CITI Technical Report 01−3

Improving AFS Performance via Selective Caching and Native ATM AAL5

Charles J. Antonelli, Kevin Coffman, Jim Rees

{cja,kwc,rees}@citi.umich.edu
http://www.citi.umich.edu/

ABSTRACT

We describe two enhancements to the AFS distributed file system intended to improve the performance of large file fetches over high-speed networks. *Selective caching AFS* bypasses the client local disk when fetching files, which improves the performance of applications such as streaming video, for which local caching is of little benefit, and gives client processor and cache resources to applications for which local caching is an advantage. *Split-path AFS* augments the conventional UDP-based connection between AFS client and server with a "fast path" over native ATM AAL5 for file data fetches.

These enhancements combine to reduce the elapsed time of large (100 MB) file fetches by 80% in our laboratory environment.

12 February 2001

Center for Information Technology Integration
The University of Michigan
Ann Arbor, Michigan 48103-4943

Improving AFS Performance via Selective Caching and Native ATM AAL5

Charles J. Antonelli, Kevin Coffman, Jim Rees

{cja,kwc,rees}@citi.umich.edu
http://www.citi.umich.edu/

1. Introduction

We describe two enhancements to the AFS [1] wide-area distributed file system intended to improve the performance of large file fetches over high-speed networks.

Selective caching AFS bypasses the client local disk when fetching files. With caching disabled, AFS file data arriving from a file server are given immediately to the requesting application, and are never cached on the client's local disk. This improves the performance of continuous media applications (such as streaming video players) by eliminating disk latencies from the data processing pipe. This is especially advantageous when the network is faster than the local disk. As a secondary benefit, selective caching frees client processor and cache resources for use by applications for which local caching is an advantage.

Split-path AFS augments the conventional UDP-based connection between AFS client and server with a "fast path" over native ATM AAL5 for file data fetches. With split-path AFS enabled, file contents travel between file server and client over a native ATM AAL5 connection. Taking full advantage of the larger MTU available and eliminating the UDP network layer, split-path AFS greatly improves transfer rates compared to conventional IP-based networks.

These enhancements combine to reduce the elapsed time of large (100 MB) file fetches by 80% in our laboratory environment.

In the remainder of this paper, we define our goals and motivations and describe our testbed. Following a short overview of AFS, we describe our implementation on the Sun Solaris and SGI IRIX platforms, analyze our experimental results, and close with a discussion of future work.

2. Goals and Motivations

It has long been known that AFS is slow [2].

In fact, as processor, disk and network speeds continue to improve, AFS continues to get slower. When AFS was developed over ten years ago, a typical client workstation sported 3 MIPS and 60 MB of fairly slow disk storage and was connected to a 10 Mbps network [3]. In this environment, AFS is an excellent alternative to local disk storage: it provides crucial semantics such as a consistent local view of a global distributed filesystem; it provides good security via strong encryption for authentication, authorization, and access; it introduces the powerful concept of volume-based storage aggregation; it makes large amounts of centrally managed disk storage available to resource-poor workstations; and it doesn't run too slowly either.

Today's commodity platforms contain 750 Mhz processors, hundreds of GB of local disk capable of transferring data at better than 20 MBps, and are routinely connected to fast networks of 100 Mbps to 1 Gbps. Moreover, AFS has been ported to high-performance platforms such as the SGI Origin 2000 family, members of which contain up to 512 R12000 processors each running at 400 Mhz, support peak system bandwidth of 160 GBps, and in today's datacenters are connected to networks exceeding OC-12 (622 Mbps) speeds.

In such environments, AFS is generally perceived to be miserably slow. Trapped in layers of network code and filesystem implementation, AFS cannot take advantage of these higher processor and network speeds to remain a viable alternative to local disk storage. AFS does continue to provide access to data stores far larger than could be accommodated on a local disk, but the performance gap is so large that users have substantial incentive to avoid using AFS.

Our work thus reflects the following goals:

- **Performance.** Our primary goal is to improve AFS performance in today's networked environments such that AFS continues to be a viable alternative to client local disk.
- **Compatibility.** Due to the size of the installed base of AFS-capable machines, it is vital that our modifications allow an unmodified AFS client to interoperate with a modified AFS server and vice-versa.
- **Mass Storage.** Because truly large collections of data, such as HDTV images [4] or the Shoah [5]

and Clementine [6] data sets, are too large to be stored feasibly on magnetic disks, mass storage systems combining slower media of higher capacity with faster media of lower capacity are popular for such data. It is important that our modifications apply in this arena, in that they interoperate with MR-AFS, a multi-resident version of AFS designed to work with mass storage devices by Goldick et al [7].

3. AFS Overview

AFS [1] is a wide-area distributed filesystem that permits transparent access by clients to files stored on servers. Client programs access these files as though they were local, using conventional file access calls via a *cache manager* running in the kernel, accessing remote files via a user-level *file server* running on the server. Client and server platforms are connected by conventional IP networks.

Some of the innovations introduced by AFS include the implementation of a single global filesystem independent of client configuration; the provision for good security via strong encryption for authentication, authorization, and access; and the organization of filesystem data into *volumes*, which represent collections of files that are treated similarly with respect to storage location, quota management, and backup.

AFS uses a remote procedure call component called Rx for communication between file servers and cache managers [8]. Rx is a connection-oriented, streaming RPC protocol built on top of UDP/IP, on which it depends for low-level packet addressing and delivery. A client and server communicate by establishing an Rx connection between themselves; this connection may be authenticated and supports data encryption. An RPC is implemented via Rx calls made over such a connection. Rx provides reliable communication via packet ordering, elimination of duplicates, and selective retransmission of dropped packets. A windowing protocol permits bulk data to be transmitted efficiently. Data is sent over an Rx connection in *chunks*, nominally 64 KB in size. Some number of Rx packets are usually transmitted as a single *jumbogram* packet for better performance.

