MFS: an Adaptive Distributed File System for Mobile Hosts

Benjamin Atkin and Kenneth P. Birman
Department of Computer Science
Cornell University, Ithaca, NY
{batkin,ken}@cs.cornell.edu

Abstract

Mobility is a critical feature of computer systems, and while wireless networks are common, most applications that run on mobile hosts lack flexible mechanisms for data access in an environment with large and frequent variations in network connectivity. Such conditions arise, for example, in collaborative work applications, particularly when wireless and wired users share files or databases. In this paper, we describe some techniques for adapting data access to network variability in the context of MFS, a client cache manager for a distributed file system. We show how MFS is able to adapt to widely varying bandwidth levels through the use of modeless adaptation, and evaluate the benefit of mechanisms for improving file system performance and cache consistency using microbenchmarks and file system traces.

1 Introduction

Mobility is now an major feature of computer systems: over the past decade, laptops and hand-held devices capable of wireless network access have become common, and wireless networks are also proliferating. Applications that run on hosts in wireless networks must cope with constraints on access to data that are generally not present in wired networks. Distance from a base station, contention with other hosts or processes on the same host, interference, and switching between different wireless media all compound the variability in network performance to which applications must adapt if they are to perform well.

This paper focuses on adaptation techniques for management of data accessed and modified by mobile hosts. We investigate adaptation in the context of MFS, a client cache manager for a distributed file system. We concentrate on distributed file systems because systems in this area are highly developed and have well understood semantics, although the techniques we describe should be broadly applicable in other application environments, such as caching dynamic Internet content or caching to improve the performance of interactions with web services. We evaluate MFS using file access traces from Windows NT and Unix, and a synthetic workload designed to emulate sharing patterns seen in collaborative engineering systems.

The authors were supported in part by DARPA under AFRL grant RADC F30602-99-1-0532, and by AFOSR under MURI grant F49620-02-1-0233, with additional support from Microsoft Research and from the Intel Corporation.

Existing work in cache management for mobile file systems [7, 13, 15, 18] incorporates mechanisms for making efficient use of available bandwidth. However, it has mostly focused on adapting existing systems to cope with periods of low bandwidth, in a style which we will refer to as *modal adaptation*. When bandwidth is high, the application communicates normally; when bandwidth falls below a threshold, the application enters a low-bandwidth mode in which communication is restricted or deferred. More generally, an application has a small number of possible modes and chooses the appropriate one based on the currently available bandwidth. For example, in the Coda file system [18], the cache manager operates in either a strongly-connected, weakly-connected, or disconnected mode, which affects the policy for writing changes to files back to the server.

Modal adaptation schemes are well-suited to environments in which changes in bandwidth are relatively predictable, such as switching network access from an Ethernet to a modem, but not as appropriate in for wireless networks, in which bandwidth availability is less predictable and varies over a larger possible range. The notion of "insufficient bandwidth" can vary depending on how much data the application is trying to send, so that it may make sense to adjust network usage when the bandwidth drops by half, rather than just when it falls to modem-like levels. Selecting a mode according to the available bandwidth can unnecessarily constrain communication, since it ignores what data the application actually wants to send over the network. Deferring writing back all modifications to files may not be a sensible policy if those are the only messages available to send.

We describe MFS (Mobile File System), a flexible cache manager for a distributed file system client, which differs from traditional cache manager design in two important respects. First, MFS uses an RPC library supporting priorities to enable *modeless adaptation* [1], which allocates available bandwidth based on the types of messages being sent. By assigning priorities appropriately, foreground activities, such as retrieving files, can proceed concurrently with background activities such as writing back changes, under the assurance that if bandwidth becomes scarce, the background activities, rather than the foreground ones, will be penalised first. Modeless adaptation using prioritised communication also allows MFS to be more flexible in response to bandwidth variations than would be possible with a modal scheme. Second, MFS incorporates a new cache consistency algorithm to efficiently provide a high degree of consistency for access to shared files, which is required for collaborative work applications.

The rest of this paper is organised as follows: Section 2 describes the MFS design and differences from existing distributed and mobile file systems, as well as giving an overview of the MFS RPC library. Section 3 describes the use of prioritised communication in MFS and experiments to evaluate its effectiveness. Section 4 presents and explains experimental results for the MFS prefetching mechanism, and Section 5 does the same for the cache consistency algorithm. Finally, Section 6 concludes and describes future work.

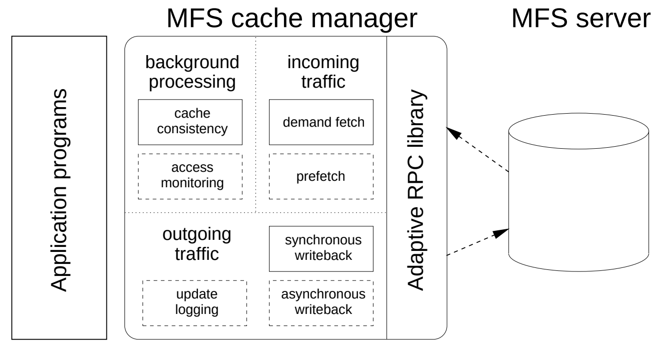


Figure 1: MFS architecture. *The most important part of MFS is the cache manager, which intercepts file system operations from application programs and resolves them into accesses to its local MFS cache or RPCs to a server. The cache manager has a number of components: those in solid boxes are part of the core system, those in dashed boxes are optional extensions which are described in subsequent sections.*

2 MFS overview

MFS differs from earlier mobile file systems in adjusting to changing network conditions using modeless adaptation. It comprises a core client-server file system, and a number of subsystems that perform different kinds of adaptation, and can be selectively enabled. Figure 1 shows the structure of the system. In this section we describe the core system, while subsequent sections do the same for the three main subsystems. We begin with an overview of mobile file system design and the relation of MFS to previous work, then briefly describe the adaptive RPC library used in MFS, and the current MFS implementation.

2.1 MFS design and related work

The core of MFS follows a design common to many mobile file systems [7, 13, 15, 18], which use techniques such as whole-file caching, and update logging combined with asynchronous writes, to cope with disconnections or intermittent connectivity.

The design of MFS is closest in structure to that of Coda [18] and the Low-Bandwidth File System (LBFS) [13]. A host acting as a client of an MFS file system runs a user-level cache manager, which receives file system operations intercepted by a kernel module, interacting with the VFS layer of the local file system. We adopt the same approach to intercepting VFS operations as LBFS, making use of the kernel module provided as part of the Arla AFS client [21].

The cache manager maintains a cache of recently-accessed MFS files on the local disk. When a VFS operation is intercepted for a file that is not in the cache, it is retrieved in full from the appropriate server, and the VFS operation is then resumed. MFS uses the writeback-on-close semantics first implemented in the Andrew File System [6]. When a dirty file is closed, the entire file contents are transferred to the server[1]. The LBFS chunk scheme for minimising bandwidth utilisation when transferring files is not used in MFS, although it is orthogonal to MFS adaptation and could be added to further improve performance.

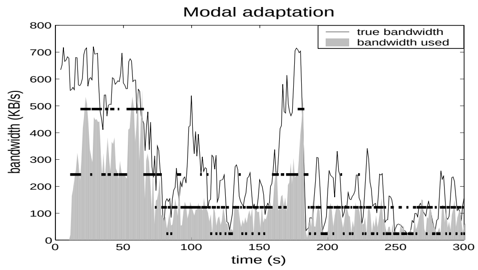
The server that stores a file is responsible for maintaining the mutual consistency of the copies cached by clients. It records which clients cache the file, and is responsible for notifying them of changes. MFS implements a variation of the scheme used by Coda: when a file is retrieved from the server, the server issues a limited-duration “callback promise”, obliging it to inform the client through a callback if another host modifies the file. If the callback promise expires without a callback being issued, the client must revalidate the file before using it. The cache consistency algorithm is described in more detail in Section 5.

2.2 Adaptive RPC library

The fundamental difference between MFS and other file systems we have described is in the communication between the cache manager and servers. While LBFS uses a variant of the NFS RPC protocol [17], MFS, like Coda, uses a customised RPC. However, unlike Coda’s RPC, the RPC used in MFS incorporates novel features to allow it to adapt to network variability. The MFS RPC library is implemented on top of the Adaptive Transport Protocol (ATP). In discussing MFS RPC, we give an overview of the parts of ATP which are most relevant to MFS; ATP and its design motivations have been described in more detail in our earlier work [1].

