For: bmb
Printed on: Fri, Mar 12, 1993 18:33:49

Document: logfs.report
Last saved on: Sun, Oct 11, 1992 11:46:52

A Synchronous File Server for Distributed File Systems*

Bradley M. Broom†

The Australian National University
Department of Computer Science

Abstract

There is currently an enormous performance differential, as much as a factor of ten, between the write performance of local file systems and that of distributed file systems, such as the popular NFS [1]. Moreover, the underlying cause of this performance gap is such that as disks with faster transfer rates, or as multiple disks in parallel, are used, this performance differential can only increase.

The performance differential is often attributed to the stateless nature of the distributed file system protocol, and many stateful protocols have been developed, partly justified by the increased file system write performance that results. Stateful protocols, however, are considerably more complex than stateless ones, and should be avoided if possible.

The poor write performance of distributed file systems is actually due to a mismatch between the stateless file system protocols, which require the file system to be written synchronously, and current file system designs, which perform very poorly at writing synchronously. This mismatch can be rectified either by changing to stateful protocols, or by developing file systems that are good at writing synchronously.

This paper describes the design of the Acacia synchronous file system, and analyses the performance of a prototype, user-level server implementation. The prototype's performance already exceeds that of NFS, and detailed analysis of its performance indicates that further development will result in write performance similar to that of existing local file systems.

* This work is supported by the Fujitsu-ANU Joint Research Agreement.
† E-mail: Brad.Broom@cs.anu.edu.au

1. Introduction

Recent, very large performance improvements in central processor speeds, of approximately 40% per year, have been accompanied by more modest improvements in the speed of secondary memory, where disk drive performance has increased by only 7% to 8% per year. Consequently, file system performance, or the lack thereof, is increasingly the cause of poor system performance.

Poor file system performance is especially detrimental in a distributed environment, where *clients*, such as workstations and / or X-terminal CPU servers access common files stored on a central *file server*, using a distributed file system protocol, such as NFS [1]. The client / server approach simplifies network administration, since only one copy of a software package needs to be maintained, and has cost advantages, since a small number of large disks is cheaper than a large number of small disks. However, this approach has performance disadvantages, which are becoming more severe as the power and number of the clients increase.

One reason for the poor performance of distributed file servers is that the design of the server file systems is not appropriate for use in conjunction with the common distributed file system protocols. To simplify crash recovery, many distributed file system protocols, such as the widely used NFS, are *stateless*: servers do not maintain state information about their clients, and the clients are not affected by server crashes or network outages (except for a significant performance drop): the client simply retries failed transactions until a positive acknowledgment is received from the server. A consequence of stateless protocols is that all client writes must be written to stable storage before the server acknowledges the request. Otherwise a server crash after the acknowledgment was sent, but before the data was committed, would result in the data being lost.

That is, such *synchronous writes* must occur at the same time as the request; in particular, before the server acknowledges the request. A *synchronous file server* is a file server that can efficiently perform synchronous writes.

Current file system designs, however, are not well suited to synchronous write operations, and are especially poor at writing, synchronously, to several files concurrently. The latter situation is common in a distributed environment, where each client is working on separate files. It is also common when small file creation is involved, since this requires writing the directory as well as the file itself. In most workstation environments, small files are created more frequently than large files [5].

This paper describes a *synchronous file system*; that is, a file system that is efficient when writing synchronously, and thus able to deliver high performance when used in a distributed file system. It has been developed in the context of the Acacia parallel / distributed file system for high performance multi-computers [4], although it is also useful for distributed systems based on local area networks.

The following section describes current file system designs and explains why they do not perform well at synchronous writing. Section 3 describes the design requirements of a file system suited to synchronous writing, and describes the log-structured file system approach. Section 4 describes the Acacia file system design, and section 5 analyses the performance of the current implementation. Finally, section 6 summarises the results and outlines further research.

