Structure and Performance of the Direct Access File System

Kostas Magoutis, Salimah Addetia, Alexandra Fedorova, Margo I. Seltzer
Division of Engineering and Applied Sciences, Harvard University
Jeffrey S. Chase, Andrew J. Gallatin, Richard Kisley, Rajiv G. Wickremesinghe
Department of Computer Science, Duke University
Eran Gabber
Lucent Technologies - Bell Laboratories

Abstract

The Direct Access File System (DAFS) is an emerging industrial standard for network-attached storage. DAFS takes advantage of new user-level network interface standards. This enables a *user-level file system* structure in which client-side functionality for remote data access resides in a library rather than in the kernel. This structure addresses longstanding performance problems stemming from weak integration of buffering layers in the network transport, kernel-based file systems and applications. The benefits of this architecture include lightweight, portable and asynchronous access to network storage and improved application control over data movement, caching and prefetching.

This paper explores the fundamental performance characteristics of a user-level file system structure based on DAFS. It presents experimental results from an open-source DAFS prototype and compares its performance to a kernel-based NFS implementation optimized for zero-copy data transfer. The results show that both systems can deliver file access throughput in excess of 100 MB/s, saturating network links with similar raw bandwidth. 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.

1 Introduction

The performance of high-speed network storage systems is often limited by client overhead, such as memory copying, network access costs and protocol overhead [2, 8, 20, 29]. A related source of inefficiency stems from poor integration of applications and file system services; lack of control over kernel policies leads to problems such as double caching, false prefetching and poor concurrency management [34]. As a result, databases and other performance-critical applications often bypass file systems in favor of raw block storage access. This sacrifices the benefits of the file system model, including ease of administration and safe sharing of resources and data. These problems have also motivated the design of radical operating system structures to allow application control over resource management [21, 31].

The recent emergence of commercial *direct-access transport* networks creates an opportunity to address these issues without changing operating systems in common use. These networks incorporate two defining features: *user-level networking* and *remote direct memory access* (RDMA). User-level networking allows safe network communication directly from user-mode applications, removing the kernel from the critical I/O path. RDMA allows the network adapter to reduce copy overhead by accessing application buffers directly.

The Direct Access File System (DAFS) [14] is a new standard for network-attached storage over direct-access transport networks. The DAFS protocol is based on the Network File System Version 4 protocol [32], with added protocol features for direct data transfer using RDMA, scatter/gather list I/O, reliable locking, command flow-control and session recovery. DAFS is designed to enable a *user-level file system client*: a DAFS client may run as an application library above the operating system kernel, with the kernel's role limited to basic network device support and memory management. This structure can improve performance, portability and reliability, and offer applications fully asynchronous I/O and more direct control over data movement and caching. Network Appliance and other network-attached storage vendors are planning DAFS interfaces for their products.

This paper explores the fundamental structural and performance characteristics of network file access using a user-level file system structure on a direct-access transport network with RDMA. We use DAFS as a basis for exploring these features since it is the first fully-specified file system protocol to support them. We describe DAFS-based client and server reference implementations for an open-source Unix system (FreeBSD) and report experimental results, comparing DAFS to a zero-copy NFS implementation. Our purpose is to illustrate the benefits and tradeoffs of these techniques to provide a basis for informed choices about deployment of DAFS-based systems and similar extensions to other network file protocols, such as NFS.

Our experiments explore the application properties that determine how RDMA and user-level file systems affect performance. For example, when a workload is balanced (i.e., the application simultaneously saturates the CPU and network link) DAFS delivers the most benefit compared to more traditional architectures. When workloads are limited by the disk, DAFS and more traditional network file systems behave comparably. Other workload factors such as metadata-intensity, I/O sizes, file sizes, and I/O access pattern also influence performance.

An important property of the user-level file system structure is that applications are no longer bound by the kernel's policies for file system buffering, caching and prefetching. The user-level file system structure and the DAFS API allow applications full control over file system access; however, the application can no longer benefit from shared kernel facilities for caching and prefetching. A secondary goal of our work is to show how *adaptation libraries* for specific classes of applications enable those applications to benefit from improved control and tighter integration with the file system, while reducing or eliminating the burden on application developers. We present experiments with two adaptation libraries for DAFS clients: Berkeley DB [28] and the TPIE external memory I/O toolkit [37]. These adaptation libraries provide the benefits of the user-level file system without requiring that applications be modified to use the DAFS API.

