Federated DAFS: Scalable Cluster-Based Direct Access File Servers

Murali Rangarajan, Suresh Gopalakrishnan, Ashok Arumugam, Rabita Sarker
Department of Computer Science,
Rutgers University, Piscataway, NJ 08854-8019
{muralir, gsuresh, ashoka, sarker}@cs.rutgers.edu

Liviu Iftode
Department of Computer Science,
University of Maryland, College Park, MD 20742
iftode@cs.umd.edu

***Abstract*— Protocols like the Direct Access File System (DAFS) leverage user-level memory-mapped communication to enable low overhead access to network-attached storage for applications. DAFS offers significant improvement in application performance using features like direct data transfer and RDMA. Our goal is to build high performance network file servers using DAFS. The benefits of the DAFS protocol can be extended to cluster-based servers, using low overhead user-level communication within the cluster.**

In this paper, we present Federated DAFS, a scalable and efficient cluster-based direct access file server. Federated DAFS combines an efficient user-space DAFS implementation with a low overhead clustering layer to present a scalable clustering solution for DAFS servers. Federated DAFS uses a portable mechanism for distribution and handling of client requests across the servers in the cluster. Federated DAFS also minimizes the intra-cluster communication by caching data blocks and by matching the file placement on the servers with the distribution of requests from the clients. Our results show that reasonable speedups(2.6 on four nodes and 4.5 on eight nodes) can be achieved using Federated DAFS on server clusters of up to eight nodes.

I. Introduction

Network storage needs are being stretched as file volumes grow and enterprises distribute storage requirements across a wider array of files, web applications and database servers. Performance of network file servers has been limited by overheads resulting from protocol processing, redundant copying and other networking costs. User-level networking architectures, such as the Virtual Interface (VI) Architecture [1], U-Net [2], and Virtual Memory Mapped Communication (VMMC) [3], designed to achieve low-latency and high-bandwidth in a SAN environment, offer an attractive solution for reducing communication software overheads.

The Direct Access File System (DAFS) protocol [4] exploits user-level memory-mapped communication to help reduce the cost of file access by enabling direct access to network-attached storage for applications. The DAFS specification includes features to improve application performance like asynchronous I/O, direct data transfer to and from application buffers, and scatter/gather I/O.

To build high performance network file servers with reasonable levels of service, we need multiprocessor or distributed network servers. Clusters of commodity computers have the potential to provide good performance with scalability, at a low cost. For cluster-based servers, user-level communication has been shown to provide a low overhead mechanism for intra-cluster communication [5]. We are interested in extending the benefits of the DAFS protocol to cluster-based servers, using low-overhead user-level communication within the cluster.

In this paper, we present Federated DAFS, a scalable cluster-based direct access file server. Federated DAFS extends the benefits of a direct access file system to cluster-based file servers. Federated DAFS takes advantage of low-latency high-bandwidth user-level communication among servers in a cluster *(i)* to present a consolidated view of the storage available on all the servers, and *(ii)* to distribute file access requests across the servers in the cluster. Federated DAFS consists of two components:

This work is supported in part by the National Science Foundation under the NSF CAREER CCR 0133366.

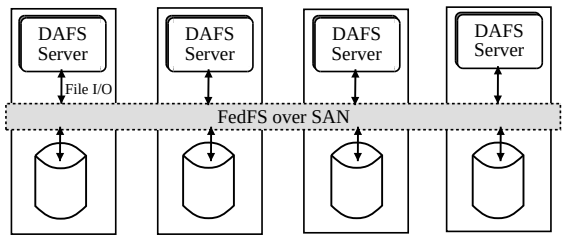

Fig. 1. Federated DAFS on a Cluster

1) *An implementation of the DAFS protocol:* We have developed a portable DAFS prototype entirely in user-space.
2) *Clustering software to implement naming and consolidate server storage across the cluster:* The Federated File System (FedFS) [6] provides a thin clustering layer which is used by the DAFS server implementation to access server storage across the cluster.

