Performance Enhancements in SunOS NFS

Suresh Sivaprakasam

Department of Computer Science
State University of New York at Buffalo
May 1993

Table of Contents

Abstract

Three components of technology particularly important for file system design are: processors, main memory and disks. Processor speeds and main memory technology are increasing at an exponential rate whereas disk technology improvements have been primarily in the areas of cost and capacity rather than in performance. Modern file systems cache recently used file data in main memory itself and larger main memories make larger file caches possible. This has two adverse effects on the file system behavior: firstly, larger file caches alter the workload presented to the disk by absorbing a greater fraction of read requests and secondly, write requests must eventually be reflected on disk for safety, so disk traffic will become more and more dominated by writes. NFS writes are inherently synchronous and couple the application's performance to that of the disk and make it hard for the application to benefit from faster CPUs.

This paper presents a new optimized algorithm for NFS write/read by allowing them to be performed in batch-parallel mode rather than sequentially, with expanded NFS buffering. We present the results of benchmarking, which demonstrate the efficiency based on cost and benefit.

This document is available as Technical Report TR 93-18 of the SUNY, Buffalo Computer Science Department. A Postscript version is available via anonymous ftp to **ftp.cs.buffalo.ed**u as **/pub/tech-reports/93-18.p**s. The author can be contacted by electronic mail as **sivap-s@cs.buffalo.edu**.

Last Updated May 1993.

TRADEMARKS

LIABILITY

Acknowledgments

This work would not have evolved without the help of my advisor **Dr. Sreejit Chakravarty**, to whom I am most grateful. This odyssey has been a long one, and wouldn't have completed without the help of several people. I owe special thanks to the following people :

Harry Delano, my supervisor for making me a GA and giving me that terminal which helped a lot to burn the midnight oil. **Kenneth Smith**, for providing the most wonderful arrangement of crash machines, Ken a special thanks for that last disk you installed for me. **Ajay Shekhawat**, who aroused my interest in Unix and unveiled the arcana of hacking to me. **Sunder**, my brother for all his die-hard friends who helped me tirelessly and for the various suggestions in problem solving which helped me in the course of the project. **Adrian Cockroft**, **Larry McVoy** for providing all those papers which saved a lot of trouble.

And all those anonymous souls on **comp.unix.wizards**, **comp.unix.internals**, **comp.protocols.nfs** whose stimulating colloquy formed my main food for thought from which I gained most of my knowledge.

All the rest are mine, of course.

Network File System(NFS)

Overview

NFS is a distributed file system based on a client-server model, that allows transparent access to remote files. NFS provides two kinds of transparency:

- ❐ The files appear to be resident on the local disk attached to the local machine and all of the filesystem entries i.e. files and directories, are viewed the same way, whether local or remote. NFS hides the location of the file on the network.
- ❐ NFS mounted filesystems contain no information about the file server from which they are mounted. The NFS file server may be of a different architecture or running an entirely different operating system with a radically different concept of filesystem structure.

NFS achieves this first level of transparency by defining a generic set of file system operations that are performed on a *Virtual File System*. The second level comes from the definition of *Vnodes*, which are related to the ubiquitous inodes of the UNIX file system but hide the actual structure of the physical file system beneath them. The architectural differences are taken care of by using a standardized representation called *RPC/XDR*. Together they are woven by the concept of mount, which is an extension to the local UNIX mount to mount remote disks.

