The WiND Filesystem (WFS)

Muthian Sivathanu
Computer Sciences Department
University of Wisconsin Madison
muthian@cs.wisc.edu

15th January 2002

Abstract

We present in this paper the design and implementation of a kernel level infrastructure aimed at facilitating rapid design, development and experimentation with network attached storage systems. The infrastructre is built in the linux kernel in the form of a distributed striped filesystem called the WiND filesystem (WFS). Based on the NASD paradigm, WFS allows direct communication between the clients and the disks without having to go through a centralized file server for all operations. The filesystem also has mechanisms for implementing efficient striping across heterogeneous disks with varying performance. Various tradeoffs in design and implementation are considered, followed by a brief evaluation of the initial implementation.

1 Introduction

The demand for high performance, highly scalable and available storage is on the rise due to ever so many factors such as the increasing mismatch between processor and disk speeds, proliferation of data intensive applications like large-scale data mining and decision support systems and rapidly increasing use of multimedia. This has led to research on new paradigms and architectures for designing storage systems, the most influential of them in the recent past being the network attached storage paradigm, which started off from the NASD project at CMU [1]. The basic idea behind NASD is to overcome the bus bandwidth bottleneck by replacing the I/O bus with a high-speed switched network. Numerous storage devices are placed on the network and direct communication is

allowed between the clients and the disks over the network. Such an architecture obviously requires that disks are capable of communicating independently over the network and executing reasonably complex protocols, but the increasingly powerful processors in modern disks makes this a realistic option. This paper presents WFS, a striped distributed filesystem that makes use of the network attached storage paradigm to provide highly scalable, high throughput I/O to client applications. The principal objective of WFS is to serve as a platform for experimenting with different strategies and solutions for building highly scalable, available and easily manageable storage systems, as part of the WiND project [2]. For this purpose, the focus has been on building it as simple and extensible as possible.

The rest of the paper is organized as follows. We discuss background material in section 2. In section 3, we consider the tradeoffs in implementing a network attached storage architecture at a device layer versus at a filesystem layer. Section 4 presents the general design and architecture of WFS. Section 5 talks about various implementation details and tradeoffs. Performance measurements are presented in Section 6 and related work is discussed in Section 7. Section 8 considers future work, and we conclude in Section 9.

2 Background

Traditional filesystems like NFS and AFS suffer from a major scalability problem - all client requests and data pass through a single fileserver or a set of few replicated file servers. The I/O bus and CPU at the file server very easily become a bottleneck for even moderately high access rates. Thus, even if the file server runs on top of a high throughput RAID system, the bandwidth realizable out of the file server is limited to a much lesser value due to these resource bottlenecks. The network attached storage paradigm solves this problem by allowing direct communication between the client and the disks, without having to pass through a central file server. By doing this, NASD allows clients to exploit the full bandwidth realizable out of the disks, without being constrained by any intermediate bottleneck. The classic techniques for improving efficiency and availability like striping, mirroring etc. that are done in traditional RAID systems [11] are also applicable in the context of managing a pool of disks attached to the network. A major difference, however, is the fact that the responsibility for striping and redundancy for performance/availablity is now

distributed across the multiple clients and the file manager, as opposed to being implemented at a centralized controller in traditional RAID systems.

The NASD project also proposed a new interface that could be exported by these network attached disks. Instead of exposing a simple block interface as a traditional disk does, the disks now export an object-level interface, with functions to create and delete objects, and perform read or write on an object, given the object identifier and a logical offset within the object. In basic terms, this could be viewed as the disk exporting a restricted filesystem interface, without any support for directories and hierarchy, since the object namespace is flat. The basic motivation behind such an object interface is that disks now have a capability of understanding relationships between blocks. A filesystem, by placing each of its files in a separate object, could indicate to the storage system that the blocks comprising the object constitute a single file, and hence should be placed together. The earlier argument about disks getting smarter makes this additional intelligence on the disks reasonable to consider.