4. CITI Testbed

Our experimental testbed shown in Figure 1 consists of four systems: `br`, a 167 Mhz Sun Ultra 1/170 with an 83 Mhz system bus, 64 MB of memory and 2 GB of disk running Solaris 2.6; `hf`, a 167 Mhz Sun Ultra 1/170 with an 83 Mhz system bus, 256 MB of memory and 2 GB of disk running Solaris 2.6; `bullrun`, an SGI O2 with a 175 Mhz R10000, 128 MB of memory, and 2 GB of disk running IRIX 6.5.6m; and `harpersferry`, an SGI O2 with a 180 Mhz R5000, 64 MB of memory, and 2 GB of disk running IRIX 6.5.3f.

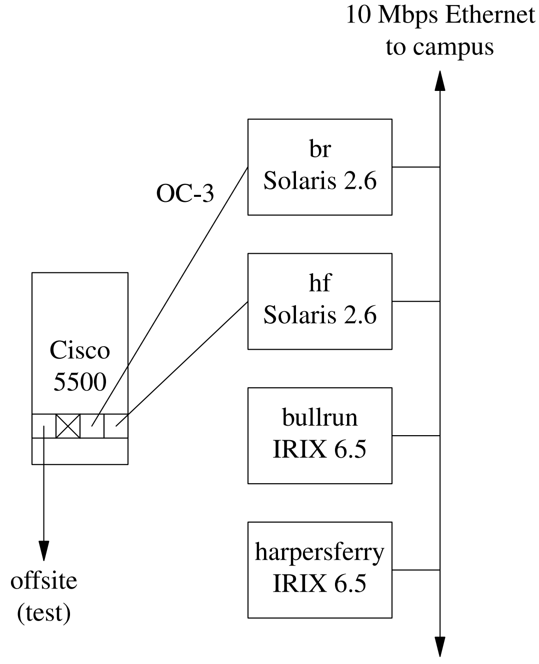

Figure 1 CITI experimental testbed. One of the Cisco 5500 ATM ports is connected to a LANE module, a second provides connectivity to an offsite test ATM network. `br` and `hf` are shown connected to the remaining two ports.

Each system is permanently attached to a 10 Mbps Ethernet for Internet connectivity and legacy traffic. ATM connectivity is provided by a four-port OC-3 ATM card located in a Cisco 5500 switch; LANE services are provided by software in the switch. The Sun and SGI machines are equipped with Fore Systems (now Marconi) SBA-200E and PCA-200E OC-3 ATM cards, respectively. All machines share two available ATM ports; this permits us to exchange ATM traffic between two hosts at a time. The Cisco switch also provides connectivity (and LANE services) to an offsite test network; it is usually lightly loaded. All of our ATM traffic passes between the two shared ports.

All hosts run an AFS 3.5 cache manager. Both `hf` and `harpersferry` run an AFS 3.5 file server; `hf` was given 4 GB of additional external disk storage for this purpose. `harpersferry` also runs an MR-AFS 3.4 server;† both servers access the same AFS partitions, but may not be run concurrently.

† As of this writing MR-AFS has not been upgraded to version 3.5.

5. Selective Caching Implementation

We began our development work on the Solaris platform as it was more familiar to us and because we had complete operating system source code. Development of the IRIX solution was delayed pending completion of the Solaris prototype.

We added code to implement selective caching (occasionally known as cache bypass in the code) for read operations. Our requirements do not include implementing selective caching for writes, which simplifies our implementation and the following analysis somewhat.

5.1. Solaris Overview

Initially, we examined some previous code written at CITI that implemented a version of selective caching in the RS/6000 AIX kernel [9]. We learned quickly that the Solaris and AIX kernels are very different in the way they treat filesystems and virtual memory; neither the previous design nor the previous code were viable.

In the Solaris kernel, the virtual memory system and the filesystem are not independent [10]. Solaris treats almost all of virtual memory as a filesystem cache, and a filesystem must be integrated with the virtual memory system to provide consistency of data when files are accessed both by memory mapping and by the conventional read/write operations.†

The AFS cache manager interacts with the Solaris kernel as follows. When a `read` call is made from a user process to read data from a file in `/afs`, the AFS filesystem vnode routine ("vnode op") `afs_vmread` is invoked. `afs_vmread` and `afs_vmwrite` both call a common routine `afs_nfsrdwr`, which calls `afs_VerifyVCache` to ensure the file status on the client reflects the actual status on the server, to verify that an AFS vnode exists for the file on the client, and to read the data from the server a page at a time. Within this read loop, `afs_nfsrdwr` calls several `segmap` routines, which are provided by the Solaris kernel to manage segments of virtual memory for both the kernel and user processes, as well as the AFS routine responsible for copying data from kernel to user space, `AFS_UIOMOVE`. The latter causes a page fault if the data are not present, *e.g.,* if this is the first time this AFS file has been accessed by this client.

To resolve the page fault, Solaris calls the `getpage` vnode op,‡ which calls `afs_GetOnePage`, which calls `afs_GetDCache`. The last of these is responsible for verifying that a DCache (AFS data cache) entry exists for the file and for determining the latest version of the file. `afs_GetDCache` may be called with various flags that dictate whether it should merely verify that a DCache entry exists, or whether it must ensure that the latest file data for that file are present in the client's cache; in the latter case, if the cached copy of the file is out of date, the file contents are fetched a chunk at a time via Rx streaming RPC from the server and stored to the AFS disk cache.