The layout of this paper is as follows. Section 2 summarizes the trends that motivated DAFS and user-level file systems and sets our study in context with previous work. Section 3 gives an overview of the salient features of the DAFS specifications, and Section 4 describes the DAFS reference implementation used in the experiments. Section 5 presents two example adaptation libraries, and Section 6 describes zero-copy, kernel-based NFS as an alternative to DAFS. Section 7 presents experimental results. We conclude in Section 8.

2 Background and Related Work

In this section, we discuss the previous work that lays the foundation for DAFS and provides the context for our experimental results. We begin with a discussion of the issues that limit performance in network storage systems and then discuss the two critical architectural features that we examine to attack performance bottlenecks: direct-access transports and user-level file systems.

2.1 Network Storage Performance

Network storage solutions can be categorized as Storage-Area Network (SAN)-based solutions, which provide a block abstraction to clients, and Network-Attached Storage (NAS)-based solutions, which export a network file system interface. Because a SAN storage volume appears as a local disk, the client has full control over the volume's data layout; client-side file systems or database software can run unmodified [23]. However, this precludes concurrent access to the shared volume from other clients, unless the client software is extended to coordinate its accesses with other clients [36]. In contrast, a NAS-based file service can control sharing and access for individual files on a shared volume. This approach allows safe data sharing across diverse clients and applications.

Communication overhead was a key factor driving acceptance of Fibre Channel [20] as a high-performance SAN. Fibre Channel leverages network interface controller (NIC) support to offload transport processing from the host and access I/O blocks in host memory directly without copying. Recently, NICs supporting the emerging iSCSI block storage standard have entered the market as an IP-based SAN alternative. In contrast, NAS solutions have typically used IP-based protocols over conventional NICs, and have paid a performance penalty. The most-often cited causes for poor performance of network file systems are (a) protocol processing in network stacks; (b) memory copies [2, 15, 29, 35]; and (c) other kernel overhead such as system calls and context switches. Data copying, in particular, incurs substantial per-byte overhead in the CPU and memory system that is not masked by advancing processor technology.

One way to reduce network storage access over-

head is to offload some or all of the transport protocol processing to the NIC. Many network adapters can compute Internet checksums as data moves to and from host memory; this approach is relatively simple and delivers a substantial benefit. An increasing number of adapters can offload all TCP or UDP protocol processing, but more substantial kernel revisions are needed to use them. Neither approach by itself avoids the fundamental overheads of data copying.

Several known techniques can remove copies from the transport data path. Previous work has explored copy avoidance for TCP/IP communication (Chase et al. [8] provide a summary). Brustoloni [5] introduced *emulated copy*, a scheme that avoids copying in network I/O while preserving copy semantics. IO-Lite [29] adds scatter/gather features to the I/O API and relies on support from the NIC to handle multiple client processes safely without copying. Another approach is to implement critical applications (e.g., Web servers) in the kernel [19]. Some of the advantages can be obtained more cleanly with combined data movement primitives, e.g., *sendfile*, which move data from storage directly to a network connection without a user space transfer; this is useful for file transfer in common server applications.

DAFS was introduced to combine the low overhead and flexibility of SAN products with the generality of NAS file services. The DAFS approach to removing these overheads is to use a direct-access transport to read and write application buffers directly. DAFS also enables implementation of the file system client at user level for improved efficiency, portability and application control. The next two sections discuss these aspects of DAFS in more detail. In Section 6, we discuss an alternative approach that reduces NFS overhead by eliminating data copying.

2.2 Direct-Access Transports

Direct-access transports are characterized by NIC support for remote direct memory access (RDMA), user-level networking with minimal kernel overhead, reliable messaging transport connections and per-connection buffering, and efficient asynchronous event notification. The Virtual Interface (VI) Architecture [12] defines a host interface and API for NICs supporting these features.