3 Device-level vs. Filesystem-level Design

There are two ways one could build a software infrastructure that enables applications make use of a set of network attached disks. The first option would be to implement the infrastructure at the device layer (as a device driver), so that normal filesystems and applications can easily and transparently leverage the benefits of a network attached pool of disks. This approach, while definitely beneficial in terms of allowing any existing filesystem to be transparently mounted over network attached storage, also has its own limitations. An ordinary client filesystem like ext2 mounted on top of NASD will behave incorrectly in the face of concurrent access from other clients. The problem is that the filesystem assumes exclusive control over the disk and hence cannot handle inconsistencies due to other clients on the network sharing access to the disk, especially when these inconsistencies arise in the case of metadata blocks like inodes or freelist bitmaps. Hence, a NASD implementation at the device layer cannot permit an unmodified filesystem to use it correctly, if the disk is also being accessed by other clients over the network. Since sharing is one of the important benefits of switching to a network-storage paradigm, this limitation on concurrency of access is likely unacceptable.

The other option to implementing such a software infrastructure is be to

build it at a higher layer, as a separate distributed filesystem. The filesystem would be aware of the fact that it is running on top of network attached disks and hence numerous optimizations that would have been impossible from a device-level implementation, now become possible. Some potential benefits of having a filesystem-level infrastructure are:

Data Sharing: This is a major advantage of a filesystem level design. Since the filesystem now knows that the network attached disks are meant to be accessed by other clients as well, it can have mechanisms that permit concurrent access and data sharing, providing whatever consistency semantics it deems fit.

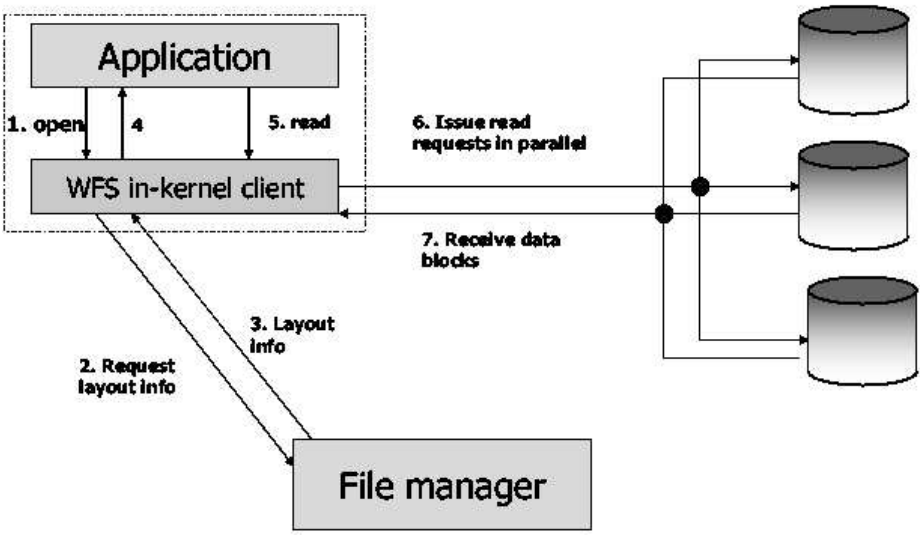
Per file redundancy and striping: Given that the responsibility of striping and replication lies with the client in the NASD paradigm, a filesystem level design could facilitate flexible per-file choices for these policies. The filesystem would be the best place to do this since it knows about access patterns and lifetimes of files. For example, the filesystem could choose not to mirror a file which it thinks would be short-lived, or it could decide between RAID-5 and RAID-1 for a file depending on the perceived/inferred access patterns of the file, similar to AutoRAID [12] . A storage- level infrastructure, on the other hand, would have to necessarily impose the same redundancy policy on all objects, since it would have no file-level information.

Flexible security: While the NASD project looked at security at a device level [7], thereby forcing a uniform security policy across all objects, a distributed filesystem can facilitate flexible per-file security, allowing one to choose the optimal tradeoff between security and performance on a per-file basis. Moreover, issues like key management are more natural to handle at a higher level as opposed to a device level. An ordinary filesystem mounted on top of a secure device-level NAS infrastructure would often end up duplicating the measures for security.

