers. Separate curves are shown for configurations that used 384, 192, and 96 metadata tables.

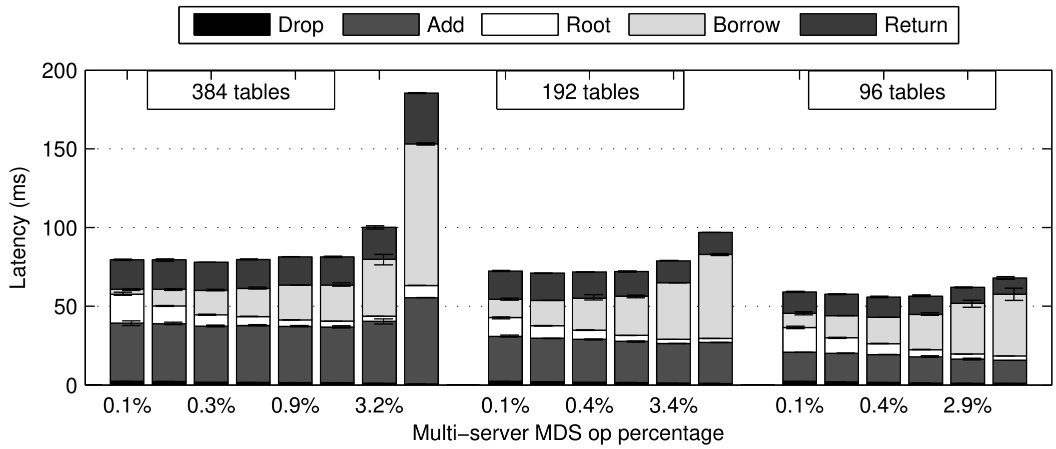


Figure 6.10: Components of multi-server operation latency. These graphs show the contribution each step of a multi-server operation makes to the latency of that operation. Each multi-server operation makes a complete cycle of BORROWing and RETURNing a table. Each BORROW and RETURN involves one server ADDing and the other DROPing the table, and the "Add" and "Drop" bars show the total time spent in the two ADDs and two DROPs in each cycle. The "Root" bar shows the time taken by the delegation coordinator's lock manager to lock both the tables in preparation for migration. The remainder of the time spent waiting for the BORROW and RETURN RPCs (not accounted for by the "Add", "Drop" or "Root" states) are shown separately. Figure 6.5.1 shows the latencies for three Ursa Minor configurations with 384, 192, and 96 tables, corresponding to the throughputs in Figure 6.9(a). Within each group, separate bars show workloads with varying percentages of multi-server operations.

Effect on multi-server operations

The latency of performing a multi-server operation, shown in Figure 6.5.1 is largely unaffected by migration granularity. This is logical, because the actions performed during a migration do not change with the size of the individual table being migrated. The only component that does change in latency of the ADDing a table. Each ADD requires reading a number of B-tree pages from the table's storage node. The latency for reading an individual page is 5.6 ms for 384 tables, 4.5 ms for 192 tables, and 3.2 ms for 96 tables, and the ADD latency decreases in similar proportion. The reason read latency is lower is that, regardless of the number of metadata server, the transaction layer described in Section 5.8 is limited to a single concurrent I/O operation per table. Thus, with fewer tables, the storage nodes see fewer concurrent requests, and are able to service them faster. The effect of table granularity on read latency would still exist even if all operations were single-server.

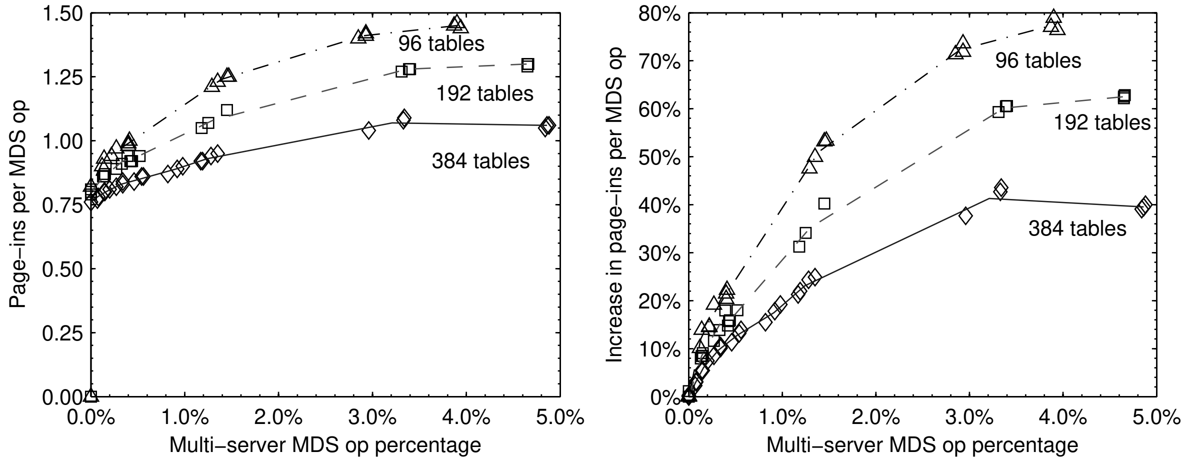
On the other hand, it is counter-intuitive that while latency decreases, throughput (shown previously in Figure 6.9(a)) also decreases. This is an artifact of how the SPECsfs97 benchmark functions; it tries progressively higher throughput targets until either the throughput achieved stops increasing or average latency exceeds a preset 40 ms threshold. Thus, a run that achieves a lower throughput will generally have fewer requests submitted than one which achieves a higher throughput. If requests were submitted at equal rates, the slower system would show higher latency, most likely exceeding the 40 ms threshold and cause the benchmark to terminate.

Effects on other operations

Each time a table is borrowed, the original server must flush that table from its cache, the destination server will have to fault in any pages it needs to service the cross-server LINK, and, once the table is returned, the original server will have to fault in any pages needed for subsequent requests. The pages would likely have been in the original server's cache if the cache were not flushed. Figure 6.11(a) shows that the total number of page reads does increase with the rate of multi-server operations. Some of these page reads are those mandatory misses required to service the LINK and accounted for in the latency of LINK. Excluding these reads yields the flush-induced increase in cache misses, shown in Figure 6.11(b), which mirrors the shape of the overall slowdown curve in Figure 6.9(c).