Direct-access transports enable *user-level networking* in which the user-mode process interacts directly with the NIC to send or receive messages

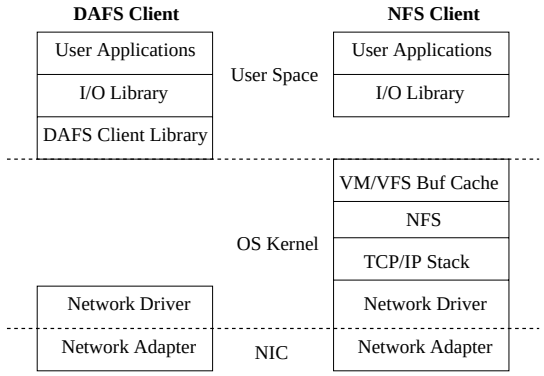

Figure 1: User-level vs. kernel-based client file system structure.

with minimal intervention from the operating system kernel. The NIC device exposes an array of connection descriptors to the system physical address space. At connection setup time, the kernel network driver maps a free connection descriptor into the user process virtual address space, giving the process direct and safe access to NIC control registers and buffer queues in the descriptor. This enables RDMA, which allows the network adapter to reduce copy overhead by accessing application buffers directly. The combination of user-level network access and copy avoidance has a lengthy heritage in research systems spanning two decades [2, 4, 6, 33, 38].

The experiments in Section 7 quantify the improvement in access overhead that DAFS gains from RDMA and transport offload on direct-access NICs.

2.3 User-Level File Systems

In addition to overhead reduction, the DAFS protocol leverages user-level networking to enable the network file system structure depicted in the left-hand side of Figure 1. In contrast to traditional kernel-based network file system implementations, as shown in the right side of Figure 1, DAFS file clients may run in user mode as libraries linked directly with applications.

While DAFS also supports kernel-based clients, our work focuses primarily on the properties of the user-level file system structure. A user-level client yields additional modest reductions in overhead by removing system call costs. Perhaps more importantly, it can run on any operating system, with no special kernel support needed other than the NIC driver itself. The client may evolve independently of the operating system, and multiple

client implementations may run on the same system. Most importantly, this structure offers an opportunity to improve integration of file system functions with I/O-intensive applications. In particular, it enables fully asynchronous pipelined file system access, even on systems with inadequate kernel support for asynchronous I/O, and it offers full application control over caching, data movement and prefetching.

It has long been recognized that the kernel policies for file system caching and prefetching are poorly matched to the needs of some important applications [34]. Migrating these OS functions into libraries to allow improved application control and specialization is similar in spirit to the *library operating systems* of Exokernel [21], *protocol service decomposition* for high-speed networking [24], and related approaches. User-level file systems were conceived for the SHRIMP project [4] and the Network-Attached Secure Disks (NASD) project [18]. NFS and other network file system protocols could support user-level clients over an RPC layer incorporating the relevant features of DAFS [7], and we believe that our results and conclusions would apply to such a system.

Earlier work arguing against user-level file systems [39] assumed some form of kernel mediation in the critical I/O path and did not take into account the primary sources of overhead outlined in Section 2.1. However, the user-level structure considered in this paper does have potential disadvantages. It depends on direct-access network hardware, which is not yet widely deployed. Although an application can control caching and prefetching, it does not benefit from the common policies for shared caching and prefetching in the kernel. Thus, in its simplest form, this structure places more burden on the application to manage data movement, and it may be necessary to extend applications to use a new file system API. Section 5 shows how this power and complexity can be encapsulated in prepackaged I/O *adaptation libraries* (depicted in Figure 1) implementing APIs and policies appropriate for a particular class of applications. If the adaptation API has the same syntax and semantics as a pre-existing API, then it is unnecessary to modify the applications themselves (or the operating system).

3 DAFS Architecture and Standards

The DAFS specification grew out of the DAFS Collaborative, an industry/academic consortium led by Network Appliance and Intel, and it is presently undergoing standardization through the Storage Networking Industry Association (SNIA).

The draft standard defines the *DAFS protocol* [14] as a set of request and response formats and their semantics, and a recommended procedural *DAFS API* [13] to access the DAFS service from a client program. Because library-level components may be replaced, client programs may access a DAFS service through any convenient I/O interface. The DAFS API is specified as a recommended interface to promote portability of DAFS client programs. The DAFS API is richer and more complex than common file system APIs including the standard Unix system call interface.

The next section gives an overview of the DAFS architecture and standards, with an emphasis on the transport-related aspects: Sections 3.2 and 3.3 focus on DAFS support for RDMA and asynchronous file I/O respectively.