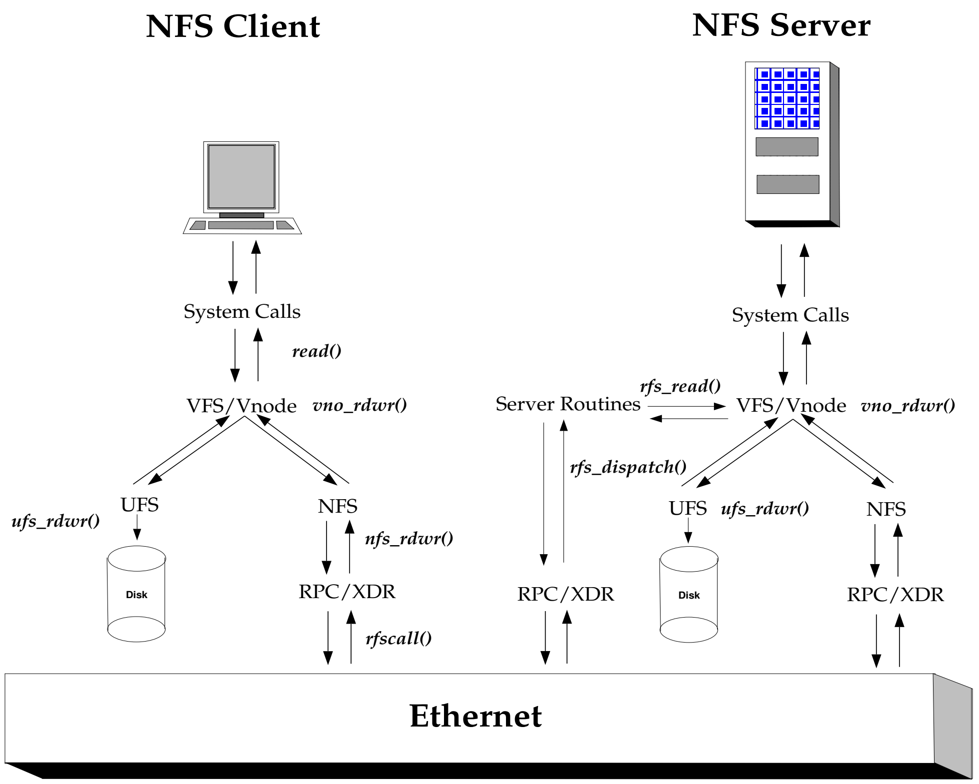

Figure 1: A kernel trace of a system call: We show the complete trace of a system call from the user-land to the kernel and back to the user-land. For simplicity sake we have taken the `read` system call which helps us in understanding the various layers of abstraction that exist within the kernel. Another reason for choosing `read` call is the different thread of execution taken for local and remote files. First, all the parameters passed are interpreted; the file descriptor passed is indexed into the *u-area's* user descriptor table to get the entry in the *file table*. Once we have the file table entry, the vnode can be retrieved. Next the generic operation for read `vno_rdwr` is called, which in turn calls the macro `VOP_RDWR`. `VOP_RDWR` based on the type of Vnode namely whether it is a local or remote file gets mapped into `ufs_rdwr` or `nfs_rdwr`. `nfs_rdwr` makes a RPC call `NFSPROC_READ` to the server, which in turn reads it from the disk and replies back to the client whereas `ufs_rdwr` reads it directly from the disk and returns to user land.

Structure

OSI Model

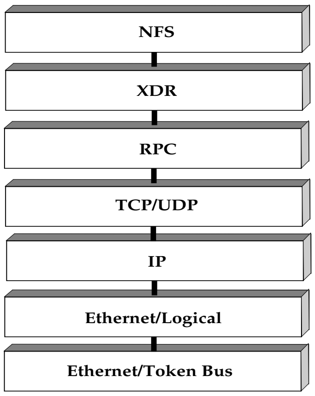

Figure 2: NFS protocol in the OSI Network model

Virtual File System

To support multiple filesystems of different architecture and design, there was needed a consistent interface which NFS could manipulate. The answer was to separate the filesystem dependent implementation from the filesystem independent implementation, and build a consistent interface between them. This interface is called the Virtual File System(VFS)/Virtual File Node(vnode).

All system calls that manipulate files or filesystems were modified to perform operations on the generic filesystem, VFS/Vnodes. A VFS is a data structure that is created whenever a filesystem is mounted. A Vnode is a data structure that represents an open file or directory. The VFS and Vnode structures provide the linkage between the generic filesystem and the 'real' filesystem underlying it which accesses the data. Within each VFS/Vnode is a set of procedures(pointers) that map the appropriate underlying filesystem operation.

RPC/XDR

RPC is a method of interprocess communication over a network. The benefit of RPCs is that they allow a programmer to program at a higher level of abstraction than most network interprocess communication mechanisms. In many cases the programmer can effectively ignore the underlying communication transport mechanism and its implementation.

The purpose of XDR is twofold. The first is to provide an architecture independent method for representing data to be passed on the network. The second is to provide a method for converting an architecture's own internal data representation into and out of XDR encoded format. XDR is necessary because of the proliferation of machines in the marketplace that have different representations of the same data type.

NFS Protocol

The NFS protocol is a set of procedures that uses the RPC mechanism to contact the appropriate NFS server on the remote host, which provides access to the actual file or directory to be manipulated. Since the file or directory maybe contained in a different architecture or operating system, a generic parameter for these procedures would be to use a file handle. To the client, the file handle is an opaque piece of data whereas the remote server understands the structure of the file handle and in turn uses it to perform the required operation.

Mount Protocol

Since NFS maps the filesystem operations to the underlying filesystem operations, it may so happen that some operating systems may not provide the UNIX concept of mounting a filesystem. It is for this reason that the mount protocol is separated from the NFS protocol itself.