After calling `afs_GetDCache`, `afs_GetOnePage` enters a loop calling `page_lookup`, a Solaris VM routine to determine whether the page already exists in the kernel's virtual memory. If not, it calls `afs_ustrategy` to read the data from the disk cache into the user's virtual memory.

5.2. Solaris Modifications

Our modifications to Solaris are grouped into two categories: changes to handle uncached fetches from the file server, and changes required to improve performance in the presence of both prefetching and locking.

5.2.1. Uncached Fetches

For our selective caching implementation, we believed that by not writing data to the disk cache file and then reading them back into virtual memory a page at a time we would improve performance considerably. We changed the `afs_getpage` vnode op to determine whether the file should be cached or not. If not, it calls our new routine `afs_GetOnePageNoCache` instead of `afs_GetOnePage`. The former is very similar to the latter, except that it does not call `afs_GetDCache` — we don't need to maintain a DCache entry for non-cached files. Like `afs_GetOnePage`, our routine calls `page_lookup` to determine if the required page is already in virtual memory. If it is not, instead of calling `afs_ustrategy` to read the disk cache file, it simply allocates virtual memory to hold a chunk of the file and then calls the AFS server to get the data.

Unlike the caching case, when the data arrive from the server they are written directly into the kernel's virtual memory segment. A pair of new routines, `afs_Get-NullCache` and `afs_NullCacheFetchProc`, are used to accomplish this. These routines are similar to the UFS and in-memory AFS cache routines. We considered making the null cache analogous to the UFS and memory caches already supported by the AFS code; however, doing so would have required the unnecessary overhead of using the `fs_GetDCache`

† Although mapping is the preferred way of accessing files and many Solaris system utilities have been rewritten to use it, the need for the conventional calls remains. So we find `cat`, which now uses memory mapping, still opening and reading a single byte from every file it maps — to update the file read time.

‡ Actually, `getpage` is called for every page access as Solaris depends on the filesystem to determine whether a filesystem's page is up to date in virtual memory.

routine. This turned out to be a good decision, as we'll see later.

After implementing this strategy, we compared its performance to the caching case. Attempting to log kernel timing data via `printf` generated too much additional overhead and resulted in many lost timing records, so we implemented a ring buffer in the kernel to accumulate timing data and a user-level program to dump the data after completion of a test. Contrary to expectations, the improvement was slight, less than 5%. Investigation showed that while the caching code does write data to the disk cache file and then read them back into virtual memory, this is not expensive because virtual memory is a filesystem cache. The data normally remain in VM between writing and subsequent readback, replacing disk I/O with a simple memory copy. When it is necessary to do the disk I/O, it is usually because the file data are not cached on the client at all — in which case the network time dominates.

5.2.2. Prefetched Data

Not satisfied with the previous results, we looked elsewhere. The original code attempts to prefetch chunks from the fileserver when it recognizes that the user is accessing the data in a sequential manner; in effect, a fetch for the anticipated next chunk of data is sent to the fileserver while the current request is being processed. Code for this exists in `afs_nfsrdwr`, but it depends on the existence of a DCache entry that we don't have for non-cached files.

We developed the `afs_PrefetchNoCache` routine, called from an asynchronous kernel daemon so that it would not interfere with the user's original request, to prefetch chunks for non-cached files. Interestingly, instrumentation of the existing prefetch code showed that it was never prefetching. Concerned that there might be an architectural reason for this that would interfere with our own prefetch code, we contacted Transarc and were sent patches to fix the existing prefetch code [11]. The patches did not improve the performance of the caching case. Upon further investigation, we suspected that kernel data structure locking was interfering with prefetching.

We received more patches that change the way the locking of the vnode structures is done during data fetches [12]. The application of the latter patches caused us to see improvements in our code, but no improvements for the caching case. The routine `afs_GetDCache` always gets a shared lock on the vcache entry for the file in question. The caching case prefetch code also acquires a shared lock. The locking semantics allow only one shared or exclusive lock to be granted at any one time, which prevents two processes from running

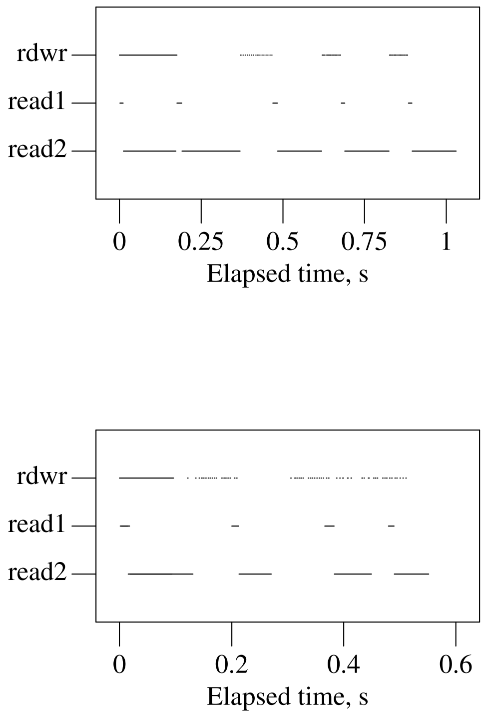

Figure 2 Locking effects. The diagrams show sequences of `afs_nfsrdwr` calls and resulting Rx data fetches, plotted against time. The top diagram shows `afs_nfsrdwr` calls and Rx fetches blocking each other; the bottom diagram shows the improved concurrency exhibited by the cache bypass code with the locking patches applied.

concurrently if they both require a shared lock. This means that the user attempting to read data is blocked while the prefetch of the next chunk is in progress, defeating the purpose of the prefetch entirely.

