CITI Technical Report 92-3

Faster AFS

Michael T. Stolarchuk
mts@citi.umich.edu

ABSTRACT

The AFS Cache Manager fetches files from the AFS file server, and caches them into a local file system. Given this model, users expect reads of locally cached files to perform at local file system rates. However, read performance of the AFS cached files is half the read performance of the local file system. This paper discusses the reasons for the large performance difference, and the modifications made to AFS so that reads of locally cached files perform within 10% of the performance of the local file system.

June 22, 1992

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

Faster AFS

Michael T. Stolarchuk

June 22, 1992

The AFS File System [1] is a distributed file system based on the client/server model. The file servers only perform file service; the clients are workstations that run additional code to network with AFS file servers. These AFS clients also have additional responsibilities; they provide caching, a central role of AFS clients. Because files are mostly read, caching the files locally enables most user read requests to be processed without using network resources.

The AFS Cache Manager keeps track of all the files cached. The cache is kept on local disk and is nonvolatile. The AFS Cache Manager also keeps a nonvolatile index of the cached files. The AFS Cache Manager uses a local file system to store the contents of the cached AFS files. Later read requests are subcontracted to the local file system.

Users then expect the read performance of cached files to be similar to the read performance of the local file system. When measured, however, the performance of the AFS 3.1 Cache Manager is about half that of the local file system. Some AFS sites have actually moved binaries to the local disk drives, away from the distributed file system, due to these performance differences.

At the Center for Information Technology Integration (CITI), we examined the read performance of the AFS 3.1 Cache Manager, attempting to improve the overall performance of the client environment.

The body of this paper is divided into three main sections. The first section discusses performance issues of the AFS 3.1 Cache Manager, the second section describes the implementation of the AFS Cache Manager and the third section describes the modifications we have made to the AFS Cache Manager, along with their performance impact.

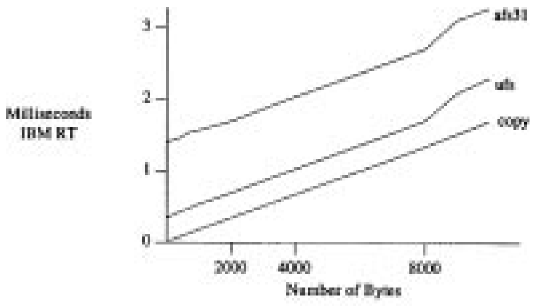
1. Performance Measurements

1.1 Benchmark

We used a very simple benchmark to compare read times. The benchmark reads bytes from the beginning of the file. It opens a file once, then performs reads. The benchmark can be directed to read any file, allowing measurement of both AFS and the local file system. The benchmark reads the same block in each iteration. That block is kept in memory; no disk I/O is performed.

Measurements

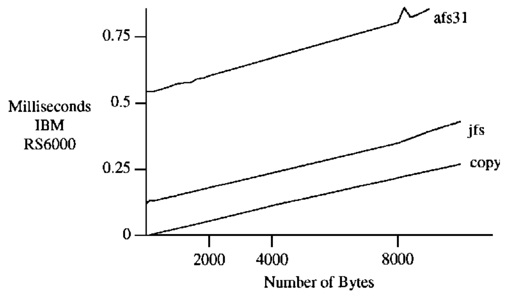
We measured the times for two different machines, both running the AFS 3.1 Cache Manager. Figure 1 describes an IBM RT, about 2 MIPS. Figure 2 describes an IBM RS/6000 520, about 20 Mhz.



IBM RT			
Bytes	copy	ufs	afs31
100	0.028	0.336	1.056
1000	0.182	0.354	1.031
2000	0.347	0.349	0.997
4000	0.675	0.350	1.010
8000	1.342	0.365	1.001

Figure 1. IBM RT Benchmark Performance Measurement

The graph in Figure 1 above shows measurements of the time in milliseconds for the AFS 3.1 (afs31) read, the Berkeley Fast File (ufs) read, and memory to memory copies (copy) on the IBM RT. The table lists the time spent in milliseconds to copy the data from the read operation and describes the overhead for the other components by listing the additional time spent above the copy. The 'ufs' column describes the time spent performing local file system operations, and the 'afs31' column describes the time spent within the AFS 3.1 Cache Manager.