3.1 DAFS Protocol Summary

The DAFS protocol derives from NFS Version 4 [32] (NFSv4) but diverges from it in several significant ways. DAFS assumes a reliable network transport and offers server-directed command flow-control in a manner similar to block storage protocols such as iSCSI. In contrast to NFSv4, every DAFS operation is a separate request, but DAFS supports request chaining to allow pipelining of dependent requests (e.g., a name *lookup* or *open* followed by file *read*). DAFS protocol headers are organized to preserve alignment of fixed-size fields. DAFS also defines features for reliable session recovery and enhanced locking primitives. To enable the application (or an adaptation layer) to support file caching, DAFS adopts the NFSv4 mechanism for consistent caching based on *open delegations* [1, 14, 32].

The DAFS specification is independent of the underlying transport, but its features depend on direct-access NICs. In addition, some transport-level features (e.g., message flow-control) are defined within the DAFS protocol itself, although they could be viewed as a separate layer below the file service protocol.

3.2 Direct-Access Data Transfer

To benefit from RDMA, DAFS supports *direct* variants of key data transfer operations (*read, write, readdir, getattr, setattr*). Direct operations transfer

directly to or from client-provided memory regions using RDMA *read* or *write* operations as described in Section 2.2.

The client must register each memory region with the local kernel before requesting direct I/O on the region. The DAFS API defines primitives to *register* and *unregister* memory regions for direct I/O; the *register* primitive returns a region descriptor to designate the region for direct I/O operations. In current implementations, registration issues a system call to pin buffer regions in physical memory, then loads page translations for the region into a lookup table on the NIC so that it may interpret incoming RDMA directives. To control buffer pinning by a process for direct I/O, the operating system should impose a resource limit similar to that applied in the case of the 4.4BSD *mlock* API [26]. Buffer registration may be encapsulated in an adaptation library.

RDMA operations for direct I/O in the DAFS protocol are always initiated by the server rather than a client. For example, to request a DAFS direct write, the client's write request to the server includes a region token for the buffer containing the data. The server then issues an RDMA *read* to fetch the data from the client, and responds to the DAFS write request after the RDMA completes. This allows the server to manage its buffers and control the order and rate of data transfer [27].

3.3 Asynchronous I/O and Prefetching

The DAFS API supports a fully asynchronous interface, enabling clients to pipeline I/O operations and overlap them with application processing. A flexible event notification mechanism delivers asynchronous I/O completions: the client may create an arbitrary number of *completion groups*, specify an arbitrary completion group for each DAFS operation and poll or wait for events on any completion group.

The asynchronous I/O primitives enable event-driven application architectures as an alternative to multithreading. Event-driven application structures are often more efficient and more portable than those based on threads. Asynchronous I/O APIs allow better application control over concurrency, often with lower overhead than synchronous I/O using threads.

Many NFS implementations support a limited form of asynchrony beneath synchronous kernel I/O APIs. Typically, multiple processes (called *I/O daemons* or *nfsiods*) issue blocking requests for sequential block read-ahead or write-behind. Unfortunately, frequent *nfsiod* context switching adds overhead [2]. The kernel policies only prefetch after a run of sequential reads and may prefetch erroneously if future reads are not sequential.

4 DAFS Reference Implementation

We have built prototypes of a user-level DAFS client and a kernel DAFS server implementation for FreeBSD. Both sides of the reference implementation use protocol stubs in a DAFS SDK provided by Network Appliance. The reference implementation currently uses a 1.25 Gb/s Giganet cLAN VI interconnect.

4.1 User-level Client

The user-level DAFS client is based on a three-module design, separating transport functions, flow-control and protocol handling. It implements an asynchronous event-driven control core for the DAFS request/response channel protocol. The subset of the DAFS API supported includes direct and asynchronous variants of basic file access and data transfer operations.

The client design allows full asynchrony for single-threaded applications. All requests to the library are non-blocking, unless the caller explicitly requests to wait for a pending completion. The client polls for event completion in the context of application threads, in explicit polling requests and in a standard preamble/epilogue executed on every entry and exit to the library. At these points, it checks for received responses and may also initiate pending sends if permitted by the request flow-control window. Each thread entry into the library advances the work of the client. One drawback of this structure is that pending completions build up on the client receive queues if the application does not enter the library. However, deferring response processing in this case does not interfere with the activity of the client, since it is not collecting its completions or initiating new I/O. A more general approach was recently proposed for asynchronous application-level networking in Exokernel [16].

4.2 Kernel Server

The kernel-based DAFS server [25] is a kernel-loadable module for FreeBSD 4.3-RELEASE that implements the complete DAFS specification. Using the VFS/Vnode interface, the server may export

any local file system through DAFS. The kernel-based server also has an event-driven design and takes advantage of efficient hardware support for event notification and delivery in asynchronous network I/O. The server is multithreaded in order to deal with blocking conditions in disk I/O and buffer cache locking.

