**CacheDAFS: User Level Client-Side Caching for the Direct Access File System**

The Harvard community has made this article openly available. **Please share** how this access benefits you. Your story matters

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Citable link</td>
<td><a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:25619461">http://nrs.harvard.edu/urn-3:HUL.InstRepos:25619461</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at <a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA">http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA</a></td>
</tr>
</tbody>
</table>
CacheDAFS: User Level Client-Side Caching for the Direct Access File System

Salimah Addetia

TR-14-01

Computer Science Group
Harvard University
Cambridge, Massachusetts
CacheDAFS: User Level Client-Side Caching for the Direct Access File System (DAFS)

A thesis presented
by
Salimah Addetia
to
The Department of Computer Science
in partial fulfillment of the requirements for the degree of Master of Engineering in the subject of Computer Science

Harvard University
Cambridge, Massachusetts
2001
CacheDAFS: User Level Client-Side Caching for the Direct Access File System (DAFS)

Abstract

This thesis focuses on the design, implementation and evaluation of user-level client-side caching for the Direct Access File System (DAFS). DAFS is a high performance file access protocol designed for local file sharing in high-speed, low latency networked environments.

DAFS operates over memory-to-memory interconnects such as Virtual Interface (VI). VI provides a standard for efficient network communication by moving software overheads into hardware and eliminating the operating system from common data transfers. While much work has been done on message passing and distributed shared memory in VI-like environments, DAFS is one of the first attempts to extend user-level networking to network file systems. In the environment of high-speed networks with virtual interfaces, software overheads such as data copies and translation, buffer management and context switches become important bottlenecks. The DAFS protocol departs from traditional network file system practices to enhance performance.

Distributed file systems use client-side caching to improve performance by reducing network traffic, disk traffic and server load. The DAFS client omits any caching. This thesis presents a user-space cache for DAFS called cacheDAFS with a careful design that avoids most bottlenecks in network file system protocols and user-level
networking environments. CacheDAFS maintains perfect consistency among DAFS clients using NFSv4-like open delegations. Changes to the DAFS API in order to add caching are minimal and results show that DAFS applications can use cacheDAFS to reap all the standard benefits of caching.
Contents

Title Page ................................................................. i
Abstract ..................................................................... iii
Table of Contents ......................................................... v
List of Figures .............................................................. vii
Acknowledgments ......................................................... viii

1 Introduction ............................................................. 1
  1.1 VI Architecture .................................................... 3
  1.2 DAFS Compared to a Network File System ................. 4
  1.3 A Case for Client Caching ....................................... 10
  1.4 Thesis Outline ...................................................... 12

2 Related Work .......................................................... 13
  2.1 Maintaining Cache Consistency ............................... 14
    2.1.1 Consistency Checks ........................................ 15
    2.1.2 Leasing ...................................................... 16
    2.1.3 Callbacks .................................................. 18
    2.1.4 Distributed Caching ...................................... 22
    2.1.5 The Consistency Spectrum ............................... 23
  2.2 Optimizing File IO ................................................ 24
    2.2.1 Copy Avoidance Across the Kernel/User Boundary . 26
    2.2.2 Combined Copy Avoidance .............................. 29
  2.3 User Space Communication .................................... 32

3 Architecture .......................................................... 35
  3.1 Choice of Consistency Protocol .............................. 36
    3.1.1 The cacheDAFS Consistency Protocol ................. 37
    3.1.2 Protocol Specifications .................................. 39
  3.2 Cache Design and Structure ................................... 43
  3.3 Changes to the DAFS API ...................................... 47
  3.4 Implementation Details ........................................ 48
3.4.1 Data Structures ................................................. 49
3.4.2 Delayed-Writes ............................................. 51
3.4.3 Asynchronous Read Ahead ................................. 53
3.4.4 Code Base .................................................. 53

4 Results ............................................................. 55
4.1 Measuring the cacheDAFS Data Path ......................... 56
4.2 Postmark Experiments ......................................... 60
  4.2.1 Varying the Read/Write Ratio .......................... 63
  4.2.2 Varying the File Set Size ............................... 68
4.3 Locality of Reference ......................................... 72

5 Conclusions and Future Work ................................. 75
5.1 Future Work .................................................. 76

Bibliography ....................................................... 78

A CacheDAFS API ................................................. 83

B Individual Operation Measurements for Postmark ............ 84
List of Figures

1.1 Possible DAFS Client Implementations .......................... 8
2.1 The Cache Consistency Spectrum ................................. 24
2.2 Copy Avoidance Techniques of Interest to this Thesis .......... 25
3.1 CacheDAFS Delegation State Diagram ........................... 40
3.2 Example Behavior of DAFS Delegations ......................... 42
3.3 Possible Architectures for cacheDAFS ............................ 43
3.4 Details of the Data Path in cacheDAFS ............................ 46
4.1 Possible DAFS Client Implementations .......................... 57
4.2 Relative Cost of Data Path Components .......................... 58
4.3 Data Path Component Costs .............................. 59
4.4 Varying the Read/Write Ratio With Postmark .................. 67
4.5 Varying the File Set Size ........................................ 70
4.6 Read Time For Changing File Set Size .......................... 71
4.7 Read Performance With Increasing Working Set Size ............ 73
Acknowledgments

A sincere thank you to my advisor, Margo Seltzer, for her advice and encouragement and for giving me the opportunity to see what research is like. It was an honor and pleasure to be your student.

A special thanks to Kostas Magoutis for introducing me to DAFS. Without his hard work and his DAFS server, the existence of cacheDAFS would have remained purely theoretical.

Working with the DAFS team at Harvard has taught me many things; I could not have asked for a more motivating or challenging group. I will miss all the fun we had (and yes, even the work).

Thanks to Richard Kisley at Duke University for his DAFS Provider, on which I have built cacheDAFS.

Last, but not least, a heartfelt thank you to my family for their love and support throughout my Harvard career. My sisters motivate me to do the very best that I can. Without my parents’ support, none of this would have been possible.
Chapter 1

Introduction

Distributed file systems have long been used to provide concurrent file sharing in networking environments. Recent trends show vast improvements in network and CPU speed [48]. Specialized storage devices serve to decouple storage from application servers. Applications have the ability to access high performance environments through virtual interfaces to networks. Traditional distributed file system protocols were not designed for such specialized environments and introduce many overheads that prevent applications from deriving full benefit from them.

The emergence of Network Attached Storage (NAS) has introduced highly available, fast file access that is decoupled from file servers. The combination of fast storage devices and reliable Gigabit speed interconnects results in a high speed, low latency local area network (LAN) environment for file sharing. Traditional distributed file systems such as Network File System (NFS) [44] were developed for unreliable, high-latency network environments and introduce overheads that are unacceptable in high performance environments. The design of network file systems that offer high
performance in new local file sharing environments has become very important.

Research in the area of User Level Networks (ULNs) addresses many of the performance issues encountered in the Gigabit network environment. This has led to the development of User Level Network Interfaces (ULNIs) or virtual interfaces to network adapters [2], bringing applications closer to the network. A ULNI permits applications to queue data transfer directly to the network adapter without kernel intervention by mapping internal interface registers into user-space. Virtual Interface (VI) Architecture [11] is a standard interface of this type. VI evolved from research in user-space communication.

Limitations of traditional distributed file system protocols and the evolution of memory-to-memory data transfer in user-space has motivated the design of a new file access protocol: the Direct Access File System (DAFS) [12]. DAFS maps file system semantics onto high speed, low latency NAS environments. Extending the use of memory-to-memory interconnects to the distributed file system, DAFS uses VI and similar transports for file access.

It is a well-known fact that client-side caching enhances file system performance. DAFS, however, has no provision for caching. This thesis argues that performance can be improved significantly by allowing the DAFS client to cache while still maintaining the client in user-space and providing perfect consistency.

The remainder of this chapter describes the DAFS environment and the way DAFS differs from traditional distributed file systems. Section 1.1 discusses the details of VI Architecture. Section 1.2 provides a description of DAFS and its differences from NFS, the most widely used distributed file system. Section 1.3 outlines the motivation
for adding client-side caching to DAFS.

1.1 VI Architecture

VI attempts to eliminate software overheads found in traditional network protocols by moving them into hardware. These overheads include error checking, reliability of reception, kernel involvement and context switching, buffer copies and CPU overhead for protocol processing on both the sender and recipient. While these software-based operations are necessary, they produce excessive overhead, preventing a system from taking full advantage of a high speed interconnect. These interrupt processing and message passing overheads can consume from 10 - 30% of CPU cycles. VI reduces this overhead to between 3 - 5% [2].

VI is a connection-oriented protocol designed with the assumption that the underlying network is highly reliable. Failure is catastrophic and messages are not retransmitted. The primary advantage of VI is to reduce network software overhead. This is done by removing the kernel from the common path of data transfer using Remote Direct Memory Access (RDMA).

VI provides two asynchronous modes of data transfer: traditional send/receive and RDMA. Send/receive transfers accommodate smaller control messages and transfer data inline with a message. RDMA data transfer avoids costly kernel traps and data copies by transferring data between pre-defined user-level buffers. Such buffers need to be registered with the network adapter prior to use. Memory registration involves the kernel and is typically performed once on buffers that are used repeatedly for data transfer. Involvement of the kernel is limited to setup and teardown of connections,
registration of buffers, interrupt processing and error handling.

A VI connection consists of a send queue and a receive queue. Applications may set up multiple VI connections and post descriptors specifying the virtual addresses of buffers used for memory transfer. These descriptors are processed asynchronously and marked with a status upon completion. An application may poll or block to determine when posted descriptors are complete. Completion queues may be used to collect the results from various data transfer queues.

Although DAFS generalizes to other VI-like memory-to-memory transports, this thesis focuses on a VI-based DAFS implementation, since this is the platform on which we operate.

1.2 DAFS Compared to a Network File System

DAFS departs from the traditional distributed file system in many ways, one of which is its potential to exist in user-space and derive full benefit from RDMA and ULNIs. Using RDMA, DAFS can bypass the kernel in the common data transfer cases, placing data directly into application buffers. This eliminates CPU overhead and data copies along the data transfer path. The cost is that applications are made aware of low-level transport details such as memory registration and descriptors through the DAFS API [13]. Arguably, DAFS parallels research in ULNs more closely than recent distributed file system research. Chapter 2 discusses ULNs and the layering of messaging protocols on virtual interfaces in more detail. This section takes a closer look at the DAFS protocol and its divergence from NFS, one of the most widely used distributed file systems. There are two factors that contribute to this divergence: a
difference in the target environment and the use of RDMA and user-level networking in transports.

The latest version of NFS, NFSv4 [47], targets wide area network (WAN) environments, which are known to have high latency and unreliability. DAFS is designed specifically for high performance LANs. Previous versions of NFS were stateless but NFSv4 introduces soft state into the server for leases (used to manage locks) and delegations (used for client caching of file data) as well as OPEN and CLOSE operations used to keep track of this state. The NFSv4 protocol also provides extensive guidelines for client-side caching (section 2.1.2 describes this caching protocol in more detail) for better performance over WANs. NFSv4 limits caching guarantees to common file sharing cases so that they do not interfere with locking and share reservations (for instance, client caching must respect mandatory locking when it is in effect). By contrast, the DAFS client is not designed to implement client caching, although the mechanisms to provide it are present through protocol aspects inherited from NFSv4. One of the goals of this thesis is to implement these mechanisms and measure the usefulness of caching.

DAFS servers manage session state consisting of up to three VI connections for each client. The main purpose of these separate connections is to separate control from data transfer by using the send/receive transmission method for control and RDMA for data transfer. The first channel, referred to as the Operation channel is a mandatory send/receive channel where clients send DAFS requests and servers send DAFS responses. An optional RDMA channel is used for server initiated bulk data transfer. The third VI channel is an optional Back Control channel where DAFS
requests are issued by the server and DAFS responses by the client. This channel is used for delegation (leases used for data caching) revocation, asynchronous notification of operation completion and other server-to-client requests. DAFS servers also maintain per session state such as flow control information, client byte ordering and other protocol information. NFS uses synchronous Remote Procedure Calls (RPC). NFSv4 also supports delegation and lease revocation when clients perform operations that result in sharing conflicts.