IBM RS/6000 520			
Bytes	copy	jfs	afs31
100	0.004	0.128	0.409
1000	0.028	0.124	0.421
2000	0.055	0.124	0.424
4000	0.112	0.122	0.435
8000	0.215	0.131	0.456

FIGURE 2. Figure 2. IBMRS/6000 benchmark performance measurement

The graph in Figure 2 shows measurements of AFS 3.1 (afs31) read, the AIX 3.1 Journaling File System (jfs) read, and memory to memory copies (copy) on the IBM RS/6000 520.

The table lists the time spent in milliseconds to copy the data from the read operation and describes the overhead for the other components by listing the additional time spent above the copy. The 'jfs' column describes the time spent performing local file system operations, and the 'afs31'column describes the time spent within the AFS 3.1 Cache Manager.

1.2 Discussion

To gain the benefit of a distributed file system, users of the AFS Cache Manager are willing to sacrifice performance somewhat. If the performance loss becomes large, the benefits of a distributed file system come into question. Our own goal is 10% overhead; we would be willing to pay 10% above the time spent processing the read request to the local file system. After all, once the file is on the local disk, the AFS Cache Manager should be able to access the data quickly.

That 10% overhead is a relative value. When the underlying file system uses significant processing resources, the AFS Cache Manager can also use significant resources. In both previous figures, the underlying file system uses little processing resources. From the table, the overhead of both the local file system and the AFS Cache Manager is static; the time stays relatively constant over the range of reads. On the RT, the static overhead to read one block from the local file system is equal to the time to move 2K of data. The static overhead of the AFS Cache Manager to process a read is three times larger.

The benchmarks measure the overhead, the processing time to complete the read in addition to the memory to memory copy. We ignored the I/O delays since the benchmark's times correlated with performance differences reported from some applications. Additionally, we have AFS Cache Managers which use main memory, not disks, as the cache. When the memory-based AFS Cache Managers exhibited the same performance loss (relative to the local file system), we decided to investigate the AFS Cache Manager.

The overhead depends on the size of the read requests typically issued by user processes. According to [2] 70% of user processes typically operate requesting 4K or less. Because we believe most of the I/O from user processes comes from the standard I/O library, and because standard I/O determines its block size from the file system, we need to be aware of the block size of the underlying file system.

The IBM RT's local file system has a block size of 8K, while the IBM AIX's file system has a block size of 4K. Typical user processes on the RT will make 8K read requests, while the IBM RS/6000's AIX will make 4K requests. The distribution of the sizes of read requests determines the perceived performance loss.

The performance loss is due to the length of the code path. The number of instructions the AFS Cache Manager needs to issue to meet its requirements is large compared to the underlying file system. The AFS Cache Manager has requirements that are not clearly outlined; it is using the code path to meet these requirements. By describing these requirements, and understanding the size of the underlying file system, we can make some significant performance improvements.

2. The AFS 3.1 Cache Manager Read Requirements

To understand why the AFS Cache Manager was using more processing time than the underlying file system, we reverse engineered the code of the read operation to determine its requirements. We then studied how the AFS Cache Manager implements each of the requirements, with the goal of reimplementing significant portions of the read procedure for higher performance.

2.1 Cache Consistency

The data in the cached file must represent up-to-date information. The AFS Cache Manager uses a lazy policy to determine if the AFS file is out of date. Before any data associated with an AFS file is referenced, the file is checked for cache consistency. For example, early in the read operation, the AFS Cache Manager tests to determine whether the file is up to date. The test is straightforward and involves several different comparisons. If the file is from a read-only volume, for example, it is assumed to be up to date.

If the file is within a read-write volume, then it is consistent if a "callback promise" exists. A callback is a promise made by the file server to inform the client if a file's status changes. Callbacks in AFS 3.1 have limited duration, depending on the number of concurrent users of the AFS file. The duration is currently quantized, with a maximum duration of 4 hours, for 0 to 7 users, and a minimum of 7 minutes, for over 64 users.