Luckily for us, we do not call `afs_GetDCache` for non-cached files. The user's read operation requires only a read lock on the vcache and the prefetch can obtain the shared lock without interference. We observe a 26% decrease in elapsed time due to the elimination of this blocking, when compared to the caching case.

Figure 2 makes clear the effect of locking on concurrency, which compares the time necessary to fetch five 64 KB chunks from the fileserver with and without locking patches. Both graphs shows a sequence of calls to `afs_nfsrdwr`, each of which requests a 4 KB page. If such a request cannot be satisfied from the cache, the next 64 KB chunk is retrieved from the server using two Rx calls: a fetch of the count of the number of bytes to be fetched (read1) followed by a fetch of that number of bytes of data (read2). In the top graph, without locking patches, the first page request triggers both a chunk request and a prefetch for the next chunk; because of locking, the two requests occur sequentially. Thereafter, a prefetch blocks access to the previous chunk — already present in the cache — while the prefetch completes. The bottom graph shows the improvement: the initial fetch and subsequent prefetch occur concurrently; thereafter, afs_nfsrdwr calls overlap with Rx fetches, and we observe a 40% decrease in time required to fetch the five chunks.†

These measurements were taken from our Solaris cache bypass implementation; the results for the Solaris caching implementation are similar to those shown in the top graph.

5.3. IRIX Modifications

For the Solaris work, we had access to the kernel source code and documentation [10]; these were invaluable. For the IRIX work, we didn't have access to either. However, we were able to infer enough information from the existing AFS code for IRIX to perform a port of the Solaris work. To our good fortune it appears that the IRIX virtual memory code does more of the work of managing VM pages for the filesystem than does Solaris. This port was made easier by the experience we gained with the Solaris work; we were able to reuse several of our routines without modification.

The IRIX read vnode op calls the AFS routine afs_xread. Both afs_xread and afs_xwrite call a common routine afsrwvp. This routine calls the IRIX routine chunkread. Comments in the AFS code led us to believe the strategy vnode op is called from chunkread. The strategy vnode op is responsible for getting the data from the fileserver and putting them into VM.

afsrwvp calls chunkread for a page of data at a time. This results in a call to afs_strategy if the page does not already reside in VM. It appears that chunkread determines page residency, whereas this task was left to the filesystem in Solaris. We changed afsrwvp to request data a chunk at a time rather than a page at a time, to obviate one-page I/O requests to the fileserver.

We also changed afsrwvp to queue a request for our prefetch routine, which does not depend on a DCache entry, rather than the normal DCache-based request. In afs_strategy, we changed the code to fetch the data from the server via routines similar to the routines created for the Solaris port. For non-caching files, we call afs_NoCacheStrategy which calls afs_GetNullCache to get the data from the server and put them into virtual memory.

† The gap near the center of the bottom graph is probably due to missing trace records, but requires further investigation.

5.3.1. Non-cached File Writes

We do not implement non-caching file writes. This would seem to require synchronous writes to the fileserver which would slow down the caching writes unacceptably. When a non-cached file is opened for writing, the file is transitioned into caching mode; when the last writer closes the file, it becomes non-caching again. The file's cache and VM pages are flushed on each transition. We assume such transitions are infrequent.

5.3.2. Caching Strategies

We recognize that the development of application-aware caching strategies continues to be an active area of research with diverse and potentially complex solutions [13, 14, 15]. We decided not to implement any of these automated caching strategies, relying instead on a simpler model proposed by Hacker [9], in which the AFS volume becomes the unit of cacheability. All files located in a given volume observe the same caching strategy, which is determined from the volume name; individual files may still observe a differing strategy as explained below. We believe that volume-based caching is a good idea, because it permits caching decisions to be made without requiring special file names or other application-level changes, while retaining the ability to influence those decisions in a way transparent to applications. More experience with the implementation will help validate this idea.

We modified afs_lookup to check the name of the AFS volume in which the file resides to determine whether the file should be cached. If the name of the volume contains two leading underscores, such as __video.data.volume, then files within the volume are, by default, not cached on the client; otherwise, files from the volume are cached. This behavior can be overridden on a per-file basis: a file in a non-caching volume is cached if its name begins with two plus signs, as in ++CacheThisFile; a file in a caching volume will not be cached if the file name begins with two leading underscores, as in __DontCacheMe.

In situations where a given file or volume name cannot be altered to achieve the desired caching behavior, we implemented a pioctl call to set the caching behavior for a file. However, should the file entry be discarded from the AFS VCache on the client, any caching behavior set in the entry by the pioctl call is lost.

6. Split Path Implementation

Again, we began our development work on the Solaris platform, intending as before to port our solution to IRIX when documentation on the in-kernel ATM support became available for that platform. However, our efforts to secure such documentation were not successful, and in this case we had no existing AFS kernel code to study. This precluded an IRIX port.

Our overall strategy is to establish a point-to-point ATM connection between AFS client and server, and route data fetched from the server over this connection. Other client/server communcations continue to use the existing Rx connection. This is a classic split-path design in which file data can take advantage of ATM's higher bandwidth and MTU sizes, while the smaller control packets travel over a control channel of smaller capacity without cluttering the data channel.†

We run a modified version of the Rx protocol over the data channel in which the basic Rx packet size has been increased to 64 KB; Rx jumbograms are disabled and Rx packets are transmitted individually. In addition, we streamlined the protocol layering by removing the UDP and IP layers from the protocol stack; Rx passes packets to the ATM network directly.