Fig. 1 shows the Federated DAFS architecture, with DAFS servers running on each node and accessing the underlying storage using the FedFS clustering layer.

We evaluated the performance of Federated DAFS using the postmark file system benchmark [7]. Our experiments show that Federated DAFS provides reasonably good speedups. Our evaluations also show that caching of data blocks can be used to improve performance by reducing the intra-cluster communication. Federated DAFS minimizes the intra-cluster communication further by matching the file placement on the servers with the distribution of requests from the clients.

The rest of the paper is organized as follows. Section II describes the background and related work. Section III describes our implementation of the DAFS protocol in user-space. Section IV presents details of FedFS, our clustering solution, and Section V describes the implementation of FedFS. Section VI presents the results from our experimental evaluation and Section VII presents the conclusions.

II. Background and Related Work

A. Reducing Networking Overheads

Various techniques have been used to reduce the overheads arising out of copies in the network data path [8]–[10]. In addition to zero-copy communication, user-level networking architectures [1]–[3] provide a mechanism for high-bandwidth and low-latency communication with minimal overheads. The Virtual Interface Architecture (VIA) [1] is a user-level memory-mapped communication standard for SANs that reduces the communication overhead by providing direct access to the network interface for applications.

B. Direct Access File Systems

The DAFS protocol [4], [11] takes advantage of direct access transports to provide clients with low-overhead access to network attached file storage. It allows efficient, portable implementations of file system clients entirely in user-space. Applications can reap the benefits of the DAFS protocol from user-space by registering the memory regions used in communication with the OS kernel. The DAFS protocol also allows better control of data movement and caching by the applications.

The fundamental performance characteristics of a DAFS-based user-level file system structure was explored in [12]. It was shown that lower client overhead in the DAFS configuration can improve application performance by up to 40% over optimized NFS when application processing and I/O demands are well-balanced. Optimistic DAFS [13] presents an enhancement of DAFS that enables clients to directly access remote memory pages of files present in the server cache.

C. Cluster-Based Network Servers

Locality-based schemes have also been used to build high performance cluster-based network servers [14], [15]. However, the above schemes use non-portable techniques like TCP handoffs to distribute requests. Previous studies [5] have revealed that portability can be achieved without significant degradation in performance and that user-level networking protocols like VIA are effective for intra-cluster communication in cluster-based servers [16]. It was shown in [16] that user-level communication can improve performance of cluster-based servers by as much as 29%, mainly due to contributions from low processor overhead, remote memory writes and zero-copy.

Federated DAFS uses a low overhead protocol like DAFS for client-server communication and a

lightweight user-level clustering protocol (FedFS) for intra-server communication, to build an efficient cluster-based file server.

III. Implementation of DAFS protocol

In this section, we describe our implementation of the DAFS protocol. Our client and server prototypes implement the DAFS protocol using the Virtual Interface Architecture [1], entirely in user-space, making it portable across any POSIX-based system like Linux, FreeBSD or Solaris. Both the DAFS client and server implementations follow a staged event driven model [17], with dynamically configurable thread pools to service each stage. Each thread pool has an associated job queue, the length of which indicates the load at this stage. These thread pools can grow in size up to a tunable maximum to handle increased loads.

A. Client and Server Implementations

The DAFS client is provided to applications in the form of a user-level library. When the application invokes a DAFS API primitive, the library translates this to an RPC request for the corresponding I/O operation on the server. Requests are sent in the context of the application thread which invokes the DAFS API. Application threads can choose between using the asynchronous DAFS API or waiting for the responses to their requests, providing the application better control and use of concurrency.

The server implementation uses a separate connection manager thread which handles connection requests from clients. Once a connection is established with a client, the server handles DAFS API requests arriving from the client using a receive thread, and a pool of threads for file system processing. The responsibility of the receive thread is to receive DAFS API requests from the client and pass them on to the file system thread pool. Threads in the file system pool perform the required I/O operation and send the results back to the client. Using a multithreaded file system processing module helps in handling concurrent client requests efficiently.

