Building a User-level Direct Access File System over Infiniband

Murali Rangarajan and Liviu Iftode
Department of Computer Science,
Rutgers University, Piscataway, NJ 08854-8019
{muralir, iftode}@cs.rutgers.edu

***Abstract*— In this paper, we present the design and implementation of a user-space Direct Access File System (DAFS) over Infiniband using channel access and portable programming interfaces viz., the Verbs API (VAPI) and the User Direct Access Programming Library (uDAPL). We present an implementation of DAFS using the Virtual Interface Architecture (VIA) for comparison.**

We discuss design issues in providing a high performance DAFS implementation over Infiniband: descriptor and buffer management, flow control, completion mechanisms and zero-copy data transfers through efficient use of scatter-gather and Remote Direct Memory Access (RDMA). We present optimizations to reduce overheads in buffer/descriptor management as well as memory registration/deregistration. We also show how an extended DAFS session management API could be used by applications to query session attributes and achieve better performance using DAFS. We present a performance evaluation to demonstrate bandwidth characteristics of our DAFS implementation, and the impact of reducing overheads in I/O request processing and memory registration.

I. Introduction

Networking architectures such as Infiniband [1], and Virtual Interface Architecture (VIA) [2], designed to achieve low-latency and high-bandwidth in a System Area Network (SAN) environment, offer an attractive solution for reducing communication software overheads in network storage systems. These networking architectures rely on two key features to provide low latency, high throughput communication - *(i)* User-level networking and *(ii)* Remote Direct Memory Access (RDMA).

The Infiniband Architecture (IBA) [1] is envisioned as the default communication fabric for future SANs. IBA provides a channel-based, switched fabric, interconnect architecture for servers. The Infiniband I/O fabric provides reliable low latency communication for servers in clustered environments.

Programming interfaces for Infiniband may be broadly classified into two main categories, *(i)* channel access interfaces (provided by adapter vendors), that communicate directly with the drivers of channel adapters, and *(ii)* portable interfaces, layered on top of these channel access interfaces hiding the platform specific details.

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

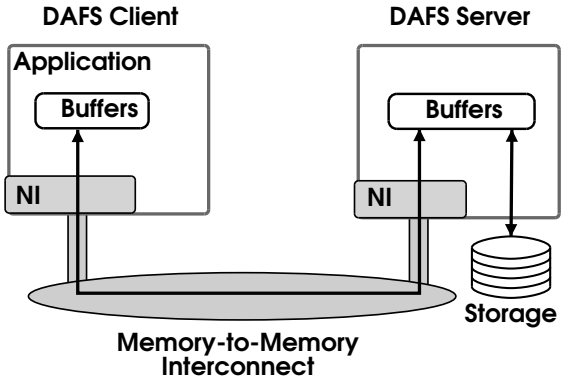


Fig. 1. Data Transfer Path in DAFS

The Direct Access File System (DAFS) [3] protocol is specifically designed to take advantage of memory-to-memory interconnect capabilities in clustered environments. DAFS enables zero-copy data transfers in and out of application buffers by allowing applications to access the network interface (NI) directly from user-space. Figure 1 shows the data transfer path between application buffers on the DAFS client and the storage physically located on the DAFS server.

In this paper, we present our implementation of the DAFS protocol over Infiniband in user-space, using both channel access and portable programming interfaces. We use the Mellanox Verbs API (VAPI) [4] for the channel access interface, and the User Direct Access Programming Layer (uDAPL) [5] for the portable interface.

We present a performance evaluation of our DAFS implementations over these interfaces and also provide a comparison with a VIA-based implementation. To our best knowledge, this is the first study of a DAFS implementation over Infiniband. To reduce overheads in memory registration/deregistration, we use a combination of the Fast Memory Registration (FMR) mechanism provided by VAPI and a lazy caching scheme to

amortize registration costs. We show that applications can use an extended DAFS session management API to query session attributes and harness the benefits of the lazy caching scheme. We combine a selective signaling scheme with pre-allocated descriptor sets to reduce the overheads in request initiation and completion handling.

We present a performance evaluation that covers performance characteristics for workloads with small and large file sizes. Using our performance evaluation, we show the following:

- *Bandwidth and CPU Utilization:* We show that applications are able to achieve more than 90% of the available bandwidth using our DAFS implementations on VIA and Infiniband. Using our DAFS implementation over Infiniband, we show that applications are able to achieve a read bandwidth of up to 675 MB/s. Based on the ratio of CPU utilization to achieved bandwidth, we predict that our DAFS implementations will not saturate the processors on our server if a higher bandwidth (1 GB/s) is available from the underlying transports.
- *Optimizations in Descriptor Management:* We demonstrate the performance impact of reducing the overheads resulting from buffer/descriptor management and memory registration. For metadata intensive workloads and small file I/O, we show improvements of up to 50% in throughput using our strategy for buffer and descriptor management.
- *Optimizations in Memory Registration/Deregistration:* For applications that use memory registration in the critical path of data transfers, performance can be improved by up to 35% using techniques supported in VAPI. We also demonstrate additional gains in throughput (about 25%) for applications that use our extended session management API to achieve a better management of data buffers.
- *VAPI vs uDAPL:* We show that programming using the VAPI layer results in higher performance compared to using uDAPL. uDAPL provides portability with bandwidth close to that of VAPI but is not suited for applications that are latency sensitive.
- *Comparison with VIA:* The VIA-based implementation exhibits good performance for workloads that are metadata intensive and perform small file I/O. However, it is limited by available bandwidth in the underlying network and does not support features like RDMA Read.