Structure Layout

With this introductory chapter we will see some of the bottlenecks the NFS protocol imposes and the possible solutions for overcoming them. The solutions outlined in this document are entirely software oriented, the intention being to offer as cheap a solution as possible, since hardware always leads to accruing of cost with the number of machines being used. In particular we have focussed on two of the NFS procedures `NFS write` and `NFS read`, and propose a new algorithm that offers promising results. Chapter 2 discusses in detail the bottlenecks in the `NFS write` and `NFS read` procedures and offers a new schema that was implemented and benchmarked with good results. Chapter 3 is the concluding chapter, in which we analyze the problems with the NFS protocol and the future work that can be done on enhancing the performance. Chapter 4 offers the various references.

Performance Enhancements

I/O Performance Gap

File system design is governed by two major factors: technology which provides the basic set of building blocks and workload, which determines the set of operations that must be carried out efficiently. Three components of technology that play an important role in design of file systems are: *processor*, *disks* and *main memory*. Processor speeds have been increasing exponentially in contrast to disks whose improvements have been primarily in the areas of costs without significant improvements in the performance. The third component is main memory whose costs as well as speed has been increasing at an exponential rate.

Modern day file systems cache the recently used file data in main memory and with the advent of cheaper and larger memories, larger file caches are possible. This has two *adverse* effects on the file system:

❐ First, larger file caches alter the workload presented to the disk by absorbing greater fraction of the *read* requests.

❐ Second, with improvement in performance, by file caching comes the potential problem of data integrity. Consistency must be maintained between the main memory and the disk, and this once again changes the workload by seeping in more *write* requests to the disk.

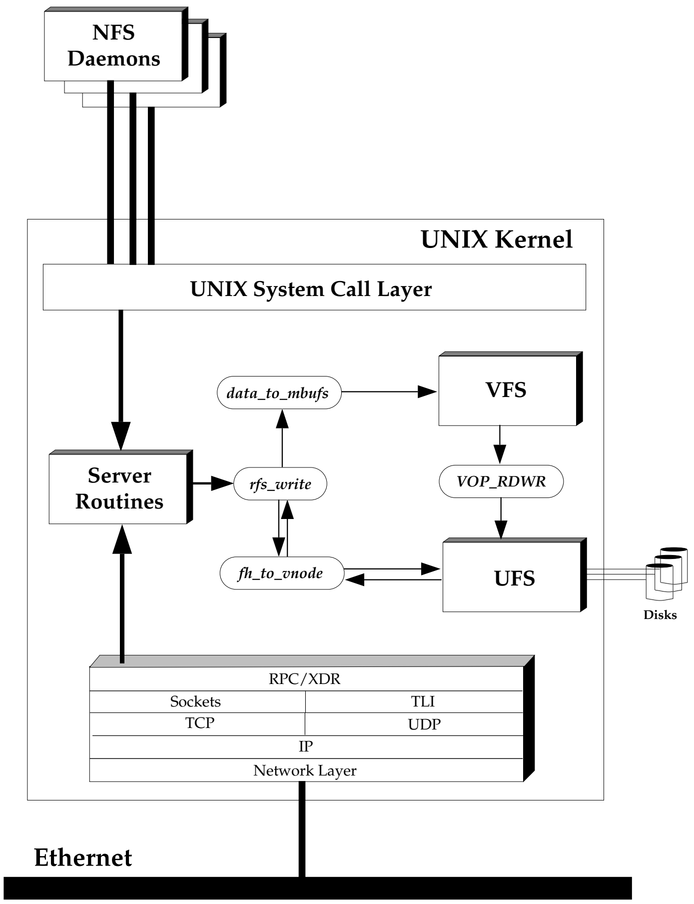

Figure 3: Anatomy of a NFS Write: Nfs daemons(*nfsds*) represent the NFS servers which serve client requests. *nfsds* are made user level processes which trap to the kernel on a system call never to return back, to bring about multithreading of NFS RPC requests in a otherwise single threaded kernel. The server's actions are defined by the NFS protocol which are available as remote procedures accessible by RPC calls to the client within the kernel. Each RPC call comes along with the name of the procedure to execute and the arguments to the procedure. NFS Write takes a `nfswriteargs` structure as a parameter which contains besides the data to be written, the file handle which identifies the file's vnode/inode. `fhtovp`[shown as *fh_to_vnode*] is the procedure that takes this opaque file handle, locks the *inode* and returns the vnode; but if the *inode* is already locked it makes the thread of execution sleep till the *inode* is released. The reason this is done is to prevent multiple writes being done simultaneously on the same file which may lead to corruption of data. Even though multiple *nfsds* can enter the `rfs_write` other than the first *nfsd*, the others get blocked inside `fhtovp` and are released only when the first *nfsd* releases the *inode*. Due to this, writes happen only in sizes of 8K and sequentially.