B. Communication and RPCs

The DAFS client establishes a VI connection with the server on which it sends DAFS I/O requests. Each VI connection is associated with a set of descriptors which makes it possible for clients to submit multiple simultaneous requests without waiting for the completion of previous requests. Our implementation supports true zero-copy data transfers between the DAFS client and server. For writes, the data is transferred directly from the application buffers without any additional copy, using the scatter/gather send available in VIA. For reads, the server uses scatter/gather RDMA to send replies directly to application buffers without incurring additional copies. In both cases, the message consists of multiple segments with the first segment carrying the header and the subsequent segments carrying data.

Each primitive in the DAFS API is implemented as an RPC on the server. The arguments to the RPC include the arguments to the DAFS primitive, an RPC procedure identifier, and a unique request identifier that is generated by the client. These arguments are marshalled into a request header using the DAFS RPC stub generator, and sent to the server. On the server, the request is unmarshalled and the requested I/O operation is performed by invoking the handler corresponding to the procedure identifier present in the request.

C. Support for Asynchronous I/O

Using an event-driven model makes it easy to support the asynchronous I/O primitives specified by the DAFS protocol, allowing the applications to pipeline multiple I/O requests and achieve better performance. An asynchronous I/O operation is described by an I/O descriptor. For asynchronous I/O primitives, the API call returns a request identifier in the I/O descriptor as soon as the request is dispatched to the server, without waiting for the response. The I/O descriptor also contains a result buffer where the result of the asynchronous operation is stored upon completion. For each pending asynchronous request, the client saves a request context that includes the request identifier, the result buffer and information about the requesting thread. When the reply for an asynchronous operation arrives at the client, its corresponding request context is identified, the result stored in the result buffer and the context released. The application can check the completion status of any previously submitted asynchronous request using the I/O descriptors.

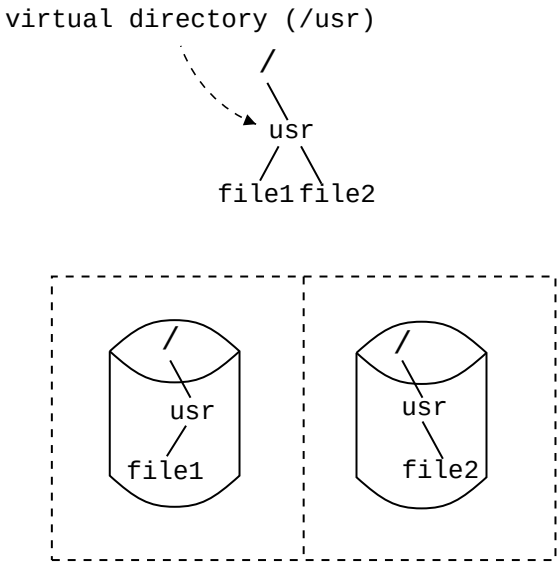

Fig. 2. FedFS Virtual Directory /usr

IV. Clustering using FedFS

The DAFS server implementation uses FedFS to access the server storage distributed across the cluster. FedFS [6] is a novel cluster file system architecture developed by us, that provides a global name space in a cluster by aggregating local file systems of cluster nodes into a loose federation. Using FedFS, stand-alone servers can access storage across the cluster and act as distributed servers. A copy of the server running on a node can operate on files located on any cluster node by performing the operation through FedFS.

FedFS creates the name space dynamically for each distributed application that runs on the cluster. The name space exists only during the lifetime of the distributed application. The location independent global naming allows easy file migration, which could be used for balancing load across the cluster nodes. In the following subsections, we'll describe the global naming scheme provided by FedFS and the protocol used in FedFS for accessing files.

A. Global Naming in FedFS

The key component in FedFS global name space management is the *virtual directory* (VD). A virtual directory in FedFS is the union of all the local directories with the same pathname, from all the server nodes. For example, if a directory /usr exists in each local file system, the virtual directory /usr in the resulting FedFS will contain the union of all the /usr directories, as illustrated in Fig. 2.