4 WFS Design

The WiND filesystem is made of three components: an in-kernel filesystem at the client, the network attached disks and a centralized file manager. The purpose of having an additional entity in the filemanager is two fold: First, it maintains metadata for each file that includes, inter alia, the layout information

Figure 1: WFS Architecture



for each file which indicates things like the redundancy level for the file and the set of disks it is striped across. This metadata information, which often spans multiple disks for a striped, replicated file, is best maintained at a centralized location common to all disks. An alternative would be to replicate this at each of the disks that contain some part of the file, but we choose the former for reasons of simplicity of design. Second, the file manager maintains access control state and thus facilitates uniform access control for all files in the filesystem. This provides for decoupling the access control policy decision-making from actual verification at the disk, thereby providing for a single, generic verification strategy at the NAS disk that can coexist under multiple access control policies. The CMU NASD security work [7] details how such decoupling of access control could be done. However, an essential requirement in this architecture is that the file manager should be used as sparingly as possible, so that it does not become a scalability bottleneck because if it does, it would frustrate the primary motivation for a network attached paradigm. Since the filemanager is contacted only for metadata operations with reads and writes of data going directly to the disk, we believe that file manager scalability is not much of a concern. However, for extremely large filesystems with high access-rates, one could have a set of replicated filemanagers with some simple load distribution strategy among them.

Figure 1 shows the general architecture of WFS by tracing the sequence of operations that take place that place as a result of a file `open()` followed by a `read()` . On an `open`, the WFS in-kernel client checks to see if the metadata information for the file is cached. If not, it sends a `lookup` request to the file manager specifying the name of the file to be looked up. The file manager returns a WFS file handle for the file, which contains, information on how the different parts of the file are to be located across the various disks. Using this information, the client now issues requests to the different disks in parallel, and once it has heard back from all the requisite disks, returns the read data back to the client. Writes happen in an analagous manner. One point to be noted here is that the metadata information for the file is cached at the client, so subsequent reads and writes and even opens need not go to the file manager. Since applications often exhibit good locality in the set of files they access, the local in-kernel metadata cache will likely have a high hit-rate, and hence the load at the file manager for these lookups is expected to be very minimal.

4.1 Support for managing heterogeneity among disks

The WiND project [2] proposes adaptive striping as a strategy for managing heterogenity among network attached disks. Traditional RAID solutions are based on the assumption that all disk perform equally well, and hence perform poorly when applied to disks with varying performance. For example, a RAID 0 system that does simple striping across N disks with varying performance will always track the performance of the slowest disk, since for each stripe, it has to wait for the slowest disk to complete. Adaptive striping looks at laying out data proportionally across the disks, in tune with the perceived rate of operation of each disk. In other words, slower disks will have less data placed upon them (proportional to rates of the disks), so that in a stripe read, the slower disks no longer limit the full throughput realizable out of the faster disks.

WFS provides a mechanism to implement proportional striping for a RAID-0 system. The layout information contained in the file metadata indicates how exactly the data is to be striped across disks, including the relative ratio of data that needs to go to each disk. The read and write routines of WFS make use of this layout information to read or write data at high throughput inspite of heterogenities across disks. However, the mechanism for gathering the data that helps fix these ratios is not in place currently. One way of gathering the data would be by sending periodic probe reads or writes to different disks to get a

measure of their relative performance. However, assuming that such a scheme for gathering the relative performance information is available, WFS already has the mechanisms for implementing adaptive striping.

An additional benefit of keeping the layout information on a per file basis is that the proportion across disks can now vary across files. This serves to balance a tension that is caused by adaptive striping. By laying out less data at the slower disks and more data at the faster disks, adaptive striping essentially increases the seek time for random access at the faster disks. Since disks mainly differ in terms of sequential throughput and not so much in random seek latency, adaptive striping is favorable only to a predominantly sequential workload and could be counter-productive for a random access workload. By enabling a per-file decision on the ratio of striping, the filesystem is given the power to monitor access patterns for various files and use adaptive striping only for files with a predominantly sequential access pattern, and do uniform striping for a random access file. Again, this is another example of something that is possible only because the infrastructure is implemented at the filesystem level.