DAFS provides two modes of data transfer, direct and inline. Direct data transfer uses RDMA to transfer data between pre-registered buffers. Inline data transfer refers to sending data with the DAFS Request/Response along the Operation channel (implying a data copy). A client invokes a DAFS direct write by sending a request with a number of <virtual_address, length> pairs describing the local data buffers to be used for transfer. The server then performs an RDMA read using the RDMA channel and sends a response using the Operation channel to signify completion. A DAFS direct read is performed similarly by specifying destination buffers for the server to post an RDMA write. NFSv4, like previous versions of NFS performs all data transfer inline.

Two prominent bottlenecks in distributed file systems are synchrony and data copies within the protocol stack. DAFS allows applications to eliminate both of these with asynchronous data transfer and RDMA of data directly into application buffers. This is done by changing the file system interface seen by applications, thereby increasing programming complexity. DAFS exposes transport details such as memory registration, and completion groups. Standard NFS implementations perform data
copies across the user/kernel boundary and perform all operations synchronously. NFSv3 [41] introduces asynchronous writes, allowing an RPC to return before data has been safely committed on the server.

DAFS has three proposed architectures, two of which are user-space implementations layered on top of a user-level VI Provider Library (VIPL). The third architecture is a standard kernel file system implementation. A user-level implementation of the DAFS client (also referred to as the DAFS Provider) precludes the use of standard kernel file system support such as file descriptors, the VFS/VNODE interface and the buffer cache unless they can be incorporated without excessive overhead. Operating systems provide extensive support for general file system architecture used by NFS and other distributed file systems. A user-level DAFS client is not designed to use this infrastructure. Figure 1.1 shows the three possible DAFS client implementations\(^1\).

NFS implementations use the buffer cache for client caching. The buffer cache is a shared pool of buffers managed from within the kernel used to store file blocks. Prefetching by NFS clients is limited to what is provided by the kernel buffer cache. This is typical of distributed file systems but is not usually sufficient for applications such as databases that have knowledge of their file access patterns and cannot instruct the buffer cache to make use of them. DAFS permits applications to control prefetching into the server cache and supply hints for future file access. Data is available in the server cache for transfer to the client when needed. In this way, a DAFS application that does its own caching (such as a database) can use the server as an extension of its client cache.

Since DAFS is designed for a local file-sharing environment, fine-grained locking is

\(^1\)Source: Direct Access File System Specification, v1.0
of particular interest. NFSv4 provides byte range file locks that may be renewed periodically using leases and are revoked upon client failure/reboot, server failure/reboot, or a conflicting lock request. DAFS adds to these AUTORECOVER and PERSIST locks, which provide stronger locking guarantees. AUTORECOVER locks are meant specifically to allow automatic recovery from failures. A client holding this type of lock is given the opportunity to roll back operations if needed, to recover from failure. If a client fails for a long time, the server may perform roll back on an AUTORECOVER lock and release it for use by other clients. PERSIST locks require the participation
of the client for recovery. A PERSIST lock becomes breakable when the lease (a lease refers to a time period during which the client is allowed to hold the lock) expires or the server fails. If conflicting access then occurs or the client fails, this lock becomes broken. Unless the client recovers, reclaims the lock, performs necessary recovery operations and releases it, any attempt to acquire the lock will return an error.

NFSv4 provides a new COMPOUND operation designed for better performance in high latency networking environments. COMPOUND allows the client to pack a series of dependent requests into a single one and send them to be processed by the server without waiting for the results of each request. This, however, requires packing of the request by the client and parsing by the server. DAFS introduces request chaining instead. Dependent requests are sent in the normal way, but with special flags to identify that they are chained and only a single response is expected. This allows the client to send multiple requests without the need to wait for round trip time and processing time for each individual request. The number of chained requests a client can send is limited by the number of outstanding requests allowed by the server. The server is responsible for checking flags to process chained requests.

DAFS clearly diverges from traditional network file system protocols for the purpose of providing better performance. One area involving performance that has not been addressed by DAFS is that of client caching. The next section discusses why client caching would be beneficial to DAFS.
1.3 A Case for Client Caching

Many optimizations have been introduced into distributed file systems, one of which is client caching. Retrieving data from the server is much more costly than local cache access, thus the client derives performance benefits by satisfying requests repeatedly from its local cache. The use of prefetching can ensure that data is available locally when needed. DAFS is specifically designed for high performance but aside from inheriting caching aspects from NFSv4, does not discuss client caching in its protocol. This section discusses reasons for this omission and produces an argument for client caching.

Network speeds have increased to within an order of magnitude of memory bandwidth, sometimes even exceeding this [8]. Standard caching techniques involve the kernel buffer cache and require kernel context switches for read and write operations as well as data copies across the user/kernel boundary from the buffer cache into application buffers. RDMA from the server cache into application buffers eliminates the data copy. With smart prefetching mechanisms, the application can use DAFS to ensure that the data is always present in the server cache when needed. When the application experiences locality of reference, we can do better by allowing the client to cache and satisfy requests at memory speed. By placing the cache in user-space, we avoid the cost of going through the kernel for cache access. As a user-level library, this cache is also easily customizable.

The DAFS protocol involves client-controlled server caching. Clients can provide the server with hints or perform read ahead into the server cache so that memory-to-memory access is always available. But if the server is not able to keep all data
that clients require in its cache, the clients will experience disk speed. Additionally, most clients have access to large amounts of local memory they can take advantage of. Prefetching by the client into the server cache can be used in cooperation with client caching to produce better overall performance. Since DAFS is a protocol in its beginning stages, the idea of client caching can be incorporated and tested early.

Finally, client caching results in the need for a cache coherence protocol. Software overheads during data transfer are undesirable in the environment of Gigabit speed networks. Cache coherence must be maintained across processes, open file instances and nodes. For this reason, the DAFS API document does not consider client caching. A coherence protocol that does not incur overhead that masks its usefulness is needed. In traditional file systems, it has been shown that the overhead of maintaining strict cache coherence like that of the Sprite network operating system [38] does not incur a lot of overhead and is well worth the benefits it provides. We propose that client caching for such environments can be incorporated easily into DAFS without incurring high overhead for maintaining perfect consistency.

This thesis postulates that careful design of client-side caching in a user-level implementation of DAFS over VI can improve the performance of the file access protocol. The key aspects of this design are maintaining a cache of registered memory in user-space, taking advantage of DAFS operations and providing perfect consistency. We show that both file data caching and metadata caching are useful.
1.4 Thesis Outline

Chapter 2 gives an overview of work related to this thesis. A summary of cache coherence techniques considered for client-side caching in DAFS is presented. Other related work involves research in copy avoidance and performance issues in similar environments. Due to the broad range of this work, only those areas relevant to this thesis are covered here. Chapter 3 describes the design and implementation details of cacheDAFS. Chapter 4 discusses performance evaluation using synthetic microbenchmarks and macrobenchmarks over cacheDAFS. Finally, Chapter 5 concludes and discusses future work.
Chapter 2

Related Work

This thesis deals with the design and implementation of a client-side caching system for DAFS. Related work falls under two categories: cache coherence protocols that preserve consistency of data cached at clients and research in elimination of unnecessary overheads in distributed file systems and VLI-like environments.

The main concern in choosing a cache coherence protocol is how to provide the best consistency possible without inducing expensive software overhead. Different strategies have evolved based on the target networking environment and studies on file access patterns. An overview consistency protocols used by the most prominent and successful distributed file systems is presented in Section 2.1. The advantages and disadvantages of each are discussed and a consistency spectrum is defined to show the relative strength of consistency provided by each method.

A distributed file system involves the interaction of network and file I/O. In an environment such as DAFS, where performance is a major concern, eliminating unnecessary overheads in file and network I/O such as data copying across protection
boundaries is a major focus. Research has shown that careful cooperation of network and file I/O copy avoidance can improve performance [7]. Section 2.2 discusses data copy avoidance in networking and file I/O including combined approaches.

DAFS eliminates kernel context switch and data copy overheads by using VI-like transports and exposing transport details to the application. Research in user-space communication with ULNs deals with similar overheads and provides some insights for careful design of a caching layer for DAFS. Section 2.3 outlines relevant research with ULNs.

2.1 Maintaining Cache Consistency

Distributed file systems have been around for a long time and much work has been done in the area of improving performance. One of the most important techniques used to improve the performance of distributed file systems is client-side caching. Clients cache data in main memory to reduce server load, disk and network traffic. When multiple clients cache files, the problem of keeping copies of files cached on different clients consistent arises. A consistency protocol ensures that changes made to a file by one client are immediately visible to others caching the file.

Many consistency protocols have been developed over the years, sometimes trading strength of consistency for other advantages such as scalability. The strength of a consistency protocol (its consistency semantics) reflects how strongly the protocol enforces the idea that changes a client makes to a file must be visible to other clients. Ideally, a consistency protocol should provide perfect consistency: changes made to a file by one client are immediately visible to all other clients. A file system provides
UNIX semantics when modifications to a file are seen on the next read or write operation. Open-close or session semantics ensure that changes are visible to other clients as soon as the file is closed. In practice, network file systems provide a range of consistency guarantees from none to perfect consistency. Cache consistency protocols for LAN inspired client-server file systems fall almost cleanly into three categories: consistency checks, leasing and callbacks.

2.1.1 Consistency Checks

A client that is performing caching may be entirely unaware of other clients’ activities and the freshness of the file data and metadata that it caches. One simple mechanism for maintaining weak consistency is the use of consistency checks - periodic queries to the server about the freshness of cached data. This method is implemented in early versions of NFS. Although the NFS protocol specifies synchronous writes, the use of consistency checks loosens the consistency guarantees. NFS implementations cache attributes, directory information and file data in separate caches. The NFS protocol also mandates synchronous writes but this is too costly in practice. NFS implementations use delayed-writes and flush all dirty data synchronously on close. This is done for two reasons: to maintain consistency and so that write errors may be reported back during the system call. Delayed-writes are performed periodically (often at thirty second intervals) by a daemon that flushes dirty data. This means that file modifications or newly created files may not be seen elsewhere for at least thirty seconds. The NFSv2 write mechanism was replaced in NFSv3 with asynchronous writes and a COMMIT operation for flushing changes to the server disk. This method
reduces delay as seen by the client but loosens the consistency guarantee in favor of performance. An asynchronous write permits the client to return once the write is initiated instead of blocking until it is safely committed. The client is not certain that this write will complete successfully and it may be lost if the server crashes before commit.

Consistency checks do not enforce cache coherence. Modifications performed by other clients in between checks are not seen immediately. The disadvantages of the consistency check mechanism are many:

- Excessive consistency checks use up bandwidth and occupy the server

- Cache consistency is not guaranteed. Between checks, the client has no idea what changes have been made to the data it is caching

- Write-through delays are incurred even when they may not be necessary (i.e. no other clients are reading or writing a file). There is no distinction between file sharing cases and no opportunity to optimize for the common sharing cases.

Consistency checks are easy to implement but maintain the weakest form of consistency. We can do much better for DAFS by adding only a little more complexity.

### 2.1.2 Leasing

A lease, first defined by Gray [21], permits a client to have use of a file for limited time duration. There are different types of leases a server can grant a client, each allowing the client to service certain operations locally, on a cached file. There are two situations in which a lease may be terminated: the lease expires or the server revokes
it when another client requests a conflicting lease. Clients renew leases periodically with an explicit lease renewal or implicitly by performing operations on the file. The server maintains what is called *soft state* corresponding to leases given to clients. This state need not be recovered when a client or server crashes; the server simply waits for the leases to expire and a client repeatedly attempts to re-establish the lease. Among caching mechanisms that use leasing are NFSv4 and Not Quite NFS (NQNFS) [28]. Leases provide the advantage of easy recovery at the expense of the additional overhead of lease requests and renewals.