The *home* of a file is defined to be the server node on which the corresponding pathname is present in the local file system. FedFS maintains information about the homes of all files in a directory in the corresponding virtual directory. Each pathname (virtual directory or file) in FedFS is associated with a *manager*. The manager node is identified by applying a constant hash function to the pathname. For a file, the manager is responsible for keeping information about the home nodes. For a directory, the manager is responsible for creating and maintaining the virtual directory contents. Directory operations such as `create` and `delete` contact the manager to update the virtual directory contents. This ensures that collisions are avoided in the global name space[1].

dirmerge: The virtual directory is constructed by the manager at the time of the first lookup involving the directory. In order to construct the virtual directory, the manager node performs a `dirmerge` operation. The `dirmerge` operation involves sending a request to the nodes which have a copy of the directory, asking for the directory contents. To determine the nodes which have a copy of the directory, each node maintains a summary information of the directory tree of all other nodes. This summary information, based on Bloom filters [18], is generated by each node and sent to other nodes at the time of initialization. Since Bloom filters only have false positives, no files will be left out.

The `dirmerge` is a potentially expensive operation since it involves communication with all the server nodes in the worst case. However, a `dirmerge` is performed on a directory only once. The merged directory information is cached in memory, and gets updated whenever any metadata operation (`create` or `delete` of a file/subdirectory) is performed on the directory.

File lookup/access: For any operation on a file, the first step is to identify the home of the file. Given a file pathname, a node determines the manager of the file by applying the hash function, and sends

[1]Currently, the implementation does not handle collisions generated by file creations performed outside FedFS.

a message to the manager requesting information about the home. If the manager doesn't have the information, it contacts the manager of the parent directory of the file. Thus, a maximum of three other nodes are involved in any lookup operation, irrespective of the number of nodes in the cluster - the manager of the file, the home of the file, and the manager of the parent directory. These nodes are involved typically only in the first access.

Directory table: FedFS maintains a directory table (DT) on each node to speed up the lookup process for files and directories. At the time of create or first access, the home of a file creates an entry in its DT. This DT entry is cached on both the manager node and the node accessing the file. Once the home information is cached, the home can be contacted directly for subsequent accesses to the file. Requests to open a file are always forwarded to the home node and the DT entry is updated.

The directory tables serve as a cache of the global name space, which is stored in a distributed manner in the manager nodes of the virtual directories. Since directory tables are stored in volatile memory and entries are created only on file access, the name space in FedFS is not persistent and is created on demand.

B. File Access Protocol

In this section, we explain the protocol for file access in FedFS. Fig. 3 shows how FedFS handles file system API calls made by applications. Fig. 4 shows how server nodes handle protocol requests generated by other nodes. For all requests shown in these figures, the wait for response is implicit.

- `create:` In order to create a file or directory, a server node first queries the manager to find the home, and then contacts the home. The home node sends an `add_entry` request to update the virtual directory at the manager of the parent directory, and creates the file if it doesn't exist already. The home node, which is the physical location of a file, is decided at the time of creation by the manager of the file. Various policies could be used to place the requested file.
  In our experiments, we have used a policy of placing the file on the manager node. We have also evaluated a round-robin file placement

```
create:
 get home_info from file.manager
 if (file.home == self) {
    send add_entry request to parent_dir.manager
    create file if it doesn't exist already
    create entry for file in DT
 } else {
    send create request to file.home
    cache DT entry from response
 }


delete:
 lookup file.home
 if (file.home == self) {
    delete file
    send del_entry request to parent_dir.manager
    delete DT entry for file
 } else {
    send delete request to file.home
    delete cached DT entry for file
 }


open:
 if (file.home unknown) {
    get home_info from file.manager
 }
 if (file.home == self) {
    open file
    register open in DT entry
 } else {
    send open request to file.home
    cache DT entry from response
 }


close:
 if (file.home == self) {
    close file
    register close in DT entry
 } else {
    send close request to file.home
 }
```