The hypothesis underlying ATP is that adapting to network variation by structuring applications according to modes is not always appropriate, and can sometimes lead to poor performance. Figure 2 shows the results of an experiment in which modeless adaptation over ATP achieves higher bandwidth utilisation than

[1]Though MFS is designed to support multiple MFS file servers, in this paper we will concentrate on a system with a single server.



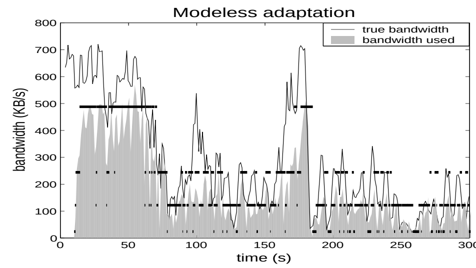


Figure 2: Modal versus modeless adaptation with ATP. *The left graph shows performance with modal adaptation, and the right graph shows a scheme in which there are four classes of messages being sent simultaneously, of increasing priorities (the lowest line corresponds to the highest priority). Dark horizontal lines represent operating modes on the left, and the highest priority of data being sent during a second on the right. The modeless scheme achieves higher utilisation (48.5 MB of data sent) because it always has messages to send, while the modal scheme is dependent on a rapid and accurate estimate of the available bandwidth in order to select its correct operating mode (41.5 MB sent). These graphs are reproduced from [1].*

an equivalent modal scheme. Other experiments have shown that modeless adaptation can achieve improvements of 10-15% in bandwidth utilisation, and it is possible to construct cases in which the improvement is even greater. Work on adaptation in mobile file systems has generally relied on modal schemes [7, 18], but our evaluation of ATP demonstrated that it could also improve the performance of file system-like workloads. We discuss the implementation of modeless adaptation in MFS further in Section 3.

ATP is implemented at user level, on top of kernel UDP. It has a message-oriented interface for communication, in which messages of an arbitrary size can be reliably transmitted with their boundaries preserved at the receiver's side. An application can send a message synchronously or asynchronously. In the latter case the sender provides a function to be executed when transmission of the message completes, and the send operation itself is non-blocking; this is similar to the Queued RPC developed for Rover [8]. Unlike Queued RPC, ATP also allows the sender to attach a priority to each message, to control the order in which the queued messages are transmitted. Messages are queued at the sender according to their receivers, and each queue is ordered by priority. Messages of the same priority within a queue are transmitted in first-in, first-out order. ATP also allows a sender to specify a send timeout for a message, which causes the transmission to be suspended if it expires, so that the sender can react to it. An analogous mechanism is available for receive operations. Besides detecting when a remote host is inaccessible, send timeouts do not play a major role in MFS. An additional use for timeouts would be to detect prefetches which are not making progress and reissue a prefetch for a different file (see Section 4).

ATP administers priorities by deriving an estimate for the bandwidth available between the sender and receiver. In order to minimise the transmission delay when a new message is sent, ATP uses a form of rate-based flow control. Each second is divided into twenty *send periods* of 50 milliseconds' duration, and at most one-twentieth of the available bandwidth is used during a single send period. Without such a constraint, ATP would send as much data as it could on receipt of a low-priority message, and this data could then be buffered at an intermediate link, delaying the transmission of any high-priority message which might be sent later. The disadvantage of this scheme is that heavy contention at the sender may delay a new message by as much as 50 milliseconds, regardless of its priority. This inefficiency of the ATP implementation is most visible when there is contention between different priorities at high bandwidth.

2.3 MFS implementation

The version of MFS described in this paper is implemented in C and runs on FreeBSD 4.5. Both the client and server have multiple threads to cope with simultaneous file system requests, and the RPC library has its own thread: therefore there are two mandatory thread context switches on any message send or receive operation. As we shall describe in subsequent sections, some subsystems have additional threads to carry out background processing. Unless noted otherwise, our experiments were conducted with a default client cache size of 256 MB.

3 RPCs with priorities

MFS RPCs are implemented on top of ATP in the natural way: an RPC request constitutes one message, and its reply another. Priorities are used to differentiate types of RPCs to improve performance: in general, small RPCs, or those which would cause an interactive client to block, are given high priority. RPCs for background activities, such as writing back files to the server, or prefetching files, are performed at low priority, so that they do not slow down high-priority RPCs. Table 1 shows the priority levels for different types of RPCs.

Assigning priorities to RPCs allows MFS to adapt to bandwidth variation in a straightforward way. At high bandwidths, all RPCs complete quickly, with or without priorities. As bandwidth

priority level	*corresponding RPC types*	*section*
VALIDATE (high)	fetch attributes, callbacks	3
FETCH	fetch file data, directory contents	3
STORE-ATTR	write back directory and metadata updates	3
STORE-FAST	write back shared files	5
STORE-DATA	write back unshared files	3
PREFETCH (low)	prefetch file data	4

Table 1: Priority levels for MFS RPCs. *Symbolic names are given for the priority levels, listed from highest to lowest priority. The third column gives the section in which the corresponding RPC types are described in detail.*

decreases, an implementation without priorities will result in the completion times for all RPCs increasing uniformly. When priorities are used, a backlog of low-priority RPCs will accumulate, while the time taken for high-priority RPCs to complete will increase more gradually. Our design is based on the assumption that when bandwidth is low, an assignment of differentiated priorities will improve the response times for interactive tasks. If a task which predominantly performs reads executes in parallel to a task which performs many writes, then with priorities, the first task will receive a higher share of the bandwidth.

In practice, many applications have patterns of interactive file access involving both reads and writes. For instance, compiling source files involves interspersed reads and writes, but does not issue concurrent RPCs frequently. Such an application will have improved read performance when there is contention with other applications, but will correspondingly be penalised on writes. This does not match our design goal of having interactive, "mostly-read" applications obtain a larger share of bandwidth. We have implemented two solutions to this problem, based on making writes asynchronous: *update logging*, used in several existing systems and incorporated in MFS for the purposes of comparison, and *asynchronous writeback*, which is new to MFS. An alternative approach is to retain synchronous writes, but assign priorities according to some notion of relative importance of processes. Unfortunately, existing operating systems and applications generally do not provide this information, so we have not investigated it further.

3.1 Update logging

Update logging, which is implemented in some mobile file systems (notably Coda [18] and Little Work [7]), removes the requirement that processes wait for writes. Rather than sending an update to the server as soon as a file is closed, the cache manager logs the update and periodically flushes logged updates to the server. These systems enable logging when bandwidth is low, to improve read performance and reduce write traffic by aggregating updates to the same file in the log before they are transmitted.

Update logging separates communication with the server into two distinct streams: updates to files and directories, and all other traffic. These two types of communication are scheduled independently and may compete. However, by making writes asynchronous, update logging pushes read-write contention "into the future", to occur at the next log flush. The designers of Little Work incorporated a low-level priority mechanism at the IP packet level to further reduce interference between writeback traffic and other network traffic sent by the client [7].

3.2 Asynchronous writeback

Though it reduces bandwidth consumption, update logging is fundamentally unsuitable for use at high bandwidth, since it imposes a delay on transmitting updates to the server. Systems using update logging must therefore switch to a synchronous writes when bandwidth is high, with a threshold controlling switches between the two modes. The mode switch also changes the semantics of the file system, and the developers of Coda have noted that undetected mode changes can surprise the user in undesirable ways [18], such as cache inconsistencies arising due to unexpectedly delayed writes.

Rather than relying on a modal adaptation scheme incorporating a transition to update logging when bandwidth is low, MFS uses a modeless *asynchronous writeback* mechanism, which is active at all bandwidth levels. Just as with update logging, when an application performs an operation that changes a file, such as a write or metadata update (create, delete, create directory and so on), it returns immediately. The update is then passed to the writeback subsystem, which sends it to the server when there is sufficient bandwidth: asynchronous writeback therefore only delays updates when there is foreground traffic. When bandwidth is high, the performance of asynchronous writeback should be comparable to purely synchronous writes, but when bandwidth is insufficient, asynchronous writes will improve the performance non-update RPCs.

The cache manager's writeback thread divides updates into metadata operations, such as directory modifications and file status changes, and file writes. The two types of operations are queued and replayed to the server separately, so that a metadata RPC can proceed in parallel with a file writeback. When an RPC from a particular queue completes, we say that the update has been *committed* at the server. The next update is then dequeued and an asynchronous RPC for it is initiated. Separating the small metadata RPCs from file writes allows remote clients to see status changes to files without having to wait for intervening writeback traffic. A similar motivation underlies the cache consistency scheme for high read-write contention environments we describe in Section 5.