Current file systems(UNIX fast file system) tend to distribute the information on the disk; the consequence being that even small reads cause too many disk accesses. The real problem surfaces when writes are to be committed to the disk to maintain data integrity. A typical write causes an average of three disk accesses and an upper bound of five accesses. In NFS, the situation is still worse, to simplify crash recovery writes are done synchronously. Synchronous writes couple the application's performance to that of the disk and defeat the very purpose of file cache.

So in conclusion we see that the I/O performance seems to be limited by the bandwidth of the I/O and memory subsystems rather than the file system's design policies. In designing the enhancements we have focussed our attention on the file system's design policies and leave it to the hardware designers to improve the bandwidth for the I/O subsystems.

NFS Write

Current Implementation

The NFS protocol specifies 16 procedures for the Server side operation one of which is `NFSPROC_WRITE` which takes care of the NFS write. The server machine at boot time spawns certain processes called the NFS Daemons or the *nfsds*, which are user level processes that listen on a particular socket on which the NFS server procedures are registered. When a request comes through, the arguments are parsed, the appropriate procedure is executed and the result is sent back as a acknowledgment on the same socket. Let us look closely how a NFS write gets executed.

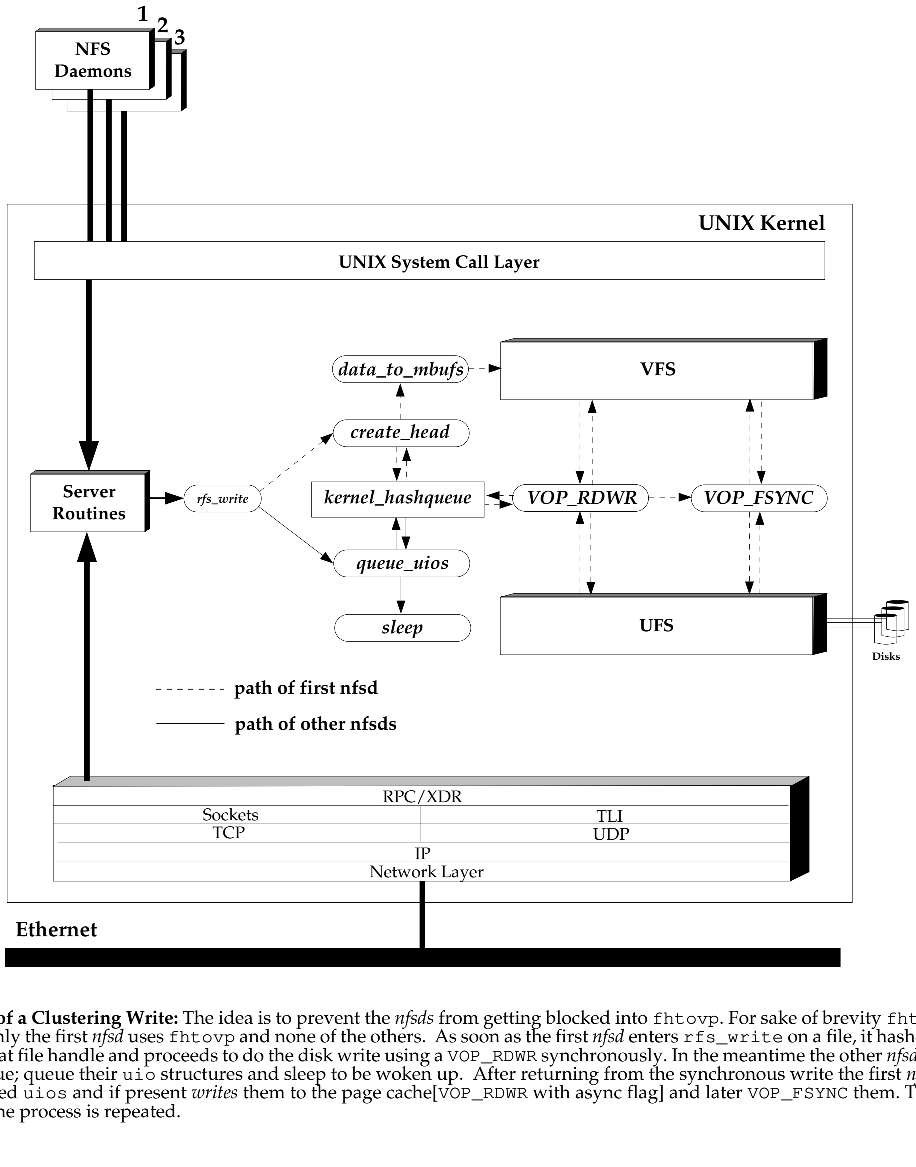

Figure 4: Anatomy of a Clustering Write: The idea is to prevent the *nfsds* from getting blocked into `fhtovp`. For sake of brevity `fhtovp` is not shown in the diagram since only the first *nfsd* uses `fhtovp` and none of the others. As soon as the first *nfsd* enters `rfs_write` on a file, it hashes on the file handle; creates a head for that file handle and proceeds to do the disk write using a `VOP_RDWR` synchronously. In the meantime the other *nfsds* check for the