The rest of the paper is organized as follows. Section II describes the background and related work. Section III presents an overview of the DAFS protocol. Section IV presents a description of our user-space DAFS implementation. Section V presents the issues in the implementation of DAFS over Infiniband and VIA, and describes the strategies used in our implementation to reduce overheads. Section VI presents the results from our experimental evaluation, and Section VII presents the conclusions.

II. Background and Related Work

A. User-level Communication and RDMA

User-level communication aims to significantly reduce the overhead associated with TCP/IP networking by bypassing the operating system in the common send-receive path. It provides applications direct access to the network interface (NI). The application must register memory buffers used in data transfers with the OS/NI. The registration locks the appropriate pages in physical memory, allowing for direct DMA operations to and from user memory buffers. The operating system is involved only for connection management and memory registration operations. Previous work in user-level communication [6], [7] has led to the development of standards like VIA [2] and IBA [1].

Infiniband associates a pair of work queues with each communication end point called a Queue Pair (QP). Applications access the NI by identifying themselves with a communication endpoint and posting requests on work queues associated with the QP. These requests are called descriptors. The network interface controller (NIC) asynchronously processes these posted descriptors on the Work Queues, and marks them complete when they are done. It is the application's responsibility to remove the completed descriptors from the queues.

User-level networking architectures include the Remote DMA (RDMA) model for data transfers in addition to the conventional Send/Receive model. The RDMA model allows applications to read/write remote memory without interrupting or consuming host processor cycles on the remote node. In the RDMA model, the initiator of the data transfer specifies both the source buffer and the destination buffer of the data transfer. In general, RDMA operations do not consume descriptors on the remote node's receive queue, and no notification is signaled to the remote node that the request has completed.

B. Programming Interface

IBA describes the service interface between a host channel adapter (HCA) and the operating system by a set of semantics called Verbs. Verbs describe operations that take place between a HCA and the operating system based on a particular queuing model for submitting work

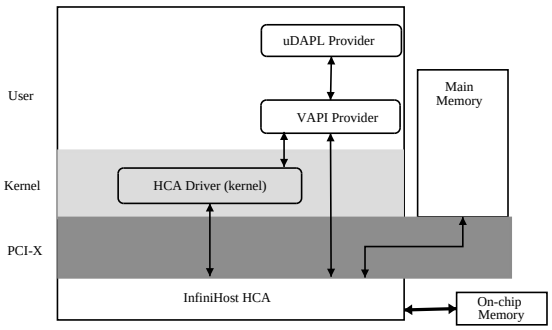


Fig. 2. Infiniband Architecture

requests to the channel adapter and returning completion status. Programming interfaces for Infiniband may be broadly classified into two categories, *(i) Channel access interfaces* that communicate directly with the HCA and *(ii) Portable interfaces* that operate above the channel access interfaces, thereby providing portability across different hardware platforms and implementations.

The Mellanox IB-Verbs API (VAPI) [4] is a channel access interface for Infiniband verbs provided by Mellanox [8]. The most notable among the portable interfaces are the User Direct Access Programming Library (uDAPL) [5] and the Sandia Portals interface [9]. uDAPL provides a generic API for network adapters capable of RDMA. uDAPL provides a common event model for notifications, connection management events, data transfer operations, asynchronous errors and other notifications.

Figure 2 shows the architecture of an Infiniband-based platform with VAPI and uDAPL. The HCA Verbs Provider Driver is the lowest level of software in the operating system and interfaces with the hardware directly. The HCA driver consists of two parts, a kernel mode driver, and a user mode library.

C. Related Work

Zhou et al [10] showed that the VI architecture can be used to improve I/O performance between the database system and the storage back-end. In [11], the author describes a storage-networking transport layer for the Lustre file system based on VI-like networks. An evaluation of file systems over various storage area networks was presented by Bancroft et al [12]. A client/server communication middleware over system area networks was proposed by Liu et al [13]. Other work has focused on using Infiniband transports to provide high performance MPI implementations in a cluster [14], [15].

Recent work has explored the performance characteristics of DAFS and demonstrated that lower client overhead using DAFS can improve application performance over optimized NFS when application processing and I/O demands are well-balanced [16]. The use of RDMA as an RPC transport for NFS was proposed and higher throughput was reported, compared to using a a conventional TCP transport [17].

More recently, work has been done on using Infiniband transports to build cluster-based file systems [18]. This work is similar to our work in exploring the use of Infiniband as a transport to build high performance file systems. They present a transport-layer implementation designed to achieve good performance for applications using PVFS [19]. In contrast, we present the first study of Direct Access File Systems over Infiniband and present techniques to reduce overhead in the descriptor handling and the memory registration/deregistration mechanisms.