Fig. 3. FedFS API Stubs

AT THE HOME OF A FILE

`create request:`
send `add_entry` request to parent_dir.manager
create file if it doesn't exist already
create entry for file in DT
send response with DT entry

`delete request:`
send `del_entry` request to parent_dir.manager
delete entry for file in DT

`open request:`
open file
register open in DT entry for file
send response with DT entry

`close request:`
close file
register close in DT entry for file
send response

AT THE MANAGER OF A FILE

`home_info request:`
determine home of file
send response with home info

AT THE MANAGER OF A DIRECTORY

`add_entry request:`
add entry for file in dir structure
send response

`del_entry request:`
delete entry for file in dir structure
send response

Fig. 4. FedFS Request Handlers

policy in which the `create` requests received at any server node are distributed across the servers in a round-robin fashion.

- `delete:` A lookup is performed to identify the home of the file and the `delete` request is forwarded to the home node. The home node deletes the file and sends a `del_entry` request to update the virtual directory at the manager of the parent directory.
- `open:` A lookup is performed to identify the home of the file and an `open` request is sent to the home node. The home node opens the file, updates the directory table entry for the file and returns a dummy descriptor.
- `close:` The `close` request is sent to the home of the file. The home node closes the file and updates the directory table entry for the file.
- `read/write:` The first access to any data block of a file has to be handled by the home node where the file resides physically. FedFS caches data blocks of files located in other server nodes in the cluster, thus optimizing subsequent accesses to the cached data blocks. The blocks are cached at the time of first access and an LRU replacement policy is used for this data block cache. Writes are performed synchronously using a write-through mechanism.

V. FedFS Implementation

FedFS is implemented as a multithreaded user level I/O library and exports the standard file system API to applications. FedFS uses VIA to implement low overhead remote memory communication among servers and relies on the local file systems to perform file I/O operations. Linking with the FedFS library enables server applications to access storage distributed across the cluster. The communication model implemented in FedFS is explained in the next subsection, followed by a separate section explaining the receive processing performed in FedFS.

A. Communication Model

On each server node, FedFS uses a pool of worker threads to handle the communication among the server nodes. Communication in FedFS involves two type of messages, request and reply. Request messages are sent either by application threads or by worker threads as part of the file access protocol. Reply messages are sent by worker threads after processing incoming requests. Currently, we use a synchronous model for processing in the application threads as well as the worker threads, i.e., a thread waits for the reply after sending out a request.

On each server node, at the time of initialization, a pair of VI channels is established with every node in the cluster. One VI channel is used exclusively for data transfers using RDMA, and the other channel is

used for the rest of the request/response communication. Currently, the RDMA channel is used only to send the response for a read request. If the response is expected to include bulk data, a handle to the reply region is included in the request header so that the bulk data can be transferred using RDMA. All other communication uses the VIA send/receive model.

Descriptors and buffers used in communication are allocated and registered at initialization time, eliminating this overhead from the critical path. For the send operation, threads do not wait in the critical path for data transmission to complete. The send descriptors are reaped from the send queue only on demand during a subsequent send.

B. Receive processing

A receive can be performed only by a worker thread whereas a send can be performed by any thread in the system. A worker thread gets mutually exclusive access to the completion queue and waits to receive messages from other servers. On receiving a message, the worker thread checks the message header to identify the message type. A request message is processed in the context of the receiving thread. The request message includes the identifier of the sender thread. The receiving thread copies this thread identifier into the reply message. A reply message is dispatched to the appropriate waiting thread using the thread identifier included in the header.

We also experimented with polling instead of waiting on the completion queue for the receive operation. However, the use of polling by the receiving thread resulted in wastage of CPU time that could have been used by application threads or other worker threads. We also tried using a dedicated processor for polling on an SMP system, by binding the worker threads to a single processor. However, this did not help since dedicating a processor to handle communication was an overkill in most of the scenarios we studied.