head under the hash queue; queue their `uio` structures and sleep to be woken up. After returning from the synchronous write the first *nfsd* checks under the same head for queued `uios` and if present *writes* them to the page cache[`VOP_RDWR` with async flag] and later `VOP_FSYNC` them. The appropriate *nfsds* are woken up and the process is repeated.

At the client side, the write goes through the VFS layer where the `uio` structure is constructed and then passed on to the NFS layer where `nfs_rdwr` does various error checking and then aligns on the appropriate page size and then calls `writerp`. From here `VOP_PUTPAGE` sends each page over-the-wire[OTW] bundling up data in a `nfswriteargs` structure along with the file handle from the vnode, the total packet not exceeding 8K size which is the maximum size that RPC layer can take. At the server side, when the packet comes in, `rfs_dispatch` dispatches it off to each *nfsd* which execute the `rfs_write` procedure. In `rfs_write` first, the file that is being written to, needs to be identified by converting the file handle to a vnode using `fhtovp`. Once the file is identified, UFS can schedule a write to the disk. Before scheduling the write, some amount of preprocessing needs to be done; the data inside the packet is either as raw data or in the form of `mbufs`, both of which need to be converted to a `uio` structure. Once the `uio` structure is constructed, `VOP_RDWR` can be called to schedule a disk write by the UFS.

Drawbacks

Two pronounced drawbacks in the above implementation are:

❒ Since the RPC/XDR layer imposes a size constraint of 8K, a disk write also goes in chunks of 8K only.

❒ At any instant of time there can be only *one* ongoing write to a particular file on disk, because the UFS locks the inode before proceeding on a write. In the above setup `fhtovp` locks the inode and returns for the first *nfsd*, whereas the other *nfsds* are put to sleep and don't return. The consequence, *writes* of any size are written sequentially to the disk.