III. Direct Access File System Protocol

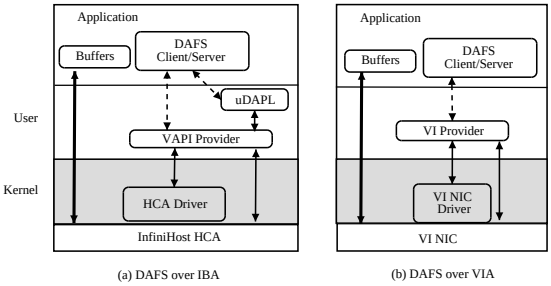
The Direct Access File System (DAFS) [20] is a file access and management protocol designed for local file-sharing or clustered environments. It addresses two primary goals: *(i)* Provide low-latency, high-throughput, and low-overhead data movement that takes advantage of modern memory-to-memory networking technologies like Infiniband [1] and VIA [2]. *(ii)* Define a set of file management and file access operations for local file-sharing requirements.

The DAFS API provides a simple and convenient file-oriented programming interface that hides many of the details of the DAFS protocol. It hides transport-layer semantics like reliability levels but exposes features like memory registration to applications, to enable zero-copy data transfers to/from application buffers. The DAFS API defines new `read` and `write` operations that include the memory address of the client's destination/source buffer.

Session Management: DAFS communication is a session-based protocol that utilizes a request-response model of message exchange between client and server. The session design incorporates a number of attributes including authentication, features related to segments of the file system name space, message flow control, and transport-level resource management.

Protocol Messaging: The request and response messages exchanged as part of the DAFS communication protocol are divided into two categories:

- *Messages that are bounded in size by the file access protocol.* These are typically protocol messages that are sent as request or response without involving any data transfer to/from the application.
- *Messages that transfer large variable-sized (usually greater than 1 KB) bulk user data.* e.g., the `dap_read` and `dap_write` operations.



(a) DAFS over IBA (b) DAFS over VIA

Fig. 3. DAFS Implementation Architecture

IV. Direct Access File System Implementation

In this section, we present our implementation of the DAFS protocol. We start with a description of our DAFS implementation architecture. We present our implementation of the DAFS session management operations along with our proposed extensions that help applications achieve better resource management. We present our *lazy cache* scheme to reduce memory registration/deregistration costs by combining it with our session management extensions. We describe the descriptor/buffer management scheme we have used in our implementation to reduce the overhead of handling descriptor completions. Finally, we present our implementation of zero-copy data transfers in and out of application buffers. In this section, we present only implementation details that are common across all three implementations (over VAPI, uDAPL and VIA).

A. DAFS Implementation Architecture

Our client and server prototypes implement the DAFS protocol entirely in user-space, making it portable across any POSIX-based system. Our DAFS implementation supports the complete DAFS API including operations for asynchronous I/O and session management. Figure 3 *a* and *b* show the architecture of our DAFS implementations using IBA and VIA respectively. The thick arrows show the data transfer path in and out of application buffers and the dotted arrows show the interaction of the DAFS client with the transport provider library.

Our DAFS client is provided to applications as a user-level library. When the application invokes a DAFS API primitive, the library translates it into an RPC request for the corresponding I/O operation on the server. Application threads can choose between using the synchronous or asynchronous API for I/O, providing the application better control and use of I/O concurrency.

Our DAFS server implementation uses a connection manager thread, along with a pool of protocol handler threads. The connection manager thread handles connection requests from clients and assigns a protocol handler thread to each established client session. This protocol handler thread receives DAFS requests from the client, performs the required I/O operation and sends the results back to the client. We use a pool of protocol handler threads which expands in size to a configurable maximum based on the load on the DAFS server.

B. Extended Session Management API

Our implementation supports the session operations specified by the DAFS protocol that cover connection/session establishment and credential registration. At the time of session set up, the client establishes various session attributes with the server. Establishing these attributes enables clients and servers to manage their transport-level resources like protocol communication buffers and descriptors on a per-session basis.

DAFS enables zero-copy data transfers in and out of application buffers, by allowing applications to manage data buffers used in communication. It is possible for applications to implement an efficient buffer management scheme if they can use information about the transport-level resources allocated by the DAFS implementation. We propose extensions to the DAFS API to query session attributes established at the time of session set up. Using these attributes, an application can tailor its buffer management scheme according to the transport-level resources associated with the session.

C. Memory Registration Using Lazy Cache

Memory registration and deregistration are expensive operations since they require pinning/unpinning of pages in physical memory and accessing the on-chip memory of the HCA/NIC. Our DAFS client uses a *lazy cache* which combines a cache of registration mappings with a lazy approach to memory deregistration. The caching strategy is similar to the pin-down cache mechanism proposed by Tezuka et al [21]. This cache is implemented using an LRU list combined with a fixed size hash table for fast access. Our DAFS client adopts a lazy approach to memory deregistration by postponing the actual deregistration until the hash table is full. This provides scope for re-using registration mappings if the same virtual address is registered multiple times.

Although this technique has been used previously [18], [21], we propose a novel mechanism for applications to leverage benefits from this technique in the context of DAFS. Applications can use the extended session management API to query the DAFS client for information about transport-level resources including the size of the buffer/descriptor set and the size of the *lazy cache*. Applications can use this information to ensure that their working set of data buffers fit in the *lazy*

cache. This results in a hit in the cache every time an application buffer is re-registered, thus amortizing the cost of memory registration.