ATM is by nature a stateful connection-oriented protocol. User processes access the ATM hardware via an XTI library; the Solaris kernel can access ATM via a mostly undocumented kernel XTI library.‡ While the AFS file server is a user process and can use the user level XTI library, the cache manager resides in the kernel and must use the kernel library. In the absence of kernel documentation, we were forced to assemble the portions of the kernel XTI API needed for the cache manager by examining the Solaris kernel sources and extrapolating from the user library API.

One issue with the XTI protocol is the inherent scaling limitation caused by point-to-point ATM connections. This limits the number of ATM-capable clients with which a given server can maintain connections, and also limits the number of ATM-capable servers to which a given client can connect. We have not been able to investigate these limits using our testbed.

6.1. Solaris Overview

Once the AFS cache manager has determined that a file's contents must be fetched from the server, `afs_GetDCache` establishes a new Rx call via `rx_NewCall`, which determines if an Rx connection

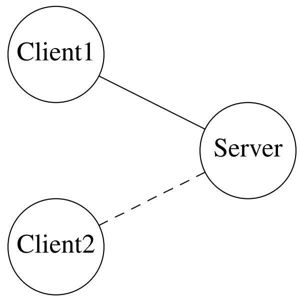

Figure 3 Split-path ATM connections. Client1 has established an ATM connection to Server, but there is no ATM connection between Client2 and Server. Client1 file fetch traffic uses our modified Rx over the ATM connection, while Client2 fetch traffic uses conventional Rx over UDP. All clients and servers use conventional Rx for traffic other than file fetches (not shown).

already exists to that server. If not, the server's IP address is found in the cache manager's data structures and an Rx connection structure is allocated. This connection structure points to a peer structure that maintains information pertinent to communicating with the server. In this way multiple Rx connections can share a single peer structure.

`afs_GetDCache` then makes a conventional Rx streaming RPC call to the server using the Rx connection, obtaining data in chunks and passing them to the kernel for further processing as explained in the section on selective caching.

6.2. Solaris Modifications

Our design augments the existing AFS client and server connections with native ATM connections accessed through XTI library calls. Fetch requests and replies travel over ATM if it is available and over a conventional Rx/UDP connection if not. An ATM connection is attempted by a cache manager the first time a fetch request occurs for a given server and the result of the attempt is remembered for future requests by that cache manager. ATM connections persist after creation for future use. Split-path connectivity is shown in Figure 3.

Our split path changes here are grouped into four classes: NSAP address discovery, ATM connection management, Rx packet routing and MTU management. Each is treated in more detail below.

6.2.1. NSAP Address Discovery

In order to establish an ATM connection, the *NSAP address* of the remote endpoint must be known to the connection initiator, *i.e.,* the AFS cache manager. A

† On machines with a single ATM connection providing both native ATM and LAN Emulation, this distinction reduces to a less efficient control channel.

‡ The XTI [16] facility originates from System V streams [17] while eliding none of its complexity.

new RXAFS_GetNSAP RPC allows a cache manager to interrogate a fileserver; if the server supports ATM it returns its NSAP address, otherwise the RPC fails.

For a server to respond to the new RPC, it must be able to discover its endpoint address. The server first attempts to retrieve its address from a configuration file, if it exists; missing or malformed data in the file prohibit the server from establishing ATM connections. If the file does not exist, the server probes the ATM hardware to obtain the NSAP address.† Implementing the operations in this order allows an administrator to prevent a file server from accepting ATM connections even though it possesses operable ATM hardware, and to prevent or override ATM hardware probes.

Once a cache manager succeeds in establishing an ATM connection, the ATM listener, *i.e.,* the AFS file server, needs to obtain the NSAP address of the initiator in order to send back replies. This information is available to the listener at connection establishment via the user-level XTI library.

6.2.2. ATM Connection Management

When a cache manager needs to create a new Rx connection to a file server, rx_NewConnection is called. We added code to this routine to establish the ATM connection as well. If the current state of the ATM connection to the server is not known, an NSAP discovery RPC is attempted; if successful, the ATM connection is created and the NSAP and IP address of the server are remembered. If the RPC fails or the server indicates it does not support ATM, this is cached in the peer structure to prevent future calls to rx_NewConnection from retrying indefinitely.

Both cache manager and file server require an *ATM listener* to wait for and process Rx packets arriving over the ATM connection; these are similar in nature to the Rx listeners that already exist. On the server the ATM listener is implemented by a pthread created when the server accepts an ATM connection; it accepts and processes incoming packets until the connection is closed, after which it exits. On the client, a small pool of kernel ATM listeners is created when the cache manager is initialized. These listeners never exit; an idle listener is chosen from the pool and allocated to a newly created ATM connection. When the connection is closed, the listener is returned to the pool.

A global ATM connection table (ACT) holds state information about each open ATM connection for both cache manager and file server, including the NSAP address of the remote endpoint, the IP address and port of the remote endpoint, a handle to the ATM connection, and the process ID and state (running *vs.* idle) for kernel ATM listeners. Multihomed file servers continue to be supported (multiple IP addresses can map to the same NSAP address), but a file server is assumed to possess only one ATM connection.

† Our implementation probes the hardware by issuing ioctls to /dev/fld; this is supported for ForeThought 5.0.0.8 and 5.2.0.3, but not for other releases or vendors.