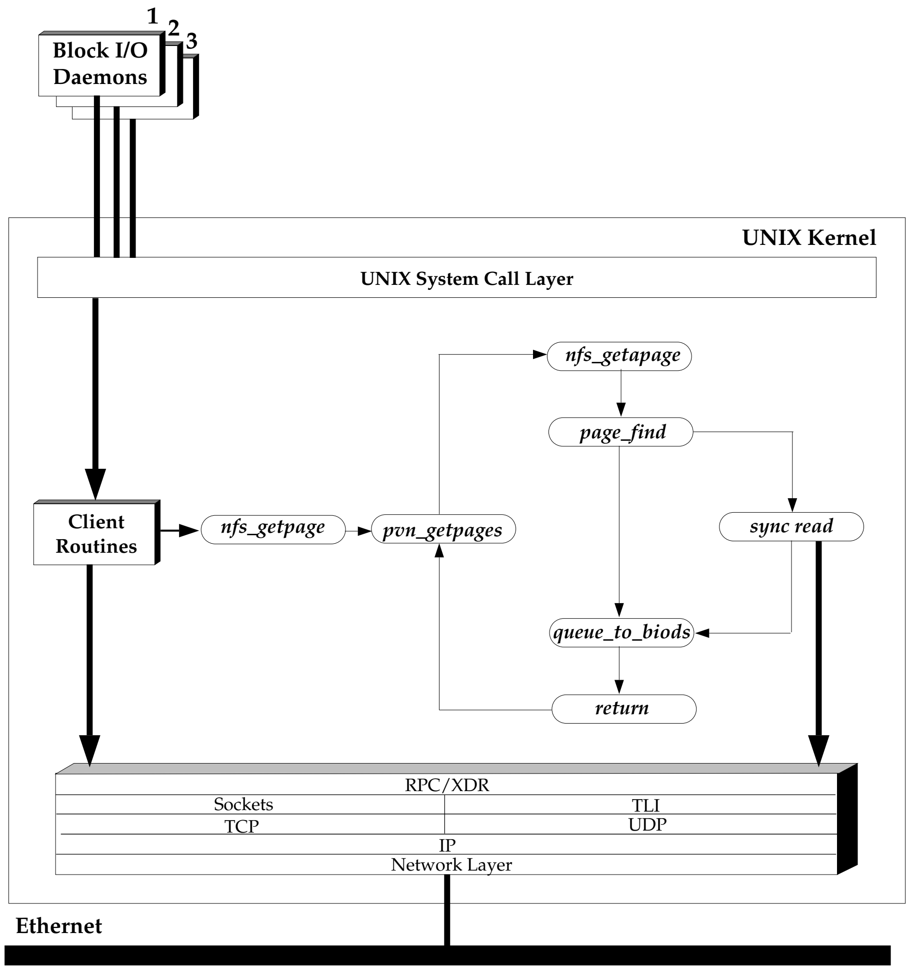

Figure 5: Anatomy of a NFS Read: The Block I/O daemons(*biods*) serve the same purpose to the clients as *nfsds* to the server and are also user level processes which trap to the kernel never to return. The client has a set of generic macros which are mapped to *nfs* routines or to *ufs* routines. The *nfs* client routine that handles reading data from the remote server is `nfs_getpage`. This routine in turn calls `pvn_getpages` when the size of the data to be read is larger than 8K else it calls `nfs_getapage`. We consider the special case of sequential reads when readaheads need to be done. First, `page_find` is used to check whether the page is available locally in the page cache else a RPC call needs to be made. Either way if the page is found locally or remotely, a readahead for the *next* page is queued to the *biods* and then the kernel thread returns to `pvn_getpage` so that the next time, `nfs_getapage` is entered optimistically it will find that page in the page cache. But invariably the kernel thread finds that the page is in transit and is forced to sleep till I/O is done.

Clustered Write

This novel algorithm for NFS Write solves the above mentioned problems. The basic idea behind this approach is to collect *writes* to the same file, and commit them to the disk in chunks larger than 8K; with proper synchronization mechanisms to bring about rendezvous between *nfsds*. For sake of brevity we will assume that all the *nfsds* are executing a NFS Write for the same file, i.e. all of them have *writes* on the same file handle. Further the new `rfs_write` is multi-threaded, i.e. the thread of execution for the first *nfsd* is different from the thread of execution of the remaining *nfsds*.

When the first *nfsd* enters `rfs_write`, it needs a way to flag a signal to the other *nfsd* that there is an ongoing *write* on the same file handle. This is needed to prevent the other *nfsds* from getting trapped in `fhtovp` which doesn't return when the inode is locked. To enable this flagging the kernel maintains a new data structure which is a linear array into which the first *nfsd* hashes into and creates a *head* whose value is the file handle itself. So when the other *nfsds* enter `rfs_write` they check up whether there is a head for the file handle; if present they know that there is an ongoing write on the same file handle, so they queue up their data after the normal preprocessing in the form of a `uio` structure and sleep on the respective `uio` structure. When the first *nfsd* returns after a write, it hashes back to look under the head whether there are any impending *writes* for the same file handle; if present it collects all the `uio` structures and puts them into the buffer cache(memory) and then does a `VOP_FSYNC` on the file handle which guarantees that all the data blocks for that file handle is on the disk. Next the sleeping *nfsds* are woken and given the reply which needs to be sent to the client who made the call. This last operation is repeated till there are no more `uios` to be written under the head, when the first *nfsd* returns from the `rfs_write`.

The performance gain is clearly evident: we have overcome the 8K barrier, because when `VOP_FSYNC` is done; typically data is written in chunks larger than 8K and above, which gives a severe hike in write bandwidth as the results show.