D. Buffer Management and Flow Control

Communication using RDMA interconnects require that buffers/descriptors be pre-posted on the receive work queue to receive messages. Our implementations use a window-based flow control scheme for buffers and descriptors used in communication. Clients and servers set their send and receive windows at the time of session setup. The send/receive window sizes are used to allocate and pre-register a set of send and receive descriptors/buffers that are associated with the session. Pre-allocating and registering this descriptor set enables pipelining of data transfers without waiting for completion of previously posted descriptors.

E. Messaging and Data Transfer

Our implementation uses the Send/Receive model for all messages that do not involve data transfer. Fixed size pre-registered buffers are used for sending/receiving such messages. The number and size of these buffers associated with each session are based on the flow control parameters established at the time of setting up the session. The receive buffers are also posted on the receive queue at this time. To send a protocol message which does not carry data, the DAFS implementation builds the header in one of the pre-registered buffers, and copies the application specified parameters into this buffer before sending it out.

Data transfers involve variable sized buffers that are transferred using RDMA or scatter-gather Send/Receive, depending on the transport. All RDMA operations are initiated by the server. On the DAFS client, the application determines when the buffers used in data transfers are allocated and registered.

For `dap_read`, the application includes a handle to a registered buffer in the request, and the server uses RDMA write to send the data directly to the application buffer without any intermediate copies.

For `dap_write`, our Infiniband-based implementations support both client-initiated writes using gather-send and server initiated RDMA Read from client memory. In our VIA-based implementation, data transfers from the client to server in `dap_write` are supported only using client-initiated writes, since the underlying transport provider does not support RDMA Read. For client-initiated writes, the client uses gather-send to transfer the data out of the application buffers without a copy. The message consists of two segments, the header and the data to be transferred. For RDMA Read, the client sends a write request to the server with a handle to the memory region containing the data, and the server uses RDMA Read to fetch the data. The type of transfer (RDMA Read or client-initiated writes) as well as the size of the write buffer, are chosen at the time of session setup.

V. Implementing DAFS over Infiniband

In this section, we present details of our DAFS implementation. We use the terms DAFS-VAPI, DAFS-uDAPL and DAFS-VIA to refer to the respective implementations. We present transport-specific implementation details of various issues including connection management and completion mechanisms. We present an outline of how our implementation uses VAPI features to reduce overheads for memory registration/deregistration and send descriptor handling.

A. Platform Details

We used Mellanox InfiniHost HCAs for the Infiniband transport and cLAN adapters from Emulex for the VIA transport. We used a VAPI provider supplied with the InfiniHost SDK for the VAPI interface. We used a version of the uDAPL library (implemented using VAPI) customized for the Mellanox HCA. For VIA, we used a native implementation provided by Emulex.

B. Connection Management

DAFS-VAPI: A reliable connection-oriented transport model is used by creating QPs with the Reliable Connection (RC) service option. VAPI does not include primitives for connection management and connections are established using a Connection Manager (CM) provided with VAPI. Servers register their services with the CM and the CM calls back the server to accept connection requests, create QPs and change their state. Clients call the CM to issue a connect request and the CM calls back the client to change the QP state, according to the connection status and parameters.

DAFS-uDAPL: uDAPL supports reliable connections using a client-server connection model. The DAFS server uses a dedicated connection acceptor thread to accept client connection requests. Once a connection request is accepted, an EndPoint (EP) is created and passed to a protocol handler thread for serving client requests. uDAPL maps EPs to queue pairs in the underlying VAPI layer and uses the CM service provided with the VAPI layer for connection management.

DAFS-VIA: Virtual Interfaces (VIs) are created with the Reliable Delivery option and explicitly connected using a client-server connection model. A dedicated

connection acceptor thread is used on the DAFS server to accept incoming connections. The newly created VI as a result of accepting a connection is passed to a protocol handler thread for serving client requests.

C. Completion Mechanisms

DAFS-VAPI: With VAPI, the completion of requests is reported through Completion Queues (CQs). The CQs associated with a QP are specified at the time of creating the QP. VAPI supports two models for request completion notifications:

- A callback-based mechanism that allows asynchronous notifications. Completion notifications can be selectively enabled for a batch of requests or even on a per-request basis.
- Checking the CQ for request completions using polling or blocking.

DAFS-uDAPL: The results of nearly all operations in the uDAPL interface are communicated through asynchronous events. Event Dispatcher (EVD) objects are created for various event streams associated with an EP. EVDs are queues to collect notifications for connection requests, connection establishment, disconnect notifications, data transfer completions, memory bind completions, asynchronous errors, and software generated events. These EVDs are associated with the EP at the time of creation of the EPs. Consumers place an operation on the EVD queues for processing and either poll or wait on the appropriate EVD queue for the result. The uDAPL interface does not support a callback-based notification or a selective signaling mechanism on a per-event basis like VAPI. In our implementation, each EP is associated with its own set of EVDs. On the DAFS server, this enables the protocol handler thread mapped to the EP to use these EVDs without any locking issues.