5 Implementation

We will discuss in this section, some of the important issues in implementing each of the three components of the system. The entire system is implemented in Linux. While the client filesystem is implemented inside the kernel, the disk and file manager are user-level service daemons. All these entities communicate through SunRPC over UDP. The choice of SunRPC over a lower level protocol like TCP was primarily motivated by the enhanced power that SunRPC provided, which greatly simplified the implementation. For example, we did not have to bother about packing and unpacking of arguments and managing retransmissions and maintaining timers for each message. More significantly, the callback functionality provided by the linux kernel SunRPC, together with the facility to initiate asynchronous RPC greatly simplified the implementation of asynchrony in read and write routines of WFS. Implementing this on top of a protocol like TCP would have been considerably more complex. The tradeoff in the choice of RPC over plain TCP is that RPC is a bit more heavyweight owing to the generality of its XDR (extended data representation) layer, among others. We did some initial performance measurements on the latency incurred by TCP versus RPC and we concluded that the performance difference was not

significant enough to motivate venturing on the more complex task of using TCP.

5.1 Client Filesystem

The client filesystem is implemented as a kernel module in Linux. It implements the VFS interface exposed by the linux kernel and registers itself as a VFS-compatible filesystem when the module is loaded. Data specific to WFS are stored in filesystem-specific fields of various VFS data structures like the inode, the directory cache entry (dentry) and the superblock. WFS-specific routines for initializing and cleaning up these specific fields will be called by the VFS layer at the relevant time. We now briefly describe the implementation of a few major VFS functions in WFS.

5.1.1 WFS `mount`

When the filesystem is mounted, the client initiates an RPC connection to the file manager and gets a handle to the root of the remote filesystem component specified. The IP address of the file manager and the remote directory to be mounted are specified as part of the mount call. The RPC connection handle is maintained persistently for subsequent requests to the file manager, as part of the filesystem specific field of the superblock pertaining to the WFS being mounted. An inode is formed with the file handle and attributes returned from the file manager (again, the WFS file handle is stored in a filesystem-specific component of the VFS inode). The superblock is initialized with this root inode for the file system and a directory cache entry mapping this inode to the corresponding pathname.

5.1.2 WFS `open`

When an application does an `open` of a file in WFS, the `lookup` routine of VFS parses the pathname component by component starting at the WFS root (whose handle is maintained in the superblock), and at each step, calls the WFS-specific `lookup` function if the pathname-to-inode mapping is not present in the in-kernel directory entry (dentry) cache. WFS now sends a lookup to the file manager, specifying the file handle of the parent directory and the name to be looked up. If the name is valid, the file manager returns a file handle for the new path component or file, along with the layout information for the

file. The layout information returned by the file manager contains the following information:

1. Number of disks across which the file is striped
2. The RAID level of the file (currently only RAID-0)
3. The stripe size for the file, and the size of a stripe fragment (for e.g. the stripe fragment size could be 4K and the stripe size could be 20 fragments, distributed in some ratio across the disks)
4. A list of drive descriptors

The drive descriptors contain information on each disk, including information on the relative proportion of each stripe that needs to go to each disk. A drive descriptor has the following fields

1. The drive ID of the disk and its IP address
2. The object ID of the corresponding object containing the fraction of file at this disk
3. Capability to access the object
4. The number of fragments in a stripe that goes to this disk (for e.g. if the stripe size is 20 fragments, 3 disks can have values of 10, 5, 5 which would mean the first disk gets twice the amount of data as the remaining two, likely because it is approximately twice as fast as the remaining two disks

5.1.3 WFS `read`

There are two ways the `read()` function can be implemented. The standard way that is encouraged by linux VFS (and used in linux implementation of NFS client) is to use the `generic_file_read` interface provided by VFS which reads a file through the VM page cache. The page cache is hashed by the inode number and the logical offset within the file. The `generic_file_read` routine splits up large reads into individual page reads (multiples of the page size), and for each page that is not in cache, invokes the filesystem specific `read_page` function. Hence it would be sufficient to just implement the `read_page` function of VFS. However, though our initial implementation followed this approach, we felt that this was highly restrictive in the context of WFS. A major limitation

of this approach comes from the fact that this approach would fail to utilize the maximum parallelism available out of striping. Since each call to `read_page` from the `generic_read_page` routine is synchronous, we lose the capability to send requests to different disks in parallel and overlapping their latencies. Of course, the generic routine does some readahead, but that is quite limited, and worse, fixed in size. We would obviously like the readahead window to be dependent on the size of the read request, in order to exploit maximum parallelism while at the same time ensuring that unnecessary data is not fetched, thereby polluting the cache. Additionally, since the WFS-specific function always sees only single-page reads now, it cannot take any decisions that are dependent on the size of the read. For example, if we have a file mirrored across a slow and fast disk, then the choice of disk doesn't matter if the read is small (since the difference in seek times would be negligible), whereas for larger reads, the filesystem needs to select the faster disk for high throughput.