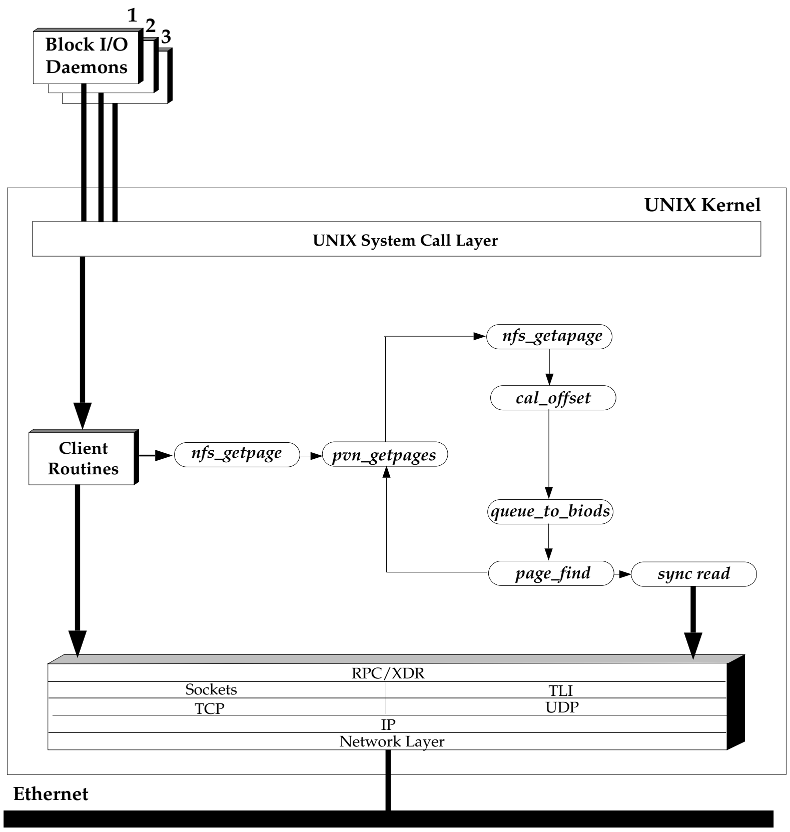

Figure 6: Anatomy of a Enhanced NFS Read: In this new *read* we change the scheduling of the readaheads. To increase the parallelism we first calculate the offset of the next read and schedule that as a readahead to the *biod* and then carry on with the normal routine of checking up in page cache and if not present we make a RPC call to the server. The advantage is that with the synchronous call a asynchronous read is also proceeding simultaneaously. This increases the probability of finding the page in memory when the kernel thread of execution returns to read the next page.

NFS Read

Current Implementation

The NFS protocol specifies `NFSPROC_READ` as one of the 16 procedures which is called by the client via RPC. The client side interface to this procedure is through a client side's `nfs_getpage`. `nfs_getpage` breaks up the data size to be read into 8K chunks and then makes the RPC call. The NFS client runs daemons similar to *nfsds* on the Server side called *biods*[Block I/O], which help in NFS I/O. `nfs_getpage` handles sequential and random in slightly different ways. Firstly the client needs to detect that it is doing a sequential read; the way this is done is to update a variable before a *read* is done with the next offset to be read. So during the next read when the variable is checked to find it contains the same offset, `nfs_getpage` knows it is doing sequential read. A good reason why we need to detect sequential read is to give readaheads to *biods* so that the NFS Read can be completed faster. Let us a take a closer look at how a sequential read is done.

`nfs_getpage` calls `pvn_getpages` when the data size to be read is larger than 8K, and this in turn calls `nfs_getapage` for each 8K chunk. Let us see a trace of `nfs_getpage`. First the next read offset is updated in the vnode's rnode. Second, the page is to be checked if it is available in the page cache. If available locally in the page cache, NFS read is easily done, else a requisite page is set up in memory along with a `buf` structure for I/O; with this setup ready a RPC call is made. After the RPC call a *read* for the next offset is given to the biod so that next time, NFS *read* will find the page in page cache rather than doing a RPC call. A point to note is that kernel thread doesn't need to wait till the *biod* gets done, rather it can queue the offset for the *biod* and continue with its thread of execution.

Drawbacks

One major drawback in the above implementation is:

- ❐ Typically when the kernel thread of execution returns for the next consequent read, it finds that the page is in I/O transit and is invariably put to sleep till the process doing I/O i.e. biod, is done.

Enhanced Read

In the enhanced read, the drawbacks are overcome by incorporating a change in `nfs_getapage`. In this new `nfs_getapage`, first the next offset to be *read* is calculated; then it is scheduled to the *biod.* Now the page cache is checked for the presence of the page containing the current offset else a RPC call is made to the server. When `nfs_getapage` is entered for the next read, with a very high probability the *biod* would have completed I/O on the page and will find the page in cache and don't need to sleep waiting for the I/O to get done.

The clear advantage is in the savings of time we get by eliminating the kernel sleep, which increases the bandwidth of the NFS Read.

Performance Results

The entire implementation and benchmarking of all the enhancements was done on a Sun 3/50(server) and Sun 3/160(client). The modified kernel with the clustering write and the enhanced read was incorporated into a base SunOS 4.1.1 kernel and was run on the Sun 3/50 which acted as the server for Sun 3/160. Both have a main memory of 2Mbytes with SCSI disks.

Since our enhancement work was primarily in the area of NFS writes and NFS reads, we needed to see the improvement in bandwidth. No state-of-art benchmark gave us this capability, so we decided to write our own benchmark which would give us this statistic. The *read* benchmark mainly `mmaps` a file and just before writing `msyncs` it with invalidation with a timer. The *write* benchmark writes in a NFS mounted file system, timing the results. The results are displayed in the table next page.

NFS Writes

We sampled on ten sets of 1000 writes each of 100K and the following are the results.