DAFS-VIA: Each VI is associated with send and receive work queues. Each VI work queue is optionally associated with a Completion Queue (CQ) when the VI is created. Our implementation does not attach any CQs to the VI work queues. On the DAFS server, each protocol handler thread uses its own VI and associated work queues without any locking issues.

D. Reducing Overheads Using VAPI

Fast Memory Registration: In order to reduce the overhead of performing the memory registration in the critical path of data transfers, VAPI provides a fast memory registration (FMR) mechanism as part of an extended API. Using the FMR mechanism, memory registration is performed by first creating an FMR, followed by mapping the FMR to a memory region in a subsequent operation. Our implementation creates a set of FMRs at the time of session setup and maps application buffers on to FMRs when the application invokes `dap_register_mem`. Even though the FMR is mapped to a memory region in the critical path, this is a relatively lightweight operation.

Efficient Send Descriptor Handling: VAPI provides an option in the request descriptor that allows request completions to go unsignaled. This helps amortize the cost of removing the completed descriptor from the work queue before reusing the descriptor or the associated data buffer. In our implementation, only the last descriptor in a set of descriptors associated with a QP has signaling turned on. After the last descriptor is submitted to the work queue, there are no free descriptors and the last descriptor in the CQ is checked for completion. The completion of the last descriptor guarantees the completion of all the previously submitted descriptors since descriptors are guaranteed to complete in the order that they are submitted.

VI. Experimental Results

In this section, we present performance results using a range of applications and benchmarks with our DAFS implementations. The goal is to evaluate the performance of the DAFS implementations over different transports, compare the performance over different programming interfaces, and quantify the impact of the various issues and features we have discussed.

A. Experimental Setup

The client and server nodes used in our experiments were equipped with an Intel E7501 chipset, two Intel Xeon 2.4 GHz P4 processors, 512 KB L2 cache, 400 MHz front side bus, 1 GB DDR SDRAM and 64-bit 133 MHz PCI-X interfaces. Each node was also equipped with a 36 GB, 15K RPM Seagate disk. Both the client and server nodes ran the Linux-2.4.18 kernel.

For the Infiniband-based implementations, we used Mellanox InfiniHost MT23108 DualPort 4x HCA adapters connected through an InfiniScale MT43132 Eight 4x Port Infiniband Switch. For the Mellanox InfiniHost HCA, we used firmware version fw-23108-1.18.000 and SDK version thca-x86-1.0.1. For the VIA-based implementation, we used Emulex cLAN adapters. In Table I, we show the raw bandwidth and four-byte one-way latency for the Infiniband and VIA networks.

B. Applications and Workloads

We used three different applications and multiple workloads for our performance evaluations.

TABLE I
NETWORK CHARACTERISTICS

	Infiniband/Mellanox	VIA/cLAN
Bandwidth (MB/s)	746	112
Latency (μs)	7.5	8

To show the bandwidth achievable by applications for file access, we used *asyncIOperf* to read/write a large file on the server using various block sizes. *asyncIOperf* is a simple benchmark developed by us, that uses DAFS asynchronous I/O operations to perform sequential I/O. It uses asynchronous I/O operations to fill the I/O pipeline to the server with a set of requests. We used a file size (758 MB) that fits in the server physical memory (1 GB), and a warmed up server buffer cache since our intention was to stress the network data transfer path.

To measure the performance small file I/O, we used Postmark [22]. Postmark is a synthetic benchmark that measures file system performance with a workload composed of many short-lived, relatively small files. Postmark 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. The READ/WRITE of a file translates to an `open`, followed by a `read`/`write` of the entire file and a `close`. The CREATE of a file translates to an `open` followed by a `write` of random text and a `close`. DELETE removes a random file from the active set. The client was configured to start with a request set of 200 files and issued 60,000 transactions for each run. All results presented were averaged over ten runs.

To illustrate the performance impact of using the memory registration optimizations, we used Berkeley DB [23]. Stubs were used to translate data access primitives to DAFS operations. These stubs perform memory registration and deregistration in the critical path, on buffers used for data access by the Berkeley DB client core. Berkeley DB manages its own buffering and caching, independent of the underlying file system buffer cache. It can be configured to use a specific block size and internal cache size.

C. Bandwidth and CPU Utilization

In this experiment, we measured the read and write bandwidth obtained with the different DAFS implementations using *asyncIOperf*. The goal is to demonstrate the maximum bandwidth that can be achieved by applications using our DAFS implementations. The application used a pipe of length eight to submit asynchronous I/O requests, and used polling to check for completion of the requests.

Figure 4 shows the read and write bandwidth achieved with our three DAFS implementations for various block sizes. The block size is the data transfer size of each file I/O request made by the application. DAFS-VAPI and DAFS-VIA achieve more than 90% of the available network bandwidth for the underlying transports, reported in Table I. The bandwidth obtained using DAFS-uDAPL is about 10% less than that obtained using DAFS-VAPI due to overheads arising from event management. The maximum read bandwidth of 675 MB/s is achieved by DAFS-VAPI since it uses Infiniband for the transport and has the minimum per-I/O overhead. The read bandwidth achieved is slightly higher than the write bandwidth across all implementations since the read operation uses RDMA Write for data transfer whereas the write operation uses gather-send.

