MORRIS: A Distributed File System for Read-Intensive Applications

David Chau, Jennifer Lin, Michael Matczynski, Nathan Palmer
{ddcc, jwlin, mikem, palmer}@mit.edu

May 12, 2005

Abstract

This paper presents the design and implementation of Modularly Optimized Round-robin Read-Intensive Storage (MORRIS), a file system which provides high throughput for read-intensive applications. `NFSStripe`, MORRIS' primary component, is an NFS loopback server that achieves performance competitive with the traditional single-server model by distributing the task of data storage and retrieval over multiple machines.

There are two main challenges associated with such a design. The first consists of structuring the underlying storage of the filesystem in such a way as to take advantage of multiple data servers, thereby allowing multiple concurrent read operations to be efficiently executed. We solve this problem by "striping" files across multiple `StripeServer` data servers in fixed-size blocks. Employing multiple machines to serve data from disk allows our system to fulfill multiple client requests at once, whereas a single-server arrangement cannot.

The second challenge is to ensure filesystem coherence as multiple concurrent client operations are issued to multiple independent servers. We solve this problem, while conferring minimal impact on the system's performance, by designing a multiple-reader/single-writer locking protocol specifically suited to our system's data structures.

1 Introduction

There are a wide variety of data mining problems where applications make frequent file reads (but infrequent writes), spending as much as $98-99\%$ of their time performing read-related operations and as little as 0.05% writing to disk [11]. Many of these data mining tasks, particularly those in the field of bioinformatics, lend themselves to a high degree of parallelism, where multiple processes working individually (on independent machines) perform reads from the same underlying database and merge their results quickly when each node completes its portion of the work [11, 5, 12].

Many of these data mining applications operate by sequentially reading large flat-file databases, processing each record in turn [1, 5, 12]. The large size of these databases, often on the order of several gigabytes [3, 2], makes aggressive client-side caching an impractical solution to improving performance. The files are often simply too large to store wholesale in volatile memory on each client, and even if the memory were available, a node rarely re-reads the same portion of a file, meaning that the data would seldom be revisited after the initial read which caused the data to be cached. Moreover, this solution fails to address the issue of contention between clients attempting the initial caching read.

Because of the high degree of parallelism exploited by these applications, providing concurrent read access to the shared data becomes an important factor in application performance. Multiple nodes requesting the same file from one networked storage device can lead to clients sit-

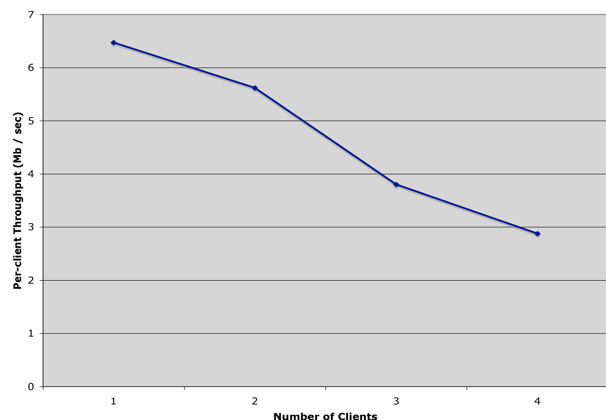

Figure 1: **Performance of a single FreeBSD NFS server serving multiple clients:** We varied the number of clients concurrently reading a 200 MB file and observed a nearly linear decrease in per-client throughput.

ting idle, wasting valuable computing time, while waiting for their data to arrive. Considering the long-running nature of these applications and the fact that they spend nearly all of their time reading, we suggest that a file server capable of improving read access times for these applications could provide a significant boost in application performance.

This paper presents an NFS file server that provides high throughput for these read-intensive applications. Our goal is to alleviate the bottleneck that occurs when a single NFS server is used as the storage medium in such environments. Figure 1 illustrates the performance degradation experienced by multiple clients reading the same file from a single NFS server.