NFS Write (Kbytes/second)	Samples										
	1	2	3	4	5	6	7	8	9	10	Avrg
Without Clustering	83.86	83.80	84.68	80.97	86.25	87.14	86.96	85.07	86.20	103.58	86.85
With Clustering	129.54	129.53	129.76	129.34	128.45	129.46	129.39	129.77	129.71	129.75	129.47

Enhancement: 49.01%

NFS Reads

We sampled on ten sets of 1000 reads each of 100K and the following are the results.

NFS Read (Kbytes/second)	Samples										
	1	2	3	4	5	6	7	8	9	10	Avrg
Current Read	349.72	340.39	340.39	342.06	341.66	341.47	340.92	342.09	340.98	340.75	342.04
New Read	373.42	370.69	376.08	374.42	373.33	375.41	376.40	375.10	371.46	366.76	373.31

Enhancement: 9.1%

Future Work

Summary

This work initially started out as a study of the design and implementation of NFS, to identify the problems and incorporate changes whereever there was scope of improvement. We identified two areas of improvement i.e. NFS Write and Read, and implemented new algorithms. Our experiments have convinced us that the clustering prototype for NFS Write outperforms the current implementation of NFS Write without disrupting the NFS protocol in any manner. The new NFS Read performed better only for sequential reads but with increasing processor speeds and memory sizes, we cannot afford to overlook any scope for improvement however little it may be. In the rest of the chapter we will outline future scope for improvement on write clustering prototype, a few other areas which can be improved upon and other problems which arise due to the NFS protocol.

NFS Write

Clustering Write

Even though the clustering prototype outperforms the current NFSWrite, there is still a lot of scope for further enhancement which can lead to proper utilization of the *nfsds* and greater bandwidth. Following are two areas with scope for further improvement:

❒ ***Freeing nfsds***. The role of the *nfsd* is dichotomous, one is to queue up the data which gives the real performance hike and second, is to send the reply back to the client whose request was being served. To accomplish this we make the *nfsd* sleep till the entire write operation is done so as to send the reply back to the client which is the modification time of the file. We can alleviate the *nfsd* of this second function by queuing enough information along with the data so that the first *nfsd* can send the appropriate replies and the *nfsds* can be freed leading to better utilization and performance.

❒ ***Reducing disk latency***. The real performance hike comes due to the fact that disk writes are done in sizes larger than 8K. For every chunk of data written the inode must also be updated and hence written to the disk. This may cause a disk latency if the inode resides in a different cylinder group. To subdue this effect, we can schedule the inode write before the data write by using `VOP_PUTPAGE` instead of `VOP_RDWR`, hence giving better performance.

Write Caching

By not having to wait for a physical disk write to complete, we can improve the bandwidth of a write operation. This improvement could directly reflect on not only the NFS Write but also `Setattributes`, `Create`, `Remove` and `Makedirectory`.

References

[1]. Russell Sandberg, David Goldberg, Steve Kleiman, Dan Walsh and Bob Lyon. *Design and Implementation of the Sun Network File System*, Proceedings of the Summer Usenix Conference pp 119-130, Oregon 1985.

[2]. Sun Microsystems Inc. *NFS: Network File System Specification*, RFC 1094.

[3]. V. Srinivasan and Jeffrey C. Mogul. *Spritely NFS: Experiments and Cache Consistency Protocols* pp 45-57. Proceedings of the 12th ACM symposium on Operating Systems Principles, Litchfield Park, AZ December 1989.

[4]. Chet Juszcak. *Improving Performance and Correctness of an NFS Server*. In Proceedings of Winter Usenix Conference pp 53-63 San Diego, February 1989.

[5]. Steven Kleiman. *Vnodes: An Architecture for Multiple File System Types in Sun UNIX*, pp 238-247, Proceedings of Summer Usenix Conference, Atlanta GA. 1986.

[6]. Robert A. Gingell, Joseph P. Moran and William A. Shannon. *Virtual Memory Architecture in SunOS* pp 81-94 Proceedings of Summer Usenix Conference, Atlanta GA. 1986.

[7]. Joseph P. Moran. *SunOS Virtual Memory Implementation* Proceedings of the European UNIX Users Group April 1988.

[8]. M. Kirk McKusick and et.al. *A Fast File System for UNIX* ACM Transactions on Computer Systems vol 2, no. 3 pp 181-197, August 1984.

[9]. John. H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols et.al. *Scale and Performance in a Distributed File System* ACM Transactions on Computer Systems 6(1) pp 51-81 February 1988.

[10]. Mendel Rosenblum and John K. Ousterhout. *The Design and Implementation of a Log Structured File* System. In proceedings of 13 Symposium on Operating Systems Principles pp 1-15.