2. Analysis of Current File System Designs

A contemporary file system design, such as the Unix Fast File System [2], divides the disk (or disk partition) into small groups of disk cylinders called *cylinder groups*. Each cylinder group contains a small number of file descriptor entries (or *inodes* in Unix terminology) and the rest is used for file data blocks. The file data is written, if possible, into the same cylinder group as the file's inode, at locations highly optimised to reduce the time subsequently taken to read the file. Once the file reaches a certain size, subsequent data blocks are written in a different cylinder group to prevent a single file from using all the available space in any particular group.

When data is written to the file, the disk must seek to the cylinder containing the file data, write the data, then seek to the cylinder containing the inode, and update the inode. Both of these seeks and both of these writes must be performed before the server responds to the client. If the client is creating a new file, the server must create and write the inode, modify and rewrite the directory file containing the new file, and update the inode for the directory file, then reply to the client, which writes the file data. For each new data block appended to the file, the server must allocate and write the new file data block, update the file's inode, and then reply to the client.

When used in a local file system, the Unix Fast File System is not required to update the disk before the client, in this case the local operating system, proceeds further; modified data blocks are stored in the system's file buffer until written to disk at the file system's convenience. Consequently, the file system can optimise the order in which blocks are written to disk, and even avoid some writes, since the file's inode need not be written to disk after every single update.

When used in a distributed file system, however, both the data and meta-data (inode) blocks must be written to disk before the server can reply to the client. Consequently, the server must do many seeks between comparatively small writes, and the effective bandwidth of the disk will be substantially below its maximum transfer rate. For example, based on the typical disk profile shown in figure 1, appending one 8 Kbyte block (the largest size allowed by NFS) to a file requires a seek of an average approximately 17 msec, an average rotational latency of 8.3 msec, a data transfer to the SCSI controller of 2 msec, a write of 3.7 msec, a seek to the inode of another 17 msec, a rotational latency of 8.3 msec, and a write of 0.5 msec (if only the single sector containing the modified inode is rewritten). The total delay of 56.8 msec is more than ten times that actually required for writing data to the disk (4.2 msec). Under these assumptions, the server write throughput would be 140 Kbyte/sec.

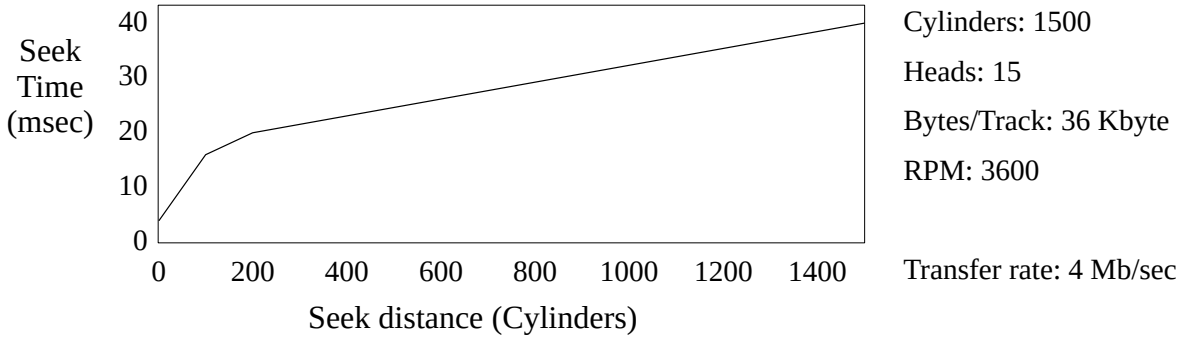

Figure 1. Typical SCSI Disk Profile

If the server is required to write multiple files concurrently, the seeks involved will take even longer, if, as is likely, the files are stored in different cylinder groups.

By way of comparison, simple experiments with NFS under SunOS 4.1, using a disk with characteristics similar to the one shown in figure 1, obtained write throughputs of at most 80 Kbyte/sec, which is somewhat less than the above analysis suggests. Although the maximum performance may be slightly greater, this figure is a reasonable estimate of NFS performance. It is much less than a tenth of the disk's bandwidth.