The chief complexity in implementing asynchronous writeback lies in resolving dependencies between metadata operations and updates to the same file. For instance, a file may be created, modified and closed, and the length of the metadata queue may be enough to mean that the file update would be initiated first: in this case the file update must wait. Alternatively, a file may be

test	*activity*	*synchronous*		*logging*		*asynchronous*	
		uniform	*priorities*	*uniform*	*priorities*	*uniform*	*priorities*
GC	grep	2.9 (0.2)	3.4 (0.4)	3.2 (0.5)	3.1 (0.4)	3.3 (0.5)	3.1 (0.4)
	compile	63.6 (1.0)	63.0 (0.9)	54.4 (11.7)	48.1 (16.9)	60.7 (0.8)	52.6 (11.4)
GW	grep	**9.2 (0.5)**	**8.9 (1.0)**	8.4 (0.2)	8.5 (0.2)	8.5 (0.4)	**8.1 (0.1)**
	write	10.7 (0.5)	10.8 (0.6)	1.7 (0.5)	1.6 (0.4)	1.7 (0.2)	1.6 (0.2)
RC	read	19.2 (0.8)	19.5 (0.9)	20.1 (0.8)	20.1 (1.4)	20.3 (1.1)	19.8 (0.7)
	compile	75.7 (1.4)	78.7 (2.3)	65.0 (12.5)	66.0 (11.8)	71.9 (1.3)	68.0 (1.2)
RW	read	**34.7 (1.4)**	**21.3 (1.4)**	35.0 (1.3)	22.3 (1.3)	35.4 (1.7)	**21.4 (1.3)**
	write	23.8 (1.2)	40.3 (1.5)	3.4 (0.4)	3.0 (0.1)	3.7 (0.7)	3.1 (0.2)

Table 2: Performance of MFS priorities and writeback schemes. *Each test consists of two concurrent processes executing different workloads. Mean times to completion are shown with standard deviations, over ten executions. Three different policies for writing back files are listed, under uniform or differentiated priorities (reads take precedence over writes). Values in bold are of particular significance. Note that elapsed times for write workloads give the time until the process running the workload finishes, not when the log is flushed (this is shown in Figure 3).*

modified and then deleted, which requires the file update RPC to be cancelled if it is still in transmission when the remove RPC is initiated. An update to a file will supersede any previous queued updates.

3.3 Performance evaluation

After adding priorities to RPCs, it is natural to ask when they are beneficial, and to what degree. In addition to comparing MFS with and without prioritised RPCs, we also investigate the performance impact of replacing synchronous RPCs for file updates with asynchronous writeback. The performance of these alternatives is compared in a set of microbenchmarks, and with workloads gathered from Windows NT file system traces.

Our experimental setup consists of two 1 GHz Pentium III desktop machines running the FreeBSD 4.5 operating system, one of which acts as an MFS server, and the other as an MFS client. The client machine makes use of the Dummynet traffic-shaping module in FreeBSD to limit its incoming and outgoing bandwidth. The experiments we conduct in this section have a constant bandwidth over the duration of the experiment, but we analyse the performance of MFS when the bandwidth varies over the course of an experiment in Section 4.5.

3.4 Microbenchmarks

The first set of experiments compares different MFS configurations for specific types of contention. Four workloads were used:

Grep: executes the `grep` utility several times on each of 256 8-KB files. The files are present in the cache, but must be validated before they are used.

Read: accesses 16 1-MB files in sequence, writing the contents of each file to `/dev/null`. The files are not initially present in the cache.

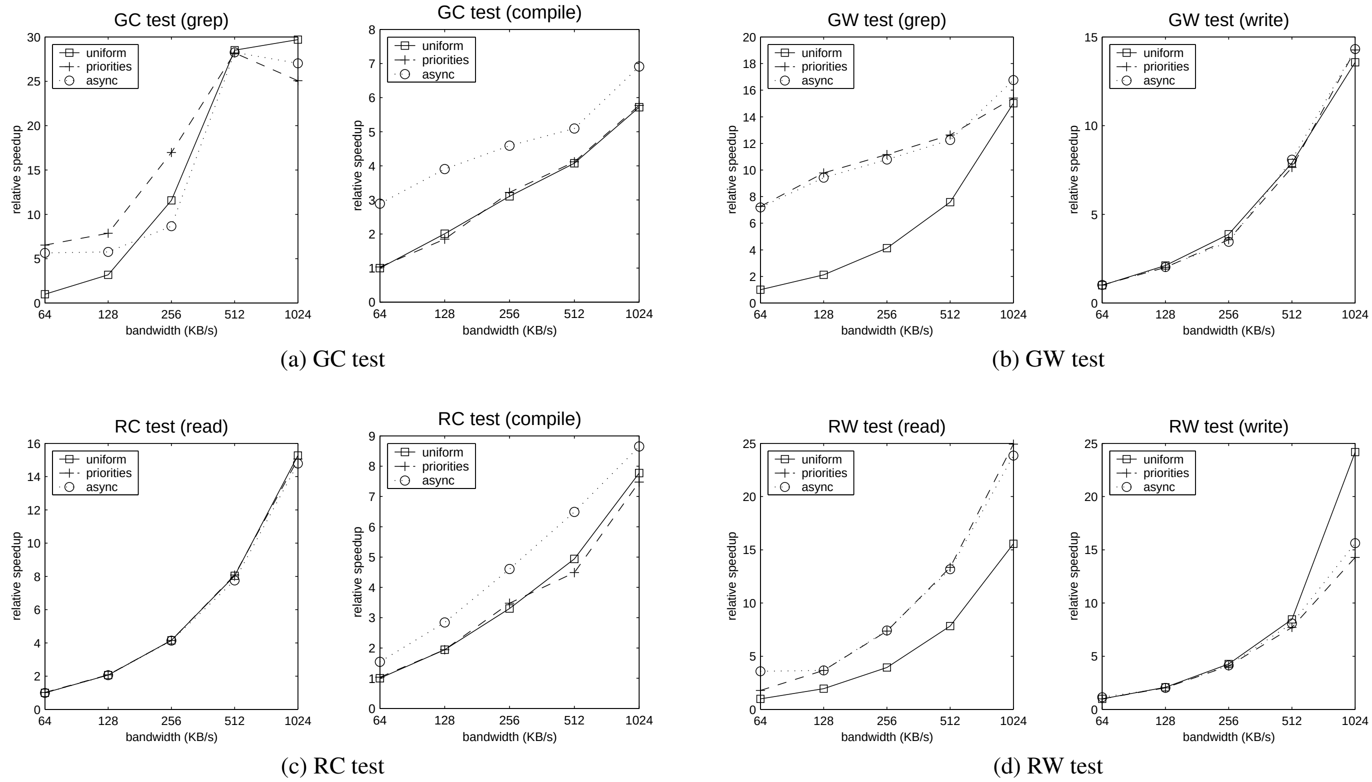
Write: copies 16 1-MB files from the local file system into the MFS file system.

Compile: compiles the entire MFS file system and its RPC library (259 files and directories comprising 1223 KB). None of the files are initially in the cache. This workload performs an intensive pattern of reads and writes files without raising the issue of concurrent accesses (a topic we tackle in Section 5).

Of these workloads, we classified Grep and Read as foreground workloads, and Compile and Write as background workloads. Four combined workloads were then generated by running a foreground and a background workload concurrently: we denote these as GC (Grep/Compile), GW (Grep/Write), RC (Read/Compile) and RW (Read/Write). Three types of RPCs predominate: cache validations, fetches of file data, and store operations for files, in descending order of priority. The aim of the experiments was to demonstrate that priorities improve the performance of the foreground workloads.

The four combined workloads were executed on top of MFS configured with either synchronous writes, update logging or asynchronous writeback. The update logging mechanism was configured to delay flushing an update for at least a second. Every experiment was repeated ten times at each of five possible bandwidth values. Table 2 shows the time taken for each workload at a bandwidth of 1024 KB/s, and Figure 3 shows overall results for selected configurations.