When metadata is partitioned into fewer tables, the number of non-LINK operations contending for the same table increases. Since each of these operations may need to read B-tree pages, the penalty for flushing a table from the cache increases as the number of tables decreases. Additionally, migrating a table makes it unavailable for serving other operations while the migration is in progress—when a single table represents a large fraction of the total metadata in the system, making one table unavailable has a large impact on overall performance. This is exacerbated in Ursa Minor, because threads within the metadata server contend for table-level locks. As a result, if a server has too few available tables, lock contention will limit performance even if there is spare CPU or I/O capacity. Given that small tables permit finer-grained load balancing, a reasonable Ursa Minor configuration might place 1%-10% of a server's capacity in a single table as suggested for other systems [7].

Increasing the number of tables increases the overhead of identifying which table any particular SOID resides in. Once the correct table has been identified, however, finding that SOID's record will be faster because the table contains fewer records, resulting in a shallower B-tree. A major penalty of having too many tables is that the delegation table will be large, possibly requiring a more efficient means of storing and distributing it. Additionally, the more tables the SOID-space is partitioned into, the more likely it is



(a) B-tree page-ins per MDS op.

(b) Increase in page-ins.

Figure 6.11: Page-ins per MDS operation vs. percentage of multi-server operations. The average number of MDS B-tree pages read during each MDS operation is shown on the left for SFS-fixed workloads with varying percentages of multi-server LINK operations. The increase in page-ins (relative to the case with no-multi server operations) for the experiments in Figure 6.9 is shown on the right.

that a multi-object operation will be multi-table. But, if the assignment of tables to servers also accounts for locality, the probability of a multi-server operation might not change.

6.5.2 Server-local state

A second factor that directly affects migration overhead is the amount of state that must be migrated from the source server to the destination server. In the current implementation of Ursa Minor, the servers maintain no local state that cannot be recovered from the metadata tables stored on the storage nodes. If Ursa Minor were to be extended in the future to support client cache consistency, each server would need to maintain a few bytes of local state per open file, per client. This may add up to tens or hundreds of kilobytes per table. If the metadata servers maintained a write-back cache, rather than write-through, the dirty data in the cache would need to be flushed either to the storage nodes or transferred directly to the new server (as is done in Ceph [52]). This may involve a few megabytes being transferred between servers. In a system designed without a back-end storage pool shared between servers, the entirety of the metadata table would need to be copied from one server to another.

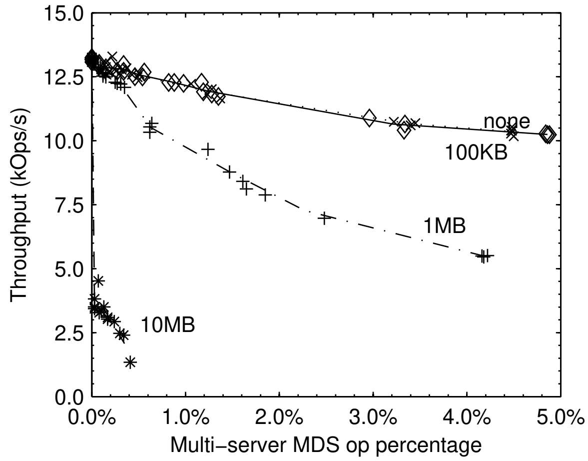
To explore this issue, we modified Ursa Minor to copy a specified amount of dummy data from the source server to the destination server during each migration. Figure 6.12 shows the effect on throughput of copying 100 kB, 1 MB, and 10 MB of local state during migration. While 100 kB of local state makes little difference in overall throughput, 1 MB causes a significant slowdown. With 10 MB, the degradation is severe enough that SPECsfs97 cannot complete a valid run when more than 0.5% of operations require migration. For comparison, for the 384 table configuration used, a single table is approximately 32 MB in size.

It is worth noting that the Ursa Minor RPC system is not specialized for bulk data transfers. On the hardware we used, the RPC system can utilize at most 450 Mbps of each server's network link. With no migrations, each server's baseline traffic is about 350 Mbps, leaving only 100 Mbps of additional capacity for state transfers. If migration requires more bandwidth, it will come at the expense of the foreground workload. The latency of state transfers, shown in Figure 6.12(c) suggest that it only achieves 15 Mbps to 66 Mbps.

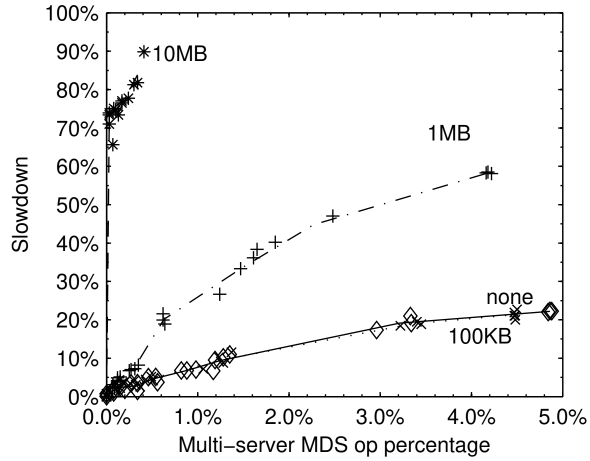
6.5.3 Server cache size

In previous experiments, we observed performance effects that could be, at least partially, attributed to changes in cache sizes or behavior. For example, changing the number of servers also changes the aggregate sizes of the server metadata and directory caches (which are a fixed size per server), and changing the number of tables changes the aggregate size of the B-tree page cache (which is a fixed size per table).