2.2 Chunk Location

The AFS Cache Manager manages every AFS file as chunks in the local cache. Files that do not fit into a chunk are broken into multiple chunks. Chunks are fixed in size and implemented as a file in the local file system. Only the chunks currently referenced by the application need to be in the cache. This means there are many chunks for one large AFS file, implying a mapping from an AFS file and offset into a chunk.

In the AFS Cache Manager, that mapping is performed through a hashed list of file identifiers. The AFS file is identified by a set of numbers, the File Identifier (FID), which consists of cell number, volume number, vnode number, and a "uniquifier."

2.3 Chunk Isolation

Although an AFS file is managed in chunks, the user process is isolated from the implementation of chunks. If the user process requests data from an AFS file, and the request spans several different chunks, the read code must break up the original request into several smaller requests, each completely satisfied from one chunk.

The vnode interface [3] of the local file system reads and writes chunks, allowing the AFS Cache Manager to be relatively portable. This implementation strategy allows us to determine the overhead of the AFS Cache Manager reads, by comparing the performance of the local file system and the performance of AFS reads.

2.4 Early Return

If a chunk is not within the local cache, the read procedure must request its contents from the AFS file server. If the user process requests a small number of bytes at the front of a file, read returns to the user process when part of the chunk is filled.

The AFS Cache Manager keeps track of the highest byte retrieved from a file server for a chunk. A flag in the chunk indicates when the chunk is actively being fetched. After the read locates the related chunk, it checks to see if the data is currently being fetched. If it is, then the read waits until the desired data is received.

This implementation is straightforward, except that the user process waits for the chunk to fill. Some other process must be filling the chunk. AFS typically configures two background processes during early system initialization to perform such activities. If a chunk needs to be fetched, the AFS Cache Manager has code to attempt to perform the fetch through the background processes, with the hope that the actual reading process can return early.

2.5 Prefetching

The AFS Cache Manager tries to hide some network and server latency by enqueuing fetch requests for the next chunks of a file. When the read is nearly completed, the background daemon receives a request to fill the next chunk.

3. A Faster Implementation

Adding code to create a frequently executed path, improves performance dramatically without making a significant investment in new code. Adding significant new code to the AFS Cache Manager is expensive because the AFS Cache Manager is routinely ported to many different platforms. Additionally, significant new code would need to be tested on many of those different platforms. Some of the current read code exists to deal with differences discovered the “hard way” from such porting efforts. Our faster implementation should incorporate the benefits of those experiences.

Our goal is to improve client performance. Our stated goal is 10% overhead, or 10% of the time to read an 8K block out of the disk cache. In the RT case, the 8K can be read in 1.7 milliseconds. We need to perform all of the AFS code in.17 milliseconds. This is empirically the time to perform a 1K move, and therefore represents some 500 memory cycles. A benchmark aids in determining how many cycles we can spare.

We decided to construct a test to find the shortest path to the local file system through the AFS read procedure. The results act as an upper bound for performance, and give us a mechanism for exploring our performance goal. We place a call to the local file system as the first executable statement within the AFS read procedure. With a benchmark performing 200 byte reads (very short reads), the test was already performing at 10% overhead. To reach our goal of 10%, we had to code the solution to call the local file system as the first executable statement within the AFS read procedure.

This test provided the implementation skeleton, we had to meet each of the additional read requirements, using little or no additional code.

3.1 Meeting The Requirements

To meet our performance objective, the short path needs to become the commonly executed path through the read procedure. In the faster implementation, we first test some conditions to determine whether we can execute the short path to the local file system. If the conditions fail, we use the long path through the read procedure.

The conditions that determine whether to use the short path are a hint [4]. The hint must be rich enough to allow the short path to execute frequently. The hint must also meet the requirements of the read procedure described previously. Because the long path is still available, we need to only implement those requirements that will help meet our performance goal.

The hint is populated during the long path through the read procedure. On the next read call, the hint tests to see if it can use the short path.

The hint prejudices the performance of the read code. For particular kinds of user code behaviors, the read procedure now provides better performance.