These inefficiencies can be partially overcome by including in the client a number of *block I/O daemons*, which write data asynchronously to the application. That is, when an application writes a disk block, that write is passed to a block I/O daemon which actually writes the block to the file server, and waits for the server to respond, while the application continues asynchronously. That is, the client machine, not the client application, must do synchronous updates.

Each of these block I/O daemons can execute their writes concurrently. If the writes are to the same file, the server may be able to write all of the data blocks concerned before seeking to and updating the inode and replying to all of the requests at once. If eight block I/O daemons are involved, the group of transactions will require a 17 msec seek, an 8.3 msec rotational delay, eight writes of 5.7 msec each (including transfer time), a seek of 17 msec, a rotational delay of 8.3 msec, and a single write of 0.5 msec. The total delay in this nearly optimum case is 96.7 msec, more than twice that required to write the data itself. Under these conditions, the server write throughput would be 660 Kbyte/sec.

The inefficiencies described above are caused by the need to seek between the data and meta-data blocks in the file system. Buying disks with faster transfer rates, or connecting multiple disks in parallel, as in RAID systems [6], will achieve limited performance improvements, since two of the significant delays—the seek and rotational latencies—are not improved by such techniques.

3. Synchronous File System Design Requirements

As identified by Ousterhout [7], file system write performance can be dramatically improved by re-organising the file system so that seeks between data and meta-data are not required, and the

effect of rotational latencies can be minimised by writing very large amounts of data at a time. In the Sprite log-structured file system, Ousterhout used write sizes of up to a megabyte, although subsequent analysis [9] shows that these large writes reduce overall read performance, and that the ideal write size for optimising both read and write performance falls in a broad range between 40 and 70 Kbytes.

Eliminating seeks between the data and meta-data means that the meta-data can no longer occupy a fixed position on the disk, as in conventional file systems. Instead, the meta-data must be written very close to the data itself.

One approach for storing data and meta-data close together is as a log-structured file system [8], in which all changed data and meta-data is appended to a (logically) write once log. When a disk block is updated, for example, both the new contents of the disk block and the updated inode are appended to the log. When the tail of the log stored in primary memory has grown sufficiently large, it is written in one large piece to the end of the disk copy of the log. During operation, primary memory tables store the locations of file meta-data so that the log need not be accessed for every read operation. When the file system is restarted, the log is read to determine the current state of the file system, and the current location of file meta-data. Periodically, the current file system state is dumped to one of two reserved locations of the disk, to limit the length of log that needs to be reread when the file system is restarted.

When the log has reached the end of the disk, deleted or updated data has left holes in the log, so the write position of the log wraps around to the start of the disk and writes in those holes. Clearly, good performance requires these holes to be reasonably large, and a free space compaction algorithm, similar to a garbage collector, must be used. Appropriate algorithms for the Sprite design are described by Rosenblum and Ousterhout [8].

The Sprite log-structured file-system was not designed specifically for synchronous operation, since Sprite uses a stateful distributed file system protocol and a complex recovery protocol. Nevertheless, the Sprite log-structured file system will likely support a stateless distributed file system protocol reasonably well for loads dominated by writes, since under such conditions the heads will rarely have to seek to the end of the log. The server can accumulate whatever writes occur within a reasonable time, and then append an appropriately sized section to the log without imposing undue latencies on its clients. However, when reads occur, the heads will have to seek from the current read position to the current end of the write log, and afterwards seek back to the next read position. Thus, for write loads interspersed in moderate read loads, a log-structured file system will deliver inferior performance, because seeks will often be required for comparatively small amounts of data read or written.

The principle of storing the meta-data on the same cylinder as the data clearly applies to a synchronous file system, but the need to seek between the current read position and the end of the log does not. A synchronous file system should allow all data and meta-data to be written in any

cylinder close to the current read position having sufficient free space, and for the write to occur without unnecessary rotational delays.

4. The Acacia File System Design

4.1. Overall Design

The Acacia file system consists of *inodes*, which contain details about individual files, such as protection information, and in particular pointers to data blocks, indirect blocks (which contain pointers to data blocks), or doubly indirect blocks. So far, the Acacia file system is very similar to the Unix Fast File System.