To overcome these limitations, we decided to implement our own routine for WFS read that sees the total read request, checks for cache hits and exploits maximum parallelism by issuing all constituent page reads in parallel, and then waiting simultaneously for all of them to complete. This is done by executing asynchronous read rpc calls to each page after locking the page, and then waiting on the pages to get unlocked. The callback function of the read rpc unlocks the page and wakes up the thread waiting on that page. We then copy the read data to the user buffer and return once all rpc's have finished.

5.1.4 WFS `write`

Writes are performed similar to reads except that the application need not be blocked until the asynchronous rpc returns. In tune with the normal Linux behavior of delayed writes, we just initiate asynchronous write rpcs to the disk and return control to the application. When the application does an fsync, we wait on each of those rpcs to return and also send a fsync message to the individual disks to make them flush their caches. The list of writes that are outstanding for a file are maintained as a write queue in the inode, so that on any flush of the file or freeing of the inode, we readily have the list of writes to wait on.

5.1.5 WFS `create, mkdir`

These are similar to open - the client filesystem sends the file handle of the parent directory and the name of the new file to be created to the file manager. The file manager creates the constituent objects on the drives and sends a handle to the new file, along with layout information for the file. The file manager currently selects the same set of disks and the same stripe ratio or each of the disks, but this could be extended to have the file manager make use of some periodically gathered state to decide which disks are free and then decide on their relative stripe assignments by making use of knowledge about their relative performance.

5.2 File Manager

The file manager is implemented as a user level RPC server that services requests from the WFS client. All metadata operations are executed at the file manager. The file manager maintains the hierarchy of the file system since the disks just export a flat object namespace. The file manager also maintains the mapping from a file name to the set of constituent objects the file is composed of. We follow a very simple procedure for maintaining these data at the file manager and looking them up efficiently. For each file in the WFS partition, the file manager has a corresponding file in its local disk. However, instead of containing the file data, the file contains just the metadata for the actual WFS file, while the data is contained in the disks. What this essentially means is that a lookup of a pathname just translates into an open on the local filesystem, reading the local file to get the metadata, and sending the metadata back as layout and attributes to the client. To facilitate easy lookup, the pathname of the file in the local filesystem is included as part of the WFS handle sent to clients, so that subsequent lookups that contain the handles of the parent directory directly point to the pathname to look for the metadata file. The unique id in the file handle is just set to the inode number of the file in the local filesystem.

A possible extension to the file manager would be to change it to store metadata in the network attached disks under well-known object ids instead of storing them in its local disk. The obvious advantage of this is that file manager replication now becomes trivial. Moreover, even in the absence of replication, if a file manager goes down, any other system can immediately take over as the file manager by reading the metadata from the well-known objects in the network disks. This would clearly improve availability of the file manager and its

resilience to crashes. The Zebra project [5] looked at a similar implementation of the file manager for better availability of the file manager.