We also evaluated a version of our Infiniband-based DAFS implementation using server-initiated RDMA Read to transfer data for the write operation. However, the write bandwidth we obtained using RDMA Read was lesser than that reported in Figure 4, due to the additional latency in initializing the RDMA Read operation.

Figure 5 shows the ratio of server CPU utilization (in %) to read bandwidth (in GB/s) for various block sizes. The user-level DAFS server performs a data copy from the file system buffers to the communication buffers in the critical path of data transfer, which contributes most to the CPU utilization. The figure can be interpreted to represent the CPU utilization (in %) for the different DAFS implementations, to achieve a read bandwidth of 1 GB/s. Our Infiniband and VIA platforms do not support a bandwidth of 1 GB/s. However, we can interpret the data in Figure 5 to speculate that on a system with similar processing power, our DAFS implementations are not expected to saturate the processors even at a bandwidth of 1 GB/s, for block sizes greater than 4 KB. We observed similar characteristics for write, but we do not present them here.

D. Performance for Small Packet Traffic

In Figure 6, we present the throughput (transactions/second) obtained with *Postmark-Mixed* across all three DAFS implementations, for various file sizes. The goal is to compare the performance of our implementations for workloads that involve small file I/O and/or are metadata-intensive.

We can see that DAFS-VAPI has the highest throughput for file sizes greater than 8 KB but the interesting

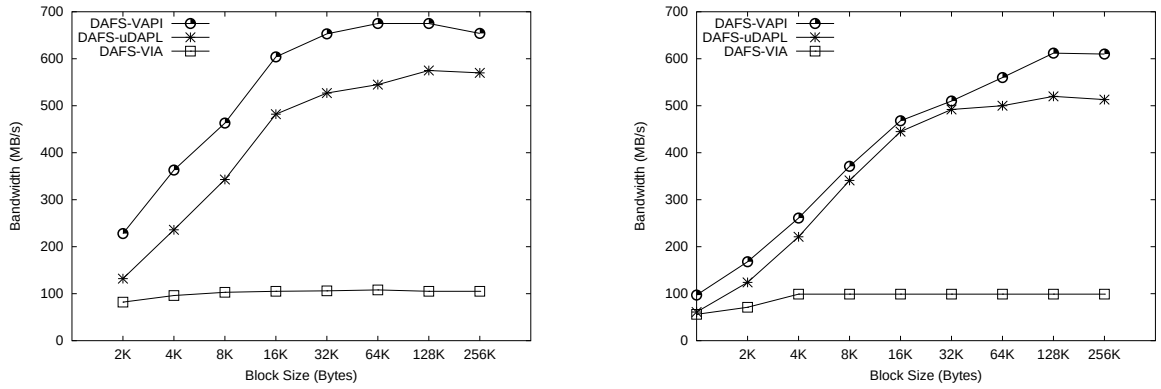


Fig. 4. Read (left) and Write (right) Bandwidth using Asynchronous I/O

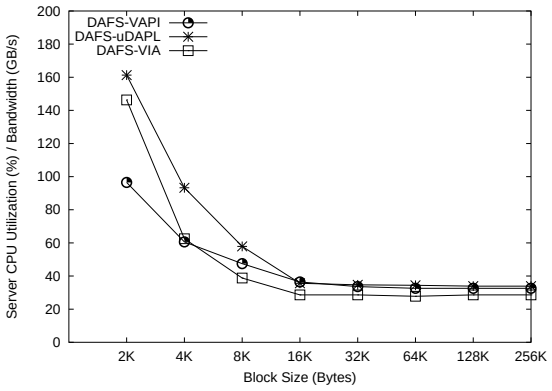


Fig. 5. Ratio of Server CPU Utilization to Read Bandwidth

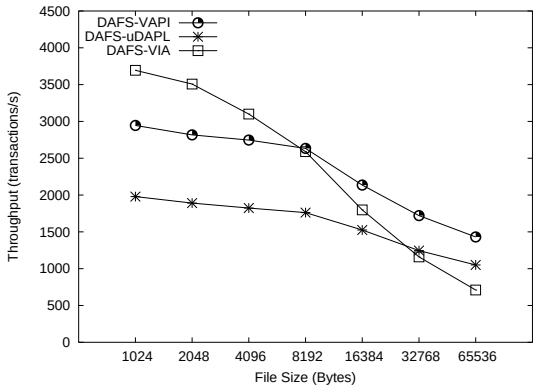


Fig. 6. Postmark Throughput

observation is that DAFS-VIA outperforms the other two implementations for small file sizes. The VIA/cLAN network exhibits good performance for small packet traffic and *Postmark-Mixed* has a high ratio of DAFS control messages to data messages. A careful examination of the message traffic for *Postmark-Mixed* for a file size of 4 KB revealed that small packets dominate the traffic and that the average packet size is small (43 B for the control messages). The main overheads suffered by DAFS-uDAPL result from checking for request completion, where it has to check an internal queue as well as the CQ in the VAPI layer. uDAPL drains the entire CQ to check for a single event completion, in order to support a threshold parameter in the primitive used to wait for event completions. Since the fixed cost of processing an I/O request is dominant for this workload, DAFS-uDAPL performs worse than the other two implementations.