Unlike the Unix Fast File System, inodes do not reside at fixed disk locations. Instead, inodes are grouped into *inode blocks*, which are written at any convenient location. To facilitate the subsequent location of an inode block, pointers to the current locations of the inode blocks are maintained. To support a large number of inodes, these pointers are maintained in *inode indirect blocks*.

The inode indirect blocks are also not written at fixed locations, and pointers to the inode indirect blocks must be maintained. The number of inode indirect blocks is small enough that these pointers can be stored in the file system's superblock.

The *superblock* contains details about the file system's organisation, including the low level name of the file system, its capacity, and so on. It also includes the pointers described above to the inode indirect blocks.

A disk write operation therefore requires writing the new data block(s), indirect block(s) (if any), inode block, inode indirect block, and superblock. All blocks in this sequence must be written, since each contains a pointer to the previous block(s) in the sequence.

However, the only ordering constraint on the above writes is that the superblock write occurs last. Before it occurs, no change in the file system is recorded. Once it has occurred, the disk copy of the superblock contains an updated pointer to the new inode indirect block, which contains a pointer to the new inode block, which contains pointers to the new data and indirect blocks. Thus, the update is now committed to disk, the disk space previously occupied by updated data can be released, and the client can be notified that the write is complete.

4.2. Seek Optimisation

Clearly, the superblock cannot be stored at a fixed location on the disk, since that would require two seeks for each update, which is the reason current file systems perform badly. In Acacia, one block per cylinder is permanently reserved for the superblock. An update operation finishes by