Our system consists of two separate components: `NFSStripe`, which is an NFS loopback server, and `StripeServer`, a data block server that `NFSStripe` communicates with. `NFSStripe` is intended to be run on each client requiring access to a common filesystem. It reads and writes data in fixed-size blocks by communicating with `StripeServer` data storage servers which maintain the blocks on stable media.

Observations about the workload characteristics of the applications that we expect our system to service, in particular, scientific data mining, have allowed us to fine-tune the interaction between these two components to enable high read throughput while simultaneously ensuring the coherence of the filesystem. That is, because the clients are expected to make frequent concurrent read operations (but infrequent writes), we allow multiple `NFSStripe` servers to simultaneously read a file, but require that operations which modify the filesystem be given exclusive access to those structures being modified. The protocol that enforces this, combined with a simple "striping" scheme where the blocks constituting a file are laid out over multiple `StripeServer` machines, enables multiple clients to read the same file (albeit different blocks) simultaneously without suffering from delays caused by lock contention or overloaded data servers.

We would like our system to provide semantics to the client that mimic as closely as possible those of a single local disk, since that is what most users and programmers have come to expect. Put another way, users and applications should notice no difference between interacting with a filesystem residing on a local disk versus one mounted via an `NFSStripe` server. In this regard, the choice of NFS as an interface to the client is a reasonable one since that was one of the original goals laid out by Sandberg *et al.* [9]. We note also that, at a more technical level, because its interface complies to the NFS v3 [7] standard, existing programs may access an `NFSStripe` filesystem without modification.

2 Related Work

In an effort to provide better performance scalability than the single-server setup achieves, we take an approach similar to that of Frangipani [10] by layering `NFSStripe` on top of multiple `StripeServer` data servers. Similar to that of the Frangipani's sister service, Petal, `StripeServer` behaves like a "network disk" in that data may be read or written in blocks. In contrast with Frangipani, however, `NFSStripe` server is a full-featured NFS v3 [7] loopback server [6], meaning that the large number of systems already including an NFS client do not need kernel modification or special device drivers

to use our system. Additionally, since both `NFSStripe` and `StripeServer` run as user processes and do not require special administrative privileges to run[1], we believe that our system provides greater administrative flexibility than does Frangipani.

FAB [8] is a distributed disk array that aims to achieve reliability through a voting-based algorithm that distributes data among a federation of bricks. `NFSStripe`'s goals differ from those of FAB in that we focus on high read throughput, whereas FAB focuses on reliability. For applications that do not need as much reliability, e.g., they backup their data, our system has the potential to better utilize disk resources and achieve similar levels of throughput.

Zebra [4] is a network filesystem that stripes its log and batches many small files together to achieve high performance; it is resilient against single-server failures because it uses a parity stripe. Zebra is optimized for sequential small file accesses however, it is not necessarily suitable for running database applications that randomly read large files. Our system is capable of handling sequential and random reads with high throughput.

3 Design

Recall that our filesystem consists of two components: `NFSStripe` and `StripeServer`. `NFSStripe` is an NFS loopback server that communicates with multiple machines running instances of `StripeServer`. We intend `NFSStripe` to be run on each client requiring access to the shared filesystem. Figure 2 illustrates the interaction between the different parts of the system. `StripeServer` stores and retrieves 8k blocks of data, and each file's content blocks are laid out and retrieved by `NFSStripe` in "stripes" across multiple `StripeServer` machines.

As mentioned earlier, there are two main challenges associated with designing a filesystem

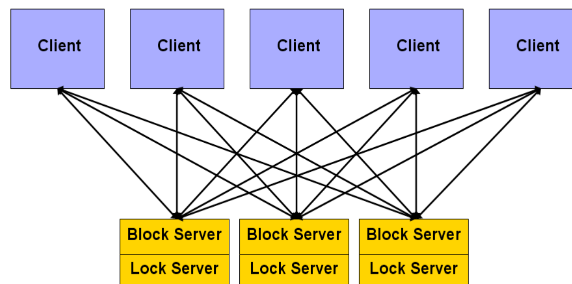

Figure 2: **Interaction between `NFSStripe` `StripeServer`:** Each `NFSStripe` client communicates with all of the `StripeServer` data block servers.

where multiple clients need fast concurrent read access. One is to guarantee filesystem coherence, even in the event of network partitions or server failures. The second challenge is to devise a scheme for efficiently distributing data among the multiple `StripeServer` instances. We address the former in this section along with a discussion on persistence across system crashes, but defer discussion of the second challenge until Section 4.