NFSv4 refers to file data-related leases as *open delegations* (in NFSv4, the term *lease* refers to lock-related leases). There are two types of open delegations - the read delegation and the write delegation. These delegations are maintained by the server and given out to clients when an OPEN operation is performed. A read delegation may be given to several clients and represents a guarantee that no other client has opened the file for writing. A client may cache reads when it holds a read delegation. A write delegation is given to only one client at a time and it guarantees that no other client has opened the file. A client holding this delegation may cache both reads and writes. Delegations must be coordinated with lock leases and the combination of messages required for an application to ensure perfect consistency results in a potential for high traffic. A delegation is recalled or revoked as soon as another client performs a conflicting operation on the file in question (attempts a conflicting sharing access). Upon recall, the client must flush modified state to the server and return the delegation. Attributes and directory information are cached using consistency checks as in previous versions on NFS. The NFSv4 server is no longer stateless, but maintains
soft state for the purpose of easy recovery at the cost of lease traffic. The addition of the open delegation serves to reduce traffic to the server by allowing a client to service OPEN and CLOSE requests locally in common file sharing situations and avoid consistency checks on metadata. The delegation and locking constructs defined in NFSv4 can be used to maintain perfect consistency of cached data.

The DAFS specification is inherited from NFSv4. It already contains the framework for implementation of caching with perfect consistency. Delegations by themselves are very simple and easy to implement. These factors make it ideal to implement delegations as the caching mechanism for DAFS. We use delegations for caching in cacheDAFS with a slight variation that makes delegations 'infinite' and more like the callbacks described in the next section.

2.1.3 Callbacks

The callback mechanism permits a client to perform certain operations (such as reading or writing from the local cache) without violating consistency. The term callback represents a promise that the server will notify the client when conflicting access requires it to invalidate its cache, write back dirty data or do both. A client obtains a callback from a server upon opening a file. The server must keep track of open files and the clients caching them in order to detect consistency conflicts and inform clients when necessary by revoking callbacks. This is a server-oriented caching method that introduces hard state into the server. Involving the server in maintaining cache coherency has one main disadvantage - it introduces the need to recover server state upon crash from clients or non-volatile memory. Explicit OPEN and CLOSE
operations on files are needed to allow the server to keep track of changes in file sharing state. The client is also required to service callbacks. The advantages of the callback mechanism are that it reduces network traffic and server load by avoiding unnecessary consistency messages (for example lease renewals). The server has control over enforcing consistency. Write-backs of dirty data are forced only when a sharing situation arises that would result in a consistency violation. Among callback-oriented caching schemes are Sprite [35], Spritely-NFS [49], AFS [46], Coda [45] and DFS [25].

The Sprite network operating system offers perfect consistency using callbacks. Sprite permits read-only sharing where many readers may cache the same file or single-writer sharing where a single writer may read or write the file while caching it but only if other clients are not accessing it. A client holding a read callback on a file is guaranteed that other clients with the file open are only performing reads. If a consistency conflict arises (for example the file is opened by a writer), the server notifies clients by revoking callbacks. Write callbacks are a guarantee that no other client is reading or writing the file. Single writers may cache dirty data and usually write back to the server using thirty-second delayed-writes. This results in periodic write traffic to the server but creates a window of delay during which file data may be modified many times without write back. Short-lived temporary files may be deleted without the need to write back at all. To avoid accessing stale data, a single writer checks a file version number on every open to see if the file has been modified by someone else since the last close. The server also keeps track of the last writer of a file. If a client other than the last writer opens the file, the server sends a callback to the last writer, requiring all dirty data to be flushed. Server state consists of the
sharing status of open files. Upon crash, this data is recovered from clients that are interested in sending their open state to the server to reclaim callbacks.

Spritely-NFS is an adaptation of Sprite’s strict consistency guarantees to NFS. The mechanism involves small changes to the NFS protocol - the addition of OPEN and CLOSE operations, introduction of server state and a callback RPC that must be serviced by the client. The implementation of Spritely-NFS does not include crash recovery. NQNFS is a leased caching scheme adapted to NFS that provides stronger consistency guarantees with the advantage of soft server state. NQNFS borrows ideas from Spritely-NFS but uses three kinds of leases instead: multiple readers, single writer and non-caching. In the single writer case, both Spritely-NFS and NQNFS adopt the policy of thirty-second delayed-writes. These two caching schemes show that strong consistency guarantees can be added to NFS without much complication of the protocol or additional overhead. Spritely-NFS performs much better than NFS or slightly worse depending on the benchmark used.

The Andrew File System (AFS) also uses callbacks to enforce consistency but provides weaker open-close semantics for consistency. Modifications done by a client to a file are only visible to other clients once that file has been closed. AFS sacrifices strength of consistency for better scalability. A workstation caches entire files whenever possible and attempts to keep frequently used files in the cache for use during disconnected operation. Dirty data is written back to the server on close only. This method requires much less client-server communication but has the disadvantage that a client may cache and modify a file for a long time before it closes and flushes changes, thus making other clients aware of them.
DCE/DFS extends the ideas of AFS but provides finer-grained cache consistency using tokens. Whenever a client performs a write to a file, all other clients see the effect immediately. Each token is much like a callback in that it provides a guarantee that the client may perform certain operations on the file without violating consistency. Tokens are available for operations on file data and metadata. A token manager on the client is invoked on every call into the VNODE interface to ensure the client possesses all tokens necessary to perform the operation. A partial ordering of locked resources prevents deadlock of clients requesting tokens from the server. Tokens are required for read, write, open and lock operations for both file data and metadata. Finer grained consistency is permitted by allowing tokens to specify byte ranges for the data they represent. Read and write metadata tokens are always incompatible (signifying a sharing conflict). Read and write tokens for data or locks are incompatible if their byte ranges overlap.

While token-based callbacks introduce additional complexity and overhead into the callback protocol, callbacks by themselves are simple and incur low overhead. The Sprite consistency protocol provides perfect consistency with low overhead. This thesis incorporates the idea of callbacks by making delegations infinite. The protocol of Sprite would be equally suitable for cacheDAFS but we use delegations, since these are already a part of the DAFS specification and provide the same consistency guarantees for common file sharing cases. However, we adopt the policy of delayed-writes as implemented in Sprite. Delegations also provide us with the ability to cache some metadata without the complexity of a token manager on the client.
2.1.4 Distributed Caching

Another set of coherency mechanisms have evolved for shared file systems, distributed shared memory and clustered systems. Cooperative caching is based on the premise that servers are expensive, specialized resources and end up being costly bottlenecks. File systems such as NFS originally designed for LANs tend to waste bandwidth with broadcasting and write-through mechanisms [53]. Moreover, the failure of an important server is costly. Cooperative caching is designed to take advantage of low latency, high-speed networks and high-powered clients with disk space, memory and CPU cycles to spare. Solutions include decentralization and symmetry between clients and the server [14]. Not all cooperative caching schemes are truly distributed and many require servers to determine which client holds cached data before contacting that client to obtain the data. Fully distributed versions of cooperative caching require the maintenance of globally replicated data structures for quick location of cached file data. To avoid false positives, the data structures must be updated frequently and transferred between clients in a way that does not generate excessive network traffic. Often servers are responsible for maintaining these data structures, invalidating caches and forwarding requests to the appropriate cache locations. To allow client-to-client data transfer, clients must be trusted or additional authentication overhead is required.

Clustered and database environments use caching for distributed shared memory. The protocols used here are often based on locking data blocks or objects and are typically designed for scalability and high availability. Such sharing environments can afford to trust clients to access data because clients themselves are trusted entities
or members of a clustered environment. Clients hold locks on data items and some
distributed or centralized form of coherency and deadlock avoidance is maintained.
One example is the method used in Frangipani [51] for maintaining cache coherence
among servers. A lockable segment is one of a log file, unallocated data block, direc-
tory or symbolic link. Locks are of two different kinds: read locks (permission to read
and cache data) and write locks (permission to read, write and cache data). Release
of a write lock implies a write back of modified data. Each server is required to use
a specific mechanism to acquire locks and avoid deadlock.

Cooperative caching with DAFS is beyond the scope of this thesis. In general,
DAFS clients are not trusted entities and authentication may be required for client-
to-client data transfer. For DAFS to take full advantage of distributed caching tech-
niques, it would require client-to-client VI connections as well as client-to-server VI
connections.

2.1.5 The Consistency Spectrum

It is useful to look at the range of consistency guarantees offered by each file
system and their relative strengths. Figure 2.1 depicts the range of possible guar-
antees from no consistency (i.e arbitrary caching) to perfect consistency and where
each mechanism discussed in the previous section falls on the spectrum. Note that
Web caching is placed on the low end of the scale. Web caching methods provide
no way of enforcing consistency between cached copies of data. This is primarily
because protocols for the Internet were established without considering consistent
caching. Typically, proxies or browsers that are caching pages perform consistency
checks. HTTP headers can be used to provide guidelines about when to check for data freshness but obeying these guidelines is not mandatory.

Figure 2.1: The Cache Consistency Spectrum

2.2 Optimizing File IO

File systems are designed to operate within the kernel. The kernel provides security, fairness, resource allocation and built-in support for file systems. UNIX systems provide a file system manager and a generic VFS/VNODE interface that is implemented by local and network file systems [32]. An application performs data I/O (including file I/O) using the standard POSIX read/write interface. File data is cached in a shared buffer cache in blocks and is accessible using a generic file descriptor. If a file block is referenced and not resident in the buffer cache, the Virtual Memory (VM) system evokes the appropriate file system pager to retrieve the block either from the disk or network. UNIX systems use separate buffers called mbufs for network I/O and IPC socket communication. This means that at least one copy from kernel to user-space is required for file data transfer.

In the recent past, network speeds have been increasing an order of magnitude
every ten years. They are within an order of magnitude of memory speed and in some cases may exceed this. CPU speeds are increasing 50-100% every year. DRAM, most commonly used for main memory, is increasing in speed much more slowly - at the rate of 50% every decade [8]. For these reasons, it has become detrimental to performance to have data copy overheads while performing I/O. For small data sizes, data copies are often efficient but as the data size grows, file system performance slows.

![Diagram](https://via.placeholder.com/150)

**Figure 2.2: Copy Avoidance Techniques of Interest to this Thesis**

Much research has been devoted to optimizing the data path throughout the network and file system protocol stack in order to improve performance. This research in *copy avoidance* is extensive. The following subsections discuss only a subset of the research in this area. Figure 2.2 depicts copy avoidance techniques of interest to this thesis, each of which is discussed in more detail during the remainder of this
Chapter 2: Related Work

chapter. Note that DAFS over VI falls into the intersection of all categories since it is a user-level file system that avoids both network and file data copying.

General research in copy avoidance across the kernel/user boundary in file systems is outlined in Section 2.2.1. Combined copy avoidance in both network and file I/O produces clear performance benefits, especially when both are combined. Section 2.2.2 discusses two methods of eliminating copies throughout the protocol stack.

DAFS may be implemented as a kernel-resident file system. However, this thesis focuses on a uDAFS (user-level DAFS) client with a user-level cache. A user-space DAFS file client operates over the user-level VI PL. Many VI predecessors exist along with user-space communications models. Research in the area of ULNs exposes many of the issues that are encountered in a user-level implementation of DAFS. Section 2.3 discusses the philosophies of ULNs and the overheads they encounter. While these are networking and not file system protocols, many lessons can be learned from previous work in user-space communication.

2.2.1 Copy Avoidance Across the Kernel/User Boundary

Standard application I/O interfaces suggest copy semantics. Once applications initiate I/O operations, they are able to reuse buffers immediately because data has been copied to system buffers for transfer. Network file systems, additionally, want to cache data and allow buffers to persist long after data transfer has taken place.

