CITI Technical Report 92–7

AFS Write Performance - A Campaign Paper

Sarr Blumson
sarr@citi.umich.edu

ABSTRACT

AFS writes are slow. Part of the reason they appear slow is that opportunities to overlap writing with computation are often not used. This paper describes some of these lost opportunities, and advocates (with some gestures toward objectivity) changes to the AFS cache manager to take advantage of them.

December 10, 1992

Center for Information Technology Integration
University of Michigan
519 West William Street
Ann Arbor, MI 48103-4943

AFS Write Performance - A Campaign Paper

Sarr Blumson

December 10, 1992

1. Introduction

One of the reasons that AFS writes appear to be slow is that, absent unusual occurrences such as revoking callbacks and cache overflows, there is no data transfer between the client and server machines until the file is closed. Opportunities for overlapping the transfer with the generation of the data by the users' application are thus lost.

We begin by describing some characteristics of the University of Michigan Institutional File System that increase the visibility of this problem, and then go on to characterize it in a little more detail. We then describe a possible solution and some of the reasons why this might or might not be a good idea.

2. IFS Overview and Intermediate Servers

The Institutional File Server (IFS) Project[1] is a joint effort of the University of Michigan and IBM to provide a campus-wide integrated file service for the University of Michigan community. This community is both large and diverse; there are thousands of potential client workstations of many different types with variations in storage and processing power of several orders of magnitude. The base vehicle for this service is AFS.

As we hope to serve a large campus from a small number of geographically centralized servers, our architecture includes Intermediate Servers[2] that reside between clients and the central servers and provide an intermediate level of caching. This is particularly important where the clients are small machines (e.g., Macintosh[1] or MS-DOS[2] machines), which cannot support the large local caches normally used by AFS clients. In addition, the limited capacity of these machines (as well as the desire to overcome user resistance by limiting the amount of special software that is required to use our services) has caused us to provide a translation service to the "native" protocols of the more popular machines (e.g., AppleTalk Filing Protocol for the Macintosh) in these intermediates. A sketch of this architecture is shown in Figure 1.

1. Macintosh is a trademark of Apple Computer.
2. MS-DOS is a trademark of Microsoft Corporation.

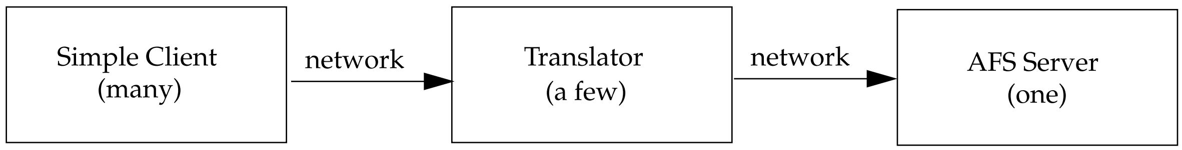

Figure 1. Architecture

A matter of concern to our potential users, and therefore to us, is that the IFS offer performance that is competitive with the alternatives that are currently available. While the IFS offers unique opportunities, such as potentially world-wide location independence, initially users do the same things they did before. Their first impression, then, is how well they can still do them.

3. The Problem

The initial impetus for this work came from a colleague at CITI who was using Macintosh tools to manipulate photographic images stored in IFS[3]. He reported that while retrieving his files was reasonable, writing them back was not.

Investigation of the causes of this eventually led to an experiment writing a very large (100 megabyte) file from a Macintosh to an IFS server. This took 3276 seconds. More interestingly from the point of view of the Macintosh, writing the file took only 1550 seconds, but closing it took 1726 seconds.

The reason for this appears to be that the intermediate/translator holds the entire file in its cache until close time, and then sends the entire file to the primary server. The data takes two hops over the underlying network; these two hops are not overlapped at all, as illustrated in Figure 2.

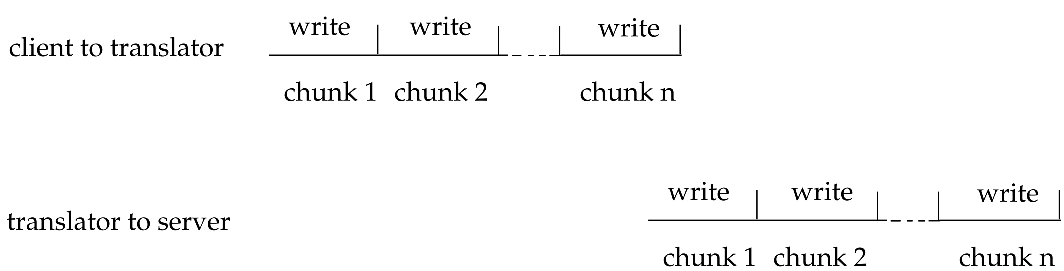

Figure 2. Non-overlapped Transfer

It is at least possible to perform this operation so that each chunk is transmitted to the primary server as soon as it is complete. This would overlap most of the transfer, as shown in Figure 3, and reduce the total time for the 100 megabyte (1600 chunk) write and close to1726 + (1550 / 1600) = 1727 seconds, a substantial improvement.