3.1 Consistency and Crash Recovery

The main goal of our filesystem is performance. Users of our system will tolerate downtime should one of the `StripeServer` instances become unreachable, and the whole filesystem will be unusable until the that server again becomes available. If an operation was in progress when a server failed, then that operation might be lost. In other words, our filesystem does not try to offer better availability than a single local disk.

Although we do not aim to offer enhanced availability, maintaining data *consistency* is important to us. A server crash should not render the filesystem forever unusable. Once the server restarts, and without performing any additional recovery operation, the directory structure should be in some usable state.

Our initial goal was to guarantee that the work done in handling every NFS call appear atomic. While achieving atomicity is fairly simple when both servers and clients never fail, it turns out to be very difficult to make operations atomic

[1]There is one daemon that requires root permissions to start, but it need only be started once to allow multiple `NFSStripe` instances to inform the operating system of their presence, and typically needs no administrative attention once it is running.

across servers if servers fail at arbitrary times[2]. Note that we do not address the issue of client failure.

Therefore, rather than trying to build extremely complex protocols to ensure atomicity, we opted for weaker guarantees. We want the directory structure to be *usable* whenever the filesystem is up: that is, if a client can access the directory structure, the directory structure that it sees will make sense. For example, a client should never see a directory entry that points off to nowhere. However, there may be garbage in the system, e.g., a file that still takes up space, even though it cannot be accessed by traversing the directory tree. We achieve this constraint by careful *ordering* of operations. (This is the same technique that UFS uses).

The general idea is that when adding data to the filesystem, we ensure that the data is stable before we update the metadata (i-node) of the structure being modified. For example, when a directory entry is being added, we update the directory entry only after ensuring that the file to which it will point is in stable storage. We do the reverse when removing data. This ordering ensures that a server crash in the middle of an operation will not result in corrupt metadata, and that the filesystem's structure will remain intact.

Moving a file across directories is ostensibly the most interesting and challenging operation to implement correctly in the presence of possible failures, so we pay it special attention here. The best we can do with operation ordering is to guarantee that a file never gets lost. However, if the server crashes, we might end up with two copies of the file. Since we implemented a move by first adding the file to the new directory, and then deleting it from the old, we will have two copies of the file in the directory structure if the crash happens between these two steps.

[2]In fact, we suspect that enforcing atomic updates to multiple servers is equivalent to the two-generals problem. In this case, guaranteeing atomicity is impossible if we want the client to be able to decide whether a call has succeeded or failed in bounded time.

3.2 Locking and Filesystem Coherence

We use locking to guarantee the coherence of our filesystem by making multi-step operations appear atomic.

Without locking, any operation that modifies the filesystem and that cannot be performed by a single call to a `StripeServer` threatens to leave the filesystem in an inconsistent state. For example, when creating a file, we need to retrieve the block containing the directory's contents, add the entry for the new file, and then put the block back onto the `StripeServer`. If a second client also updates the directory's contents between the first client's fetch and replacement of the block, the first client's modifications will overwrite the second client's changes. We therefore use locking to ensure strict consistency so that when a client modifies a directory, that client is granted sole access to the directory's contents and metadata.

We chose the i-node of a file or directory as the object that we lock. A client with an exclusive lock on the i-node of a file can be sure that no other client will modify any part of that file until the lock is released.