The results in Table 2 demonstrate the benefit of priorities when there is high contention between high-priority RPCs and writes. In both the I/O-bound GW and RW workloads, adding priorities decreases the time required for the foreground workload to execute, by up to 38% (see elapsed times for RW-read with synchronous writes in the table). This is particularly true in the RW test, where the foreground workload generates heavy contention by fetching a large volume of data. The greatest benefits are observable for the combination of asynchronous writes with priorities, since here the performance of the background workload can also improve by not having to wait for its writes to be committed at the server. In the GC and RC tests, where there is lighter contention, the impact of priorities is negligible, and in some cases results in a slight overhead, but this is chiefly



(a) GC test (b) GW test

(c) RC test (d) RW test

Figure 3: Performance of prioritised RPC with respect to bandwidth variation. *Each pair of graphs in shows the speedup of one of three cache manager configurations, relative to the time taken by uniform priorities with synchronous RPCs at 64 KB/s. As well as uniform priorities and synchronous RPCs ("uniform"), the graphs also show curves for differentiated priorities and synchronous RPCs ("priorities") and differentiated priorities and asynchronous RPCs ("async"). The values plotted for bandwidth of 1024 KB/s are the same as shown in Table 2.*

due to the overhead of priorities for small RPCs mentioned in Section 2.2.

Comparing the execution time of the foreground workloads with synchronous writes, update logging and asynchronous writeback reveals that the latter two options generally perform comparably to or better than synchronous writes. Logging and asynchronous writeback greatly improve the performance of the background workloads, as has been noted previously [7, 18]. We focus on MFS with asynchronous writeback in the rest of this paper because it provides comparable performance to logged updates, allows straightforward modeless adaptation to bandwidth variation, and is easily extensible to more than one level of priority, which is required for our cache consistency algorithm.

Since reducing available bandwidth increases the contention between RPCs of different types, the benefits of RPC priorities should be more apparent at lower priorities. Figure 3 shows the experiments of Table 2 extended to a wider range of bandwidth values. In these and later experiments, we evaluate MFS performance with bandwidths from 64 to 1024 KB/s: while 64 KB/s is not "low" in the sense of prior work, it is low enough to cause significant contention for the workloads we have considered, and we believe that our results will hold if available bandwidth and traffic are scaled down further in parallel.

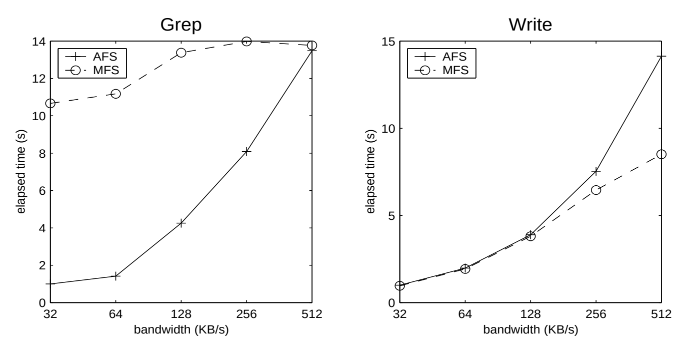


Figure 4: Comparison of MFS and AFS performance. *MFS with synchronous RPCs and priorities is compared to a version of the Andrew File System. Speedups for the two workloads of the GW test are shown, relative to the performance of AFS at 32 KB/s.*

The graphs in Figure 3 validate the incorporation of RPC priorities, since all the foreground workloads improve their performance substantially at lower bandwidths, relative to MFS with no priorities. Furthermore, the decrease in throughput for the

Grep in the GW workload even is less than would be expected with reduced bandwidth: here uniform priorities result in throughput linear in the bandwidth, while differentiated priorities are less sensitive. The RC and GC tests show the benefit of asynchronous writeback, since the updates from the compile workload are committed sooner to the server than with synchronous writes, due to the overlap of "think time" with asynchronous writes. Finally, though uniform priorities provide better performance for the Write component of the RW test at 1024 KB/s (as is to be expected, since we are prioritising reads), this benefit largely vanishes at lower bandwidths.

Though we have concentrated on determining the benefit of RPC priorities by a comparison of different configurations of MFS to one another, we have also performed a few experiments to compare the performance of MFS to a standard distributed file system. Figure 4 illustrates the result of running the GW test over MFS and an Andrew File System (AFS) setup; we used the Arla implementation of the AFS cache manager [21] and the OpenAFS server. AFS uses a UDP-based RPC library without priorities. The results largely correspond to those in Figure 3. MFS significantly outperforms AFS for the foreground Grep workload, since AFS effectively uses synchronous RPCs with uniform priorities. In the background Write workload, AFS slightly outperforms MFS, but it is both a more mature system, and more optimised than MFS for this sort of communication. Since the results of running the other tests are similar, we omit them for brevity.

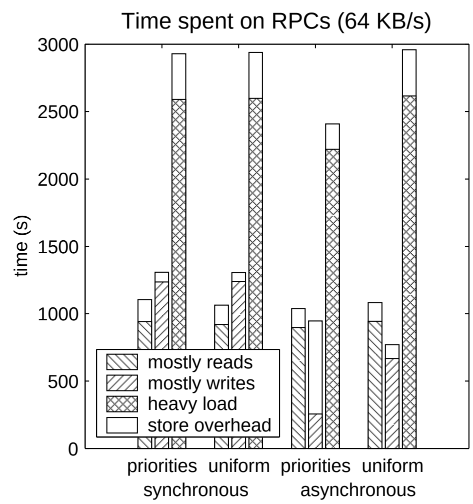
3.5 NTFS workloads

In addition to measuring the performance of MFS with synthetic workloads, we have also conducted experiments with traces gathered from the Windows NT file system (NTFS) [20]. Although MFS is implemented on a variant of Unix, and NTFS has a somewhat different interface to the file system, the traces were converted to run on top of MFS with little difficulty. The original traces recorded file accesses on a set of machines in a LAN. A majority of the accesses were local but some were to remote machines. We extracted subintervals from the traces which featured interesting file system behaviour and processed them to remove accesses to files over 4 MB in size. This preprocessing was necessary to eliminate the influence of extremely large NT system files, which made up 50% of the file system traffic in some portions of the original traces. Given that MFS retrieves and writes back whole files, including these system files would have distorted the experiments at low bandwidths.

Table 3 gives statistics for the three traces: a trace in which reads predominate, a trace in which writes predominate, and one containing exceptionally heavy file system traffic. Each trace was run over MFS with the combinations of synchronous and asynchronous writes and differentiated and uniform priorities in previous experiments, and the results are given in Figure 5. To interpret these graphs, look for instance at the "heavy load" bar

parameter	*trace*		
	mostly reads	*mostly writes*	*heavy load*
duration (s)	101	106	34
applications	25	15	41
unique files	2952	276	3312
total file sizes (MB)	46.61	16.51	125.31
read traffic (MB)	10.82	12.74	27.82
write traffic (MB)	3.44	19.92	9.59

Table 3: NTFS trace parameters. *These traces are representative periods of mixed read and write activity. The durations are from the original NTFS traces. Note that the total file sizes represent the amount fetched by MFS during the trace. Where this is exceed by the write traffic, the additional traffic is due to new files being created or existing ones extended.*



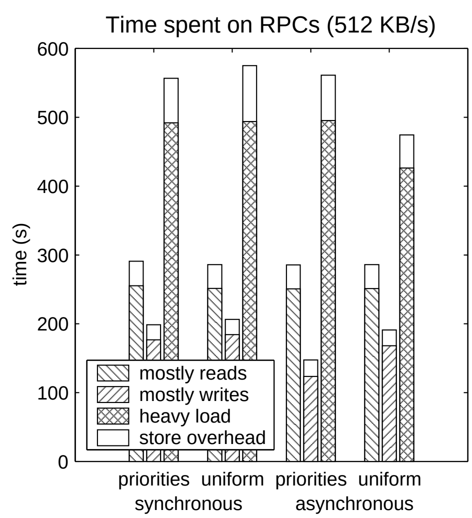


Figure 5: Graphs of NTFS traces. *Each trace ran with synchronous or asynchronous writes and uniform or differentiated priorities. The total height of each bar denotes the time from the first to last write, and the shaded portion denotes the time from the first to last read. The white portions denote the extra time required to complete all writes after the last read has finished.*

for asynchronous writeback with priorities in the 64 KB/s graph. This shows that the total duration of the trace with this MFS configuration is 2409 seconds, but all the fetch traffic is completed within 2220 seconds of the start: this is a significant improvement over the alternative configurations measured. The remaining 189 seconds of the trace are taken up by asynchronously writing back file updates.