VI. Experimental Results

In this section, we present performance results from our experiments using Federated DAFS on a cluster of eight PCs. Each server node in the cluster was equipped with dual 300 MHz Pentium II processors, 512 MB SDRAM, a 9 GB 10K RPM SCSI Quantum hard disk and a GigaNet cLAN adapter. All the server nodes ran the Linux-2.4.16 kernel.

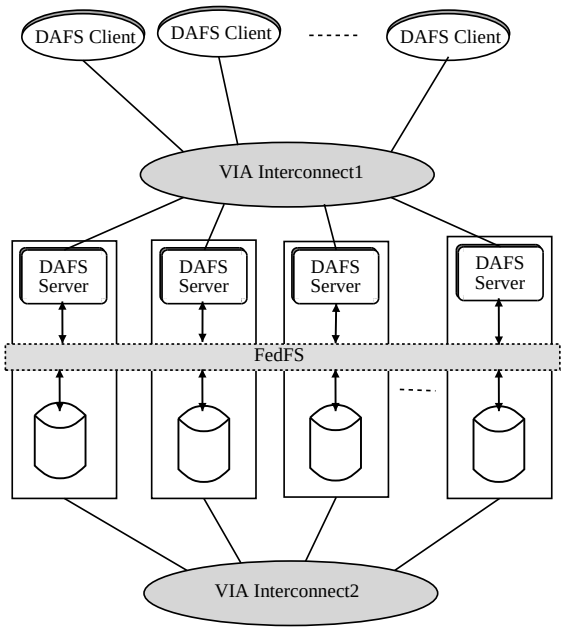

Fig. 5. Experimental setup

An implementation of VIA over cLAN was used for both client-server communication and intra-server communication. The Linux drivers for GigaNet cLAN were able to achieve a bandwidth of 105 MB/s and a one-way latency of 8 μs(one-byte packets). The eight server nodes were connected using a dedicated 8-port Emulex switch. The server nodes were connected to the clients using a 32-port Emulex switch in full bandwidth configuration. Fig. 5 shows the architecture for client-server communication as well as intra-server communication used in our experimental setup. We also performed our experiments using an alternate configuration in which the same VIA interconnect was used for both client-server communication and intra-server communication. The results obtained with the alternate configuration were identical to those reported in this section.

A. Workload

In our experiments, we have used postmark [7], a synthetic benchmark aimed at measuring file system performance over a workload composed of many short-lived, relatively small files. Postmark

TABLE I

DISTRIBUTION OF FILES USING THE HASH FUNCTION

# Servers	2	4	8
# Clients	6	10	16
Server 1	452	377	287
Server 2	448	363	305
Server 3	-	374	295
Server 4	-	385	311
Server 5	-	-	311
Server 6	-	-	295
Server 7	-	-	307
Server 8	-	-	289

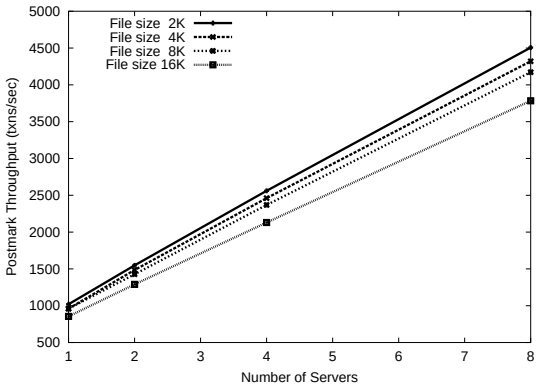

Fig. 6. Postmark throughput for various file sizes

workloads are characterized by a mix of metadata-intensive operations. The benchmark begins by creating a pool of files, performs a sequence of transactions and concludes by deleting all the files created. Each transaction consists of two operations - a randomly chosen create or delete paired with a randomly chosen read or write.

In our experiments, each client issued 30000 transactions. Multiple postmark clients were used to measure maximum throughput sustained by each server cluster configuration, with each client configured to use a request set of 150 files. For each configuration, we measured the maximum throughput sustained by the Federated DAFS server, by increasing the number of clients until the server CPUs reached saturation.