To enhance performance, our filesystem differentiates between read and write locks. Multiple clients may have read locks out on the same i-node, but only one client may have a write lock. (The clients need to acquire read locks before reading to prevent them from seeing parts of an operation in progress.) The lock server, implemented as part of `StripeServer`, will not grant a write lock until all other read and write locks on the i-node are released. Once the lock server grants a write lock, it will not grant *any* other locks on the i-node until the write lock is released.

The lock service, like the block service, is distributed. The server that is responsible for serving a given i-node (discussed in Section 4) is also responsible for managing the locks associated with that i-node. In other words, every `StripeServer` has integrated into it a lock server. This design eliminates the bottleneck of having just a single lock server, which also helps to improve performance.

For performance reasons, the client does not store its locks in permanent storage. It is therefore possible that a client may fail after acquiring, but before releasing, a lock. In this case, the lock may never be released. We decided not to handle this mode of failure in our filesystem, although one possible solution is to allow users to manually release locks using a separate utility when users discover that the client holding a lock has crashed.

3.2.1 When to Perform Locking

Not every NFS call requires locking. Because we implement UFS semantics by writing the i-node last, a call that only reads metadata, such as GETATTR, does not require locking.

The operations that require locking are the ones that examine an i-node and then decide which data blocks to read or write. For example, a READ call needs to lock, because if it did not, then after it read the i-node to determine the file's size, another client might shorten the file before the READ call has a chance to actually fetch the data blocks. We also need locking for any structural modifications that are made to the directory tree in order to prevent another client from simultaneously modifying the same structure.

As an example, the locking for the RENAME call presents a special challenge: we need to lock both the source and the destination directories, but there is the risk that another client may try to move a file in the opposite direction at the same time, resulting in a deadlock. To solve this problem, we decided to lock the i-nodes of the two directories in lexicographic order.

4 Implementation

4.1 Distributing Data Among Multiple Servers

Although an NFS file handle logically identifies a file in our filesystem, we need a method to locate content blocks and metadata for each file among the multiple `StripeServer` instances in the system. To represent this mapping, a 96-bit identifier (ID) is used internally to name each block. This identifier consists of a 64-bit NFS file handle onto which we concatenate a 32-bit integer "offset." By convention, offset 0 is reserved for the block containing the file's metadata. The content blocks for a file are sequentially segmented into 8k units and stored with ID offsets ranging from 1 to $2^{32} - 1$. Since each block is 8k, each file can theoretically contain up to 35 terabytes of data.

To determine which `StripeServer` a particular block resides on, we add the ID's file handle to its offset. We then take this 64-bit sum modulo the number of servers k to get a server identifier in the range of 0 to $k-1$. We assign an arbitrary but fixed ordering of the `StripeServer` instances with which `NFSStripe` communicates, labeling each uniquely with one of $i \in \{0 \leq k-1\}$. In this way, the server identifier computed above maps to a `StripeServer` responsible for the block in question. Assuming an unbiased distribution of file sizes, this scheme results in uniform expected load balancing, as each new file created in the filesystem will have its i-node stored on the next server in the sequence. The 8k data blocks are distributed in a round-robin fashion beginning with the next block server after the one holding the i-node. In practice, we have found this scheme to result in a reasonably uniform distribution of 8k blocks among the `StripeServer` machines.

4.2 Reading a File

When `NFSStripe` receives a request requiring it to fetch a block of data, it first calculates which block server holds the file's i-node. It then asks the server holding the i-node for a read lock on the file. This read lock on the i-node is valid for any part of the file's contents. The `NFSStripe` server then calculates which `StripeServer` instances hold the blocks containing the data for the portion of the file being read. This calculation may result in more than one `StripeServer`, as a client may read data past an 8k-block boundary. Finally, each individual `StripeServer` is asked to send the relevant data blocks to `NFSStripe`. The blocks

are then merged and the data is sent back to the client to satisfy its original request. If `StripeServer` finds that some client has currently locked the specified file for writing, it delays granting the read lock and performing the subsequent fetch until the write operation finishes and a read lock can be acquired.