In all cases the traces take significantly longer than they originally did in NTFS, where they were mostly accessing the local file system and therefore had no bandwidth constraints. The results largely repeat those seen in the microbenchmarks, to the extent that the greatest performance improvements are seen at low bandwidth when there is high read-write contention, such as in the mostly-writes trace where there is an 79% decrease in the time spent to read all the files. However, even at the higher bandwidth of 512 KB/s, there is a decrease of 30%. The mostly-reads trace is not much affected by changes in the configuration, although there is a slight decrease in both read and write times for prioritised asynchronous writeback. Unusually, at 512 KB/s the heavy-load trace performs best with uniform asynchronous writeback: we once again attribute this to inefficiency in the RPC protocol, since under extremely heavy load and high bandwidth it performs better when all messages have the same priority.

4 Prefetching

Prefetching is commonly used to improve the performance of local file systems, as well as distributed file systems. However, in a file system with whole-file access, a mechanism is required to determine appropriate prefetching hints. Earlier work in file system prefetching has used clustering to derive file groups from cache access statistics [11], predicted future file accesses from recent accesses [5], or allowed applications to specify prefetching hints explicitly [16, 19].

Inter-file dependencies can also be used as a source of hints. For instance, it may be known that a certain shared library is required to run a text editor: in this case it would be advantageous to retrieve the shared library from the server as well as retrieving the text editor executable. Alternatively, explicit information such as the operating system's database of installed software packages, or other application-specified dependency information can be used.

Any of these techniques could be used to derive hints for use by the MFS prefetching subsystem; our evaluation uses hand-specified dependency information, which is inaccurate in some cases. Rather than reimplementing an existing hint-generation mechanism, we focus on the performance of MFS with prefetching, using a deliberately simple hint mechanism for the purposes of evaluation. Dependencies between files are conveyed using a *file group*, which is a list of file identifiers for the related files. It is assumed that after one file in the group has been accessed, it becomes advantageous to prefetch the remainder of the files in the group. A file group is implemented as a special type of file within the MFS file system, with its own file identifier, but not attached to any specific directory. The file group a file belongs to, if any, is one of its attributes.

The MFS prefetching subsystem derives much of its effectiveness from being combined with prioritised RPCs. While the prefetching algorithm in MFS is straightforward, it can still make bad decisions without a large overall performance penalty because the interference of prefetching with other file system activity is minimised. In the same way that some local file systems execute speculative operations to improve performance [3], MFS makes use of the "speculative communication" of prioritised RPCs in the hope of achieving a benefit through prefetching files.

4.1 MFS prefetching implementation

The MFS cache manager incorporates a small prefetching module, which can be optionally enabled at start-up. When it is initialised, a prefetching thread starts and initiates prefetch requests in parallel with the main activity of the cache manager.

The core component of the cache manager alerts the prefetching module every time an application reads or writes a file, by calling the `file_access` routine. This routine checks whether the file belongs to a file group – if not, the access is ignored. If it is a member of a file group, the group is put at the head of the *prefetch list*. The prefetch thread periodically examines the group at the head of the list. If the group file for the group is not in the cache, it retrieves it from the server. Then it scans the files in the group in order until it finds the first one which is not in the cache, or not validated, and issues a prefetch request or validation request for it. If all the files are valid and are in the cache, the group is moved to the end of the prefetch list. Once the prefetch completes, the thread rechecks the head of the list to find the next file to prefetch: a new group may now be at the head of the list as a result of further application accesses to files.

Prefetch requests are similar to regular fetch requests for files, with the exception that they are issued at the lowest level of priority: all other RPC traffic takes precedence over a prefetch RPC, as shown in Table 1. Prefetches are synchronous, and only one prefetch is made at a time. This is more a matter of implementation convenience than a design decision: other work has shown the benefits initiating multiple concurrent prefetches from different servers [19]. MFS does not currently make use of timeouts for prefetches, as we have noted earlier, but it could easily to extended to abandon a prefetching attempt that does not complete in a timely manner.

The main complexity in implementing the prefetching subsystem lies in handling a *demand fetch* (a compulsory fetch to service a cache miss) for a file which is already being prefetched. This conflict arises very frequently, particularly when an application performs a fast linear scan of files in a file group. An efficient implementation of prefetching requires that the demand

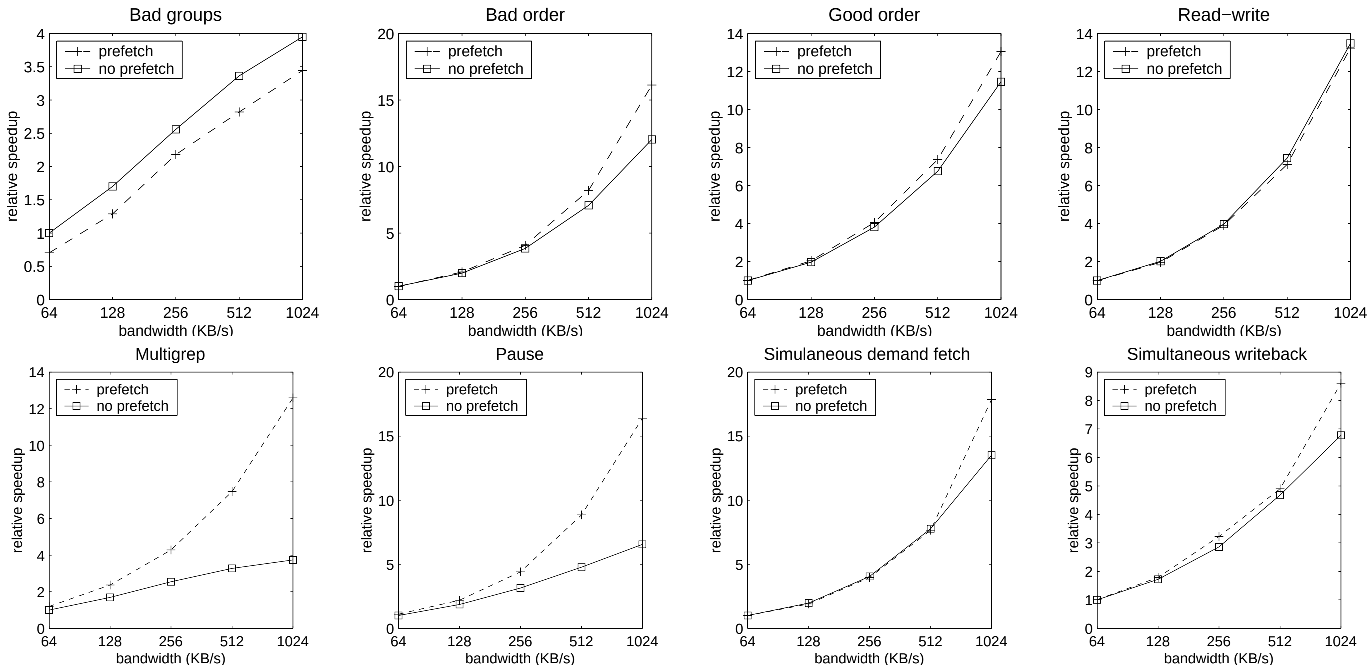


Figure 6: Relative speedup of workloads with prefetching. *These graphs show the speedup gained by adding prefetching for a range of bandwidth values, relative to the time taken with a bandwidth of 64 KB/s and no prefetching. Where a test comprises two separate processes, only the speedup for the foreground process is shown.*

fetch wait for the prefetch to complete, or that the prefetch be aborted. Issuing a fetch RPC at the same time as a prefetch is in progress needlessly wastes bandwidth, since it retrieves the same file from the server twice. The same could be true if we opt for aborting prefetches, since an aborted prefetch could be very close to completion. MFS therefore makes the demand fetch wait for the prefetch, but also raises the priority of the prefetch RPC to that of a regular fetch operation, to prevent a priority inversion. This requires an additional "raise-priority" RPC to the server, which results in more overhead than the case where a demand fetch occurs without a fetch-prefetch conflict. On the other hand, the fetch can frequently make use of the data already transferred and so still results in a faster response to the application.

As we have explained, the implementation of the prefetching subsystem is not sophisticated. While it will reach an equilibrium if the total size of the file groups in the prefetch list is less than the cache size, there is no mechanism to prevent the prefetching subsystem "running ahead" of actual file accesses and evicting useful files from the cache, or evicting files which it has prefetched but have not yet been referenced by the user. Techniques for preventing this behaviour have been discussed elsewhere [19].