E. Reducing Descriptor Management Overheads

In this experiment, we use the *Postmark-Read* workload with small file I/O to demonstrate the performance impact of the descriptor management strategy used in DAFS-VAPI. In DAFS-VAPI, removing completed descriptors from the send queue involves an `ioctl` to check for completion and a potential wait inside the kernel. The descriptor management scheme helps improve performance by amortizing the cost of checking for request completions. Two strategies are used: *(i)* Using multiple buffers/descriptors for each session, thus avoiding the check for descriptor completion in the critical path of every send operation. *(ii)* Using a selective signaling scheme to enable notifications only on the last descriptor of a descriptor set.

In Figure 7, we compare the *Postmark-Read* throughput using two versions of DAFS-VAPI, for various file sizes (with file size used as the block size). We compare a version that uses a descriptor set of size 32 (DAFS-VAPI32), with another version that uses only one descriptor per session. For small file sizes, we can see an improvement of up to 50% in *Postmark-Read* throughput by using our descriptor management scheme. The gain in throughput comes from using descriptor sets and selective signaling.

F. Optimized Memory Registration Using Lazy Cache and FMR

In this experiment, our goal is to demonstrate the performance impact of using optimizations for memory

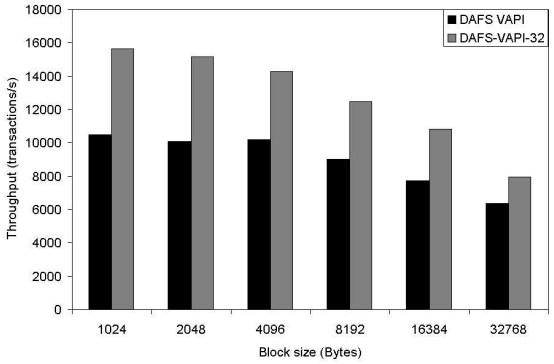


Fig. 7. Impact of Buffer/Descriptor Management

TABLE II
COST OF MEMORY REGISTRATION (μS)

Buffer Size	VAPI	VAPI-FMR	uDAPL
1 KB	61	6	110
2 KB	61	6	110
4 KB	61	6	110
8 KB	74	18	115
16 KB	87	33	126
32 KB	117	63	150
64 KB	174	120	217

registration/deregistration, and the benefit of using our proposed extensions to the DAFS session management API. In Table II, we present the cost of memory registration for various buffer sizes using VAPI and uDAPL. VAPI-FMR shows that using the FMR mechanism provided in VAPI reduces the cost of memory registration by about 55 μs. uDAPL provides a two stage registration process - the first stage pins the buffer in physical memory and the second stage makes the buffer eligible to be the destination of an RDMA operation. A wait for completion of the second stage increases the latency of the uDAPL registration operation.

To illustrate the impact of memory registration on performance, we present experimental results using an application that performs memory registration in the critical path. Figure 8 shows the bandwidth obtained with Berkeley DB for various block sizes, with and without the optimization strategy for memory registration. The workload consisted of a single sequential read of all records in a data set of size 758 MB, from a warmed up buffer cache on the server. For this workload, reading each record involved a memory registration, followed by a `read` and a memory deregistration operation. The memory registration and deregistration costs dominate the overheads for DAFS-VAPI, and using the FMR mechanism improves the throughput by about 35%.

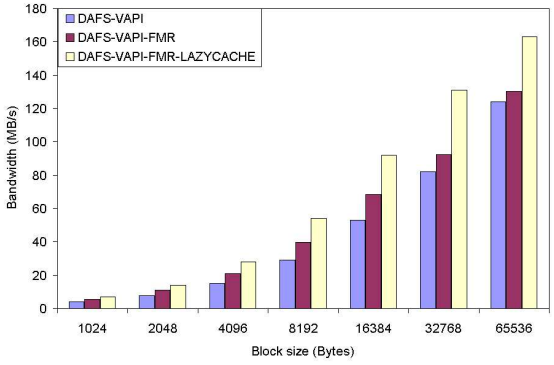


Fig. 8. Berkeley DB Bandwidth

To achieve the maximum benefit from the *lazy cache* scheme (Section IV-C), the entire working set of application data buffers should fit in the *lazy cache*. This ensures that the registration mappings of the entire working set are cached by the DAFS client, resulting in near-zero registration/deregistration costs. For Berkeley DB, the set of buffers used in data access is determined by its internal cache size. We modified Berkeley DB to use our session API extension to query session attributes pertaining to size of the *lazy cache* and size of the descriptor set used in the DAFS implementation. This information was used to tune the Berkeley DB cache size so that *lazy cache* could cache the registration mappings of the entire working set. In Figure 8, DAFS-VAPI-FMR-LAZYCACHE shows an additional performance gain of 25% as a result of this modification.

G. Summary

We showed that applications were able to achieve 90% of the available bandwidth using our DAFS implementations. We demonstrated a read bandwidth of 675 MB/s with our DAFS implementation using VAPI.