By striping the data across multiple block servers, the individual load on any particular block server will remain low, thus allowing the aggregate throughput to scale as more `StripeServer` instances are added. That is, since a client attempting to read sequentially through a large file will cause `NFSStripe` to retrieve blocks from each `StripeServer` in round-robin fashion, each `StripeServer` will become immediately available to serve other `NFSStripe` requests, presumably from other clients.

One weakness of this design centers around the need for each READ request to retrieve a file's i-node, which, for any given file is stored on a single `StripeServer`. Although our multi-reader locking scheme mitigates the performance hit on an i-node's `StripeServer`, this issue may cause scalability problems and is an area for future research. One approach to alleviating this bottleneck may involve caching metadata on each `NFSStripe` server. Another solution may involve replicating each metadata block onto every `StripeServer`, allowing `NFSStripe` to acquire metadata from the same `StripeServer` it intends to read content from. Both of these schemes, however, would require significant changes to our locking protocol.

4.3 Writing to a File

When `NFSStripe` receives a request requiring it to modify a data structure in the filesystem, such as writing to a file, it first determines which `StripeServer` holds the file's i-node. `NFSStripe` then sends that server a request for a write lock on the file. The lock server running as part of that `StripeServer` instance then waits until all read locks on that file have been released. Once that happens, a write lock is issued to the `NFSStripe` server that requested it. This lock allows exclusive access to all data in the file.

The procedure for writing follows from that used when reading: a block is fetched from the appropriate `StripeServer`, modified, then written back to the same server (if no block previously existed for the extent of the file being written, one is created, rather than fetched). Afterwards, `NFSStripe` contacts the i-node's lock server to release the write lock.

4.4 `StripeServer` Operations

Each of the following calls execute atomically on `StripeServer`, in the order they are received.

`acquire_read(key)`: Lock the block named by `key` for reading. The server will allow other `acquire_read()`s on the same key, but will block `acquire_write()`s.

`acquire_write(key)`: Lock the block named by `key` for writing. The server will block other `acquire_write()`s and `acquire_read()`s on the same key.

`get(key)`: Return the block named by `key`.

`put(key, data)`: Store the block `data` named by `key`.

`delete(key)`: Delete the block named by `key`.

`unlock(key)`: Unlock the block named by `key`.

4.5 NFS Calls

We implemented all of the relevant NFS RPCs, including `RMDIR` and `RENAME`. For `ACCESS`, we chose not to implement permissions and allowed execution of all operations. To give the reader a sense of how `NFSStripe` actually implements these operations via the `StripeServer` interface, we give a detailed explanation of the `CREATE` routine.

In `CREATE`, we acquire a write lock on the i-node of the directory in which the file will be

```
CREATE RPC
create(dir file handle, file name):
  acquire_write(dir file handle)
  get(dir inode)
  get(dir listing)
  if (dir does not already contain filename):
    put(inode for new file)
    put(updated dir list with new file entry)
  put(dir inode) with updated mtime, ctime
  unlock(dir file handle)
```

Figure 3: **Pseudocode of CREATE RPC:** Create a file, where `inode(fh)` represents the key for the inode for the given file handle and `data(fh)` represents the key(s) for the data corresponding to the file handle.

created, as shown in the pseudocode in Figure 3. We retrieve the block containing the directory's contents, add the entry for the new file if it does not already exist, and then put the i-node for the new file followed by the updated block of directory's content on the server. We then update the i-node of the directory and release the write lock.

Locking is necessary to prevent, for example, `LOOKUP` from accessing the new i-node for the new file until the `CREATE` operation is done, thus ensuring atomicity. If the server crashes before `CREATE` finishes, then we will have the i-node for the new file stored somewhere, but the new file will not be accessible from the directory tree, which is acceptable.

Note that for efficiency, we do not update the `atime` on files when we access them. Only the `mtime` and `ctime` are updated upon modification.

5 Performance

We evaluated the performance of our system on the Emulab [13] testbed, using two different configurations of eight 850 Mhz Pentium III machines running FreeBSD 4.9, each containing 512 MB of RAM. All of the tests presented here consisted of reading a single 200 MB file.