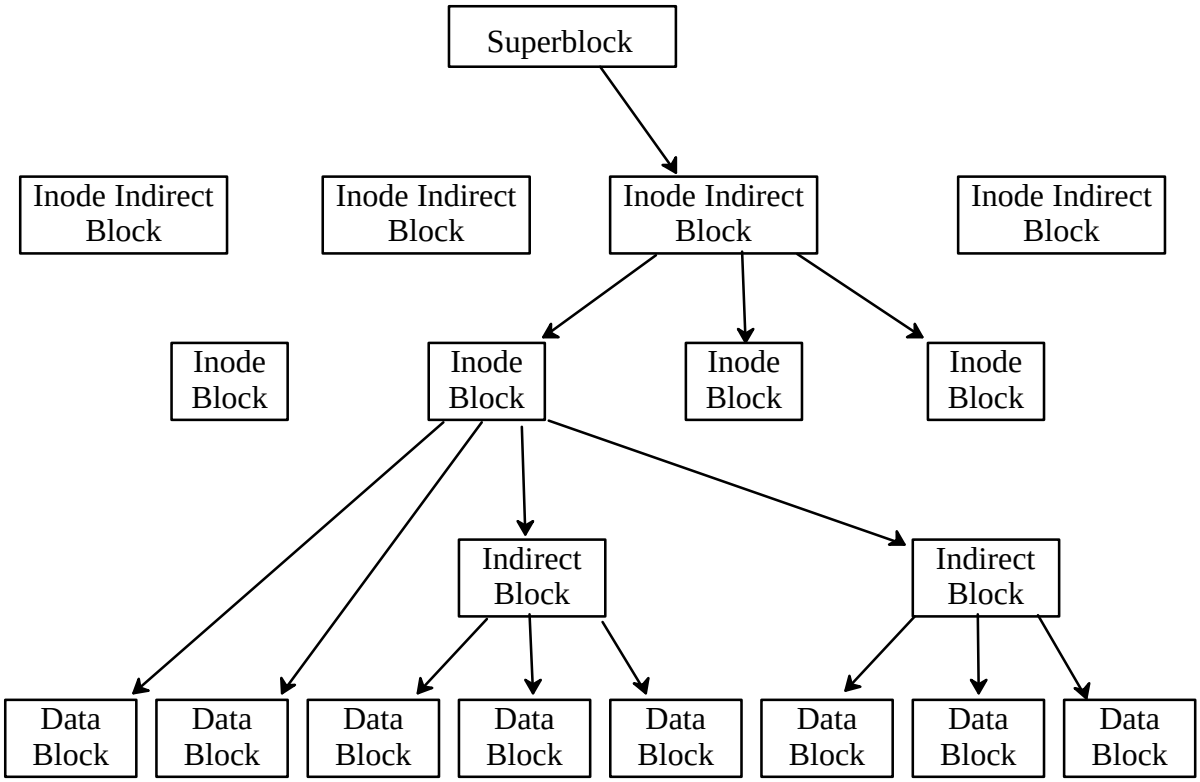

Figure 2. Logical File System Organisation

writing the superblock into the reserved block in the current cylinder. So that the current superblock can be identified when the file system is restarted, the superblock contains a sequence number which is incremented each time the superblock is updated; on restart, the file system simply searches for the superblock with the greatest sequence number.

The cylinder(s) on which an update is written is not constrained by the file system. If only one update is pending, the file system might seek to the closest cylinder with sufficient free space, whereas if many updates are pending, it may seek slightly further to locate a group of cylinders with more free space.

The reservation of a superblock entry in each cylinder is a classic space/time trade-off. For current disks, having the typical disk characteristics shown in figure 1, one block per cylinder is approximately one percent of the available space. Additional free space is highly desirable so that writes can usually be performed in a single cylinder, close to the current position. Although detailed performance studies for nearly full disks have not yet been done, it is likely that the necessary free space will be similar to that of the Unix Fast File System, in which ten percent of the file system is so reserved.

4.3. Rotational Optimisation

Rotational delays are second greatest source of delay after seek delays. The majority of disks rotate at 3600 rpm, and so each write suffers an average rotational latency of 8.3 msec. The simplest way of avoiding rotational delays is to do large consecutive writes, as in log-structured

file systems. However, this strategy is not applicable to the Acacia design since the superblock must be written to a fixed location in each cylinder. Thus, each update must consist of at least two writes: one for the non-superblock data and one for the superblock. Also, it is unlikely that all the free space in a cylinder is located contiguously so that a single large write can be issued.

Moreover, the Sun operating system automatically decomposes large (≥ 64 Kbyte) writes into several smaller ones, so the file system may as well decompose such large writes itself, so that it can allocate disk resources appropriately.

Since the superblock location is fixed, the file system allocates the write immediately before the superblock so that it will finish several blocks before the rotational position of the superblock, thus allowing time for the superblock write to be issued before it passes the disk heads. Similarly, the write before that one is allocated so that it finishes several blocks before the rotational position of the start of the next write, and so on. Consequently, the rotational latencies are such that they just cover the time taken to issue the following write command. The only rotational latency that cannot be so covered is the one before the very first write in the sequence.

If the disk involved is on the SCSI bus, as are most currently available cheap disks, the time required to issue a write command includes the time required to transfer the data into the controller's write buffer. The maximum write speed is thus given by the following equation: $Speed_{Write} = \frac{1}{\frac{1}{Speed_{Rot}} + \frac{1}{Speed_{Transfer}}}$. For the typical disk described above, this is approximately 1.4 Mbytes/sec. In practice, such a perfect match between data transfer and disk writes will not occur, and actual write rates will be somewhat less than the above figure.

5. Performance of the Prototype Implementation

To analyse the performance of the file system, a simple, single-threaded server has been implemented and measured. Clients communicate with the server via the stateless Acacia protocol [4] over TCP. The server, which runs in user-mode on the remote machine, reads incoming requests, updates the disks, and then replies to the client.

Because the server is single-threaded, it cannot overlap request input and disk updating, nor can it combine several small requests into a single larger request. Consequently, its performance will be substantially below that of a multi-threaded, kernel-mode server.

Figure 3 shows the file server's write speed as seen by the client, for one, two, and three block I/O daemons. The dotted line shows the speed for a single client, the dashed line the speed for two clients, and the solid line the speed for three clients. Write speed, which is between two and three times that of NFS, increases marginally as the request size increases from 40 to 100 Kbytes, whereas the number of clients does not significantly affect performance.