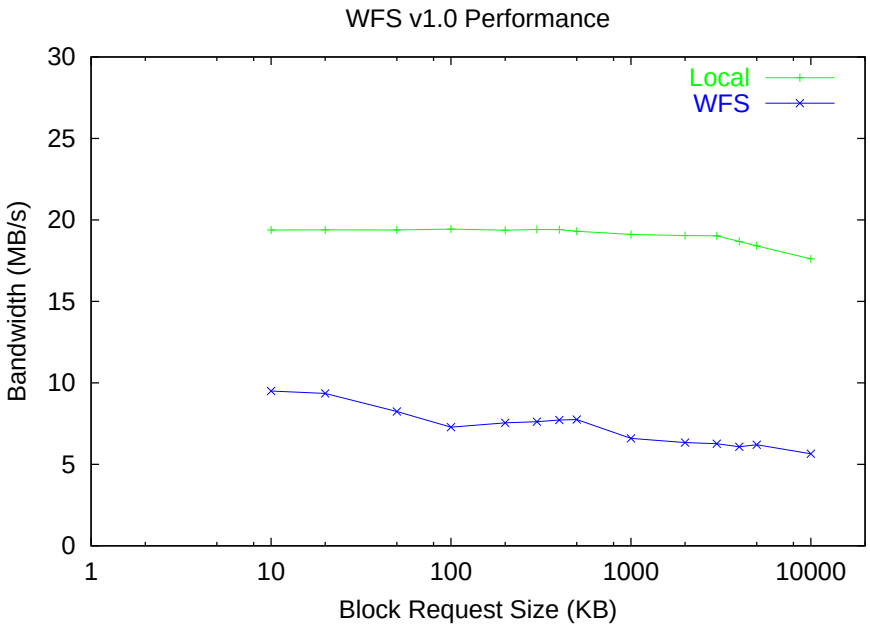
5.3 Network Disks

Similar to the file manager, the network attached disks are also implemented as user level service daemons that service rpc calls from both the file manager and the client. The disks currently implemented store data in the local ext2 filesystem, with a file for each object stored. The disk implementation is multithreaded for higher throughput. Again, instead of using the local filesystem to store objects, we could have stored data over raw disk, with our own simple metadata, instead of accepting the generality of ext2, much of whose functionality we do not require. However, doing so may have made the disk servers more inefficient since the multiple service threads would now have to lock metadata and freelist blocks that they update inorder to avoid races, and this would negate the advantage of avoiding ext2's extra overhead (Locking inside the kernel as part of a read/write syscall is much cheaper than explicit system calls to lock and unlock everytime an update is made). Hence we are convinced that the current choice of using the local filesystem to store objects is a reasonable compromise. Obviously, in an actual network attached disk, we might need to implement a lightweight special purpose filesystem since it is questionable if it could run a full-fledged Linux kernel.

6 Performance

The main initial objective of our performance measurements was to identify the overhead of accessing data over the network through RPC compared to local disk access. We thought this was a very pertinent comparison to make especially in view of our debatable choice of a heavyweight protocol like RPC for all filesystem operations. We mainly measure the sequential read and write throughput realizable from the system and compare it with the throughput of the local disk, which turns out to be around 20 MB/s in our case. The experimental setup consisted of three machines, one each for the client, file manager and a single disk server, each being dual P-III 500MHz with 1GB of RAM and SCSI disks of peak throughput 20 MB/s. The machines were connected by a Gigabit Ethernet.

Figure 2: WFS Initial Read Performance



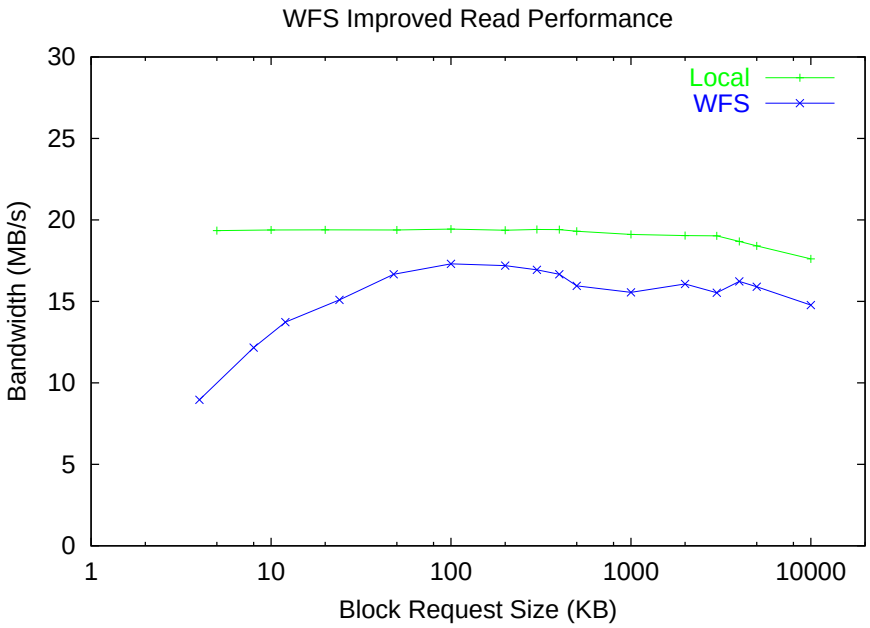
6.1 Read performance

The read throughput one can get out of WFS is obviously dependent on the block size of the read, i.e the size specified in each read request. A large size would facilitate greater parallelism since the client issues all page read requests within a single read request in parallel. On the other hand, if the read is just in 4K chunks, then each read would be synchronous, incurring the network round-trip latency for every read request, which, together with the disk read latency, would reduce read throughput.