The VIA-based implementation exhibits good performance for workloads that are metadata intensive and perform small file I/O. For such workloads, we demonstrated the benefits of reducing overheads from descriptor management. We showed that improvements of up to 50% in throughput could be achieved using our strategy for buffer and descriptor management.

For applications that use memory registration in the critical path of data transfers, we showed that performance could be improved by up to 35% using the fast registration mechanism provided in VAPI. An additional gain of up to 25% in throughput was demonstrated using the *lazy cache* scheme which amortizes the registration and deregistration cost over repeated registrations.

VII. Conclusions

In this paper, we presented the design and implementation of the DAFS protocol over Infiniband using two different programming interfaces viz., VAPI and uDAPL. To our best knowledge, this is the first study of a DAFS implementation over Infiniband, and the first performance characterization of applications using the uDAPL interface. Our Infiniband-based DAFS implementations support zero-copy data transfer operations using RDMA Write, RDMA Read and scatter-gather Send/Receive. We presented design issues involved in the different implementations with the objective of achieving high performance. We presented techniques to reduce overheads related to descriptor management and memory registration/deregistration.

We presented the performance of DAFS implementations over Infiniband using two different programming interfaces (VAPI and uDAPL), and compared the performance with that of a VIA-based implementation. We presented bandwidth characteristics and demonstrated the performance impact of reducing overheads in I/O request processing and memory registration/deregistration. Our implementation using VAPI exhibits the best performance characteristics. Programming at the VAPI layer provides the advantages of a richer API, lower per-I/O overhead and higher bandwidth. DAFS-uDAPL demonstrates good bandwidth but suffers from event management overheads which are dominant for workloads with small packet traffic. DAFS-VIA offers good performance for workloads with small packet traffic but is limited by the available bandwidth in the underlying network.

VIII. Acknowledgements

We thank Mellanox Inc. for donating the Mellanox InfiniHost HCAs and the InfiniScale Switch to enable our research using Infiniband. We also thank Network Appliance for supporting this research.

References

[1] "The Infiniband Trade Association," http://www.infinibandta.org, August 2000.

[2] 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.

[3] 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 and Storage Technologies*, 2003.

[4] "Mellanox IB-Verbs API (VAPI). Mellanox Technologies Inc." 2001.

[5] James Lentini, Vu Pham, Steven Sears, and Randall Smith, "Implementation and Analysis of the User Direct Access Programming Library," in *2nd Workshop on Novel Uses of System Area Networks, SAN-2*, February 2003.

[6] 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.

[7] 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.

[8] "Mellanox Technologies Inc. http://www.mellanox.com."

[9] Ron Brightwell, Arthur B. Maccabe and Rolf Riesen, "The Portals 3.2 Mesage Passing Interface Revision 1.1," in *Sandia National Laboratories*, November 2002.

[10] Yuanyuan Zhou, Kai Li, Angelos Bilas, Suresh Jagannathan, Cezary Dubnicki, and James F. Philbin, "Experiences with VI Communication for Database Storage," in , February 2003.

[11] R. Zahir, "Lustre Storage Networking Transport Layer," in *http://www.lustre.org/docs.html*, 2002.

[12] M. Bancroft, N. Bear, J. Finlayson, R. Hill, , R. Isicoff, and H. Thompson, "Functionality and Performance Evaluation of File Systems for Storage Area Networks (SAN)," in *Eighth NASA Goddard Conference on Mass Storage Systems and Technologies*, 2000.

[13] J. Liu, M. Banikazemi, B. Abali, and D. K. Panda, "Portable Client/Server Communication Middleware over SANs: Design and Performance Evaluation with Infiniband," in *2nd Workshop on Novel Uses of System Area Networks, SAN-2*, February 2003.

[14] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Peter Wyckoff, and Dhabaleswar K. Panda, "High Performance RDMA-Based MPI Implementation over InfiniBand," in *Proceedings of 17th Annual ACM International Conference on Supercomputing*, June 2003.

[15] Vijay Velusamy, Changzheng Rao, Srigurunath Chakravarthi, Jothi Neelamegam, Weiyi Chen, Sanjay Verma and Anthony Skjellum, "Programming the Infiniband Network Architecture For High Performance Message Passing Systems," in *ISCA 16th International Conference on Parallel and Distributed Computing Systems (PDCS-2003)*, August 2003.

[16] 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.

[17] Brent Callaghan, Theresa Lingutla-Raj, Alex Chiu, Peter Staubach, and Omer Asad, "NFS over RDMA," in *Workshop on Network-I/O Convergence: Experience, Lessons, Implications*, 2003.

[18] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda, "PVFS over InfiniBand: Design and Performance Evaluation," in *International Conference on Parallel Processing (ICPP 03)*, October 2003.

[19] P. H. Carns, W. B. Ligon III, R. B. Ross, R. Thakur, "PVFS: A Parallel File System For Linux Clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*, 2002.

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

[21] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa, "Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication," in *12th Int. Parallel Processing Symposium*, March 1998, pp. 308–315.

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

[23] M. Olson, K. Bostic, and M. Seltzer, "Berkeley DB," in *USENIX Annual Technical Conference*, June 1999.