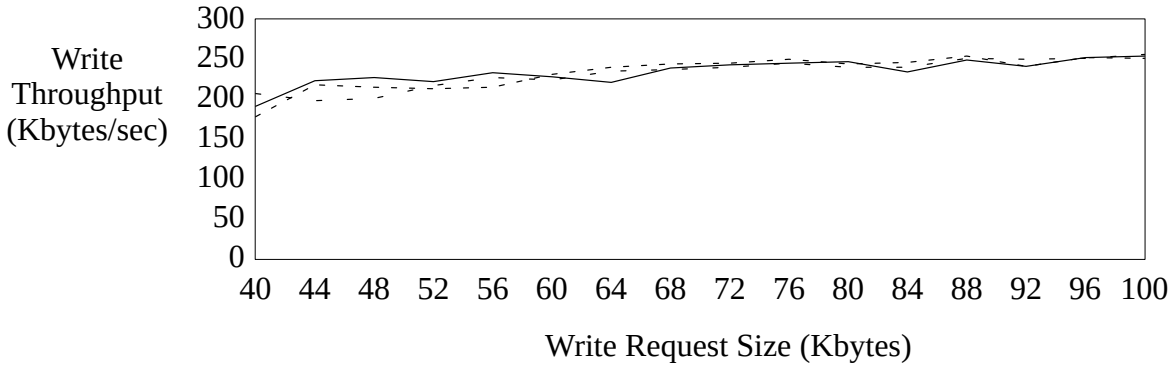

Figure 3. Average write speed seen by the client.

Figure 3 is more a reflection of the server's single-threaded implementation than a true test of the file system's design. The server reads an input request, computes the necessary disk writes, and then does those disk writes. Detailed traces show that the first two steps consume a significant portion of the server's time, and hence dominate performance. In a multi-threaded server, the first two steps can be performed while the previous request is being written to disk. Consequently, the "raw" performance of the server's disk updating algorithm was measured by recording the time immediately before and after each disk write sequence. The results are shown in Figure 4.

Figure 4. Average Raw File System Performance (user data).

This graph is a better guide to the ultimate file system performance achievable by a multi-threaded server. It shows write performance increasing from 550 Kbytes/sec for 40 Kbyte writes to 850 Kbytes/sec for 100 Kbyte writes. The latter is comparable to the speed of existing local file systems. In practice, such speeds might not be achievable using existing local area networks such as ethernet.

These speeds, however, are still substantially below the maximum write speed for the disk concerned (1.4 Mbyte/sec), especially for the smaller writes in the figure. The low performance for the smaller writes is due to the larger proportion of the disk's write bandwidth being used for file system meta-data. Figure 5 shows the speed at which all data, user and meta-data, is written to disk. While the write bandwidth for smaller writes is still below that of the larger writes, the difference is not so great. In a multi-threaded server, several small write requests could be combined

into a single update, with substantial savings in the amount of meta-data written. Also, the current implementation stores user-data and meta-data in the same size blocks, 4 Kbytes, requiring substantially more meta-data to be written than is strictly necessary. Reducing the block size or being more sophisticated with the meta-data could reduce this overhead.