B. Request distribution

FedFS uses a hash function on the pathnames to identify the manager for each file. In our experiments, FedFS has been configured to place files on the manager node unless mentioned otherwise. The clients apply the same hash function on file pathnames to distribute requests to the servers in the cluster. This makes sure that file requests from the clients are sent to the server on which the file is located. Table I shows the distribution of requested files across the servers, obtained by applying the hash function on the request trace of pathnames generated by the postmark benchmark.

C. Throughput and Speedup with Postmark

In Fig. 6, we show the throughput obtained with the postmark benchmark for file sizes ranging from 2KB to 16KB, for various cluster sizes. In this experiment, each transaction performed a create/delete coupled with a file read operation (no writes). We see that for larger file sizes (16KB), the performance drops since the latency from client-server data transfers dominates. With the help of the FedFS clustering layer, we were able to achieve speedups of 2.6 on a four node cluster and 4.5 on a eight node cluster relative to throughput on a single node. We are currently optimizing the performance of the FedFS layer by using file migration to relocate files based on load information, and we believe that this will help us achieve better speedups.

D. File Placement and Caching

In the previous experiment, none of the requests arriving at any server node translated to a remote file access across the cluster. This was achieved by matching the request distribution scheme used by the clients with the file placement scheme on the servers. If the above schemes do not match, client requests arriving at a server node could result in file access to a remote server node. To study the impact of the file placement policy on the communication overhead, we implemented a round-robin policy as an example. When the round-robin policy is used, file create requests arriving at any node are assigned home nodes in a round-robin fashion. In such a scenario, the number of requests at any server node that translate into local file access is approximately 1/N of the total, where N is the size of the cluster. The remaining requests result in remote requests for file access across the cluster. We verify this with an experiment using a round-robin policy for file

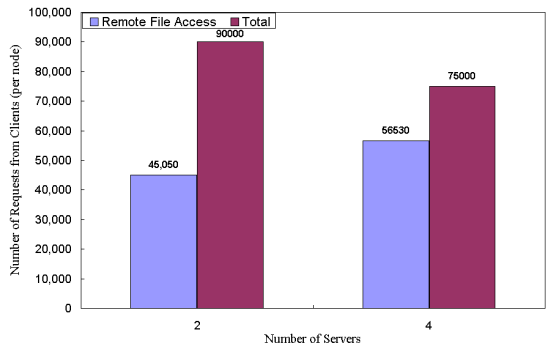

Fig. 7. Communication overhead using round-robin file placement

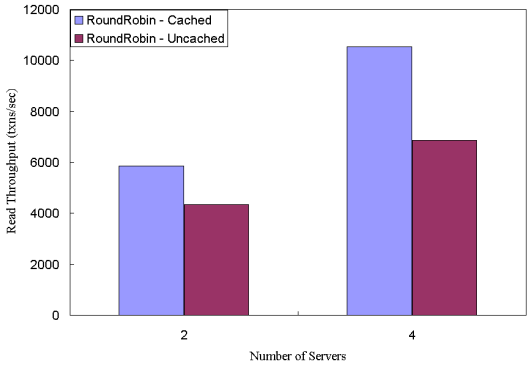

Fig. 8. Impact of data block caching

placement on the servers. Fig. 7 shows the average number of client requests on each server node that translate into remote file accesses.

FedFS uses caching to minimize the communication overhead resulting from remote file access. Using the round-robin policy for file placement and a modified postmark benchmark, we evaluate the effectiveness of caching in minimizing communication overhead. In Fig. 8, we present the results of our experiment with a modified postmark benchmark, in which all files are created before running the transactions, and each transaction performs only a read operation. We can see that the caching layer in FedFS helps improve the throughput by about 35% for the scenario studied.