6.2.3. Rx Packet Routing

Recall afs_GetNullCache makes an RPC call to the file server when higher-level cache manager code has determined that a file must be fetched from the server. We inserted code to check the state of the peer flag; if set to indicate a valid ATM connection to the server exists, a corresponding flag is set in the call structure associated with the current RPC call, and processing continues to prepare packets for transmission to the server. The call structure flag is later checked in rxi_Send and rxi_SendList; if set, the ATM connection handle is placed in the associated packet structure. Later, the low-level routine rxi_SendPacket sends the packet via the stored handle, effectively routing the packet either via our ATM or the conventional UDP connection to its destination. Packets containing data from the file server are similarly routed back to the cache manager via the correct kind of connection by using the server's copy of the ACT.

6.2.4. MTU Management

In order to achieve maximum performance over the ATM connection, we need to send packets at the native MTU size, 64 KB. Accordingly, a set of MTU-related data is added to the peer structure for an ATM-capable peer, augmenting the data already kept for the conventional Rx connection. This permits link level metrics and so forth to be kept independently for both kinds of connections. All ATM-related MTUs are set to the native ATM MTU size less the size of the Rx header.

Minimal tuning of Rx's retransmission logic has been done to date. Retransmissions are not much of an issue over uncongested ATM connections, but for other conditions more work needs to be done here.

7. Experiments

This sections summarizes our experimental results. All experiments were conducted on the CITI testbed.

7.1. Rx Protocol Stack Measurements

Initially, we obtained some comparative measurements on a pair of available 70 Mhz SparcServer 5 machines, each equipped with a 23 Mhz system bus, 1 GB of disk and running Solaris 5.6; one machine contains 32 MB of memory, the other 64 MB. An OC-3 ATM network connects the two machines. The goal of these measurements is to compare the performance of various layers of the Rx protocol stack in order to determine bottlenecks, not to obtain performance numbers comparible to larger and faster machines such as those running at the sponsor's site.

The measurements of the Rx protocol stack include the raw ATM AAL5 bandwidth as measured by the `svc_send/svc_recv` utility that is supplied with the ATM card; UDP throughput as measured by `sendudp/recvudp`, a locally-written instrumented UDP packet sender and receiver; and Rx throughput as determined by the AFS utility `rxtest`, which measures throughput rates between two user-level processes. For comparison, we also show TCP throughput as measured by `ftp` in binary mode. The results are shown in Table 1.

These results show that native ATM performance is good. However, the UDP layer reduces this by a little less than half, and Rx performance is dismal at 12 Mbps; this is less than a third of TCP's performance under the same conditions.

However, observe that the native ATM test is performing little packet processing at either end of the connection. This confirms our suspicions that the network is not the bottleneck. We verified this by comparing FTP binary and ASCII-mode transfers; the latter were significantly slower, probably due to the CR/LF processing necessary in that mode. This pointed to the CPU as a bottleneck.

From these results we also identified two other likely bottlenecks: the UDP layer itself reduces throughput considerably, and while the native ATM test uses packet sizes of 64 KB, LAN Emulation limits IP packet sizes to 9 KB, so the smaller packet sizes seem to be hurting UDP throughput, and to a similar extent, TCP.

Accordingly, we built a version of `svc_send` that eliminates the UDP layer, instead communicating directly using native ATM, and ran the native `svc_send` with both 9 KB and 64 KB packet sizes. We repeated some of the tests on the newly-arrived Ultra 1 machines; the results are shown in Table 2.

The results show an almost four-fold increase in performance when eliminating UDP from the Rx protocol stack; Rx performance is approaching the physical OC-3 limit, and this performance occurs while Rx is processing packets at both ends of the connection.† In contrast, increasing the packet size had a negligible effect on performance.

7.2. Selective Caching Results

We initially implemented selective caching, with prefetching and without any split-path modifications, for both Sun Ultra 1 (Solaris 5.6) and SGI O2 (IRIX 6.5), as well as MR-AFS on SGI O2 (IRIX 6.5). We measured the performance of our implementation by repetitively flushing a test file from the client AFS cache and measuring the time to fetch the file. Steady-state performance was measured by discarding initial runs. File sizes of 1 MB, 10 MB, and 100 MB file were used. The results are shown in Table 3.

With selective caching enabled, an average improvement of about 19% in elapsed time is observed when reading a 100 MB file. We see an 85% average improvement in system time as well, but the elapsed time dominates. User time is essentially unchanged.

7.3. Split Path Results

We implemented split path processing by adding this capability to both cache manager and file server, for Sun Ultra 1 (Solaris 5.6). As this cache manager had

Rx Protocol Stack	
Test	Mbps
ATM	92
UDP/IP	55
Rx	12
TCP/IP	40
Rx/SGI	35

Table 1 Rx Protocol Stack. This table shows measurements of the Rx protocol stack, including native OC-3 ATM, UDP, and Rx protocols. running on a pair of Solaris 5.6 SparcServer 5 machines connected by an OC-3 ATM network.. TCP performance is shown for comparison, as is the Rx performance on a pair of SGI machines running IRIX 6.5.

Modified Rx Protocol Stack	
Test	Mbps
Rx/ATM, 64 KB	127
Rx/ATM, 9 KB	126
Rx/UDP, 9 KB	38.5

Table 2 Modified Rx Protocol Stack. These tests were run on Solaris 5.6 Ultra 1 machines, which are faster than SparcServer 5's. Rx over native ATM shows performance independent of packet size approaching the physical OC-3 limit, while Rx over UDP shows a marked performance drop.

† Both sides of the Rx connection are serviced by user-level processes in this test; when serving files, the AFS cache manager uses a kernel Rx implementation.