Figure 2 shows the read performance observed as a function of the read block size. The experiment consists of reading a 100 MB file sequentially. Before each measurement, the cache at the disk is flushed by umounting and remounting its local filesystem. Similarly, WFS is unmounted and remounted at the client side before each measurement.

As can be seen, the read throughput obtained is dismally poor compared the local disk bandwidth, in some cases as low as one-third of the local bandwidth. After some analysis into the unexpected behavior, we tracked it down to a problem with the SunRPC implementation for the Linux kernel. The scheme

Figure 3: WFS Improved Read performance



for congestion control used in SunRPC seems to be overly conservative in deducing congestion, and whenever congestion is suspected, the asynchronous RPC issue process would block till the congestion clears up. The ultra-conservative congestion control, thus almost totally eliminates parallelism among multiple requests (basically it does not permit overlapping of network latency of multiple requests - even worse, it leads to additional queueing delays at the rpc queue before being issued). When we turn congestion check off (by changing the RPC transport structure returned by the SunRPC layer), we observe much better read throughput figures closer to the local read bandwidth. Figure 3 shows the improved read throughput with the modification to the congestion control field in the rpc structure.

6.2 Write Performance

Unlike the read performance, the write throughput in WFS does not depend on the size of each write request. This is because writes are always performed lazily in that control is returned to the application as soon as the asynchronous rpc is initiated. In our experiments, we write 100 MB to a file sequentially, and do

Table 1: WFS Write performance

System tested	Write throughput (MB/s)
Local disk	17.1
Initial WFS	6.2
WFS with 'async'	16.1

an fsync at the end, to ensure that we are accounting for the latency to write to the disk at the remote end. As mentioned earlier, during an fsync, WFS waits for all pending write RPCs for the file to return, and then issues fsync requests asynchronously to all the disks which contain parts of the file, to ensure that the cache at the remote end is also flushed. Table 1 contains the sequential write bandwidth that we obtained for write to a local disk and to WFS.

Again, as can be seen, the write performance of the initial WFS write implementation was quite low compared to the local disk bandwidth. We then tracked the problem to an artifact of the linux kernel delayed write mechanism at the disk server. Since these machines have a large amount of memory (1GB), the entire 100 MB initially ends up being buffered, and when an fsync request is sent, that is when the entire data is actually committed to disk. Because of this, there is no overlap between the network latency and the disk latency at the remote end, thereby leading to the client incurring first the full network latency and then waiting on the disk latency. A better implementation would permit the disk to keep writing data to disk periodically between rpc requests, so that these latencies are improved.

To do this, we added a new interface to the kernel at the disk server called an 'asynchronous sync'. This allows an application to specify, through an fcntl call, the threshold on a per-file basis for the maximum amount of data that can be dirty in the file. When the amount of dirty data for the file exceeds the threshold, the kernel initiates asynchronous writeback of the buffer cache pertaining to the file. When this new interface is used at the disk with a threshold of 5MB for the file, the write throughput improves heavily to closely match the local disk bandwidth. The reason for the improvement is that we now permit the disk server to overlap its disk latency with the network latency incurred by the client, so that when the final fsync is done by the client, very little data remains to be written.

7 Related Work

WFS was developed to serve as a possible infrastructure for the WiND[2] project, which looks at building highly manageable, self-adaptive storage that copes with heterogenity among disks in a network-attached storage paradigm. The NASD[1] work at CMU is also closely related to this work, in that the basic network attached storage paradigm is being exploited by WFS. However, a key difference is that NASD mainly advocates operating at the device layer, whereby different filesystems and applications can leverage the benefits of the NAS disk pool. Though it does consider modifying standard filesystems like NFS and AFS to use the NASD interface [3], issues like concurrency, redundancy and striping are managed only at the device level[4, 8], thereby suffering from the drawbacks of not using filesystem-level information to carry out optimizations. By allowing the filesystem total control over management of the network attached disks, WFS has the potential of exploiting these optimizations as detailed earlier. In this sense, WFS follows more along the philosophy of systems like Zebra [5]and Berkeley xFS[6], which give filesystems total control over the disks together with the responsibility to maintain redundancy for availability and performance. WFS differs from these systems in that both zebra and xfs depend on a log-structured layout at the disks, while WFS operates on top of a plain object interface similar to the NASD disk interface. The Petal[9] and Frangipani[10] projects at DEC illustrate a clear modularity between the responsibilities of the storage system and the filesystem, by designing the storage layer Petal to abstract much of the complexity of the network attached disk servers. Again, we believe that this demarcation results in information loss that could rule out implementing features like per-file security and redundancy, while at the same time not duplicating what is done beneath the filesystem