VII. Conclusions

This paper explores the issues related to building a scalable cluster-based direct access file server. Using features like direct data transfer and RDMA, Direct Access File Systems offer significant improvement in application performance by reducing overheads. FedFS exploits low overhead user-level networking for intra-cluster communication to present a low overhead clustering solution for servers. Federated DAFS combines an efficient user-space DAFS implementation with a thin clustering layer (FedFS) to present a scalable clustering solution for DAFS servers. Federated DAFS uses a portable mechanism for distribution and handling of client requests across the servers in the cluster. Federated DAFS minimizes the intra-cluster communication by caching data blocks of remote files and by matching the file placement on the servers with the distribution of requests from the clients.

Our results show that reasonable speedups can be achieved using Federated DAFS on server clusters of up to eight nodes. Federated DAFS has the potential to achieve better performance than demonstrated with the planned optimizations to migrate files based on load information on the servers.

Acknowledgment

The authors would like to thank Florin Sultan and Aniruddha Bohra for their help in preparing the final version of this manuscript.

References

[1] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd, “The Virtual Interface Architecture,” *IEEE Micro*, vol. 18, no. 2, 1998.

[2] A. Basu, V. Buch, W. Vogels, and T. von Eicken, “U-Net: A User-Level Network Interface for Parallel and Distributed Computing,” in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[3] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg, “A Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer,” in *Proceedings of the 21st Annual Symposium on Computer Architecture*, Apr. 1994, pp. 142–153.

[4] J. Katcher and S. Kleiman, “An Introduction to the Direct Access File System,” in *Whitepaper (www.dafscollaborative.org)*, Jun 2000.

[5] E.V. Carrera and R. Bianchini, “Efficiency vs. Portability in Cluster-Based Network Servers,” in *8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2001.

[6] S. Gopalakrishnan, A. Arumugam and L. Iftode, “Federated File Systems for Clusters with Remote Memory Communication,” in *FAST (Work in Progress Session)*, 2002.

[7] J. Katcher, “Postmark: A New File System Benchmark,” Network Appliance, Tech. Rep. 3022, October 1997.

[8] J. Chase, K. Yocum, and A. Gallatin, “End-System Optimizations for High-Speed TCP,” IEEE Communications, special issue on TCP Performance in Future Networking Environments, vol. 39 no. 4, April 2001, 2001.

[9] M. Thadani and Y. Khalidi, “An Efficient Zero-Copy I/O Framework for UNIX,” 1995.

[10] V. S. Pai, P. Druschel, and W. Zwaenepoel, “IO-Lite: A Unified I/O Buffering and Caching System,” *ACM Transactions on Computer Systems*, vol. 18, no. 1, pp. 37–66, 2000.

[11] Matt DeBergalis, Peter Corbett, Steve Kleiman, Arthur Lent, Dave Noveck, Tom Talpey, and Mark Wittle, “The Direct Access File System,” in *Proceedings of the 2nd Usenix Conference on File Storage and Technologies*, 2003.

[12] K. Magoutis, S. Addetia, A. Fedorova, M.I. Seltzer, J.S. Chase, A.J. Gallatin, R. Kisley, R.G. Wickremesinghe, and E. Gabber, “Structure and Performance of the Direct Access File System,” in *USENIX Annual Technical Conference*, 2002.

[13] Kostas Magoutis, “Optimistic Direct Access File System,” in *Proceedings of the 1st Workshop on Novel Uses of System Area Networks*, 2002.

[14] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel, “Scalable Content-Aware Request Distribution,” in *USENIX Annual Technical Conference*, June 2000.

[15] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum, “Locality-Aware Request Distribution in Cluster-based Network Servers,” in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[16] E.V. Carrera, S. Rao, L. Iftode, and R. Bianchini, “User-Level Communication in Cluster-Based Servers,” in *8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2001.

[17] M. Welsh, D. Culler, and E. Brewer, “SEDA: An Architecture for Well-Conditioned, Scalable Internet Services,” in *Proceedings of the Eighteenth Symposium on Operating Systems Principles*, October 2001.

[18] B. Bloom, “Space/time tradeoffs in hash coding with allowable errors,” *CACM*, vol. 13, no. 7, pp. 422–426, 1970.