One method of avoiding copies while performing I/O is to change the semantics seen by applications. Other well-known data passing semantics for I/O systems are move semantics and share semantics \cite{8}. Move semantics are analogous to moving the
buffer along with the data. Once data has been transferred, the buffers are removed from the application’s address space. Share semantics require the application to share the buffer during I/O operations by not modifying it. This is typically implemented by removing write permissions from pages used for transfer. Move semantics are not useful for file systems that require caching of data since buffers should persist in the cache. Share semantics do not allow modification of buffers, permitting at best read-only sharing. An application that uses the direct variants of the DAFS asynchronous interfaces essentially obeys share semantics. Registered buffers are used to queue asynchronous data transfer and the application polls or waits for completion before reusing the buffer. A third classification of data passing semantics is weak move. This is an optimization whereby buffers used for transfer are volatile, meaning the system does not enforce their immutability during transfer.

A well-known and widely supported copy avoidance technique for file I/O is the \texttt{mmap} interface supported by most UNIX systems. The \texttt{mmap} system call maps file data into a contiguous region of the application’s address space. Applications that use \texttt{mmap} to create a shared mapping of a file can make changes that are immediately visible to others. The main disadvantage of \texttt{mmap} are that it requires a contiguous region of address space and the initial mapping is not only expensive but may require copying. For a user-level file system cache in a high performance environment, the overhead of the \texttt{mmap} system call may be too much.

Research has been conducted in the area of sharing the buffer cache at user-level without too much kernel involvement. This research in service decomposition [29] introduces the cache segment abstraction and a trusted cache manager to reduce the
overhead of memory mapped shared file access. Cache segments consist of a descriptor region and an array of slots. File data blocks are stored in descriptor regions and may be accessed rapidly to determine which data blocks are present in the cache. A cache segment manager in a separate protection domain is responsible for mapping segments from its cache into the address spaces of applications when needed. In addition to this, it maintains mappings from the system buffer cache to its own address space for all blocks accessed by its clients.

Whenever an application requires file data, it first checks the cache segment descriptor section. If the data is present, it may be accessed immediately from the corresponding slot. If not, a miss occurs and the application interfaces with the cache manager. Two types of misses are possible: the application misses and the cache manager finds the block in its cache or the application misses and the manager in turn misses. In the former case, the cache manager maps the block from its cache to a free cache segment slot chosen by the application. In the latter case, the block must first be mapped from the buffer cache to the manager’s cache and then into the application’s free slot.

This infrastructure is based on the premise that the cached case is common and expensive misses that require interaction with the buffer cache are rare. The cache manager fetches and writes data only when requested by applications, which poses a consistency problem not only between applications using the cache manager but also with other applications using the buffer cache.
2.2.2 Combined Copy Avoidance

A significant amount of research has been devoted to eliminating data copies within the kernel. However, many file systems still perform copies across the kernel/user boundary into application buffers. There are two reasons for this: applications do not request data aligned along page boundaries and applications expect copy semantics from I/O interface implementations. Research has shown that combined copy avoidance in both the network and file I/O protocol stack results in better overall performance. Moreover, techniques must be chosen carefully so that they do not conflict or provide incorrect behavior. This section discusses two separate frameworks that allow caching of file data without data copies in the file and network protocol stack.

Emulated Copy and Header Patching

Genie is an I/O framework developed at CMU [8]. Work on Genie shows that file mapping can be combined with emulated copy and header patching to eliminate all copies and preserve standard I/O interfaces. These techniques require kernel support and modification of the VM system and networking layer. As with other copy avoidance techniques that use the VM system, preserving the standard UNIX read/write interface requires some tricks.

Emulated copy is used to provide weak move semantics for network data transfer. When data is passed to the network interface, the ideal situation is for buffers to remain unmodified until transfer is complete. An application, however, cannot always be trusted to leave buffers untouched. A common method for dealing with this is
to remove write permissions from application buffers while they are being used for transfer, which requires kernel involvement. They are promoted to system buffers and Copy On Write (COW) is performed if the buffers are touched. Zero-copy NFS uses COW to avoid copies from application to system buffers across the user/kernel boundary. COW is an expensive operation that can be optimized through page swapping. Genie aligns system buffers to application buffers wherever possible instead of copying. It also implements a more efficient Transient Copy on Write (TCOW) that uses reference counts to avoid copies.

The file system may use memory mapping to interact with a networking interface that implements emulated copy and provide I/O without data copies. To transfer data directly from memory mapped file regions, assuming move semantics, the system simply de-allocates and unmaps pages after transfer. If there are outstanding references to a page, copying and swapping is performed as needed. In order to input data into mapped file regions, header patching is required because network data is preceded by a header that the application mapping files does not want to see. The network layer inputs data at its preferred alignment and is required to peek at a header to determine where file data begins. After patching the first few bytes of data at the header location, the file is mapped where the application expects to see it.

IO Lite

IO Lite [39] is designed to unify sharing among the different UNIX I/O systems and avoid all copies along the common data path. IO Lite provides improved performance at the cost of a new API, departing from the standard POSIX read/write
I/O interface. Backwards compatibility is provided, but at the cost of data copies. This approach to eliminating copies is not uncommon and has been adopted by other systems providing high performance including DAFS.

Applications using IO Lite do not have control over the location or layout of data, nor do they define its alignment. Data is stored in immutable buffers that are page-aligned and always a multiple of the page size. Ordered sequences of data are represented by buffer aggregates, which are directed acyclic graphs (DAGs) of buffers. While their corresponding buffers are immutable, buffer aggregates are not. When modifications are made to data, new buffers are allocated to store the modified data. The buffer aggregates are updated to point to new data where changes have occurred and old data where they have not, thereby avoiding copies. Aggregates persist while references to them still exist. Concurrent data sharing is supported by making buffers volatile and passing them by reference between different protection domains.

IO Lite provides a user-level library with primitives IO_L_read, IO_L_write, and IO_L_mmap, among others. Applications never modify aggregates directly; instead they use the user-level IO Lite library. Kernel support is required for security and fairness. The IO Lite file cache consists of mappings from \(< start\_address, length >\) pairs to \(< file\_offset, length >\) pairs. Unlike the standard buffer cache, file data is not stored in contiguous blocks. Cache insert and lookup operations are invoked by IO_L_write and IO_L_read and require system calls to a kernel module that manipulates the cache. A cache lookup operation returns a sequence of \(< location, length >\) pairs.

IO Lite is implemented using fbufs data structures that provide weak move semantics for network I/O [16]. Several difficulties are encountered when implementing
the IO Lite API for a UNIX system. Data copying is needed to support IO Lite’s version of `mmap`, the kernel must be modified to use fbufs for network I/O and separate support must be provided for each file system. Supporting buffer aggregates in UNIX file systems is difficult because file data is page aligned, page sized and modified in place. In addition, the file system is strongly tied to the VM system. Applications that use IO Lite must use a special programming model if they need to keep data read in intermediate data structures and avoid copies. If data access is complex such as random access to large matrices, the application may be better off using another alternative such as `mmap`.

### 2.3 User Space Communication

The VI industry standard came into existence after much research in virtual interfaces for user-space network communication. VI deals with various overheads encountered in message passing with a virtual interface to the network adapter, asynchronous data transfer and RDMA using virtual addresses. DAFS enables the use of these features in file systems by exposing them to applications while providing the necessary file system abstractions. In order to provide a useful caching component for DAFS, cacheDAFS must either expose the same details to applications or hide them by providing a standard interface without degrading performance. This section presents some of the issues of user-space communication. Although mainly used for message passing systems, ULNs and the implementation of thin protocols over them provide insights for careful design of a user-level file system.

User space communication is motivated by recent improvements in network tech-
Chapter 2: Related Work

Networks are evolving into fast, switch-based interconnect and intelligent network boards with DMA capabilities and the ability to do network protocol processing previously performed by the CPU. Hardware and wire latencies are extremely low, while software overheads are orders of magnitude larger and can prevent an application from seeing wire speed data transfer rates [40]. A recent study on user-space communication protocols evaluated on a Myrinet (1.28 GBps) cluster [2] shows a latency of 5 - 17µs for small messages. Highest bandwidths are achieved using RDMA and true zero copy for both send and receive operations. The most common sources of overhead in traditional communication software are found to be traps into the kernel for security purposes, buffer management and alignment, multiple data copies of messages and synchronous messaging.

The challenge of user-space communication is to provide sufficient amount of functionality for the application without incurring any of these overheads. One example is the improvement of Fast Messages [40] over Active Messages [52]. Active Messages are based on the idea of transmitting the address of a user-level message handler at the head of a message. Fast Messages are based on similar handlers, but provide reliable, in-order delivery and decoupling of the processor and network. If the communication layer does not provide enough functionality, applications will need to add what is missing possibly at a higher cost, reintroducing the overheads removed to make communication efficient in the first place. In order for applications like DAFS to provide data transfer close to wire rates they must adopt the same practices as the communication layers beneath them. An example of this is shown by the implementation of Sun RPC on SHRIMP [6]. Overheads of the original Sun RPC protocol are
progressively removed by eliminating context switches with the use of virtual memory mapped communication, reducing data copies in the RPC protocol stack from six to two and finally customizing RPC to SHRIMP. The result is a reduction in round trip null RPC latency from 33\(\mu s\) to 9.5\(\mu s\). Further work on user-space communication can be found in [5], [17], [50].
Chapter 3

Architecture

CacheDAFS is designed to operate at user-level, eliminate kernel overheads, provide perfect consistency and export the standard POSIX API. As a user-level library, cacheDAFS can be extended easily to support application-specific caching policies. For example, the application has specific knowledge of its access patterns, it can extend cacheDAFS (much more easily than it could the kernel buffer cache) to incorporate a different replacement or prefetching policy. By designing cacheDAFS as a user-level cache with a standard interface, we eliminate any overhead that requires kernel interaction. We do not change the interface seen by the application or require hooks into the kernel to protect and adjust buffers. Although the current cacheDAFS implementation has a synchronous interface, an asynchronous one can easily be built on DAFS operations.

This chapter describes the architecture and design of cacheDAFS. Section 3.1 gives a detailed description of the cacheDAFS caching and consistency protocol, the motivation for choosing it and the aspects of the DAFS protocol that are used to
support client caching. Section 3.2 describes the possible architectures for cacheDAFS and the rationale for implementing it as a DAFS application. Section 3.3 outlines the changes to the DAFS API required to implement cacheDAFS. Finally, Section 3.4 describes implementation details.

3.1 Choice of Consistency Protocol

The biggest concern in providing caching with perfect consistency is whether the overhead of the consistency protocol will negate the benefits of caching. The overhead of a consistency protocol is largely based on the traffic it incurs and the amount of processing required on the client and server. We design client caching for a file sharing environment. Studies [3] have shown that the overhead of a perfect consistency protocol - that of the Sprite Network Operating System - is not significant in standard file sharing environments except when write sharing occurs. Write sharing is very infrequent in these environments, making locking at the file level (delegations, callbacks or leases) sufficient. NFSv4, like Sprite, is designed to optimize the common file sharing cases of multiple-reader and single-writer sharing. DAFS also operates in other environments, such as the database environment. Here write sharing is frequent and finer granularity of sharing is required. Dealing with maintaining consistency in such environments is beyond the scope of this thesis.

Large caches are effective in absorbing read traffic [4]. Approximately 60% of read operations and 10% of write operations are absorbed by caches. Most write traffic is due to the need to protect freshly written data in the cache from being lost due to a crash. Since main memory access time is still faster than most network accesses, a
client cache with a full consistency protocol for DAFS would be useful for satisfying both read and write traffic locally. A DAFS client that also caches would be able to make use of asynchronous RDMA or server-initiated RDMA to speed write traffic. For instance, the server can keep track of virtual addresses of buffers that the client has cached. Dirty data may be written back without client intervention, minimizing latency as seen by the client. In this thesis, we use asynchronous client-initiated delayed-writes since our platform does not support RDMA reads for server-initiated flushing.