5.1 The Marginal Penalty for an Additional Reader

In the first experiment, we set up four of the machines running `StripeServer` instances, while the remaining four acted as clients running `NFSStripe` instances. Our goal in this experiment was to understand the marginal performance penalty, if any, that our system incurred when an additional client attempts a concurrent read of a file striped across multiple `StripeServer` instances. To study this, we measured the per-client throughput when one, two, three or four clients (all running their own `NFSStripe` instances) concurrently read the same 200 MB file striped across the four `StripeServer` instances.

In order to compare our system to a standard single-server NFS setup, we exported (via FreeBSD 4.9's native NFS server) a directory stored on local disk on one of the machines that served as a `StripeServer` in the above experiment. We placed the same 200 MB file in that directory, and took the same measurements as above, measuring per-client throughput when one, two, three, and four clients attempted to read the file at the same time.

Figure 4 illustrates the results of these two experiments. With only one `NFSStripe` client reading, our system provides an average read rate of 2.58 MB/sec, roughly 40% of FreeBSD's 6.50 MB/sec average. We highlight, however, the fact that when all four clients are reading, FreeBSD's per-client throughput drops substantially to 36% of its one-client throughput. In contrast, with four readers, `NFSStripe`'s per-client throughput falls only slightly to 81% of its single-reader throughput.

Preliminary analysis of the activity on the `StripeServer` machines seems to indicate that the drop in performance that `NFSStripe` sees when multiple clients read the same file can be attributed to the metadata for a file being stored on only one `StripeServer`. Each NFS `READ` operation requires that the file's i-node be fetched and thus, the server holding the i-node for our 200 MB test file received an inordinate amount of traffic in relation to the other block servers. As

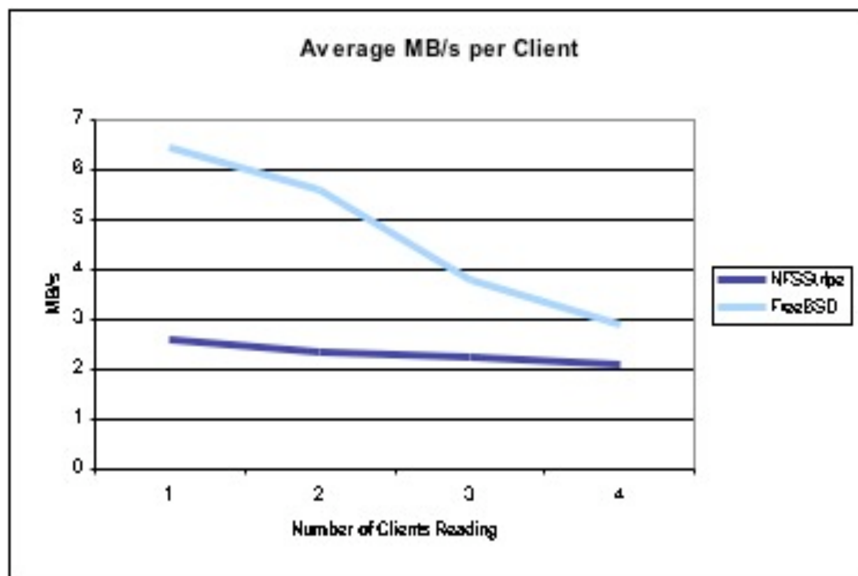

Figure 4: **Performance of `NFSStripe` communicating with four `StripeServer` instances compared to FreeBSD's NFS server serving multiple clients:** We varied the number of clients concurrently reading a 200 MB file and observed only a 19% decrease in per-client throughput for `NFSStripe` between the 1-client and 4-client scenarios, and an increase in total throughput by a factor of 3.25. FreeBSD's NFS server suffers roughly a 64% decrease in per-client throughput, while aggregate throughput remains basically constant.

mentioned in Section 4.2, there are several possible solutions that we intend to explore, including caching and replication of metadata. Real-world workloads, however, are likely to consist of more than one file, and should not present such a drastic problem.