Selective Caching Performance

Config	Size MB	Elapsed Time Cach s	Elapsed Time Bypass s	Elapsed Time Impr %	User Time Cach s	User Time Bypass s	User Time Impr %	System Time Cach s	System Time Bypass s	System Time Impr %
Solaris	1	1.24	1.15	7	0.13	0.16	-23	0.08	0.02	75
	10	13.5	11.45	15	1.42	1.41	1	0.53	0.17	86
	100	149.19	117.15	21	14.43	14.16	2	4.9	1.71	85
IRIX	1	1.4	1.28	9	0.02	0.02	0	0.15	0.04	73
	10	14.93	12.34	17	0.17	0.16	6	1.45	0.21	86
	100	155.09	127.41	18	1.67	1.61	4	14.31	2.15	85
IRIX MR−AFS	1	1.37	1.23	10	0.02	0.02	0	0.15	0.04	73
	10	14.65	12.27	16	0.17	0.16	6	1.42	0.21	85
	100	163.97	132.12	19	1.7	1.6	6	14.76	2.07	86

Table 3 Selective caching performance. This table compares the elapsed (wall clock), user, and system times when reading a file, for three file sizes in three configurations: a Solaris cache manager with a conventional AFS server, an IRIX cache manager with a conventional AFS server, and an IRIX cache manager with an IRIX MR-AFS server. Each test compares the caching and non-caching (bypass) performance and the percentage improvement shown by the bypass code.

Combined Selective Caching and Split Path Performance

Config	Size MB	Elapsed Time Cach s	Elapsed Time Comb s	Elapsed Time Impr %	User Time Cach s	User Time Comb s	User Time Impr %	System Time Cach s	System Time Comb s	System Time Impr %
Solaris	1	2.11	0.42	80	0.13	0.16	-23	0.07	0.02	71
	10	15.38	3.37	78	1.47	1.43	3	0.55	0.14	75
	100	155.27	26.83	83	14.55	14.32	2	5.42	1.72	68
	1	1.51	0.24	84	0.15	0.15	0	0.07	0.02	71
	10	15.3	2.56	83	1.43	1.45	-1	0.55	0.11	80
	100	157.94	29.21	82	14.28	14.3	0	5.54	1.73	69
	1	0.9	0.26	71	0.15	0.14	7	0.07	0.03	57
	10	14.83	2.61	82	1.43	1.39	3	0.55	0.16	71
	100	152.27	27.9	82	14.72	14.23	3	5.44	1.8	67

Table 4 Combined performance. This table compares the elapsed (wall clock), user, and system times when reading a file, for three file sizes, in a single configuration with a Solaris cache manager and a conventional AFS server. The test was repeated three times.

already been changed to accommmodate selective caching, this had the effect of implementing the combined features. Although the two features are separable, we did not create a version implementing split path processing only.

Again, we measured the performance of our combined implementation by repetitively flushing a test file from the client AFS cache and measuring the time to fetch the file. We used the same file sizes as before, and measured steady-state performance by discarding initial runs. The results are shown in Table 4.

Combining selective caching and split path processing, we observe a remarkable average reduction of 80% in elapsed time for file transfers across all file sizes tested. For example, the 100 MB file is transferred over five times faster, in less than half a minute as opposed to over two and a half minutes. Figure 4 shows these results graphically for the 100 MB file.

Figure 5 compares throughput versus file size for the caching and combined selective caching and split path cases. Data for Figures 4 and 5 are taken from the first row of Table 3; the remaining rows show similar results.

While much better than the caching figures, we note that these throughput values are one quarter of those observed in the Rx protocol stack measurements discussed in Section 7.1. There are two main differences between the protocol stack measurement code and the combined implementation: the latter implements the full AFS client and server, not just the Rx layer; and the AFS client runs in the kernel. We were not able to duplicate Table 2 for the combined implementation.

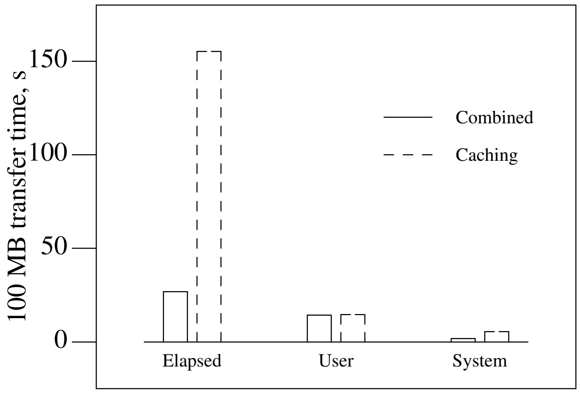

Figure 4 100 MB transfer time. This graphs compares elapsed, user, and system time consumed for the caching transfers *vs.* cache bypass and selective caching combined.

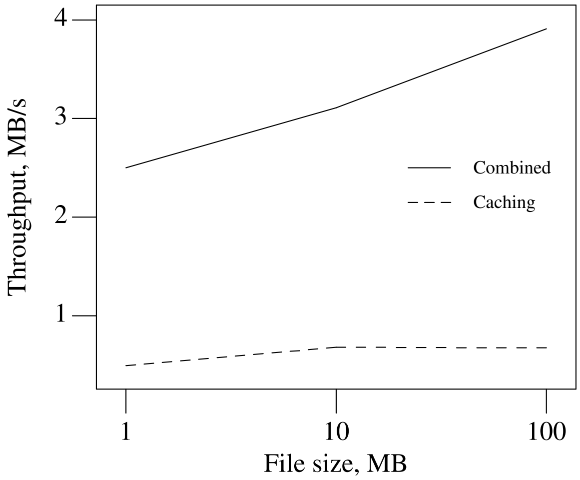

Figure 5 Throughput *vs.* file size. While the caching case shows poor performance and little improvement with file size, cache bypass and selective caching combine for greatly improved throughput that increases with file size.