The DAFS specification already incorporates some features that can be used to implement coherent caching. These features are: NFSv4-like delegations, a callback channel for back control messages from server to client and maintenance of open file state on the server with the use of open/close operations.

3.1.1 The cacheDAFS Consistency Protocol

The consistency protocol we chose for cacheDAFS is that of NFSv4 with a slight modification that makes delegations infinite. Infinite delegations are similar to Sprite callbacks. In our variation of open delegations, the client assumes the delegation is valid until it receives a delegation recall from the server. This eliminates the need for delegation renewal and makes implementation easier since we do not deal with crash recovery (described in Chapter 2). It is easy to add implicit or explicit delegation renewal, which makes it unnecessary for the server to maintain delegation state.

The client may cache data for as long as it holds a delegation. Cached data is not flushed on close, neither is it revalidated on open. We use periodic asynchronous
writes initiated by a separate thread to update changes on the server or write only when necessary - when a recall is received from the server. This is similar to the behavior of NQNFS and Sprite. If a client holds a write delegation, periodic flushes are sufficient and help to avoid flushing large amounts of data when the client finally receives a delegation recall. A DAFS delegation recall consists of a callback message from the server sent via the back control channel. This message contains only the file handle of the recalled file, thus the client is responsible for deciding whether to flush and invalidate or do both. Upon receiving the callback, we implemented the client to do the following:

- Invalidate cache blocks associated with the file
- Flush any dirty blocks
- Send a close operation to the server, if there is one outstanding

Sprite makes use of both callbacks and version numbers to deal with caching writes. We avoid version numbers and use only callbacks, which means that the last writer to a file cannot reuse its cache blocks if it closes the file. A DAFS callback specifies only a file handle while a Sprite callback can specify whether to invalidate, flush or do both. Thus, when cacheDAFS receives a callback, data is flushed and invalidated. On the next OPEN, the file cache is not used even though there may be valid data in it. There is only one case, however, where this happens: a writer has been caching dirty data and its delegation is recalled because of a conflicting open by a reader. This writer could potentially flush data, and still reuse its cache if it reopens the file and finds the version number sent by the server and the one it has
corresponding to the cache blocks to be the same.

3.1.2 Protocol Specifics

CacheDAFS is a DAFS application that does not require any changes to the underlying DAFS protocol. We incorporate caching using delegations, which already exist in the DAFS protocol and are based on the NFSv4 notion of delegations. The NFSv4 specification provides two methods for caching: *share reservations* and *locks*. Caching with locks relies on the premise that clients are well behaved. A lock on a data region represents the right to read or write that region. If the client caches data corresponding to a locked region, it must validate this data when it first locks the region and flush any modifications before releasing the lock. Share reservations use delegations. A client must use open and close operations (requesting read access, write access or both) for correct functioning of share reservations. DAFS inherits aspects of NFSv4 that allow both of these schemes to be implemented. This thesis implements delegations only for caching. These delegations behave the way NFSv4 delegations (described in Section 2.1) do, except that they do not expire. It is the server’s responsibility to revoke delegations when a sharing conflict arises.

Use of delegations transfers the responsibility of managing file sharing state to the server. However, the client is required to service some operations locally when holding a delegation for a file. These are as follows:

- Open operations are serviced locally. If there is a request for conflicting access (i.e., the client holds a read delegation but the application asks for write access) the open is sent to the server
• Close operations are serviced locally. The client sends an outstanding close operation to the server when the delegation is revoked or the file is no longer in use or when an operation that needs to go to the server is performed.

Any open with the flag 0_CREAT must be sent to the server. A remove operation is also always sent to the server. Upon remove (or another operation that needs to be sent to the server), if a local close has been performed, the client first sends a close operation followed by the remove.

Figure 3.1 summarizes types of delegations a client can hold and illustrates the conditions under which the client will lose a delegation or need to acquire a different one.

![Diagram](image)

**Figure 3.1: CacheDAFS Delegation State Diagram**

When the client holds a write delegation, it is delegated responsibility for changes to the file. This includes maintaining (and thus caching) size and change attribute information. Moreover, the client is guaranteed that no one else will be modifying the file. Thus cacheDAFS caches the file name in a local name cache and attributes
in a VNODE-like structure. This effectively serves as a name/attribute cache for that file while the write delegation persists. Similarly, when the client holds a read delegation, it does name and attribute (size, in particular) caching. This is allowed because the client is guaranteed that the file will not be modified while it holds the read delegation. No modifications may be made, however, to attributes when holding a read delegation.

A client that holds a write delegation is also safe from having the file renamed or removed while it is caching metadata. In the event that one of these happens, the client’s delegation will first be recalled. This means the client literally owns the file and may cache data as well as metadata. If another client wishes to get attributes for this file, the server sends a get_attr callback to retrieve attribute information from the client holding the write delegation as described in the DAFS specification. Once a delegation is revoked, all name and attribute access must go to the server. Aside from metadata cached when the client holds an open delegation, cacheDAFS does not perform other metadata caching.

Figure 3.2 shows an example of the behavior of delegations. Client 1 acquires a DAFS open write delegation for a file and proceeds to read and write data, caching as it goes. While holding this delegation, Client 1 may perform open and close operations locally as well. When Client 2 opens the same file to read, this results in a sharing conflict detected by the server. While Client 2 blocks, a callback is sent to Client 1 and all dirty data is flushed to the server before Client 2 receives a response. Since a sharing conflict has arisen, neither client now holds a delegation.

The overhead of enforcing consistency is low when this kind of situation does not
Figure 3.2: Example Behavior of DAFS Delegations

occur often or involve too many clients. The worst case occurs when many clients are reading a file and all hold delegations. A single writer can then attempt to open the file and require all delegations to be recalled before anyone can continue to access the file again. We were not able to measure the cost of revoking delegations as it scales with the number of clients since the server implementation used here did not handle more than three clients.
3.2 Cache Design and Structure

This section describes in detail the architectures considered for cacheDAFS and the reasons for choosing an implementation that places the cache entirely in user-space.

Figure 3.3: Possible Architectures for cacheDAFS

![Architectures Diagram]

Figure 3.3 illustrates the possible architectures for cacheDAFS. Architectures 2 and 3 are possible without any change to the DAFS API or DAFS protocol. Applications that use the DAFS API are likely to provide their own buffer management. DAFS operations require the application to hand in buffer handles and descriptors used to perform RDMA operations directly into buffers. Caching would require the Provider to maintain its own registered buffers and copy from these into application buffers when necessary, which is counterintuitive to what the DAFS API suggests.
Chapter 3: Architecture

Architecture 3 illustrates the use of the kernel buffer cache for uDAFS. Traditional file systems use the kernel buffer cache for file data caching. In order to allow the use of the buffer cache in a user-level implementation of DAFS, it is necessary to integrate with the kernel file system manager, which introduces an awkward architecture into uDAFS. Also, it would reintroduce overheads such as context switching and data copies across the kernel/user boundary that user-space communication is designed to avoid. Another option is to map buffers into user-space as described in Section 2.2.1. We must then find an inexpensive way to share these buffers between processes and keep them synchronized with the kernel buffer cache. For these reasons, we propose an architecture with a user-space cache for uDAFS. A kernel implementation of the DAFS Provider would lend itself easily to the use of the buffer cache. However, a user-level implementation would require context switches on every buffer operation to lock and use buffers or the use of \texttt{mmap} to map pages into an address space accessible by the Provider and applications. Kernel support would be required to perform data transfer and registration of buffer cache buffers. This defeats the purpose of the user-level implementation of DAFS, which is to make use of memory-to-memory data transfer directly into application buffers for faster file access.

A decision not to use the buffer cache also has several disadvantages. We cannot take advantage of built in kernel support for file systems and the ability to share cached data with other local and distributed file systems using generic file descriptors. Thus the cache we implement consists of user-space structures similar to those found in the buffer cache and is implemented as a linkable library. Each client that links the library has its own cache.
Chapter 3: Architecture

Since this thesis focuses on performance we implement the cache as a user-space cache layer called cacheDAFS situated on top of the DAFS Provider as illustrated in Architecture 1. The cache layer will take advantage of memory registration and RDMA transfer provided by DAFS and can export one of two interfaces: the POSIX I/O interface for standard applications or the DAFS API with hidden caching. This thesis implements the former version at the cost of a data copy between cacheDAFS pre-registered buffers and application buffers. Chapter 4 analyzes the overhead of this data copy and its significance when applications show locality of reference.

At the cost of small changes to the DAFS API, which already departs significantly from the standard file system API, we implement a simple cache at user-level. CacheDAFS exports the POSIX API for applications that do not manage their own buffers and expect copy semantics. Data is transferred in cache block sizes (this size is adjustable but the default size is 16K) from the server using DAFS operations and a data copy is performed into application buffers. Some DAFS applications such as databases and Web servers manage their own caches. Operating over cacheDAFS, they will not benefit from double buffering, but they may still use the cacheDAFS POSIX API, with the cache disabled. In this case, cacheDAFS serves simply as a transparency layer: data transfer is copy-free and application buffers are registered on the fly.

The one main issue that is not addressed by this design is that of sharing the cache across processes. DAFS clients are often servers to Web clients. This means that we do not often expect more than one application to run over the DAFS client, so we do not consider sharing between applications here. However, sharing between
applications is an outstanding issue and would require kernel support for management and protection of shared buffers.

The architecture of cacheDAFS is shown in more detail in Figure 3.4. This figure illustrates exactly how the different APIs are used and where the data copies occur. Note that writes are performed inline and necessitate two copies. Our CLAN platform does not support RDMA reads, thus the server supports only DAFS inline writes.
3.3 Changes to the DAFS API

The DAFS API is different from standard file APIs for the purpose of better performance. It is not intended to integrate with existing file access infrastructures. The DAFS Provider presently is not designed to perform read-ahead or write-behind for applications. The main reason for this is the cost of maintaining consistency. Although the DAFS protocol includes provision for delegations, they cannot be accessed by applications above the DAFS API. As of yet, there are no hooks in the Provider for an application to control client caching (although there are hooks for the application to control server caching). This permits DAFS caching only inside of the Provider - which introduces an awkward architecture unless it is a kernel implementation using the buffer cache. Once inside the Provider, we no longer control file data buffers; they are handed to the Provider and registered by the application above. Since these data buffers cannot be shared between applications, it is necessary to keep a separate set of buffers inside the Provider to be used as the cache and copy from these buffers to the application buffers. The DAFS API suggests and advertises copy-free semantics but introducing file data caching into the Provider introduces a copy into the data path. Having the cache above the API, however, allows us to control buffers, use DAFS features and export the standard POSIX API, allowing applications to use DAFS without modification and with the benefit of caching.

In order to access delegation information above the DAFS API, two changes to the API were required: the delegation returned from the server had to be returned as an argument to `dap_open_file` and registration of a callback by the DAFS application was required. This callback is executed from within the DAFS Provider when the
client receives a callback from the server. The DAFS Provider was modified to include a back control channel as a separate thread, listening for DAFS Requests from the server and posting DAFS Responses to these. When the server recalls a delegation, the Provider executes its own callback, which simply changes the type of delegation and then makes a call to an application-registered callback (in our case the cacheDAFS callback that invalidates, flushes and sends any outstanding local close), handing it the file handle of the target of the recall.