4.2 Prefetching evaluation

Having added prefetching to MFS, we evaluated whether such a straightforward algorithm can have a benefit for some representative workloads. In order to characterise the effect of adding prefetching, we ran a set of eight microbenchmarks. The experimental setup was the same as in the priority tests, though this time MFS was configured to run with asynchronous writeback, and RPC with priorities, and only prefetching was either enabled or disabled. The tests were run at a range of bandwidth values, as in the previous section.

Each microbenchmark consists of one or two processes accessing files, with some or all of the files forming file groups. The Read-Write test is the same as in Section 3, with a file group added for the Read data. The Compile MFS test has six file groups for the main directories of the system. Multigrep accesses 2 MB of data in 75 text files, forming a single file group. Pause accesses 4 MB of small files, waits for 4 seconds, and accesses 4 MB more; all the files are in a single file group. Simultaneous Demand-Fetch runs as two process. One process accesses 64 files of 64 KB each, which form a file group. The other does the same, but without a file group. Simultaneous Writeback executes in the same way, but the second process writes the files to the server instead of reading them.

The remaining tests investigate the overhead paid for weaknesses in the prefetching algorithm. Bad Groups uses 16 directories, each containing 64 128-KB files and forming its own file group; on its first iteration, the workload accesses the first file in each directory, on the second, the thirty-third, to provoke a large amount of useless prefetches. Good Order and Bad Order investigate the effect of the ordered list of files in a file group: Good

Order accesses the files in the group in the same order as the list; Bad Order accesses them in reverse order.

4.3 Analysis of prefetching

The graphs in Figure 6 show the results of the experiments. Where a test such as Simultaneous Demand-Fetch incorporates more than one workload, only the elapsed time for the foreground workload (the one accessing a file group) is given.

In most of the microbenchmarks, adding prefetching from the file groups specified has a substantial improvement on the performance of the workload, varying with how amenable it is to prefetching. In general, more surplus bandwidth and more think time result in improved performance: this naturally means that the greatest improvements from prefetching are evident at higher bandwidths (six out of eight microbenchmarks run at least 10% faster when bandwidth is 1024 KB/s). In contrast, at low bandwidth most workloads see no benefit, since all the bandwidth is dedicated to higher-priority traffic.

Only two tests perform worse with prefetching than without. The Read-write test performs slightly worse due to its already heavy network contention. The Bad Groups test, which exploits poor prefetching hints, consistently under-performs when prefetching is used. This effect is due to the useless prefetching RPCs flooding the outgoing link and imposing minor delays on each demand fetch: cumulatively these slow down the overall performance. An usual phenomenon is that the Bad Order test consistently outperforms Good Order, even though the latter triggers prefetches in the "correct" order. The explanation is that, by design, the Good Order test suffers from the fast linear scan phenomenon described in Section 4.1 (all prefetches in this test conflict with demand fetches). In contrast, at the start of the Bad Order test, the prefetching subsystem is able to prefetch some files accessed at the end of the test, without conflicting with a demand fetch. It can therefore achieve a greater speedup.

4.4 Summary of results

Despite the simplicity of the MFS prefetching implementation, we have shown that workloads which are amenable to file-level prefetching can achieve speedups of 70% at high bandwidth, and as much as 10% at bandwidths as low as 64 KB/s. Prefetching carries a small performance overhead, even when performed at the lowest priority, which can reduce its effectiveness for fast linear scan workloads. It is possible to construct combination of file groups and a workload for which prefetching can significantly degrade performance.

Within the constraints imposed by our file group representation, the main conclusion we draw from the test cases exhibiting a "prefetch penalty" is that the implementation could be improved to incorporate a mechanism to inhibit prefetching. The current prefetching algorithm does not correlate file accesses with the processes which make them, but if this were done, two changes

p	*RPC type*							
	fa(p)	*fa(n)*	*fd(p)*	*fd(n)*	*pf(p)*	*df(p)*	*sd(p)*	*sd(n)*
1	23	18	1	7	13	0	7	9
2	51	41	0	15	26	2	22	20
3	54	69	0	42	24	24	35	35

Table 4: Number of RPCs by type in bandwidth variability test. *The entries under 'p' denote periods in the test. Figure 7 gives the abbreviations for RPC types.*

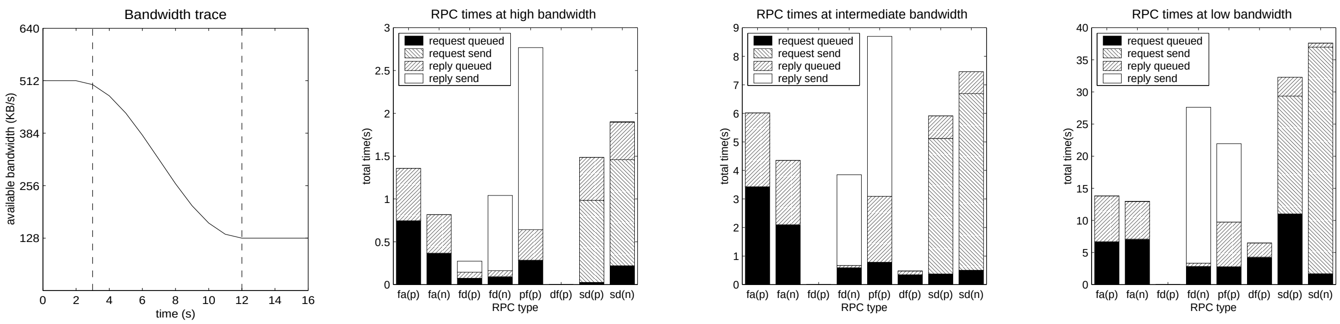
are likely to be beneficial. The first would reduce the aggressiveness of prefetching (for instance, setting a byte threshold) from a file group if it appeared that a process was not using the files prefetched based on its prior accesses. This would reduce the overhead in the Bad Groups case. The second would explicitly detect a fast linear scan by a process, by counting the instances of prefetch and demand fetch conflict for a file group, and then disable prefetching from the group.

4.5 Prefetching and bandwidth variability

So far, our experimental results have demonstrated the benefits of MFS adaptation mechanisms at various levels of bandwidth availability, but not when the bandwidth is changing over the duration of the test. To conclude this section we will describe an example of MFS traffic under the execution of the Simultaneous Writeback test described in Section 4.2. This test involves two simultaneous workloads: one writes files 64 KB to the server and the other reads 64 KB files from the server, but is slightly modified from original version to use a longer "think time" of 0.25 seconds when accessing each file (improving the potential for RPCs to overlap). We enabled asynchronous writeback and ran the test with the synthetic bandwidth trace shown in Figure 7(a), which changes the bandwidth once per second. This has three sections, a brief period when the bandwidth is at 512 KB/s, a gradual decrease to 128 KB/s over the course of ten seconds, and then the maintenance of the 128 KB/s rate until the end of the test.

The test was executed once with prefetching enabled, and once with no prefetching, and the RPCs were then divided according to which period of the trace they terminated in. For each RPC, four quantities are calculated: the time spent queued for both the RPC request and reply, and the time taken for each to be received, from the first to the last packet. This ignores the time spent at the server servicing the RPC, and the round-trip time between the client and the server, but these quantities are small compared to the other costs. These values are added up for each of the RPCs within a particular period, and the results are shown in the Figures 7(b), 7(c), and 7(d).

The graphs show how priorities affect RPCs and how prefetching changes MFS behaviour. In all three time periods, more time is spent on RPCs to fetch file attributes with prefetching enabled than without. Since the time to receive a fetch-attributes request or reply is negligible, the increased time is due to a greater queue-



(a) (b) (c) (d)

Figure 7: RPC traffic with varying bandwidth. *Graphs (b), (c) and (d) show the time spent on RPCs during an execution of the Simultaneous Writeback test from Section 4.2, with the bandwidth varying according to the curve in (a). RPCs are labelled as follows: fa = fetch-attributes, fd = fetch-data, pf = prefetch, df = demand fetch to raise priority of a prefetch RPC, sd = store-data. The time spent on RPCs is shown with prefetching enabled, denoted by (p), and disabled, denoted by (n). Note that RPC interactions can overlap so the quantities for different RPC types are not additive. For some RPC types, the time spent on particular activities is negligible in proportion to the overall time: for instance, fetch-attribute requests are small and have a very low transmission time relative to their queueing delays.*

ing delays; as we have seen earlier, high traffic can cause delays in the round-trip time for small RPCs. Conversely, store-data RPCs have a higher outgoing queueing delay in the absence of prefetching: this is due to the majority of the competing RPCs being high priority fetch-data RPCs. With prefetching, these RPCs are mostly replaced by prefetches, which operate at a lower priority than store-data RPCs, until any point where a concurrent demand fetch RPC raises their priorities to the fetch-data level.