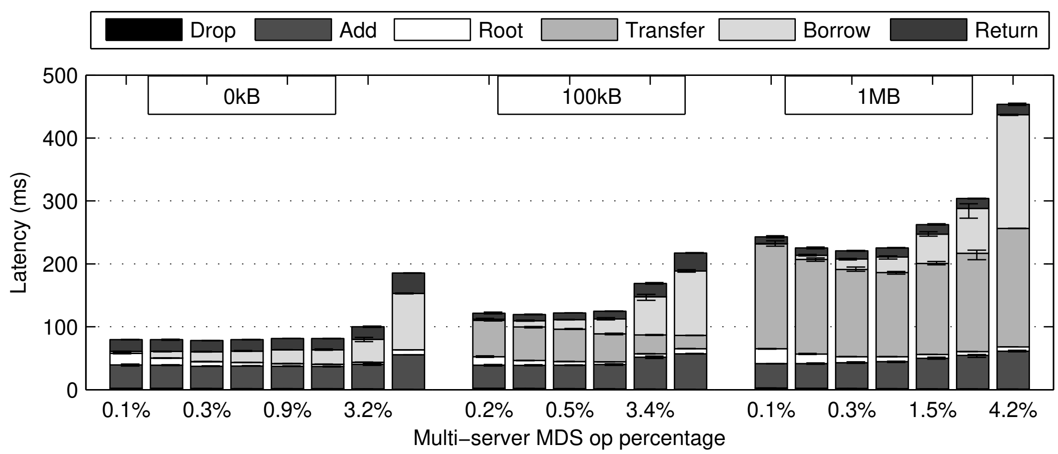
To separate the effect of cache size from the effect of the other variables, we performed a series of experiments that varied only in cache size. Figure 6.13(a) shows the throughput when the workload and Ursa Minor configuration are constant, but the size of the server's metadata cache varies. When the workload contains no multi-server operations, increasing cache size increases throughput, as should be expected. As the percentage of multi-server operations increases, the mandatory flushes triggered by migration mean that even a small cache does not have enough time to refill before the next migration forces it to be flushed again. Thus the throughputs converge the the same curve when more than 1% of the workload is multi-server. The corresponding slowdown plot in Figure 6.13(b) shows a persistent difference, but this is only because the baseline each curve is normalized against (its throughput with no multi-server operations), varies.



(a) MDS throughput.

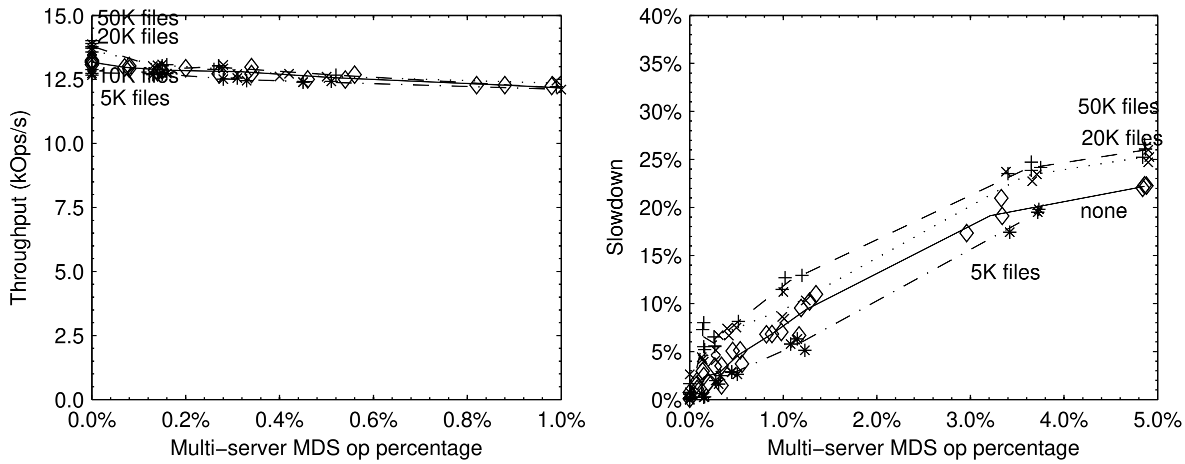


(b) MDS slowdown.



(c) Multi-server operation latency.

Figure 6.12: Influence of server-local state. The numbers of MDS operations completed per second are shown in Figure 6.12(a) for SFS-fixed workloads with varying percentages of multi-server LINK operations. The slowdown (relative to the case when no multi-server ops are present) is shown in Figure 6.12(b). Separate curves are shown for configurations in which 0 kB, 100 kB, 1 MB, and 10 MB of state must be transferred between servers on every migration. Figure 6.12(c) shows the latency overhead of multi-server operations for each configuration, broken into the same categories used for Figure 6.5.1, with the addition of a "Transfer" category for the time taken to migrate any server-local state. All runs used the small constellation with 4 million files, 384 tables, and 12 metadata servers. The 0 kB configuration corresponds to the 384 table configuration in Figure 6.9.



(a) MDS throughput.

(b) MDS slowdown.

Figure 6.13: Influence of server cache sizes. The numbers of MDS operations completed per second are shown in Figure 6.13(a) for SFS-fixed workloads with varying percentages of multi-server LINK operations. The slowdown (relative to the case when no multi-server ops are present) is shown in Figure 6.13(b). Separate curves are shown for configurations in which the metadata server could cache metadata for 5000, 10000, 20000, and 50000 files. All runs used the small constellation with 4 million files, 384 tables, and 12 metadata servers. The 10000 file line corresponds to the 384 table line in Figure 6.9. For clarity, the region plotted corresponds the magnified region in Figure 6.9; the curves are indistinguishable at higher multi-server operation rates.

Changing the size of the B-tree page cache, from 1 MB per table to 2 MB or 0.5 MB, does not, however, have any significant effect on throughput. Even at the largest size, the B-tree page cache can hold only 11% of the total B-tree pages, so it is likely is caching mostly the frequently-used internal nodes of the B-tree, but not many of the leaves, so most requests still incur a miss for the leaf nodes—the average operation reads least 0.7 pages, as shown by Figure 6.11(a). Much larger cache sizes might allow caching enough leaf nodes to see a difference, but would exceed the memory available on the metadata server nodes. These results suggest that the experiments that varied the number of servers or number of tables were not affected by their side-effect of varying aggregate cache sizes.

6.6 Implementation difficulty

Our motivation for using migration to handle multi-server operations was that it was the simple solution for the problem at hand. From the starting point of a metadata service that supported migration and single-server operations (over 47000 lines of C code), it only required 820 additional lines of code to support multi-server operations. Of these 820 lines, the global lock manager (necessary for avoiding deadlock) accounted for 530 lines, while the remainder were additional RPC handlers and modifications to the local transaction layer to trigger a BORROW when necessary. In contrast, implementing migration correctly represented 9000 lines of the original metadata server and several months of work.

To provide a basis for comparison, we created a version of Ursa Minor that implements multi-server operations using the traditional two-phase commit protocol. This version is not nearly as robust as the main version, particularly with regard to failure conditions, so the 2587 lines required to implement it can be considered to be a lower bound. The global lock manager is included in the total because it is still necessary, but the code to implement an undo/redo log is not.