These changes to the DAFS API are shown below in italics:

```c
DAP_ERROR
dap_open_file(DAP_DIRECTORY_HANDLE dir_handle,
DAP_CRED_HANDLE cred_handle, DAP_CHAR * path, DAP_FLAGS flags,
DAP_CREATE_MODE mode, DAP_FILE_HANDLE * file_handle,
dafs_open_delegation_type *delegation );

DAP_ERROR
dap_recall_callback( void (*recall_cb)(DAP_FILE_HANDLE fh) );
```

### 3.4 Implementation Details

The client cache implemented in cacheDAFS exists entirely in user-space but is modeled after the FreeBSD kernel buffer cache, using block-based caching with a fixed block size, buffer queues and hash lists. This section describes the details of the cache implementation including data structures, the implementation of delayed-writes and asynchronous read ahead.
3.4.1 Data Structures

Buffers can be placed on one of three queues: the EMPTY queue, the CLEAN queue or the DIRTY queue. All buffers are pre-registered with the network adapter. Buffers that have not been used before are in the EMPTY queue. Used buffers that have been written are placed in the DIRTY queue. The main reason for maintaining a DIRTY queue is for quick access to dirty buffers during delayed-writes. Used buffers that have not been written are found in the CLEAN queue. The cache uses an LRU replacement algorithm that selects first from the EMPTY queue and then from the CLEAN queue. As a last resort, the DIRTY queue is used. When the DIRTY queue needs to be used, the selected block must be flushed first. A recently used buffer that has not been written is released to the tail of the CLEAN queue. A recently used dirty buffer is released to the tail of the DIRTY queue. An invalidated buffer is released to the head of the CLEAN queue. Once the EMPTY queue runs out of buffers, there is no way for it to regain buffers. One further use of this queue is to hold buffers that are not registered but can be registered as the demand for buffers increases. Likewise, they could be deregistered and returned to the EMPTY queue when the demand is low. This provides a way to moderate the use of registered memory by the cache and would be somewhat analogous to the kernel buffer cache queue with buffers that have no physical memory backing them.

Every file has a unique data structure similar to the FreeBSD VNODE associated with it. We will refer to this structure as the dafsunode. It stores the file handle, a reference count for open instances, a list of pointers to cache buffers associated with this file, cached file attributes (file size and name) and information about outstanding
close operations to be performed for files. Each open instance has a file pointer indicating the current position in the file for this open instance.

Hash lists are used to quickly locate a valid buffer in the cache. A file block is hashed using the block number and dafsnode address. Each block is fetched in its entirety using an RDMA write by the server (a DAFS read direct). Then the data required by the application is copied into user buffers from the cache. A read or write that hits in the cache results only in a data copy. Reads and writes that miss require the data to be fetched (as many cache blocks as are necessary) and copied. When the client holds a write delegation, writes beyond the end of the file are performed by allocating new cache blocks, performing a copy, marking them dirty and updating attribute information locally. These writes are eventually flushed to the server as the cache fills up. Reads beyond the end of the file are not sent to the server while a delegation is held since they will fail; the file size is known and will not change. With write delegations, the client is permitted to extend the file or append to it in the cache, delaying notification of changes to the server. In general, this only requires a local copy and setting of a flag to mark the buffer in question as dirty. Cache misses can occur when writing and this requires a buffer to be fetched (RDMA read) and a local copy. Even when extending a file, this can happen. A read or write hit in the cache always requires a local data copy. A read miss requires a DAFS direct read to fetch the block and a copy to transfer it to the application buffer. A write miss requires a DAFS direct read to fetch the block and a copy from the application buffer to the cache. Caching benefit is expected to be seen when read and write hit ratios are high. This is because the speed of local memory exceeds the time required to marshal a
DAFS request, send it and receive a response. CacheDAFS was implemented so that features could easily be turned on and off for experimentation. The number and size of cache buffers is tunable, up to nearly all of the memory available (Note that the DAFS Provider has a limit on the number of external registered regions permitted. This was increased for our experiments). By default, cacheDAFS registers all cache buffers beforehand. By setting a flag, registration can be made to occur on the fly as I/O is performed into cache buffers. This is less demanding on the operating system but requires kernel interaction on every registration and deregistration. In Chapter 4, we analyze the overhead of registering on the fly.

### 3.4.2 Delayed-Writes

As dirty data accumulates in the cache, it is necessary to flush to the server periodically. Since the protocol maintains perfect consistency, we do not have to worry about the freshness of recently written data as seen by other clients. We do, however, have to protect fresh dirty data from being lost upon crash and keep the number of clean cache buffers available at a reasonable level by flushing dirty buffers to the server periodically. The client selects buffers from the DIRTY queue as a last resort but workloads that are heavy on writes will result in buffers being flushed on demand if a background thread does not flush periodically.

CacheDAFS incorporates a delayed-write policy to keep the level of dirty buffers reasonable. We use the parameters defined by the 4.3 FreeBSD buffer cache to decide when flushing is necessary and employ asynchronous writes like NFSv3. A flushing thread performing asynchronous flushing when the following threshold is exceeded:
(hidirtybuffers + lodirtybuffers) / 2

Where:

hidirtybuffers = totalbuffers / 4 + 20

lodirtybuffers = hidirtybuffers / 2

This means that we begin to flush when approximately 20% of the cache is dirty. Once the number of dirty buffers falls beneath the threshold, the thread sleeps, waiting to be woken up when necessary. Every time a buffer is dirtied, the threshold is examined to see if waking the thread is necessary.

Flushing of dirty data is performed using the DAFS asynchronous write interface and completion groups. Once the threshold is exceeded and the flushing thread is woken up, it locks a window of n dirty cache buffers (the least recently used buffers on the head of the DIRTY queue). The thread only 'tries' for these locks, preferring to lock another buffer rather than wait on one that is in use. DAFS completion groups allow a number of asynchronous I/O requests to be grouped together. Waiting or polling on this completion group produces a handle to the first completed I/O operation\(^1\). Asynchronous writes are issued for the window of locked buffers. When a completion is acquired, the associated buffer is marked clean, unlocked and placed on the tail of the CLEAN queue. If the number of dirty buffers still exceeds the threshold permitted, another buffer is locked and an asynchronous write is performed, tied to the same completion group. For the experiments run in this thesis, we use \( n = 16 \) to provide a window size that yields good throughput and locks only an insignificant number of buffers so that the cache is not strained by flushing writes. Results shown in Chapter 4 demonstrate that even when the write load is heavy, the flushing thread

\(^1\)Polling was very CPU intensive so we used wait with a short timeout instead
is effective in keeping buffers clean so that flushing on demand is never required.

### 3.4.3 Asynchronous Read Ahead

CacheDAFS buffers are fixed in size (this size is adjustable when first creating the cache), unlike those in the FreeBSD kernel buffer cache. As a result, a request for a block of data larger than the cache buffer size requires synchronous transfers of as many cache data blocks as are necessary followed by a data copy. This inefficiency is dealt with using asynchronous read ahead within a data block. For a read request larger than cache buffer size, the total number of buffers required are acquired and locked. Asynchronous reads are issued for all but the first block in the hopes that they will be completed while the first block is read synchronously. In the worst case (which very rarely occurs), asynchronous reads that are not complete on time are waited on before the data is copied to the application buffer.

### 3.4.4 Code Base

CacheDAFS was implemented on top of a user-level DAFS Provider designed and implemented at Duke University (Details of the client implementation can be found in [26]). Most required functionality was present in this client implementation. We added the following for in order to support caching:

- Delegation support
- DAFS back control channel support
- Additional efficiency and functionality to file tables in the client
• Support for atomic append

• Thread-safe capability within the Provider

• API changes for implementing our protocol

• SDK 1.0 support

The DAFS server used is a kernel resident, multi-threaded, asynchronous server developed at Harvard [30]. It implements the full DAFS specification and supports open delegations as described in previous sections. The server keeps track of all clients that have a file open and is responsible for detecting sharing conflicts. Both client and server code are compliant with v1.0 of the DAFS SDK\textsuperscript{2} modified to operate on FreeBSD. The cacheDAFS API can be found in Appendix A.

\textsuperscript{2}Available on the Web: www.dafscollaborative.org
Chapter 4

Results

This chapter evaluates cacheDAFS and compares it to DAFS using microbenchmarks and macrobenchmarks. We measure the cost of individual operations and the performance of interesting workloads produced by Postmark [24], a synthetic benchmark. We ported Postmark to run over DAFS and cacheDAFS and instrumented it to measure the cost of individual file access operations. Section 4.1 describes the use of cacheDAFS to determine the cost of various parts of the data path. Section 4.2 describes experiments with microbenchmarks that measure the effect of the different Postmark operations. We present and analyze two interesting Postmark workloads in Sections 4.2.1 and 4.2.2. Section 4.3 presents a simple read benchmark exhibiting locality of reference that shows the benefit of caching without the metadata effects that result from Postmark.

We expect the client cache to be much more effective than remote access when the working set fits mostly in the cache - especially for small data accesses, because local copies at memory speed are very fast. When misses are frequent, performance
is expected to be similar to DAFS except in the cases where data copies are large and there is no locality of reference; here we expect cacheDAFS to perform a little worse than DAFS because the misses are expensive and data is not reused.

The experiments in this chapter were performed on a patched version of FreeBSD 4.3 with two Pentium III 800 MHz machines (one client, one server) with 133MHz memory buses and 9G 10000 RPM Seagate Cheetah disks. The server had 1GB of RAM. All working sets in the experiments run here fitted in the server cache. We do not experiment with workloads that go to server disk because disk I/O is expected to be a limiting factor for all file system environments. Interesting workloads for DAFS (and cacheDAFS) are those in which memory-to-memory data transfer is the bottleneck. For all experiments, we wired 225 MB of RAM for the client cache, which is what we expect a client that has 256 MB of RAM to have available for its cache. This amount was determined by running the same experiments described in following sections using NFSv3 on a client with 256 MB of RAM. Monitoring memory usage showed that up to 225MB of memory was available to NFSv3 on such a client and almost all of it was used for the client cache.

4.1 Measuring the cacheDAFS Data Path

In this section, we use cacheDAFS to measure the cost of the various aspects of the data path. The purpose of this experiment is to provide some insight into the different overheads involved with cacheDAFS and similar applications that perform I/O on a DAFS platform. We examine the read data path in the case of a cache miss (with registration of cache buffers on the fly) and use this to break the cost down into
the following components:

- Registration/deregistration
- DAFS operation (DAFS request, RDMA write, DAFS Response)
- Data copy
- Other (additional software overhead on the client)

![Image of diagram]

Figure 4.1: Possible DAFS Client Implementations

Figure 4.1 illustrates the details of this path. When a miss occurs and the cache is configured to register buffers on the fly, a buffer is selected using the LRU algorithm and registered (1). A DAFS operation (2) is issued to perform direct I/O into the buffer. The cache buffer is then deregistered (3) and a copy (4) is performed into
the application buffer. Additional software overhead labeled as other refers to cache lookup, hashing, table updates in cacheDAFS and other software overhead in the Provider (such as descriptor processing).

Figure 4.2: Relative Cost of Data Path Components

This gives us some idea of how the costs increase as more data is transferred. Note that these measurements focus on synchronous I/O\textsuperscript{1}. We measure each component for read block sizes ranging from 4K to 512K, reading a total of 1GB of data into the cache. A single block resident in server memory is read repeatedly and the cache block is invalidated immediately afterwards to simulate random access. We measured the cost of registration and deregistration by taking the measurements first with pre-registered cache buffers and again with registration on the fly. The difference of the two measurements is taken as the overhead of registration. The results of this

\textsuperscript{1}Note that asynchronous I/O would behave differently
experiment are summarized in Figure 4.2.

![Data Path Component Costs](image)

**Figure 4.3: Data Path Component Costs**

Figure 4.2 shows the operation overheads as percentages of the entire read operation. This illustrates how important each component becomes as the data transfer size increases. All reads were performed synchronously. Thus the DAFS Op portion consists of the time taken to send the DAFS request, perform RDMA data transfer, wait for and receive the DAFS response. It can be seen that, as the block size increases, the copy becomes more significant and comparable to the time for data transfer. Yet this copy is still much less expensive than the transfer time. If this block is expected to be referenced multiple times in the future, the overhead of data transfer + data copy for the initial miss is worthwhile because future accesses will be less expensive. However, when the block is large and not referenced again in the
future, we simply experience cache pollution. For small block sizes, software overhead is relatively important because the time for transfer and copy is small.