Another possible cause for `NFSStripe`'s performance degradation on READ operations may be that the interaction between `NFSStripe` and `StripeServer` required to fulfill a `READ` request results in significantly more network messages than the standard NFS setup. `NFSStripe` must first wait for a read lock to be granted (two messages), then send a request for the file's metadata (another two messages), and finally request a particular data block and wait for its arrival (another two messages). At a total of six messages, `NFSStripe` and `StripeServer` are three times as chatty as the interaction between a standard NFS server and client. It is possible that network latency is the bottleneck causing our system's slower performance. While sending thrice as many messages, our system is slightly more than twice as slow (in the single-client scenario) as the FreeBSD NFS server. In future versions, we would like to explore methods for integrating lock requests with their respective block requests in order to reduce the number of messages passed between `NFSStripe` and `StripeServer`.

Since average per-client throughput dropped to only 81% when all four clients were reading, the total throughput of the system increased by a factor of roughly 3.25 to 8.37 MB/sec. Increasing the number of clients reading from our system *increased* its total throughput, whereas FreeBSD's total throughput remains constant (with the exception of the single client scenario) at just over 11 MB/sec.

5.2 Determining an Optimal MORRIS Configuration

In a second experiment, we wanted to study how our system performs relative to a standard NFS server when both systems are pushed to their limit with READ requests. Figure 5 sumarizes the results of this experiment. We first set up one machine running `StripeServer` and six machines acting as clients, each running its own `NFSStripe` instance. After placing our 200 MB test file in the filesystem managed by `NFSStripe`,

we measured average total system throughput (the average sum of the throughput measurements taken for each `NFSStripe` server). We then repeated this test, but increased the number of `StripeServer` instances servicing `NFSStripe` to two. For comparison, we measured the total throughput of six clients reading the same file from FreeBSD's NFS server, and observed, predictably, that it was just above 11 MB/sec.

We drew two conclusions from this experiment. The first, given the sharp upward trend in performance when we introduced the second `StripeServer`, is that when there are multiple clients performing reads, employing several data servers is an effective strategy to scale total throughput, assuming the number of clients is greater than the number of servers. Though we lacked the resources to present evidence in this paper, we expect that there is nothing to be gained by using more `StripeServer` instances than readers. Intuitively, we would expect that this will result in some servers sitting idle while the clients are busy requesting blocks from others.

The second, and perhaps more interesting, conclusion comes from the observation that the arrangement of four `StripeServer` machines serving four `NFSStripe` machines (explained in section 5.1) provided lower total throughput than did the two `StripeServer`, six `NFSStripe` arrangement employed here. This implies that one `StripeServer` is capable of serving data quickly enough to keep multiple `NFSStripe` servers satisfied at the same time. Again, we suggest that this performance characteristic is a result of network latency; the `StripeServer` machines are sitting basically idle while waiting for messages to arrive at and from the `NFSStripe` servers. Put another way, this experiment suggests that, for a fixed set of resources (machines), obtaining optimal throughput using the MORRIS system requires using significantly more client `NFSStripe` machines than `StripeServer` machines.

We assume that after progressively increasing the ratio of readers to `StripeServer` machines, total throughput performance will plateau once demand on the `StripeServer` machines meets or exceeds their ability to respond. Again, we lacked the resources to determine this threshold, since we were limited to eight machines when testing.

6 Conclusion

MORRIS achieves high throughput and filesystem coherence for read-intensive applications. It efficiently distributes data across multiple `StripeServer` instances to attain load-balancing. Our multiple-reader/single-writer locking protocol allows this design to scale better than a single NFS system. With further research, this design has the potential to outperform existing filesystems for read-intensive applications.