3.1.1 Cache Consistency

In AFS 3.1, the code must constantly check to ensure the cached file is up-to-date. The callback includes a timeout value that is compared with the current time. When the time out is passed, the callback expires. Because a callback is only tested when it is necessary to check the validity of a file, this is a lazy policy. There is no central management of all the callbacks of the entire pool of cached files.

In some early performance analysis of AFS 3.1, we determined the ratio of AFS system calls to callback validity tests: 7 callback validity tests for each AFS system call. To determine how often these validity tests were performing valuable work, we needed additional insight into the distribution of expiration times.

We extracted callback timeout values from the AFS Cache Manager. Most values expired far in the future. Many close entries were usually several minutes from timing out. We performed a very limited study of the callback expirations. A few local workstations used for development were studied. Due to the bursty activity of the machines studied, callback timeouts were clustered around many different times. However, most of the callbacks would not expire for at least several minutes.

Because most timeouts would expire far into the future, very few of the validity tests made repeatedly by the AFS client were performing valuable work. This situation suggested that the expiration test should be performed using some other policy. Therefore, we reimplemented cache consistency to manage the callbacks actively.

We use a doubly linked list sorted by timeout to collect the callback promises. Once a second, we test the top element to determine whether its callback should expire. If so, we modify the associated file to reflect the expiration. If the server delivers its callback promise to the client, then the callback of the file expired, and it is removed from the list.

For any AFS file protocol request that returns a callback, the timeout is computed, and the vcache entry is sorted on the callback-expiration list. We currently search for the correct insertion point by starting at the end of the linked list (furthest into the future) and then move towards the beginning of the list (towards current time), on the assumption that returned callback timeouts tend to be distant events rather than immediate events.

3.1.2 Chunk Location

The AFS Cache Manager searches for the chunk associated with the file request at each read. The file and offset request are mapped to a chunk reference. Even though the chunk entries are hashed, the search is expensive. To keep from searching the list at each read, the long path saves the last chunk referenced as part of the hint. That chunk is typically 64K large. A relatively large number of sequential reads can be satisfied by that one chunk. The chunk size is configured at AFS Cache Manager initialization; if there isn't enough locality of reference in 64K chunks, the chunk size can be increased.

The short path needs to check if the hint is describing the file currently being read. We could compare the file identifier of the file being read and the chunk. If they match, the hint is describing the correct file. We can, however, construct a much simpler test. The structure describing the AFS file and the chunk can be stamped with a 32 bit value. If the two values match, the file ids are considered equal. The 32 bit value is a monotonically increasing number, incremented once for every tuple we want to relate. The stamp is computed once for the file/chunk pair, and the file and chunk structures are stamped with the same value. The stamp can then test for the file match in one comparison.

The short path also needs to ensure that the read request is requesting this particular chunk. Chunks are commonly described by chunk numbers, while read operations request offsets. To make the test in the short path simple, when we save the chunk reference we will also compute the offset of the chunk in the file and make it part of the hint.

3.1.3 Chunk Isolation

The short path is used only when the request is totally contained within one chunk. By testing to see if the user's request can be satisfied within one chunk, we don't have to be concerned about providing support for isolation directly. We use the hint when the user's request is within one chunk, and ignore it otherwise. As mentioned earlier, if we find too many requests processed by the long path, we simply increase the chunk size. The chunk size can be modified only at system startup during AFS Cache Manager initialization.

No additional code is included in the short path to process reads that cross chunk boundaries. Instead, the short code depends on the existing long code path to process long read requests. This method allows the short code path to focus on providing performance for typical applications, while still correctly processing large read requests. Additionally, the larger read requests can tolerate longer overhead, due to the time spent processing the request in the local file system. We wouldn't realize the same significant performance gains by decreasing overhead on the large read requests.

To perform the test quickly in the short path, we need to test the bounds of the chunk against the user request. When the request is completely contained within the chunk, the local file system can service the request directly. When we save the chunk reference in the long path, we also compute the bounds of the chunk as offsets from zero. The read request uses the same units, making the comparison to use the short path simple.

3.1.4 Early Return

The hint cannot be populated until the chunk is completely filled. If the AFS Cache Manager sends a read request contained within one chunk to the local file system layer while the chunk is still being filled, the read could return the data from the partially filled chunk. But the read would return without satisfying the entire request. The user application is unlikely to have the additional code to retry for additional data intended to be within that chunk. To preclude this event, we can't use the short path for chunks which are currently requested from the file server. We implement this by not populating the hint until the chunk is completely filled.