Figure 4.3 shows the actual values for the same experiment and gives a better idea of the read operation cost as the data transfer size increases. It can be seen from this figure that both the data copy and registration cost become increasingly significant, as the block size gets large. The cost of registration increases with the block size. But the dominant portion of the data path is the DAFS operation, which in turn consists mostly of the time required to transfer data.

### 4.2 Postmark Experiments

Postmark is a synthetic benchmark aimed at measuring file system performance over a workload composed of many short-lived, relatively small files. Such a workload is similar to that of mail and net-news servers used by Internet Service Providers. Postmark consists of three phases: file creates, transactions and file deletes. During the file create phase a user-specified number of files is created and populated with random data. The next phase is a sequence of transactions. These transactions are chosen randomly from a file create or delete paired with file read or append. File create creates and writes random text to a file. File delete removes a random file from the file set. File read reads a random file in its entirety and file write is performed as an *atomic append*; a random amount of data is appended to the end of the file. The final phase of Postmark deletes all remaining files. Statistics are reported for each phase of transactions. Postmark behavior can be tuned by changing several parameters:
• The number and size range of files

• The number of transactions

• The ratio of creates to deletes and the ratio of reads to writes

• The number of creates, deletes, reads and writes in transactions (any one of these can be set to high bias, low bias or be removed from transactions).

Postmark workloads are metadata-intensive. For every read or write, a file is opened and closed before and after the data transfer. Creates and deletes also require metadata updates on the server and must always go to the server (even if we are caching). To reduce synchronous disk I/O at the server on metadata operations, we ran the experiments with the FFS file system using soft updates. A Postmark workload with \( N \) files, \( M \) transactions and a read/write ratio of \( k \) can be described as follows:

\[
(N + M/2)\text{creates} + (N + M/2)\text{deletes} + M \text{ opens} + (M + N)\text{closes} + \\
(N + M/2)\text{writes} + (M(10 - k)/10)\text{writes} + ((k-10)M)\text{reads}.
\]

One half of transactions involve creates and the other half deletes. All creates and opens are paired with a close. These operations are the metadata operations. The I/O operations consist of reads and writes that occur during transactions. Of the writes performed, those paired with creates and used to populate files are not considered in the read/write ratio - these are shown in italics in the equation above.

Based on this operation breakdown, we expect the results of experiments to be strongly influenced by metadata-intensive operations. For DAFS, all operations go to the server. But cacheDAFS can perform open and close operations locally.
We ported Postmark (v 1.5) to use DAFS and cacheDAFS. Modifications to use cacheDAFS consisted only of changing read, write, open, close and unlink functions to the cacheDAFS equivalents. To use DAFS, we simply allocate and pre-register buffers for I/O operations and keep track of file handles for DAFS operations. We further instrumented Postmark to compute the costs of individual operations (create, delete, open, close, read and write). This helps to explain the results and show not only how DAFS and cacheDAFS perform with the given workloads but also how individual operations compare, explaining the results. We first look at individual operations using the instrumented version of Postmark on FFS, NFSv3, NQNFS, DAFS and cacheDAFS to examine their behavior and identify the source of overheads. We run the experiment described in Section 4.2.1 with a read/write ratio of 10 (all transactions contain reads). The results show the basic behavior of the local case (FFS) for this workload and the behavior of other systems. NFSv3 behavior gives us a standard to compare the DAFS numbers against and NQNFS gives us an idea of caching effects in the NFSv3 environment, showing some differences we might expect to see between DAFS and cacheDAFS. In summary, we found the following (actual numbers can be found in Appendix B):

- Create is the most expensive operation. As the number of files increases, so does the cost of create, since the files are all created in the same directory. This is due to the linear cost of lookup in FFS. When the number of files remains fairly constant, the time to create also remains constant.

- Delete has a high variance in all systems but does not increase much as the number of files increases.
- Open has a high variance for all systems except cacheDAFS. NFSv3, DAFS and FFS in general take much longer to perform open operations than NQNFS or cacheDAFS. The variation in open is due mainly to lookup time. For NFSv3, NQNFS and FFS open sometimes is of the order of a few microseconds - this is because they can sometimes use cached attributes and perform open locally. CacheDAFS can always perform open locally and use cached attributes because of the consistency model. For all systems except cacheDAFS, as the number of files increases, so does the cost of open (due to lookup time).

- The overhead of close is fairly consistent, regardless of the number of files. This cost is much lower for FFS, cacheDAFS and NQNFS. This is because NFSv3 has to fulfill the requirements of an open-close consistency model and wait for asynchronous writes to complete. All DAFS close operations go to the server.

These general trends give us some idea of what to expect from the Postmark experiments we run in the following sections.

### 4.2.1 Varying the Read/Write Ratio

In this experiment, we measure the performance of read and writing a changing file set that fits reasonably in the client cache. This experiment examines how performance is affected by varying the read/write ratio. We vary the read/write ratio (also referred to as the read bias) from 10 to 0. The ratio of creates to deletes remains the same. This means that the file set is constantly changing but the total number of files remains almost the same as the initial number of files. We use 10,000 files and perform 30,000 transactions, each of which is a file create or delete (with equal
probability) paired with a read or write. When the read bias is 10, all transactions have reads and when it is 0, all transactions have writes. File sizes are randomly chosen from between 512 and 32K bytes (average size 16K bytes). The file set fits both in the client and server caches. Both read and write blocks were set as larger than the file size. This means that DAFS reads and writes are performed in a single operation. CacheDAFS buffers were set to be 16K in size but read ahead was enabled. The client cache size is set to 225 MB, pre-registered memory (14,400 buffers). We ran each experiment several times and averaged the results.

Since Postmark writes (except while populating files during the create phase) are all atomic appends, cacheDAFS is able to perform all of these locally when holding a delegation. Postmark occurs on a single client, thus there is no write sharing and all appends will be local. In the DAFS case, appends are required, by definition, to be stably committed on the server. We implemented atomic appends in DAFS similar to the way NFSv3 does them. An append was performed by locking the file, getting the size to be used as the position for append, committing all previously written data on the server for this file, appending the data using a write and then unlocking the file. We found that running Postmark with appends provides a disadvantage for DAFS (compared to cacheDAFS) that is simply an artifact of the way Postmark was written, since atomic appends are generally used in write sharing situations. For this reason, we modified Postmark to use regular writes (or file extension) instead of atomic append. Thus DAFS performs writes with unstable writes (inline write to server memory, which is never committed to disk). CacheDAFS performs a local copy, flushed later or a read followed by a local copy when a miss occurs. Since cacheDAFS
is able to cache metadata while holding open delegations, it has potential for better performance than DAFS.

Table 4.1: Average Individual Operation Costs for cacheDAFS

<table>
<thead>
<tr>
<th>Read Bias</th>
<th>OPEN</th>
<th>CLOSE</th>
<th>READ</th>
<th>WRITE</th>
<th>CREATE</th>
<th>DELETE</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>6</td>
<td>3</td>
<td>60</td>
<td>231</td>
<td>2319</td>
<td>678</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>3</td>
<td>86</td>
<td>268</td>
<td>2319</td>
<td>706</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>3</td>
<td>98</td>
<td>298</td>
<td>2589</td>
<td>738</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>3</td>
<td>113</td>
<td>313</td>
<td>2427</td>
<td>826</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>3</td>
<td>N/A</td>
<td>318</td>
<td>2196</td>
<td>895</td>
</tr>
</tbody>
</table>

Since metadata operations are prevalent in this workload, we examine first what the costs of individual operations are. We expect that the overhead of metadata operations will dominate the results and the read/write ratio change will have little effect on the results. Tables 4.1 and 4.2 give the average cost per operation in µs for each operation. We see that cacheDAFS open and close operations are low overhead because they are performed locally, while DAFS needs to send these operations to the server. Each read or write transaction requires and open and close.

Table 4.2: Average Individual Operation Costs for DAFS

<table>
<thead>
<tr>
<th>Read Bias</th>
<th>OPEN</th>
<th>CLOSE</th>
<th>READ</th>
<th>WRITE</th>
<th>CREATE</th>
<th>DELETE</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>381</td>
<td>78</td>
<td>408</td>
<td>549</td>
<td>2060</td>
<td>909</td>
</tr>
<tr>
<td>7</td>
<td>383</td>
<td>80</td>
<td>552</td>
<td>805</td>
<td>2049</td>
<td>896</td>
</tr>
<tr>
<td>5</td>
<td>375</td>
<td>81</td>
<td>623</td>
<td>908</td>
<td>2024</td>
<td>841</td>
</tr>
<tr>
<td>3</td>
<td>386</td>
<td>83</td>
<td>689</td>
<td>958</td>
<td>2040</td>
<td>878</td>
</tr>
<tr>
<td>0</td>
<td>406</td>
<td>83</td>
<td>N/A</td>
<td>995</td>
<td>2050</td>
<td>932</td>
</tr>
</tbody>
</table>

The performance of create and delete operations is comparable for cacheDAFS and DAFS because both operations need to go to the server. CacheDAFS also sends an outstanding close operation before delete, since it holds delegations for all files.
In cacheDAFS, there is a little more overhead while creating because of delegation support.

Given these results, we expect cacheDAFS to outperform DAFS because it can serve reads mainly out of the cache, create files in the cache and write into the cache. But there is a significant additional benefit from being able to do open and close operations locally. This benefit, however, is the same regardless of the read/write ratio; thus we expect any trend that appears as the ratio of writes to reads increases to be due mostly to the increasing number of writes. As the write ratio gets higher, DAFS should be stressed more by copying overhead since it performs more inline writes and fewer direct reads. CacheDAFS will be stressed as well because of the need to flush dirty data more often and keep buffers available for use. Since Postmark runs on a single client, cacheDAFS holds open delegations for every file, permitting attribute caching (name and file size). The results of this experiment are shown in Figure 4.4. As can be seen, cacheDAFS performs consistently better. This is because files are created locally in the cache. The initial create must go to the server but population of data occurs locally. Since many files are created in the cache and the file set fits in the client cache, most reads will be hits. Postmark files are short-lived and rarely is a file read more than once during its lifetime. For these experiments, cacheDAFS experiences from 95% - 86% read hits as the ratio of reads to writes declines and from 100% and 97% write hits as the write ratio increases. Performance declines as write activity increases because delayed-writes become more intensive. We also do not see 100% hit ratios even though the file set fits in the cache. This is partly because the

\footnote{This was determined by counting the average number of times each file was accessed during a Postmark run}
file set is constantly changing. But it is also due to fragmentation. Cache buffers are fixed in size and any file that is larger than 16K takes up two 16K buffers, wasting space. This problem is further aggravated when the file size does not match the buffer size at all. A more advanced implementation would need to allocate variable sized buffers. Furthermore, some writes do not hit in the cache. If we need to write to a cache block that has been evicted, that block must be read first and then written to avoid having holes in the cache.

![Graph showing read/write ratio with Postmark](image)

Figure 4.4: Varying the Read/Write Ratio With Postmark

With all reads, the cache flushes around 13,300 buffers asynchronously, mostly during the create phase. With all writes, the cache needs to flush 30,000 buffers throughout creates and transactions. DAFS performance declines as writes increase because writes are performed inline.

The results of this experiment show the benefit of local writes. For applications with high write activity that don’t experience write sharing, this is an advantage.
DAFS asynchronous writes can also be used to flush these writes quickly without disrupting the application. For an application that does not cache, cacheDAFS is useful to avoid unnecessary write-through. As a further advantage, cacheDAFS is aware of delegations and knows when delayed-writing is possible. With delegations exposed above the DAFS API, applications that cache can use them in the same way. By flushing less eagerly, cacheDAFS may be able to avoid flushing many files before they are deleted.

4.2.2 Varying the File Set Size

The previous section shows an experiment where the file set fits well in the client cache. This section shows how cacheDAFS performance compares to DAFS as the working set size exceeds the client cache size (resulting in lower hit rates). Note that, in all experiments, the working set fits in the server cache.