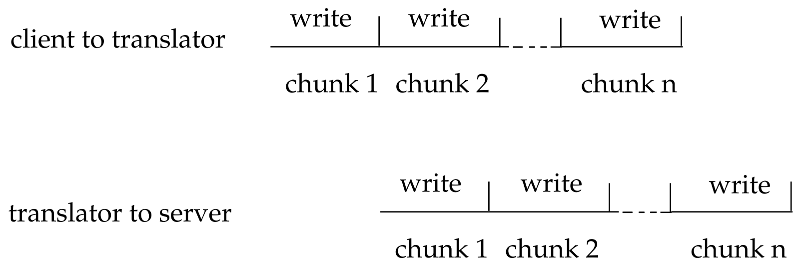

Figure 3. Overlapped Transfer

While the use of intermediate servers makes the lack of overlap more apparent, it is a general issue for the AFS cache manager. This behavior will be typical of many distributed applications in which a client communicates with a server, which in turn makes use of some distributed file service. In addition, there are many more conventional situations that can lead to similar aggravation. Any UNIX[3] application that is normally "compute bound," but which writes large amounts of output, will develop a previously unseen I/O bound phase when the output is written to an AFS server. This new phase will be particularly unsettling to users as it occurs at a point where most applications have announced that they are finished.

In this paper we are primarily concerned with a single user view of file system performance. From a system wide point of view, there are other effects of a flush on close policy that are more difficult to describe and quantify. For example, closing a large file will result in a large burst of write activity at the server. Carson and Setia[4] have shown that this batching of write traffic will often degrade other file system activity.

4. The Solution

Writing a chunk "when it is complete" is not very well defined. However, it is established gospel[5] that most write activity is sequential. In this case, it is easy to determine when a chunk is complete: it is complete when the last byte is written. We therefore propose the following algorithm:

- If two successive client write operations are sequential and cross a chunk boundary, flush the first chunk now.
- Flush any remaining dirty chunks when the file is closed.

This algorithm involves a very simple local heuristic for sequentiality, combined with a simple check for completeness that depends on the assumption of sequentiality.

5. Other Approaches

The most obvious alternative solutions involve some form of periodic flush of dirty cache pages. This is often done for system integrity reasons, as in the UNIX buffer cache, and is, in fact, done in AFS4 for exactly that reason. While these techniques reduce the dramatic effects described in sec-

3. UNIX is a trademark of Unix System Laboratories.

tion 3, the problem remains for any application whose running time is less than the flush period. These periods are typically on the order of thirty seconds. Making a program that would run for twenty seconds take forty seconds is not as dramatic as our original example, but it is still serious. Programs of this order are much more common. In addition, these methods still tend to batch write operations, with possible negative results for system performance as a whole[4]. In addition, this approach is totally unsynchronized with the writing program, increasing the probability of interference. Shortening the flush period in order to reduce batching effects would only make this worse.

Another approach would be to do the close itself in the background, rather than making the user wait. This would eliminate the apparent delay, but at a cost we believe would be much greater. The data transfer *can* fail, and this would eliminate any opportunity to inform the application of the failure. It would also break a guarantee that AFS currently makes to cooperating programs, that they can depend on a write being complete after the close.

6. Problems With the Solution

6.1 AFS Write Semantics

This change would make some subtle changes in the write behavior of AFS. In particular, data that currently does not appear at the server until the file is closed will begin appearing gradually over the life of the user program. The question is whether this matters.

It has been suggested[6] that AFS guarantees, or at least attempts to guarantee, all or nothing semantics on file updates. We believe that this is not truly an issue for several reasons.

- Our review of the AFS literature[7],[8] suggests that delaying stores until close was derived from the original whole file caching scheme, and was always viewed as a performance and scalability compromise, which did not seriously **break** the desired UNIX semantics, rather than a goal.
- This guarantee is already seriously broken in AFS3. It is clearly impossible to maintain while writing files that are larger than the local cache. In general, AFS3 will flush the dirty chunks of a file in response to cache overflows. Even at close, the new data is written to the server as a series of independent, chunk-sized operations that can interleave with reads by other clients.
- The guarantee is completely broken by the periodic cache flush in AFS4.
- The current behavior compromises another AFS goal: location independence. Currently a reading process on the same host as the writer sees different data than a reader on another host. This is a reasonable performance compromise for the reasons described by Kazar[8], but inconsistent behavior is not a plausible goal for its own sake.