3.1.5 Prefetching

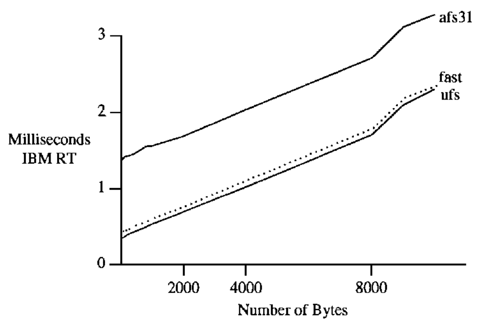
We currently have no additional code to support prefetch. We depend on the existing code in AFS 3.1 to perform some prefetch of chunks.

3.2 Measurements

The performance of the Faster AFS modifications appears in figures 3 and 4. These figures represent approximately 9% overhead for the IBM RT, and 15% overhead for the IBM RS/6000 for 4K character reads.

The IBM RS/6000 incurs a larger overhead for the fast reads than does the IBM RT. We are unsure why the overhead for the RS/6000 is larger. It may be due to the longer time to perform indirect subroutine calls. The IBM

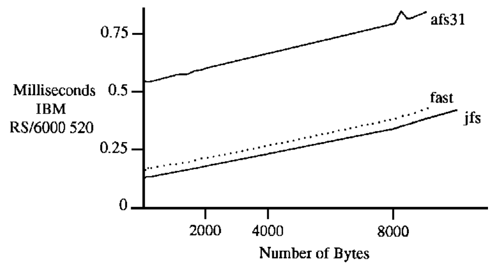
RS/6000 has housekeeping to perform, which requires about 10 instructions for indirect function calls. This housekeeping may also help explain some of the larger overhead values for AFS 3.1 Cache Manager.



			IBM RT		
Bytes	ufs	afs31	overhead	fast	overhead
100	0.364	1.056	290%	0.083	22.8%
1000	0.536	1.031	190%	0.069	12.8%
2000	0.696	0.997	140%	0.068	9.7%
4000	1.025	1.010	98%	0.089	8.6%
8000	1.707	1.001	58%	0.083	4.8%

Figure 3. Figure 3.IBM RT Faster AFS Performance

The graph in the figure above shows performance measurements of the IBM RT for the Berkeley Fast File system (ufs), the AFS 3.1 Cache Manager (afs31) and the short path through the AFS 3.1 Cache Manager (fast). The table compares the read performance, measured in milliseconds, of these three implementations. The table also displays overhead for 'afs31' and 'fast'. Overhead is computed by dividing the observed AFS Cache Manager performance by the local file system performance.



			IBM RS/6000 520		
Bytes	jfs	afs31	overhead	fast	overhead
100	0.132	0.409	309%	0.035	26.5%
1000	0.152	0.421	276%	0.034	22.3%
2000	0.179	0.42	236%	0.034	18.9%
4000	0.234	0.435	185%	0.035	14.9%
8000	0.346	0.456	131%	0.042	12.1%

Figure 4. Figure 4.IBM RS/6000 Faster AFS Performance

The graph in the figure above shows performance measurements of the IBM RS/6000 520 for the AIX 3.1 Journaling File System (jfs), the AFS 3.1 Cache Manager (afs31) and the short path through the AFS 3.1 Cache Manager (fast). The table compares the read performance, measured in milliseconds, of these three implementations. The table also displays overhead for 'afs31' and 'fast'. Overhead is computed by dividing the observed AFS Cache Manager performance by the local file system performance.

3.3 Additional Concerns

Because the hints leave vnodes open, Faster AFS can act as a resource hog. Because the number of AFS stat structure entries is limited, and because each AFS stat structure can potentially have one open vnode (as a hint), large numbers of vnodes could be left open. AFS bounds the number of vcache entries, however, and this simple mechanism keeps the number of open vnodes low. As the AFS stat structures in the AFS stat pool are reused, open vnodes (hints) are freed.