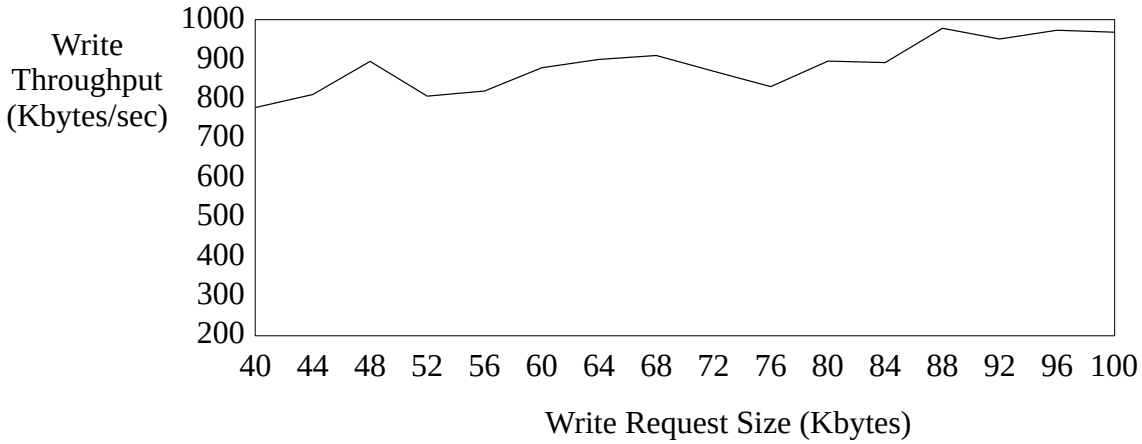

Figure 5. Average Raw File System Write Performance (all data).

There are several reasons why overall write performance falls below the disk's maximum write rate. Firstly, the first write in each write sequence occurs at a random rotational location, which results in a random latency of up to 16.7 msec before the write sequence actually commences. The effect of this latency will be greater for smaller write requests. Figure 6 shows the distribution of write speeds for individual write requests (for sizes from 40 Kbytes to 100 Kbytes). Part of the distribution is caused by random rotational latencies.

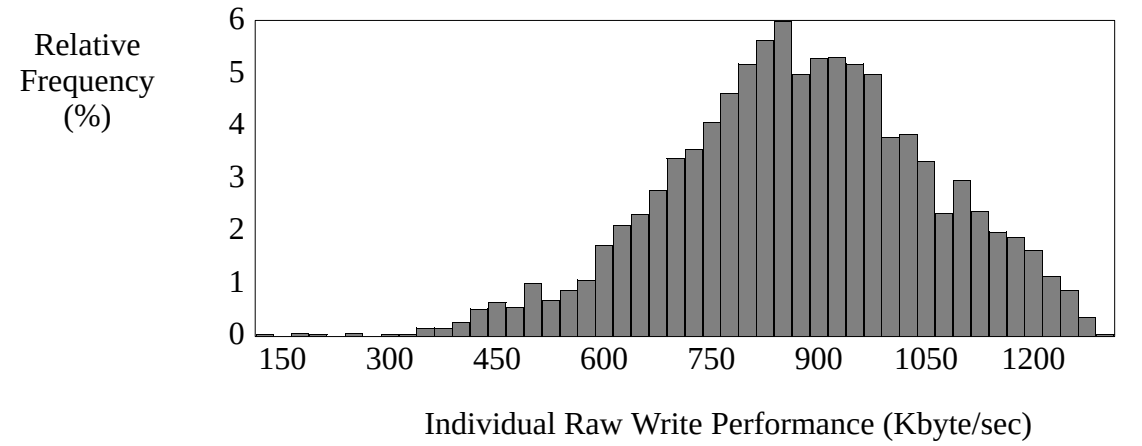

Figure 6. Histogram of Raw File System Write Performance (all data).

Secondly, the server does not separate writes by the optimum rotational amount, because scheduling delays of the user-mode server would often result in a write being missed, and the server would have to wait an additional full rotation time for the write to complete. Consequently, some leeway has been built in to the allocation of write blocks. In a kernel-mode implementation, more nearly optimal write locations could be allocated. In any case, however, there will always be some mismatch between transfer delays and the rotation time of an integral number of blocks, so that performance equal to the theoretical maximum should not be expected. Also, occasional seeks

between adjacent cylinders are required, which will add approximately 4 msec (the minimum seek time) to the required write time.

Thirdly, the algorithm currently used to allocate the write blocks at optimal (in the above sense) rotational positions is very simple, and thus computationally expensive. When the algorithm cannot determine the optimal write sequence in a reasonable time, it aborts the computation and allocates the required disk blocks using the simplest possible method, which usually results in a write sequence with long internal latencies. These poor sequences would not occur as frequently if better allocation algorithms could be developed.