In this experiment, we run Postmark, varying the file set from 5,000 files (78M comfortably fits in the client cache with no fragmentation effects) to 60,000 (937MB just barely fits in the server cache) where the average file size is 16K. We perform 50,000 transactions - reads only, biasing out create and delete. This is because file create, as the most expensive of Postmark operations, dominates the time taken for transactions. As the number of files increases, creates within transactions become significantly more expensive and dominate the results. We cannot have a delete on each transaction since this will quickly reduce the file set to nothing. We use a read/write ratio of 10 so that the effects of inline operations are not shown. CacheDAFS again has 225MB of pre-registered cache. While files can be created in the cache, as the file
set increases, these files will only include the most recently created files. Reads are random and not as likely to hit in the cache unless the total file set is small.

CacheDAFS read ahead within blocks is enabled. This does not provide much advantage; it simply means that we can read the second file block asynchronously for files larger than 16K and experience a hit on this block. Most cache hits are due to asynchronous reads from read ahead within a file as the file set becomes very large. The benefit that cacheDAFS derives from asynchronous reads is to avoid having to perform more than one synchronous I/O operation for an entire file that misses in the cache. The time taken to read the first block synchronously is about the same as what it takes for DAFS to read the entire file. There is not much difference between the times taken for DAFS to fetch 16K or 32K using an RDMA. CacheDAFS can fetch both blocks at the same time but must do the copy afterwards. We might see a more significant benefit if file sizes were larger.

Figure 4.5 summarizes the results of varying the file set size (Read hit ratio for cacheDAFS is shown above each bar). We can see that cacheDAFS is consistently better than DAFS, even when the file set starts to fall out of the cache. At first glance, it may seem that this is due to local reads hitting in the cache. This is partially true but as the file set falls out of the cache, most of the benefit cacheDAFS experiences is from being able to open and close files locally and not needing to fetch attributes from the server frequently.

As measured previously, open becomes more expensive for DAFS as the number of files increases even though close remains the same. This explains why cacheDAFS continues to do far better than DAFS, even as the file set falls out of the cache and
misses are frequent. All DAFS opens must go to the server and since an open and close is performed on every read transaction, this is expensive. Note also that the cost of open increases as the number of files increases. This is because the files all fall in one directory and metadata updates are expensive. These results suggest two things:

- DAFS would benefit from being able to cache attributes, especially to avoid metadata updates on the server

- DAFS applications can benefit from caching if they can achieve high hit rates. CacheDAFS shows significant performance advantages when file sets are small: files are created in the cache and often already there when Postmark requests to read them.
The last bar in Figure 4.5 shows both cacheDAFS and DAFS performance deteriorating to about the same. This is because the file set barely fits in the server cache and disk I/O becomes the bottleneck.

It is interesting to look at performance of the read operations only for the same experiment and compare cacheDAFS to DAFS. Figure 4.6 shows the average time taken for a read during file transactions.

![Figure 4.6: Read Time For Changing File Set Size](image)

Figure 4.6 shows the average time taken for a DAFS or cacheDAFS read call as the file set size grows. As expected (with the exception of the last bar, where there is disk I/O), DAFS values remain the same, which is the time for the DAFS request/response exchange and RDMA data transfer from server memory. CacheDAFS performs better until the file set reaches around 468 MB. At this point, only 30% of reads hit in the cache, most of which are due to asynchronous read ahead within a block. Looking
at Figure 4.5, the behavior after file set size 468 MB shows the benefit of local open and close for cacheDAFS. It is interesting to observe that, for file set size 234 MB, we do not see significant benefit with cacheDAFS. Most files should fit in the (225MB) cache but the read hit ratio is only 63%. This is mainly due to fragmentation, as described earlier.

This experiment shows that client caching is beneficial for applications especially for metadata. DAFS falls significantly behind when open and close operations that could be local are sent to the server. By observing the sharing state with cacheDAFS, we can eliminate this overhead.

4.3 Locality of Reference

The previous experiments show how cacheDAFS can be used to derive benefit from using delegations to perform certain operations locally. The benefit of being able to perform I/O operations locally in the cache is not as obvious, since metadata operations are dominant in a Postmark workload.

In this section we present another experiment designed to show how applications with access patterns that show locality of reference can take advantage of DAFS. This experiment reads data randomly from a working set of a certain size. The working set consists of a single file that is opened and closed only once. The working set size is varied from 100MB (fits in the client cache) to 900MB (fits in the server cache without causing disk I/O) and blocks are read randomly from it. The read block size is fixed to 16K and the client cache holds 8192 buffers (128 MB). We use a smaller client cache size than in previous experiments so that we can observe a wider range of
working sets and hit ratios with cacheDAFS. For each run, a total of 500,000 buffers were read - many more than the largest working set size, increasing the probability that buffers will be accessed multiple times for all working sets. This provides locality of reference for working sets and high hit rates for working sets that fit well in the client cache. This experiment performs synchronous reads only and runs from a warm server cache and cold client cache.

Figure 4.7: Read Performance With Increasing Working Set Size

We expect to see cacheDAFS perform better while most of the working set fits in the client cache. Figure 4.7 displays the average time taken to read a block with this experiment. CacheDAFS hit ratios go from 98% at a working set size of 100MB to 14% at a working set size of 900MB. As expected, it shows that applications that exhibit high locality of reference can benefit from client caching. Once the hit rate decreases to about 25% (at a working set size of 500MB), we no longer see much
benefit but at this point, the working set size is four times larger than the cache size and random access no longer provides locality of reference. DAFS performs the same regardless of the access pattern or working set size (which is expected because all accesses go to the server and hit in the server cache). At a working set of around 800MB, cacheDAFS is repeatedly missing and each miss requires a data transfer and data copy. This experiment shows that applications with access patterns that have locality of reference can experience significant benefit from cacheDAFS.
Chapter 5

Conclusions and Future Work

This chapter presents conclusions and some ideas for future work. We successfully implemented and tested a user-level cache for DAFS. CacheDAFS can be used by applications that do not perform their own caching and require the standard POSIX API. As a user-level library, cacheDAFS is easily extensible and avoids undesirable kernel overheads. It requires minimum changes to the DAFS API and no changes to the underlying DAFS protocol. CacheDAFS does not employ copy avoidance since it necessitates some kernel interaction. To the best of our knowledge, this is the first implementation and analysis of client caching using delegations with DAFS. Our results show that DAFS applications such as a mail or news server can benefit from metadata caching and data caching. The main disadvantage of cacheDAFS is that it does not allow multiple processes to share buffers at the user-level, which requires kernel interaction for protection and resource management.

DAFS delegations can provide perfect consistency in a standard file-sharing environment. We show that additional benefits are derived from the ability to perform
open and close operations locally while holding a delegation and cache metadata. As is expected from a typical implementation of client caching, applications can derive benefit from cacheDAFS if their access patterns exhibit good locality. Even when the miss rate is high, cacheDAFS does not degrade DAFS performance since it uses relatively small buffer sizes and read ahead. For applications that have heavy write workloads, dirty data is efficiently flushed using periodic asynchronous writes and DAFS completion groups.

5.1 Future Work

This section presents some ideas for future work including future improvements to cacheDAFS and extensions that are not possible with the current platform.

Perhaps the most significant outstanding issue is that of sharing the cache between processes. This avoids wasting space but may introduce overheads because kernel support is required for sharing.

CacheDAFS can be improved by more efficient buffer allocation and management. We experience fragmentation in the current implementation because of the fixed buffer size. A more advanced implementation would allow the allocation and use of different buffer sizes. Registration on the fly is expensive and increases with the block size but pre-registering the entire cache ties down most of the memory resources. More efficient resource management can be achieved by maintaining a pool of registered buffers but not pre-registering the entire cache as described in Section 3.4.1.

With cacheDAFS, we implement delayed-writes as client-initiated DAFS asynchronous inline writes. While this is shown to be efficient, it does cause a small
degradation in client performance as write traffic becomes heavy. On a platform that
is capable of RDMA reads, another option is to implement delayed-writes as server-
initiated RDMA reads. The server can remember the addresses for buffers that need
to be written and transfer the data without client knowledge.

CacheDAFS could further be extended to use NFSv4-like lock based sharing. For
workloads that include more write sharing, this allows finer granularity consistency
enforcement.

Finally, it would be interesting to implement cacheDAFS as a copy-free caching
version of DAFS that would exist within the Provider and export the DAFS API.
This would require a user-level cache with kernel interaction to protect buffers during
data transfer.
Bibliography


[38] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and
B. B. Welch, The Sprite Network Operating System, Computer Magazine of
the Computer Group News of the IEEE Computer Group Society, ; ACM CR

Dept. of Computer Science, Rice University, 1998.

[40] S. Pakin, Fast Messages (FM): Efficient, Portable Communication for Work-
station Clusters and Massively-Parallel Processors, January 1997.

[41] B. Pawlowski, C. Juszcak, P. Staubach, C. Smith, D. Lebel, and
D. Hitz, NFS Version 3: Design and Implementation, in USENIX Summer,

[42] M. Rangarajan and L. Iftode, Software Distributed Shared Memory over
Virtual Interface Architecture: Implementation and Performance, in Proceedingsof

Log Structured File System, in Proceedings of the 13th ACM Symposium on

[44] R. Sandberg et al., Design and Implementation of the Sun Network Filesys-
tem, in Proceedings of the Summer 1985 Conference, Portland, OR, June 1985,
pp. 119–130.

Workstation Environment.

[46] ———, Scalable, Secure and Highly Available Distributed File Access, in IEEE
Computer, May 1990.


[49] V. Srinivasan and J. Mogul, Spritely NFS: Experiments with Cache Consis-
tency Protocols, in In Proceedings of the Twelfth ACM Symposium on Operating


Appendix A

CacheDAFS API

int cdafs_open(const char *path, int oflag, int mode);
int cdafs_close(int filedes);
ssize_t cdafs_read(int filedes, void *buffer, size_t nbytes);
ssize_t cdafs_write(int filedes, const void *buffer, size_t nbytes);
int cdafs_fsync(int filedes);
int cdafs_unlink(char *path);
int cdafs_rename(char *from, char *to);
int cdafs_lseek(int filedes, off_t offset, int whence);
cdafs_err_t cdafs_init(char *server);
cdafs_err_t cdafs_disconnect();
Appendix B

Individual Operation

Measurements for Postmark

Table B.1 gives the measurements for individual Postmark operations used to assess the effect of metadata operations on the overall Postmark workload. The Postmark workload here has the following parameters:

- Initial number of files 10,000
- Files sizes range from 512 - 32K bytes
- 30,000 transactions are performed
- read/write ratio is 10
- create/delete ratio is 5
- read and write transfer size are larger than the file size
Table B.1: Individual Postmark Operation Costs in $\mu$s)

<table>
<thead>
<tr>
<th>System</th>
<th>CREATE (0-10,000 files)</th>
<th>CREATE (10,000 files)</th>
<th>OPEN (avg)</th>
<th>CLOSE</th>
<th>DELETE (avg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFS</td>
<td>50-1200</td>
<td>1200</td>
<td>236</td>
<td>15</td>
<td>797</td>
</tr>
<tr>
<td>NFS</td>
<td>450-2300</td>
<td>3000</td>
<td>500</td>
<td>8</td>
<td>1349</td>
</tr>
<tr>
<td>DAFS</td>
<td>130-2000</td>
<td>2100</td>
<td>383</td>
<td>71</td>
<td>216</td>
</tr>
<tr>
<td>NQNF5</td>
<td>430-2500</td>
<td>3300</td>
<td>651</td>
<td>7</td>
<td>1285</td>
</tr>
<tr>
<td>cacheDAFS</td>
<td>205-1900</td>
<td>2500</td>
<td>6</td>
<td>3</td>
<td>658</td>
</tr>
</tbody>
</table>

For operations with high variation, we report the average only. For create, we report the range as files are being created and the cost once 10,000 files is reached and the number of files remains fairly steady. All measurements were averaged over several runs.