None of this should be interpreted as meaning that atomic update semantics is not a reasonable goal in itself. Quite the contrary. Implementing it, however, requires an architecture very different from AFS, which would guarantee that updates are always atomic and appear so to all potential readers. Such an implementation would be independent of the low level issues of when to physically transfer the data that we are addressing here.

6.2 Program Behavior

All system designers know that programs are malevolent. The heuristic proposed here is only a heuristic, and cannot guarantee that the program will not immediately rewrite the chunk we have just decided to transmit. At the least, this breaks one benefit of the AFS3 scheme, minimizing the amount of data transferred to the server, because this chunk will now be transmitted twice.

A more subtle problem is the race between transmitting the chunk and modifying it that arises in this scenario. We suggest, however, that this is a "don't care." The final version will eventually arrive at the server (in the absence of failures). The underlying semantics are already so variable (e.g., a local reader could have seen the intermediate value, even if it were never written to the server) that the temporary presence of intermediate values on the server doesn't matter.

In any case, we claim (without any evidence) that this sort of program behavior, while clearly not impossible, is unlikely. If desired, the probability of a false determination of sequentiality can be made smaller by requiring a larger number of sequential writes, rather than the two we have suggested.

6.3 Implementation Details

Currently the actual transfer of the chunk is done by the thread triggering the operation, typically the user process doing the close. For the transfer to be done asynchronously, it would have to be queued to be performed by the AFS daemon.

Currently the dcache entry and the corresponding vnode are locked while a chunk is being transmitted to the server. It is not clear to us why or whether this is necessary, or what effort would be required to change it. If it cannot be changed, there is some potential for the user process to be blocked when it attempts the next write to the following chunk. While this does have the advantage of removing timing dependencies on which data is written when a chunk is unexpectedly rewritten, as discussed above, it is certainly less than ideal. However:

- At worst, this is better than the present case. Part of the transmission wait has been moved from the close operation to a write operation, but it has not been increased. Any computing that the user process did before the write (and there must have been some!) is an improvement.
- The periodic flushes done by AFS4 face the same problem. While they probably occur less often (although this depends on the user program), they will, in general, take longer because they will send more chunks. In addition, the scheme proposed here will be better synchronized with user programs because transmission will be provoked by a write, and the full interval between writes will be overlapped.

An important concern is the vital importance of keeping the code paths added by AFS short[9]. The additional tests we are proposing are trivial, however, and should not add significantly.

7. Conclusions

The write performance of AFS can be greatly improved by some simple changes that will change some behavior that is largely a vestige of earlier, "whole file caching" versions.

As an incidental benefit, the problems that are caused by the many UNIX utilities, and other user programs, that do not check the result of close operations will be reduced. Many problems that now occur only on close will begin to occur on the preceding write instead.

Last, we do not believe that the periodic stores done by AFS4 provide an adequate remedy to the problems we raise here. As discussed earlier, they are poorly synchronized with user program activity, in a way that is equally likely to result in a chunk being written too often (when the file is being written slowly) or to be of no effect at all (when the file is written and closed within one thirty second interval).

Acknowledgments

Many colleagues at the University of Michigan played major roles in the development of these ideas. Marcus Watts performed the original write experiment described in section 3. The issues around AFS write semantics were developed during some energetic conversations with Peter Honeyman. Mike Stolarchuk was a great help in clarifying the actual behavior of AFS. Lyle Seaman of Transarc also made a number of useful suggestions.

References

1. T. Hanss, "University of Michigan Institutional File System," /AIXTRA: The AIX Technical Review, pp. 25-32, (January 1992).
2. James Howe, "Intermediate File Servers in a Distributed File System Environment," CITI Technical Report 92-4, (June 30, 1992).
3. Wafik Farag and Fred Remley, "Digitized Image Data Compression and Transfer," CITI Technical Report 92-5, (October 30, 1992).
4. Scott D. Carson and Sanjeev Setia, "Analysis of the Periodic Update Write Policy For Disk Cache," IEEE Transactions on Software Engineering, vol. 18, no. 1, pp. 44-54, (January 1992.)
5. John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," Proceedings of the 10th Symposium on Operating System Principles, pp. 15-24, (December 1985).
6. Peter Honeyman, Personal Communication, (November 19, 1992.)
7. J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M. West, "Scale and Performance in Distributed File Systems," ACM Transactions on Computer Systems, vol. 6, no. 1, pp. 51-81, (February, 1988).
8. M.L. Kazar, "Synchronization and Caching Issues in the Andrew File System," Technical Report CMU-ITC-058, (June 1987).
9. Michael T. Stolarchuk, "Faster AFS," CITI Technical Report 92-3, (June 22, 1992).