Fourthly, the precise geometry of a SCSI disk is not known. Operating systems such as SunOS maintain a disk label that includes quantities called the number of cylinders, the number of heads, and the number of sectors per track. However, for SCSI disks these figures need not have any relation to the actual disk geometry, except that the total number of sectors implied by those figures is at most the actual number of sectors on the disk. Also, the SCSI controller hides surface defects from the host computer, remapping logical sectors to functional areas of the disk. Although these actions make the device much simpler to use, it also makes performance optimisation more difficult.

Finally, the user-mode server is subject to scheduling delays beyond its control, the other partitions on the disk may be referenced by other processes, causing the heads to move away from the current cylinder, and other traffic on the SCSI bus may delay the server's operations. All of these contribute very large random delays to the file server's performance. Disks reserved in their entirety for use by Acacia, and a kernel-mode implementation would reduce these delays.

6. Conclusions and Further Research

The Acacia synchronous file server has demonstrated that distributed file servers using stateless protocols can achieve the same write performance as existing local file systems, and significantly greater performance than existing distributed file systems such as NFS. Moreover, the file system need not seek while updating the disk, so the file system's performance will scale as disk transfer speeds increase, and as multiple disks are connected in parallel, as in RAID systems.

The prototype server is currently being ported to the AP1000 multi-computer [3], where it will control SCSI disks attached to individual processing nodes within that machine. This port will effectively be a kernel-mode implementation, and a multi-threaded implementation is being developed. Using the high bandwidth communications network inside the AP1000, this implementation will be able to verify the claims made here.

The algorithm used by the current server to allocate write blocks within a cylinder is very poor. It is computationally expensive, and does not always produce a reasonable result within a reason-

able time. Better algorithms for finding appropriate sequences of write blocks would decrease the processor requirements of the file system and improve the average server speed.

As with log-structured file systems, the synchronous file system requires a "good" distribution of free space within the disk to achieve high performance, although the requirements are somewhat different and a little less stringent. Rosenblum and Ousterhout [8] have done a detailed analysis of cleaning algorithms for log-structured file systems; a similar analysis for synchronous file systems is required.

Finally, as processor speeds continue to outstrip disk speeds, multi-disk file systems will become more common. The optimum file system organisation for such disks needs to be investigated. Particular issues include the number and placement of reserved superblock locations, and the appropriate use of different disks for combinations of reading and writing.

References

[1] Sun Microsystems, Inc., "NFS: Network File System Protocol Specification", *RFC 1094*, Network Information Center, SRI International, Mar. 1989.

[2] M. K. McKusick, "A Fast File System for Unix", *Transactions on Computer Systems*, 2(3), 181–197, 1984.

[3] H. Ishihata, T. Horie, S. Inano, T. Shimizu, and S. Kato, "CAP–II Architecture", in *Proc. First Fujitsu-ANU CAP Workshop* (Kawasaki, Japan), Nov. 1990.

[4] B. Broom and R. Cohen, "Acacia: A Distributed, Parallel File System for the CAP–II", in *Proc. First Fujitsu-ANU CAP Workshop* (Kawasaki, Japan), Nov. 1990.

[5] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, "Measurements of a Distributed File System", in *Proc. Thirteenth ACM Symposium on Operating Systems Principles* (Pacific Grove, CA, USA), Oct. 1991.

[6] D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *ACM SIGMOD 88*, 109–116, Jun. 1988.

[7] J. Ousterhout and F. Douglis, "Beating the I/O Bottleneck: The Case for Log-Structured File Systems", *Operating Systems Review*, Jan. 1989.

[8] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System", in *Proc. Thirteenth ACM Symposium on Operating Systems Principles* (Pacific Grove, CA, USA), Oct. 1991.

[9] S. Carson and S. Setia, "Optimal Write Batch Size in Log-Structured File Systems", in *Proc. Usenix File Systems Workshop* (Ann Arbor, MI, USA), 79–91, May 1992.