Since lines of code do not always equate to the complexity of a system, another metric to consider is the number of states a system can be in and the decision points that transition between them. Figure 6.14 shows a flowchart of the execution of a multi-server operation using migration. The regions highlighted in gray represent components that are assumed to already exist and used in unmodified form. For comparison, Figure 6.15 shows the execution of a multi-server operation using two-phase commit. In order to reduce the size of the diagram, some states, such as “Abort” and “Begin”, represent subprocesses that may themselves involve two or more steps. Even with this simplifacation, two-phase commit requires over 33 states, whereas reusing migration requires only 17, of which 8 already exist. Reusing migration clearly involves fewer states and transitions, because much of the complex failure recovery cases that two-phase commit must handle are encapsulated within the migration system.

6.7 Additional observations

Many of the choices we made in designing the MDS were guided by the properties of the rest of Ursa Minor. Since the Ursa Minor storage nodes include NVRAM, logging, and shared access, the MDS relies on these features rather than implementing its own write-ahead log or using locally-attached storage. Other systems with different underlying storage or failure models might choose to store metadata on the local disks or NVRAM of each metadata server. Migration in such a system would be more expensive, because it would require copying metadata from server to server.

One surprising observation in our experiments was that the network traffic between storage nodes and metadata servers was an order of magnitude greater than that between the metadata servers and their clients. While each object requires approximately 1 kB of metadata, the B-tree implementation we use does not pack pages tightly. Furthermore, every modification to a page requires that the entire 32 kB page be written to the storage node. The page size was chosen to match the storage node's most efficient block size, but if network bandwidth is a limiting resource, the bandwidth savings of using a smaller block size may outweigh the disadvantage of more overhead at the storage nodes. Likewise, the effects of page size on read bandwidth also call for further examination.

6.8 Discussion

6.8.1 Optimizations

The concept of migrating tables rather than performing a multi-server operation, can be thought of as using coarse-grained exclusive locking to control access to shared tables. Some multi-object operations, such as READDIRPLUS do not modify any state, and could benefit from read-only locking. Allowing tables to be shared for read but be exclusive for write, as is done in other systems [30, 47], would benefit both overall scalability and avoid the need to flush caches during a migration. It would, however, introduce the need for a potentially complex cache consistency protocol amongst metadata servers. When using migration to avoid multi-server operations, borrowed tables are generally returned to the original server with only a few modified pages. If the original server knew which pages those were, it would only need to flush those modified pages from its cache instead of flushing all pages. Including a list of modified pages in the migration protocol should a relatively simple enhancement. Previous experiments on a partial implementation of the MDS suggest that such an optimization would reduce slowdown by an order of magnitude. The improvement would come at the cost of the extra implementation complexity of the cache invalidation protocol.

In general, determining an appropriate tradeoff of costs and benefits is the responsibility of the system designer. Taken to an extreme, additional performance improvements (e.g., modifying the migration mechanism to migrate single objects) may end up being more complex than implementing a general distributed

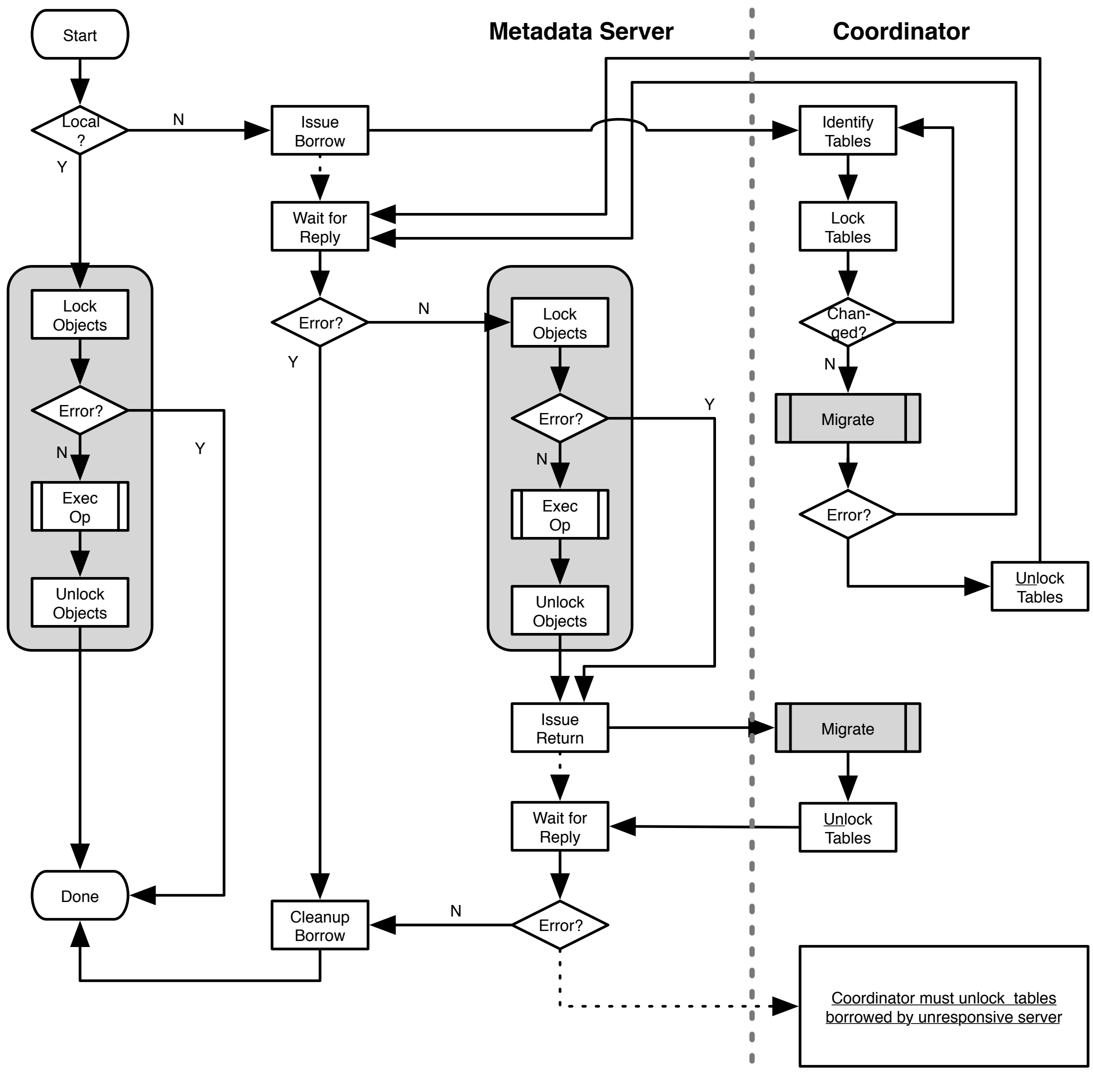


Figure 6.14: Multi-server operations implemented using migration. The significant states and decison points involved in executing an operation are shown. The gray regions represent processes that already exist for handling single-server execution and migration.

transaction protocol in the first place. Using a dedicated protocol for multi-server operations is a single point in the cost vs. benefit space. Reusing the existing migration system is another single point. Extensions to the migration mechanism (e.g. cache consistency, faster state transfer, or fine-grained migration) add additional points in the tradeoff space for the designer to consider.