8 Future Work

The immediate candidate for future work would be taking more detailed performance measurements on the filesystem to examine its scaling with addition of disks, and in the face of multiple clients sharing access to the disks. We would also like to examine the benefit of adaptive striping tuned to the access patterns on a per-file basis (basically, turning off adaptive striping for files with predominantly random access pattern to avoid escalating the seek overhad). This would require very little work since WFS already has the mechanisms in place to do

adaptive striping. Crucial to adaptive striping is the availability of dynamic information on the relative performance of the disks - this would be another component to look at. Currently, WFS does not support any well-defined consistency semantics when multiple clients share access to a file. Implementing a suitable concurrency architecture by introducing some locking and cache invalidation will be another direction for future work. Implementing other RAID levels in WFS (currently the implementation supports only RAID-0) and exploring how the other RAID levels could be made adaptive to heterogeneity, would be other possible candidates for future work.

9 Conclusion

In tune with the argument in favor of implementing a network attached storage infrastructure at a filesystem level as opposed to a device level as outlined above, we have implemented a new striped distributed filesystem that manages a pool of network attached storage devices. WFS has support for per-file redundancy and striping, adaptive striping across heterogeneous disks (subject to availability of information on the dynamic performance of the disks). We have also shown that the implementation, despite over a generic heavyweight protocol like rpc, imposes very little overhead compared to local disk performance.

References

[1] A Cost-Effective, High-Bandwidth Storage Architecture. Gibson, G.A., Nagle, D.F., Amiri, K., Butler, J., Chang, F.W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D. and Zelenka, J. Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems.

[2] Manageable Storage Via Adaptation in WiND Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, John Bent, Brian Forney, Florentina Popovici, Samvabi Muthukrishnan,and Omer Zaki 2001 IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01), Brisbane, Australia, May 2001

[3] Filesystems for Network-Attached Secure Disks. Gibson, G., Nagle, D.F., Amiri, K., Chang, F.W., Gobioff, H., Riedel, E., Rochberg, D. and Zelenka, J. CMU SCS Technical Report CMU-CS-97-118, 1997.

[4] Scalable Concurrency Control and Recovery for Shared Storage Arrays. Amiri, K., Gibson, G.A. and Golding, R. CMU SCS Technical Report CMU-CS-99-111, February 1999.

[5] John H. Hartman and John K. Ousterhout. "The Zebra Striped Network File System" , ACM Transactions on Computer Systems 13, 3, August 1995, 279-310

[6] Serverless Network File Systems. 15th Symposium on Operating Systems Principles, ACM Transactions on Computer Systems , 1995. Tom Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, Randy Wang.

[7] Security for Network Attached Storage Devices. Gobioff, H., Gibson, G.A. and Tygar, D. CMU SCS Technical Report CMU-CS-97-185, 1997

[8] Highly Concurrent Shared Storage. Amiri, K., Gibson, G.A. and Golding, R. Proceedings of the International Conference on Distributed Computing Systems, Taipei, April 2000.

[9] E.K. Lee and C.A. Thekkath. Petal: Distributed Virtual Disks. Proceedings of ASPLOS-7, Oct. 1996

[10] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In Proceedings of the 16th ACM Symposium on Operating Systems Principles, Oct. 1997

[11] Patterson, D., Gibson, G., and Katz, R., A Case for Redundant Arrays of Inexpensive Disks (RAID) Proceedings of the 1988 ACM SIGMOD Conference on Management of Data, Chicago IL, June 1988

[12] The HP AutoRAID hierarchical storage system. John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. Appears in ACM Transactions on Computer Systems 14 (1):108-136, February 1996