A comparison of fetch-data and prefetch RPCs reveals the effect of the bandwidth decrease. To begin with, the test run with prefetching performs a fetch-data RPC to get the first file, which triggers prefetching from its file group. Because of the large delay between file accesses, prefetches complete entirely without any overlapping demand fetches. Over the course of the second period of time, bandwidth becomes insufficient for a prefetch to complete during the 0.25 s delay between accesses, and raise-priority RPCs are triggered by the consequent cache misses. As the bandwidth decreases, the queueing delays increase as a proportion of the total time spent on prefetches.

5 Cache consistency

Studies of distributed file systems have largely concluded that file sharing is infrequent in general-purpose environments [2, 10]. However, we have identified a class of cache consistency scenarios as being of high importance and inadequately served by existing mobile file systems. Suppose that a complex engineering design is maintained on a server and updated by teams of designers (architects, electrical contractors, engineers, and so on) while on-site supervisors work from those designs using mobile devices. These supervisors read from the server and may also change the design, for example to reflect one of the contingencies encountered and resolved only as construction proceeds. When such users happen to be working on the same element of the design, it is clear that satisfying a request from stale data (whether in from the cache, or on a server that has yet to see a delayed writeback) would be visible to the user and costly[2].

Strong cache consistency is certainly achievable in distributed file systems [22], but must be implemented with synchronous RPCs, and requires either readers or writers to incur a delay to ensure that only the latest version of a file is accessed. In contrast, as we have noted in Section 3, sending file updates to a server asynchronously has two potential benefits: the process modifying the file need not wait for the write to complete, and, if the update is delayed in the log for some interval before being written back, it may be superseded by a later update, and therefore can be omitted entirely. However, these benefits come at the cost of reduced cache consistency, since the version of the file stored at the server is inconsistent during the time that the update remains queued for transmission. Even though asynchronous writes in MFS are not delayed to aggregate updates, a burst of updates to a sequence of files may flood the link to the server and increase the delay before updates towards the end of the burst are committed. Any other client accessing the file will access the stale version, rather than one which incorporates the pending update. We therefore refer to this as a "hidden" update, and the cache consistency problem caused by asynchronous writes as the *hidden update problem*.

Mobile file systems such as Coda [18] rely on optimistic concurrency control to resolve the conflicts generated by hidden updates. An alternative approach is to use a variant of callbacks to allow a client to replay writes asynchronously, but retain strong cache consistency. For example, the Echo file system [12] forces

[2] We thank Larry Felser and his team at Autodesk for their help in understanding the file access patterns that arise in collaborative work applications for very large architectural and engineering design firms [4].

the modifying client to flush its updates whenever another client accesses the file, and Fluid Replication [9] separates invalidating a file from transmitting its update. We have implemented a similar scheme in MFS, in which an access to a file which has an uncommitted update at a different client will force the writeback. The MFS consistency algorithm differs in its incorporation of file access information. Rather than enforce the same level of consistency for all files, MFS differentiates between “private” files, which have recently only been accessed by a single client, and “shared” files, which are accessed by multiple clients. Enforcing cache consistency between clients necessarily requires that shared files are kept highly consistent, but modifications to private files can be written back to the server less aggressively. The technique of using file access patterns to adjust a cache consistency protocol has been used in the Sprite distributed operation system [14], though in Sprite changes in caching policy were made when a file was opened simultaneously at different clients, while MFS uses longer-term access statistics.

The remainder of this section describes our consistency algorithm in detail, and an evaluation of its effectiveness in reducing cache inconsistencies.

5.1 The consistency maintenance algorithm

The MFS cache consistency algorithm is intended to achieve a high degree of consistency, subject to the constraints imposed by file semantics and the desirability of minimising overhead. We have opted for a compromise which results in a small overhead but admits the possibility of a transient inconsistency.

The algorithm requires information about client accesses in order to divide files according their status, either shared or unshared. Since the file server always assumes that an unshared file has an uncommitted write when it is accessed by an additional client, incorrect information about the status of a file only affects the efficiency of the algorithm. Detection of such a misclassification results in the file being marked as shared.

The status of files can be specified by the user or by applications, or can be inferred by the file server according to how it is accessed. To be effective, automatic inference should incorporate a heuristic for the sharing status of new files, and a mechanism for converting shared files to be unshared if they cease to be accessed by more than a single client. The current implementation in MFS assumes that every new file is unshared, and monitors client accesses to a file according to an overlapping series of time periods to ensure that files which are regularly accessed remain shared. Since the MFS file monitoring component operates on a larger time scale than the experiments considered in this paper, we omit its details for brevity.

When a process modifies a file, an update is scheduled to be appended to the log, and the process continues executing without having to wait for the server to be contacted. The writeback thread then checks the status of the file the update modifies: if the file is unshared, the update is queued for transmission at the regular store-data priority. If the file is shared and no other shared update is being sent, the thread begins transmitting the update at the store-fast priority. If another shared update is being written back, a synchronous “forward invalidation” RPC is made to the server at the highest priority, and then the update is queued for later high-priority transmission. Effectively, a forward invalidation is only made if the update cannot be transmitted immediately: in practice it can therefore be omitted at high bandwidth or when traffic is low. However, sending a forward invalidation RPC without requiring the modifying process to wait introduces a transient inconsistency.

host	*parameter*	*value*
reader	delay between accessing modules	0.25-2 s
	operations per module	4-10
	delay between operations	50-100 ms
writer	delay between accessing modules	1-4 s
	operations per module	4-20
	delay between operations	50-100 ms
	size of external files	0-128 KB

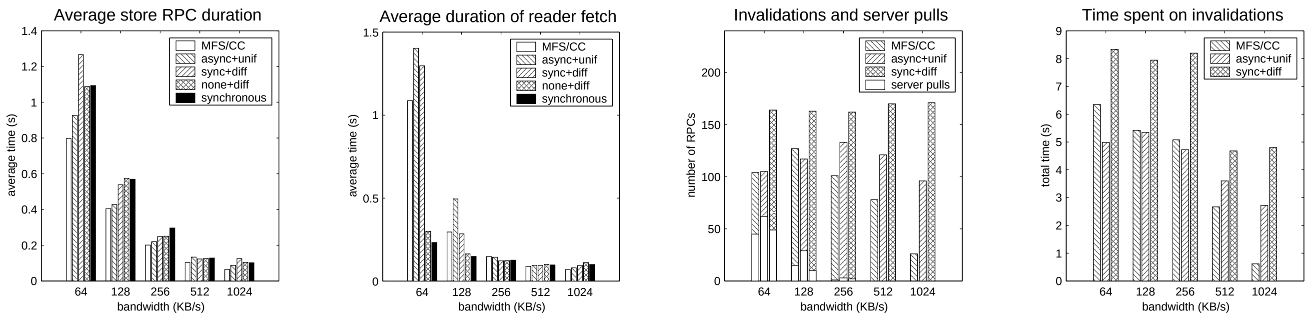
Table 5: Configuration parameters for the cache consistency evaluation. *Individual instances are uniformally distributed within the listed ranges.*

When the server receives a forward invalidation for a shared file, or begins receiving an update for a file, it records the identity of the writer, marks the file as dirty and issues callbacks to all the clients caching it. If one of these clients fetches the file before the update has been committed, the server sends high-priority “server pull” RPCs to the clients with outstanding updates, which causes them to raise the priority of any store-data RPCs to expedite transmission. A fetch RPC for an unshared file which is already cached by a different client always triggers a server pull, since the server has no way of knowing if the file has outstanding updates.

Finally, since updates to shared and unshared files are written back to the server at different priorities, the original order of the sequence of updates is no longer entirely preserved. The updates to shared files form a subsequence of the original updates, as do the updates to unshared files. However, implicit dependencies between file updates are preserved, since the combination of forward invalidations and compulsory server pull RPCs for unshared files prevents a client from accessing new versions of files in contravention of their update order.

5.2 Experimental setup

At the start of this section we identified large-scale collaborative engineering design as an example of a scenario which features a high degree of read-write sharing. At present we have evaluated the MFS cache consistency algorithm using a synthetic trace, though we are hoping to obtain real data from such an environment in the future.