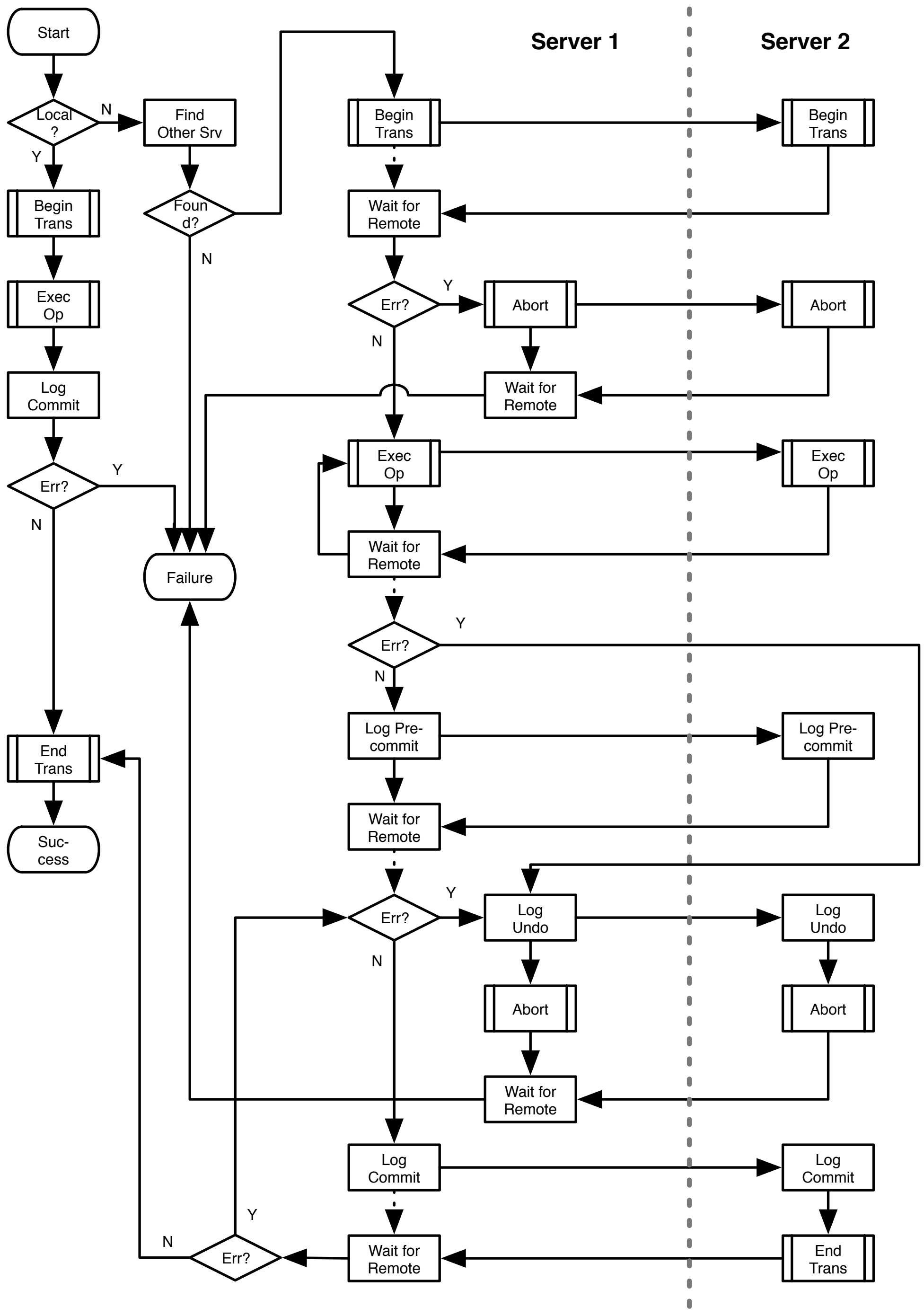


Figure 6.15: Multi-server operations implemented using two-phase commit. The significant states and decison points involved in executing an operation are shown. Common sub-operations, such as "Begin", which must acquire locks and perform deadlock avoidance, and "Abort", which must roll back logs, are encapsulated in a single state consisting of several sub-states and transitions.

6.8.2 Adverse workloads

The experiments in this chapter consider only the average, steady-state behavior of Ursa Minor. Other workloads may be non-uniform or bursty, which are difficult to emulate with the benchmark and experimental infrastructure used here. Also, while our approach to multi-server operations is intended for workloads where multi-server operations are rare, the possibility exists that a system may be presented with an adverse workload where multi-server operations are common, or that a workload with a low average rate of multi-server operations may contain bursts with higher rates.

While the experiments performed do not include those cases, the behavior under those conditions can be extrapolated from existing experiments and knowledge of the system. The worst possible workload is one that consists of only multi-server operations involving the same object. Because that object's table will be involved in every operation, and there can only be on active transaction on a table at a time, every operation in progress across the system will be serialized on that single table. Thus, the throughput of the entire MDS will be limited to that of a single table and further limited by the overhead of migrating that table between servers for every operation.

A workload consisting of only multi-server operations but without contention for the same table would perform somewhat better. In this case, every operation will involve a table migration that has to be coordinated by the root metadata server. Because the operations do not conflict with each other, all metadata servers could execute operations in parallel. The overall throughput of the system would be limited by the throughput of the root metadata server, and scaling the number of metadata servers would not increase the overall MDS throughput once the root metadata server is saturated.

In general, any system that relies on parallelism in order to scale will be be penalized by a workload that requires serialization. In a workload with a mix of single-server and contending multi-server operations, the single-server operations will be largely unaffected by the degree of contention, but each contending multi-server operation will have to wait to be serialized. This results in an increasing difference in latency between the two classes of operations, which may be a concern for latency-sensitive applications.