6.1 Other Applications

In addition to providing a high throughput server for data-mining applications, MORRIS can be set up as a decentralized workgroup data store that allows workstations to combine spare disk space into a unified shared filesystem. For example, if 10 users contribute 5 GB each, the workstations can share a single 50 GB filesystem. Aside from saving the disk space and hassle associated with multiple clients maintaining independent copies of the same file, this arrangement also provides natural load balancing, as all workstations contribute to serving files. Thus, the effect of one slow or busy workstation will be minimized.

References

[1] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI–BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, 25:3389–3402, 1997.

[2] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and D. L. Wheeler. Genbank: update. *Nucleic Acids Research*, 23, Database issue:D23–D26, 2004.

[3] H.M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T.N. Bhat, H. Weissig, I.N.

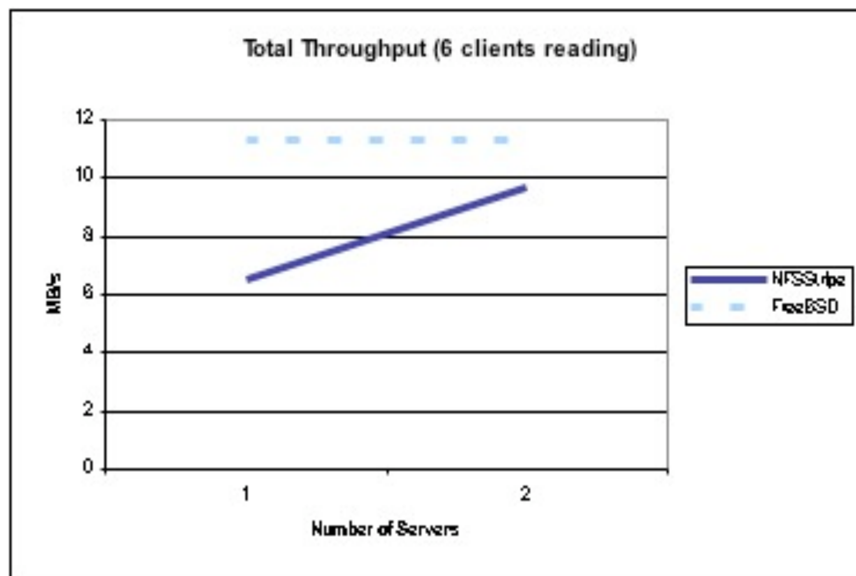

Figure 5: **Performance of `NFSStripe` communicating with one and two `StripeServer` instances:** We varied the number of `StripeServer` instances serving `NFSStripe` while clients concurrently read a 200 MB file. We see a significant increase in total system throughput with the introduction of only one additional `StripeServer`. With two `StripeServer` instances, our system comes close to the 11.28 MB/sec served by FreeBSD's NFS server.

Shindyalov, and P.E. Bourne. The protein data bank. *Nucleic Acids Res.*, 28:235–242, 2000.

[4] John H. Hartman and John K. Ousterhout. The zebra striped network file system. *ACM Trans. Comput. Syst.*, 13(3):274–310, 1995.

[5] K. Li. Clustalw-mpi: Clustalw analysis using distributed and parallel computing. *Bioinformatics*, 19, no.12:1585–1586, 2003.

[6] D. Mazieres.

[7] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. Nfs version 3 design and implementation. In *Proceedings of the Summer USENIX Technical Conference*, pages 137 – 152, 1994.

[8] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. Fab:building distributed enterprise disk arrays from commodity components. *ACM SIGPLAN Notices*, 39, issue 11:48 – 58, 2004.

[9] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.*, pages 119–130, Portland OR (USA), 1985.

[10] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Symposium on Operating Systems Principles*, pages 224–237, 1997.

[11] P. Vaidyanathan and T. Madhyastha. Dynamic replication to improve input/output scalability of genomic alignment. In *Proceedings of the International Workshop on Challenges of Large Applications in Distributed Environments*, 2003.

[12] A. Waugh, G. A. Williams, L. Wei, and R. B. Altman. Using metacomputing tools to facilitate large scale analyses of biological databases. In *Proceedings of the Pacific Symposium on Biocomputing 2001*, 2001.

[13] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 255–270, 2002.