(a) (b) (c) (d)

Figure 8: Graphs for cache consistency trace. *These graphs show various features of the performance results; in the legends, "async" denotes asynchronous invalidations; "sync" synchronous invalidations, and "none" no invalidations; "diff" denotes differentiated writeback priorities for shared and unshared files, and "unif" denotes uniform priorities. "MFS/CC" is the MFS cache consistency algorithm. In graph (c), the height of a bar counts the number of invalidations; the white portion counts the number of server-pull RPCs.*

Our experimental setup consisting of three hosts: one server, a "reader" client, and a "writer" client. The bandwidth from the reader to the server was fixed at 1024 KB/s, and the bandwidth from the writer to the server was varied according to the experiment. The writer was configured in one of seven different ways: with synchronous writes; or with asynchronous, synchronous or no invalidations, and differentiated or uniform priorities for writing back shared and unshared files. The MFS concurrency control algorithm (MFS/CC) corresponds to asynchronous invalidations with differentiated priority for shared files. Both clients access a shared repository of files stored on the file server, which consists of 40 modules. Each module has a descriptor file and a set of 4-12 member files. Module descriptor files are about 1 KB in size and the 334 member files take up an average of 64 KB. The total size of all the files in the collection is 21 MB.

The writer workload consists of the writer updating modules in a random order. An update to a module consists of a sequence of operations, 20% of which are reads and 40% are writes to a file in the module; the remaining 40% consist of writes to unshared "external" files, which are each created with a unique name. There is a pause between each operation and a longer pause between updates to modules. The reader workload is similar, but an access to a module consists of a series of reads, and external files are never accessed. The configuration parameters used to generate the reader and writer workload are listed in Table 5. The writer workload has a nominal duration of two minutes, while the reader workload is extended to terminate at the same time as the writer workload actually finishes (since low bandwidth could extend its running time beyond two minutes).

5.3 Analysis of the results

Figure 8 shows graphs of some selected results from the experiments. In general, while synchronous writes provide strong concurrency control, they resulted in the lowest rate of completed writes in all the tests, since the writer had no possibility of overlapping think time with asynchronous writeback. At all bandwidth levels the MFS/CC algorithm outperformed synchronous writes by at least 20%, and was among the options with the highest write throughput. This is clear from graph (a), which shows the average time to complete store RPCs initiated by the writer (excluding invalidations). Here MFS/CC outperforms all of the alternatives. This is because of the reduced number of invalidations it generates, and also since, in contrast to most of the other schemes, it is able to take advantage of both differentiated writeback, and of server-pull RPCs to raise the priority of its writes.

Graph (b) shows the performance from the reader's perspective. While the writer is able to decrease its time spent performing store RPCs, the reader's average time spent on fetches increases sharply when the file in question must be pulled from the writer. Naturally, this cost must be weighed against the benefit of substantially increased writer throughput. Differentiated writeback succeeds in reducing the time the reader has to wait when accessing a shared file.

Graphs (c) and (d) show statistics for invalidations and server-pull RPCs for those writer configurations which make use of them. MFS/CC significantly reduces the number of invalidations it must transmit by putting off invalidating a file until it is added to the log, yet the effect of this policy on the number of server-pull RPCs is minor. The "async+unif" policy, which differs from MFS/CC in omitting differentiated writeback, makes more invalidations and incurs more server-pull RPCs, because its store RPCs must compete with the RPCs to write back external files. This increases the commit delay for each file and the likelihood of it being accessed by the reader while it is being written back.

In conclusion, these experiments demonstrate that for the trace we have examined, the MFS algorithm of asynchronous invalidations and differentiated writeback is able to maintain cache consistency between the two clients and to allow the writer to write back changes to the stored data faster than is possible with the alternative schemes. We intend to further evaluate the performance of the algorithm to determine its effectiveness under other workloads, and with more clients.

6 Conclusion

The growing use of mobile computers and wireless networks has greatly increased the scope for adapting data access to varying network characteristics. This paper has explored applying the technique of modeless adaptation to a distributed file system to improve its performance. The cache manager for our MFS file system incorporates features that are not present in existing file systems for mobile hosts: adaptation to bandwidth variation through the use of prioritised communication, and an efficient cache consistency protocol using file access information to improve performance.

We have evaluated the effect of these features on performance at varying bandwidth levels and under both synthetic and real workloads, including a workload emulating collaborative data access with high read-write contention, and found that while the additional costs imposed are mostly hidden, they can have benefits which are very visible. Additionally, the non-modal nature of adaptation in MFS allows clients to adapt quickly to a variety of bandwidth conditions without substantial changes in operation. Our evaluation has included comparisons of MFS to cache manager configurations corresponding to prior work, and confirmed that there are situations in which MFS would outperform AFS, Coda and Little Work. However, these earlier systems were designed for a mobile environment which is substantially different from that available today. Essentially, MFS is able to provide improved performance in periods of high network contention by favouring cache validation and RPCs to retrieve files over other types of traffic. We have not compared MFS with LBFS since their approaches are orthogonal, and LBFS-style algorithms are not present in the earlier systems we have compared against. We anticipate that implementing LBFS file chunks in MFS would further improve performance its performance.

In future work, we plan to investigate the performance of modeless adaptation and MFS in wide-area and more web-like environments, as well as further evaluating the performance of the MFS cache consistency algorithm. We also intend to use MFS to further examine the benefits achievable from the automatic generation of caching policies for files.

Acknowledgements

We would like to thank Robbert van Renesse, Werner Vogels, Emin Gün Sirer and Paul Francis for comments and suggestions regarding MFS. We also thank Rimon Barr, Indranil Gupta, and Kevin Walsh for helpful discussions and corrections to this paper.

References

[1] B. Atkin and K. P. Birman. Evaluation of an adaptive transport protocol. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (Infocom 2003)*, San Francisco, California, Apr. 2003.

[2] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 198–212, Pacific Grove, California, Oct. 1991.

[3] F. W. Chang and G. A. Gibson. Automatic I/O hint generation through speculative execution. In *Operating Systems Design and Implementation*, pages 1–14, 1999.

[4] L. Felser. Personal communication, Sept. 2003.

[5] J. Griffioen and R. Appleton. Performance measurements of automatic prefetching. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems*, Sept. 1995.

[6] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988.

[7] L. B. Huston and P. Honeyman. Partially connected operation. *Computing Systems*, 8(4):365–379, 1995.

[8] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek. Mobile computing with the Rover Toolkit. *IEEE Transactions on Computers: Special issue on Mobile Computing*, 46(3):337–352, Mar. 1997.

[9] M. Kim, L. P. Cox, and B. D. Noble. Safety, visibility, and performance in a wide-area file system. In *Proceedings of the First USENIX Conference on File and Storage Technologies*, Monterey, California, Jan. 2002.

[10] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

[11] G. H. Kuenning and G. J. Popek. Automated hoarding for mobile computers. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 264–275, Saint Malo, France, Oct. 1997.

[12] T. Mann, A. Birrell, A. Hisgen, C. Jerian, and G. Swart. A coherent distributed file cache with directory write-behind. *ACM Transactions on Computer Systems*, 12(2):123–164, 1994.

[13] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, Lake Louise, Alberta, Oct. 2001.

[14] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, Feb. 1988.

[15] T. W. Page, R. G. Guy, J. S. Heidemann, D. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software – Practice and Experience*, 28(2):155–180, Feb. 1998.

[16] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 224–244, Copper Mountain Resort, Colorado, Dec. 1995.

[17] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Proceedings of USENIX Summer Conference*, 1985.

[18] M. Satyanarayanan. The evolution of Coda. *ACM Transactions on Computer Systems*, 20(2):85–124, May 2002.

[19] D. A. Steere. Exploiting the non-determinism and asynchrony of set iterators to reduce aggregrate file I/O latency. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 252–263, St. Malo, France, Oct. 1997.

[20] W. Vogels. File system usage in Windows NT 4.0. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 93–109, Kiawah Island, South Carolina, Dec. 1999.

[21] A. Westerlund and J. Danielsson. Arla – a free AFS client. In *Proceedings of the 1998 USENIX Conference, Freenix Track*, New Orleans, Louisiana, June 1998.

[22] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume leases for consistency in large-scale systems. *IEEE Transactions on Knowledge and Data Engineering*, 11(2):563–576, July 1999.