In the SFS-based experiments, contention for the same table was rare, except at high multi-server operations rates, simply because multi-server operations themselves were rare. Additionally, the benchmark was structured such that any particular table would only be involved with multi-table operations with a preset group of other tables — those used by the same NFS head-end. Thus, the number of operations that could potentially contend for the same table was limited to 16 in most experiments. During the untimed setup phase of the benchmark, some tables did experience the maximum degree of contention (because that table contained a top-level directory that was analogous to a "directory of mountpoints"), but during the timed execution phase, multi-server operations on a given table were spaced far apart enough in time that one would complete before the next one began.

Contention for the same objects will result in the same table migrating continuously between servers. One way to improve performance under contention is to detect when a table, T, is being continuously migrated and determine the set of other tables that are involved in multi-table operations with table T. If the entire set were to be served by one server, then all the multi-table operations will be single-server, avoiding the overhead of coordination and migration. It would, however, require more intelligence on the part of the delegation coordinator to detect this condition and optimize for it.

6.8.3 Applicability to other systems

The experiments in Section 6.4.1 characterize performance as as a function of the fraction of the workload that involves multiple servers. While the general reduction in throughput with increasing multi-server operation percentage should generalize to other systems that implement multi-server operations using migration, the shape of the curve is specific to this implementation of Ursa Minor. As shown in Section 6.5.1, two factors that affect the degree of slowdown are:

- The latency of migration.
- The penalty that migration imposes on single-server operations.

The range of migration latencies studied in Section 6.5.2 are intended to reflect those that could be expected from other systems that use shared storage, as is the case in many OSD or SAN-based systems. Ursa Minor's migration mechanism was not particularly optimized, and it is possible migration may be faster in other systems. In particular, migration in Ursa Minor has the same priority for access to the CPU, disk, and locks as any other operation. Giving migration a higher priority would reduce the queueing time of a migration request and substantially speed up migration. While such an optimization would assist multi-server operations, it would also assist load-balancing by speeding up the migration of objects away from an overloaded server, and thus be a reasonable optimization for other systems to use.

The high penalty of migration on single-server operations seen in Section 6.5.1 reflects Ursa Minor's naive approach to maintaining metadata server cache consistency. Such a simple approach is appropriate if migrations are extremely rare, however, systems such as AFS and OntapGX that expect migration to be even less frequent still take measures to mitigate migration's effect on other concurrent operations [14, 30]. Thus, other systems can be expected to have the same or lesser penalties than seen in the Ursa Minor experiments. As discussed in Section 6.8.1, it is possible to reduce the penalty in Ursa Minor at the cost of additional complexity.

Additionally, in a system that maintained client callback state, as Ursa Minor does not, migration would have to either break callbacks or transfer the callback state to the next server. The first approach would both affect clients, which would have to reacquire callbacks, and the next server, which would have to process a flood of callback reacquisition requests from active clients. The latter approach adds to the state that must be transferred during migration, potentially increasing the migration latency and requiring extra code to perform the transfer. Simply breaking callbacks would be the simplest approach, because the callback break and reacquire code paths are already used in normal operation and would be reasonable for a system that expected migration to be used for load-balancing alone and could tolerate brief periods of client slowdown during migration. If migration were used more frequently, the client-visible slowdown may become unacceptable. Existing systems use both approaches, but the use of migration to support multi-server operations may influence the choice of which approach a future system should use.

6.9 Summary

These results show that the Ursa Minor metadata service is transparently scalable for both standard workloads and for workloads far more severe than experienced by large deployed file-systems as represented by the traces analyzed in Chapter 3. In addition to scaling linearly, Ursa Minor's throughput when executing multi-server operations is only slightly less than optimal, as long as the amount of server-local state that needs to be migrated is under 100 kB. When the amount server-local state is large, the overhead of migration is impractically large, except at very low multi-server operation rates, such as those observed in the traces.

The key architectural limitation to scalability is the single, centralized, root metadata server, which can be addressed using techniques described in Section 5.7. The load seen by the root metadata server depends on the aggregate number of multi-server operations across the entire constellation, and not directly on the number of metadata servers—at the rates seen in the traces, the root would be able to handle far more servers.

Chapter 7

Conclusion

This dissertation that shows that reusing migration to implement multi-server operations is a simple and efficient method of providing consistent semantics in a transparently scalable distributed storage system. It presents the following three conclusions:

Multi-server operations are very rare in file system workloads. Analyzing traces of deployed large-scale file systems reveals that potentially multi-server operations make up less than 0.01% of the operations in each trace. If each server stores a contiguous subtree of the file system, operations that would involve multiple servers occur less than 4 times in every million operations.

Object-IDs can be assigned in a manner that ensures that subtrees receive numerically similar Object-IDs. Using an namespace flattening policy that encodes the namespace into the Object-ID preserves the benefit of keeping subtrees together in a contiguous range of Object-IDs. Assigning ranges of Object-IDs to each server allows the use of a simple mechanism for identifying which server hosts a particular object, while reducing the liklihood of multi-server operations. This provides the benefit of subtree-based division without the extra complexity required to identify which server to contact in such a scheme.

The overhead of using migration to implement multi-server operations is small for practical workloads. A prototype transparently scalable system can execute multi-server operation, at the rates seen in traces, with negiligible overhead. Even when multi-server operations make up 0.1% of the workload, 10× more frequent than the most pessimistic expectation from the trace analyses, the prototype system is within 1.5% of the optimal throughput as long as server-local state is small. This approach is suitable for building other transparently scalable file systems and also for adding transparent scalability to existing scalable, but not transparently scalable, systems.

Appendix A

Appendix A

A.1 MDS operation list

Name	Objects	Tables	Description
MDSCreateObject	1	1	Creates a new object.
MDSReleaseObject	1	1	Deletes an object.
MDSEnumerateObjects	0	1	Lists a range of objects.
MDSCreateStream	1	1	Creates a new stream in an object.
MDSReleaseStream	1	1	Deletes a stream from an object.
MDSReleaseData	1	1	Delete's a stream's data, but retains attributes.
MDSEnumerateStreams	1	1	Lists the streams in an object.
MDSLookup	1	1	Returns metadata for an object or stream.
MDSLookupExtra	1-N	1	Same as lookup, but may prefetch nearby objects.
MDSApproveWrite	1	1	Acquires a write capability.
MDSFinishWrite	1	1	Uses write capability to extend a stream.
MDSSetAttr	1	1	Sets an existing object's attributes.
MDSGetAttr	1	1	Gets an existing object's attributes.

Table A.1: List of MDS operations used by the NFS head-end.