A large number of in-use vnodes can be a concern in systems with very limited statically allocated vnodes. In these situations, a pool allocator for vnode references can be used. The hint could save the pool reference, along with an ownership stamp. The short path would then also need to test ownership of the vnode reference in the pool by testing the stamp.

Currently the code surrounding the hint promotion and clearing does not lock the contents of the hint structure. We avoided locks not due to performance issues, but rather due to possible deadlock conditions resulting from the server delivering callback promises. We consider this issue open, and need to spend more time to determine a good solution.

4. Additional Work

Additional work can improve local caching, both by using faster caches, and better cache replacement policies. We considered making changes to the underlying file system to better meet the AFS Cache Manager needs. We plan to study one client with 128 MB of real storage, using a memory cache, and chunk sizes of 64 K to see the upper boundaries of performance. We have also considered using cost-based cache replacement policies, to hold on to data that is more costly to recreate.

Our immediate concern was the AFS read operation, due to the ratio of reads to writes of user programs. The same modifications are equally suited for write operations. More needs to be done to better adapt to the needs of program loading. For load-on-demand paged style text, the hinting mechanism already provides some benefit. For text-shared executables it is likely the load request will span chunks. Because the hint only works for requests within one chunk, text-shared program loading doesn't benefit from this hinting mechanism. This may not be an issue since the kernel read requests for text-shared program loads are often done for the complete contents of text (and data). The static overhead of AFS and the VFS is only incurred once.

There may be some opportunity to use the same mechanism in other parts of AFS. The AFS path lookup would seem to be a likely candidate, but AFS already incorporates an additional caching mechanism for directories.

We have code to implement multiple hints. The second hint exists to allow background daemons performing prefetch operations to populate a hint. When the user process performs a read, the second hint can be used to execute the short path. Multiple hints may not improve performance, because the additional tests are made through the longer path through the AFS read procedure.

The AFS 3.1 Cache Manager is similar to the Open Software Foundation's (OSF) Distributed File System (DFS) Cache Manager. The same changes made to improve the performance of the AFS Cache Manager will also improve the performance of the DFS Cache Manager. Cache consistency is managed differently in DFS, using a token manager to coordinate read and write access. The token manager side steps the callback issues central to this paper, using an even more active policy than this paper does.

5. Conclusions

The benefit of the hinting mechanism comes from the interaction of different parts of AFS. With typical chunk sizes of 64 K, and typical user read requests of 4 K and 8 K, it seems natural to provide a short code path to the local file system cache.

A layered service needs to be aware of the resource needs of the underlying layer. Users will expect the layered service to perform similarly to the underlying service when the services are similar. We expected reads in AFS to perform favorably when compared to the underlying file system. We knew the AFS Cache Manager service was layered above the local file system, but we believed the AFS costs were small compared to the underlying service.

The direct impact of these modifications to application level programs is unclear. It is easy to build micro-benchmarks to show the direct effect of the modifications, but characterization of the workloads of our user community could only be done for clusters of users. Even so, our concern is read performance, the most common file system operation.

6. Acknowledgements

Brian Renaud for helping me keep my nose in the paper. Peter Honeyman for pointing me at Tufte. Mike Kazar for being Mike Kazar. David Richardson for reviewing over and over again. Lyle Seaman for spending more time then I ever expected. Mary Jane Northrop for editing and editing and editing.

7. References

1. M. Satyararayanan, J.H. Howard, D. A. Nichols, R. N. Sidebotham, and A. Z. Spector, "The ITC Distributed File System: Principles and Design," *Proceedings of the 10th ACM Symposium on Operating System Principles*, 1985.

2. Songnian Zhou, Herve Da Costs, and Alan Jay Smith, "A File System Tracing Package for Berkeley Unix," pp. 407—419, in *Summer 1985 USENIX Conference Proceedings*.

3. S. R. Kleinman, "Vnodes: an Architecture for Multiple File System Types in Sun Unix," in *Summer 1986 USENIX Conference Proceedings*.

4. Butler W. Lampson, "Hints for Computer System Design," vol. 21, no. 5, pp. 33—48, in *ACM Operating Systems Review, Special Issue 1983*.