However, we believe the disparity in throughput is mostly due to CPU costs in the combined implementation; this is corroborated by the sharp drop in throughput for the protocol stack measurement when ASCII-mode FTP is selected. This supports the notion that our testbed is CPU-limited, and that the performance of our implementation will improve when ported to a platform with a faster CPU.

8. Future Work

While the results of this work have been successfully demonstrated in the laboratory, several issues remain to be addressed before this can be considered a robust implementation suitable for general deployment. A physical or logical link failure occasionally causes the associated kernel ATM listener to fail gracelessly, *i.e.,* the cache manager crashes. This must be fixed, and in-progress Rx calls correctly dealt with, and the ATM link state should be reset so future Rx calls will attempt to recreate the connection.

The link-level metrics have not been extensively tuned for native ATM networks. It will be necessary to deploy to or simulate a congested ATM network in order to adjust link parameters such as retransmission timeouts appropriately.

Currently, the combined implementation provides native ATM connectivity only between AFS clients and servers. In the MR-AFS domain, it is useful to extend ATM connectivity between residencies, so the benefits accrue to those transfers as well.

9. Conclusion

We have discussed two enhancements to the AFS distributed file system intended to improve the performance of large file fetches over high-speed networks: selectively bypassing the client disk when local caching is of little benefit, and implementing a high-performance native ATM connection between AFS client and server.

We have implemented these enhancements in our laboratory and measure an 80% reduction in the elapsed time required to fetch a large (100 MB) file, while retaining compatibility with unmodified AFS clients and servers.

Some work remains to develop a robust implementation suitable for general deployment, including tuning for performance in congested ATM networks, and graceful handling of failed ATM links. It is also necessary to evaluate the results of our work on machines faster than those available in our laboratory.

10. Acknowledgements

We thank Pradip Patel of Information Technology Communications Services for making the Cisco 5500 and associated ATM module available to us for the duration of our project, and for getting the switch to work with our ATM cards when no one else could.

Thanks also go to Seth Meyer of the Information Technology Division's Login Team for lending us Ultra 1 Solaris machines. Timely access to these machines was essential to the successful completion of our project.

Sushila Subramanian helped us in many ways, including conducting porting work and performance tests at the sponsor site and in general being an excellent liaison between our project team and the sponsor. We are grateful for her assistance.

Marcus Watts provided Solaris kernel expertise, divined the kernel XTI library API, and implemented much of the ATM Rx packet routing and MTU code.

Bob Hyer of IBM kindly supplied us with patches for the cache manager locking code, and provided valuable insights.

This work is supported by the Naval Research Laboratory under Grant No. N00173-98-1-G017.

References

1. J.H. Howard, "An Overview of the Andrew File System" in *Proc. Winter USENIX Conf.,* p. 23−26, Dallas (February, 1988).
2. M.T. Stolarchuk, "Faster AFS" in *Proc. AFS Users Group* (Spring, 1992). Available at `http://www.citi.umich.edu` as CITI Technical Report 92-3.
3. C.J. Antonelli, W.A. Doster, and P. Honeyman, "Access Control in a Workstation-Based Distributed Computing Environment" in *Proc. IEEE Workshop on Experimental Distributed Systems,* Huntsville (October, 1990). Available at `http://www.citi.umich.edu` as CITI Technical Report 90-2.
4. "NRL Motion Imagery Lab (MIL)" in `http://www.nrl.navy.mil/CCS/people/kern/HDTV/`.
5. "Survivors of the Shoah Visual History Foundation" in `http://www.vhf.org/`.
6. "Clementine - Deep Space Program Science Experiment" in `http://www.nrl.navy.mil/clementine/`.
7. Jonathan S. Goldick, Kathy Benninger, Christopher Kirby, Christopher Maher, and Bill Zumach, "Multi-resident AFS: An Adventure in Mass Storage" in *Proc. Winter USENIX Conf.,* p. 47−58, New Orleans, LA (January 16-20, 1995).
8. Bob Sidebotham, *Rx: A High Performance Remote Procedure Call Transport Protocol,* Information Technology Center, Carnegie-Mellon University (February, 1989).
9. Thomas J. Hacker, *Cache Bypassing in AFS* (1992). Unpublished work.
10. Berny Goodheart and James Cox, *The Magic Garden Explained: The Internals of UNIX System V Release 4, an open systems design,* Prentice-Hall (1994).
11. Bob Hyer, personal communication, Transarc Corporation (July, 1999).
12. Bob Hyer, personal communication, Transarc Corporation (July, 1999).
13. Jim Griffioen and Randy Appleton, "Reducing File System Latency using a Predictive Approach" in *Proc. Summer USENIX Conf.,* pp. 197-207, Boston (Summer, 1994).
14. Brian Noble, M. Satyanarayanan, and Morgan Price, "A Programming Interface for Application-Aware Adaptation in Mobile Computing" in *2nd USENIX Mobile and Location-Independent Computing Symposium,* pp. 57-66, Ann Arbor (April, 1995).
15. Todd A. Anderson and James Griffioen, "An Application-Aware Data Storage Model" in *Proc. USENIX Conf.,* pp. 309-322, Monterey, California (June, 1999).
16. W. Richard Stevens, *UNIX Network Programming, Volume 1: Networking APIs - Sockets and XTI,* Prentice-Hall (1997).
17. Dennis M. Ritchie, "A Stream Input-Output System" in *UNIX Research System Papers,* p. 503−511, Murray Hill (1990).