A.2 NSS operation list

Name	Objects	Tables	Description
NSSCreate	2	2-4	Creates a new file or dir and inserts in its parent.
NSSLink	2	1	Links an existing file into an existing directory.
NSSUnlinkByName	2	2-4	Unlinks a file or directory and deletes it if empty.
NSSLookupByName	2	2-3	Returns metadata for the object with given name.
NSSReaddirBySoid	1	2	Returns the names and SOIDs for all children.
NSSReaddirBySoid+	1-N	2-N	Same as above, but includes children's metadata.
NSSSetAttrBySoid	1	1	Sets a file's attributes.
NSSGetAttrBySoid	1	1	Gets a file's attributes.
NSSRename	2-4	2-6	Renames a file to a different name.

Table A.2: List of NSS operations used by the NFS head-end.

A.3 NFS induced Ursa Minor operations

NFS operation	Resulting Ursa Minor operations
NFS3GetRoot	NSSGetAttrBySoid
NFS3GetAttr	NSSGetAttrBySoid
NFS3GetAttr	NSSSetAttrBySoid
NFS3Lookup	NSSLookupByName
NFS3Access	NSSGetAttrBySoid
NFS3Readlink	MDSLookup
NFS3Read	MDSLookup
NFS3Write	MDSLookup, MDSApproveWrite, MDSFinishWrite
NFS3Create	NSSLookupByName, NSSCreate
NFS3Mkdir	NSSLookupByName, NSSCreate
NFS3Symlink	NSSLookupByName, NSSCreate, MDSLookup, MDSApproveWrite, MDSFinishWrite
NFS3Mknod	NSSLookupByName, NSSCreate
NFS3Unlink	NSSLookupByName, NSSUnlinkByName
NFS3Rmdir	NSSLookupByName, NSSUnlinkByName
NFS3Rename	2xNSSLookupByName, NSSRename
NFS3Link	2xNSSLookupByName, NSSLink
NFS3Readdir	NSSReaddirBySoid
NFS3ReaddirPlus	NSSReaddirBySoid+
NFS3FSInfo	NSSGetAttrBySoid
NFS3Commit	NSSSetAttrBySoid, NSSFinishWrite

Table A.3: Ursa Minor operations generated by the NFS head-end when servicing NFS operations.

A.4 Power consumption

All the equipment used to evaluate Ursa Minor was located in the Data Center Observatory, which has detailed power monitoring support. The most common experiment configuration was the small constellation using 12 metadata servers. A single run, determining one data point, of this configuration increased the total DCO current draw by 21 Amps for approximately 1.5 hours. Note that the baseline idle current draw is not included. Assuming a power factor of 1.0:

$21\ A \times 208\ V \times 1.5\ H = 6.6\ kWH$ per small run.

Runs on the large constellation used twice the hardware of and thus should double the small constellation, and the complete set of experiments consisted of 1899 runs, 1416 on the small constellation and 483 on the large. Thus:

$1416\ runs \times 6.6\ kWH/run + 483\ runs \times 13.2\ kWH/run = 15.7\ MWH$

To put that in perspective, the calorific content of coal is 12000 BTU/lb-15000 BTU/lb, and that of ordinary office paper is 6234 BTU/lb [19]. Thus, the same amount of energy could be produced by combusting 3456 lb-4464 lb of coal or 8593 lb of paper with perfect efficiency. As this does not account for generation and transmission inefficiencies, far more fuel may be required in practice.

$15.7\ MWH/(6234\ BTU/lb \times 0.29307\ WH/BTU \times 10^-6\ MWH/WH) = 8593\ lb$

For comparison, this dissertation, when printed on 20 lb paper, weighs:

$122\ pages \times (20\ lb/ream/2000\ pages/ream) = 1.22\ lb$

Bibliography

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-Scalable Byzantine Fault Tolerant Services. *ACM Symposium on Operating System Principles* (Brighton, United Kingdom, 23–26 October 2005), pages 59–74. ACM, 2005.

[2] M. Abd-El-Malek, G. R. Goodson, G. R. Ganger, M. K. Reiter, and J. J. Wylie. Lazy verification in fault-tolerant distributed storage systems. *Symposium on Reliable Distributed Systems* (Orlando, FL, 26–28 October 2005). IEEE, 2005.

[3] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: federated, available, and reliable storage for an incompletely trusted environment. *Symposium on Operating Systems Design and Implementation* (Boston, MA, 09–11 December 2002), pages 1–15. USENIX Association, 2002.

[4] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. *ACM Symposium on Operating System Principles* (Stevenson, WA, 14–17 October 2007), pages 159–174. ACM, 2007.

[5] D. C. Anderson, J. S. Chase, and A. M. Vahdat. Interposed request routing for scalable network storage. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 22–25 October 2000), 2000.

[6] E. Anderson, J. Hall, J. Hartline, M. Hobbs, A. R. Karlin, J. Saia, R. Swaminathan, and J. Wilkes. An experimental study of data migration algorithms. *International Workshop on Algorithm Engineering* (Arhus, Denmark, 28–31 August 2001). Published as *Lecture Notes in Computer Science*, **2141**:145–158. Springer-Verlag, 2001.

[7] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO, 3–6 December 1995). Published as *Operating Systems Review*, **29**(5):109–126, 1995.

[8] S. Baker and J. H. Hartman. *The Mirage NFS router*. Technical Report TR02–04. Department of Computer Science, The University of Arizona, November 2002.

[9] B. Callaghan, B. Pawlowski, and P. Staubach. *RFC 1813 - NFS version 3 protocol specification*. RFC–1813. Network Working Group, June 1995.

[10] A. L. Chervenak, V. Vellanki, and Z. Kurmas. Protecting file systems: a survey of backup techniques. *Joint NASA and IEEE Mass Storage Conference* (March 1998), 1998.

[11] S. Dayal. *Characterizing HEC Storage Systems at Rest*. TR CMU-PDL-08-109. July 2008.

[12] J. R. Douceur and J. Howell. Byzantine fault isolation in the farsite distributed file system. *USENIX Annual Technical Conference* (Santa Barbara, CA, 27–28 February 2006), 2006.

[13] J. R. Douceur and J. Howell. Distributed directory service in the Farsite file system. *Symposium on Operating Systems Design and Implementation* (November). USENIX Association, 2006.

[14] M. Eisler, P. Corbett, M. Kazar, D. S. Nydick, and J. C. Wagner. Data ONTAP GX: a scalable storage cluster. *Conference on File and Storage Technologies* (San Jose, CA, 13–16 February 2007), pages 139–152, 2007.

[15] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of email and research workloads. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–2 April 2003), pages 203–217. USENIX Association, 2003.

[16] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of email and research workloads. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–2 April 2003), pages 203–216. USENIX Association, 2003.

[17] D. Ellard, J. Ledlie, and M. Seltzer. *The utility of file names*. Technical report TR-05-03. Harvard University, March 2003.

[18] D. Ellard and M. Seltzer. New NFS tracing tools and techniques for system analysis. *Systems Administration Conference* (San Diego, CA), pages 73–85. Usenix Association, 26–31 October 2003.

[19] A. U. Erdincler and P. A. Vesilind. Energy Recovery From Mixed Waste Paper. *Waste Management & Research*, **11**(6):507 - 513.

[20] V. Fuller, T. Li, J. J. Yun)Yu, and K. Varadhan. *Classless Inter-Domain Routing (CIDR): an address assignment and aggregation strategy*, RFC–1519. IETF, September 1993.

[21] FUSE: File System in Userspace, Apr 2010. http://fuse.sourceforge.net/.

[22] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit grouping: exploiting disk bandwidth for small files. *USENIX Annual Technical Conference* (Anaheim, CA), pages 1–17, January 1997.

[23] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *ACM Symposium on Operating System Principles* (Lake George, NY, 10–22 October 2003), pages 29–43. ACM, 2003.

[24] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. *Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 3–7 October 1998). Published as *SIGPLAN Notices*, **33**(11):92–103, November 1998.

[25] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Seattle, WA, 15–18 June 1997). Published as *Performance Evaluation Review*, **25**(1):272–284. ACM, June 1997.

[26] J. N. Gray. Notes on data base operating systems. In , volume 60, pages 393–481. Springer-Verlag, Berlin, 1978.

[27] J. Hendricks, R. R. Sambasivan, and S. Sinnamohideen. *Improving small file performance in object-based storage*. Technical report CMU-PDL-06-104. Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, May 2006.

[28] J. Hendricks, S. Sinnamohideen, G. R. Ganger, and M. K. Reiter. Zzyzx: Scalable Fault Tolerance through Byzantine Locking. *International Conference on Dependable Systems and Networks* (Chicago, IL, 29–31 June 2010). IEEE, 2010.

[29] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. *Winter USENIX Technical Conference* (San Francisco, CA, 17–21 January 1994), pages 235–246. USENIX Association, 1994.

[30] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, **6**(1):51–81. ACM, February 1988.

[31] W. Katsurashima, S. Yamakawa, T. Torii, J. Ishikawa, Y. Kikuchi, K. Yamaguti, K. Fujii, and T. Nakashima. NAS switch: a novel CIFS server virtualization. *IEEE Symposium on Mass Storage Systems* (San Diego, CA), pages 82–86. IEEE, 7–10 April 2003.

[32] S. R. Kleiman. Vnodes: an architecture for multiple file system types in Sun Unix. *Summer USENIX Technical Conference* (Atlanta, GA), pages 238–247. USENIX, 1986.

[33] A. J. Klosterman and G. R. Ganger. *Cuckoo: layered clustering for NFS*. Technical Report CMU–CS–02–183. Carnegie Mellon University, October 2002.

[34] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, **16**(2):133–169. ACM Press, May 1998.

[35] P. J. Leach. *A Common Internet File System (CIFS/1.0) Protocol (Working Draft)*. Technical report. Internet Engineering Task Force, December 1997.

[36] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. *Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 1–5 October 1996). Published as *SIGPLAN Notices*, **31**(9):84–92, 1996.

[37] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. *USENIX Annual Technical Conference* (San Diego, CA, 14–19 June 2008). USENIX Association, 2008.

[38] Lustre, Apr 2006. http://www.lustre.org/.

[39] M. K. McKusick. Running 'fsck' in the background. *BSDCon Conference* (San Francisco, CA, 11–14 February 2002), 2002.

[40] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: a read/write peer-to-peer file system. *Symposium on Operating Systems Design and Implementation* (Boston, MA, 09–11 December 2002). USENIX Association, 2002.

[41] When to Use Transactional NTFS, Apr 2006. http://msdn.microsoft.com/library/en−us/fileio/fs/when_to_use_transactional_ntfs.asp.

[42] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. *Summer USENIX Technical Conference* (Monterey, CA, 06–11 June 1999). USENIX Association, 1999.

[43] S. V. Patil, G. A. Gibson, S. Lang, and M. Polte. GIGA+: Scalable Directories for Shared File Systems. *ACM Symposium on Principles of Distributed Computing* (Reno, NV, 11–11 November 2007), pages 26–29. ACM, 2007.

[44] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. SnapMirror: file system based asynchronous mirroring for disaster recovery. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 117–129. USENIX Association, 2002.

[45] Reiser4 Transaction Design Document, Apr 2006. http://www.namesys.com/txn-doc.html/.

[46] Y. Saito and C. Karamanolis. *Name space consistency in the Pangaea wide-area file system*. HP Laboratories SSP Technical Report HPL–SSP–2002–12. HP Labs, December 2002.

[47] F. Schmuck and R. Haskin. GPFS: a shared-disk file system for large computing clusters. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 231–244. USENIX Association, 2002.

[48] S. Sinnamohideen, R. R. Sambasivan, J. Hendricks, L. Liu, and G. R. Ganger. *A Transparently-Scalable Metadata Service for the Ursa Minor Storage System*. Technical report CMU-PDL-10-102. Parallel Data Laboratory, Carnegie Mellon University, March 2010.

[49] SPEC SFS97 R1 V3.0 Documentation, Jan 2010. http://www.spec.org/sfs97r1/.

[50] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO, 3–6 December 1995). Published as *Operating Systems Review*, **29**(5), 1995.

[51] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: a scalable distributed file system. *ACM Symposium on Operating System Principles* (Saint-Malo, France, 5–8 October 1997). Published as *Operating Systems Review*, **31**(5):224–237. ACM, 1997.

[52] S. A. Weil, S. A. Brandt, E. L. Miller, and D. D. Long. Ceph: A scalable, high-performance distributed file system. *Symposium on Operating Systems Design and Implementation* (December). USENIX Association, 2006.

[53] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the Panasas file system. *Conference on File and Storage Technologies* (San Jose, CA, 26–29 February 2008), pages 17–34. USENIX Association, 2008.

[54] K. G. Yocum, D. C. Anderson, J. S. Chase, and A. M. Vahdat. Anypoint: extensible transport switching on the edge. *USENIX Symposium on Internet Technologies and Systems* (Seattle, WA, 26–28 March 2003). USENIX Association, 2003.