5 Adaptation Libraries

Adaptation libraries are user-level I/O libraries that implement high-level abstractions, caching and prefetching, and insulate applications from the complexity of handling DAFS I/O. Adaptation libraries interpose between the application and the file system interface (e.g., the DAFS API). By introducing versions of the library for each file system API, applications written for the library I/O API can run over user-level or kernel-based file systems, as depicted in Figure 2.

DAFS-based adaptation libraries offer an opportunity to specialize file system functions for classes of applications. For example, file caching at the application level offers three potential benefits. First, the application can access a user-level cache with lower overhead than a kernel-based cache accessed through the system call interface. Second, the client can efficiently use application-specific fetch and replacement policies. Third, in cases where caching is an essential function of the adaptation library, a user-level file system avoids the problem of *double caching*, in which data is cached redundantly in the kernel-level and user-level caches.

One problem with user-level caching is that the kernel virtual memory system may evict cached pages if the client cache consumes more memory than the kernel allocates to it. For this reason, each of these adaptation libraries either pre-registers its cache (as described in Section 3.2) or configures it to a "safe" size. A second problem is that user-level caches are not easily shared across multiple uncooperating applications, but the need for this sharing is less common in high-performance domains.

To illustrate the role of adaptation libraries, we consider two examples that we enhanced for use with DAFS: TPIE and Berkeley DB.

5.1 TPIE

TPIE [37] (Transparent Parallel I/O Environment) is a toolkit for *external memory* (EM) algorithms. EM algorithms are structured to handle

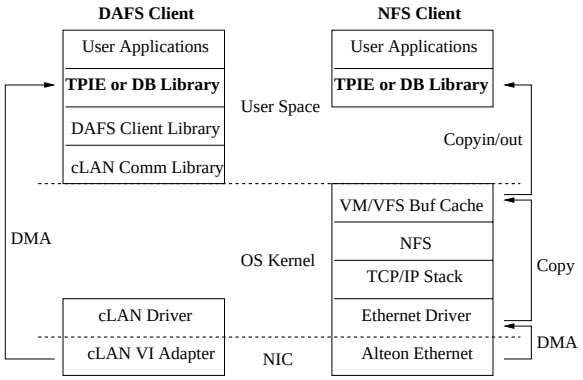

Figure 2: Adaptation Libraries can benefit from user-level clients without modifying applications.

massive data problems efficiently by minimizing the number of I/O operations. They enhance locality of data accesses by performing I/O in large blocks and maximizing the useful computation on each block while it is in memory. The TPIE toolkit supports a range of applications including Geographic Information System (GIS) analysis programs for massive terrain grids [3].

To support EM algorithms, TPIE implements a dataflow-like streaming paradigm on collections of fixed-size records. It provides abstract stream types with high-level interfaces to "push" streams of data records through application-defined record operators. A pluggable Block Transfer Engine (BTE) manages transfer buffering and provides an interface to the underlying storage system. We introduced a new BTE for the DAFS interface, allowing us to run TPIE applications over DAFS without modification. The BTE does read-ahead and write-behind on data streams using DAFS asynchronous primitives, and handles the details of block clustering and memory registration for direct I/O.

5.2 Berkeley DB

Berkeley DB (*db*) [28] is an open-source embedded database system that provides library support for concurrent storage and retrieval of key/value pairs. *Db* manages its own buffering and caching, independent of any caching at the underlying file system buffer cache. *Db* can be configured to use a specific page size (the unit of locking and I/O, usually 8KB) and buffer pool size. Running *db* over DAFS avoids double caching and bypasses the standard file system prefetching heuristics, which may degrade performance for common *db* access patterns. Section 7.4 shows the importance of these effects for *db*

performance.

6 Low-Overhead Kernel-Based NFS

The DAFS approach is one of several alternatives to improving access performance for network storage. In this section we consider a prominent competing structure as a basis for the empirical comparisons in Section 7. This approach enhances a kernel-based NFS client to reduce overhead for protocol processing and/or data movement. While this does not reduce system-call costs, there is no need to modify existing applications or even to relink them if the kernel API is preserved. However, it does require new kernel support, which is a barrier to fast and reliable deployment. Like DAFS, meaningful NFS enhancements of this sort also rely on new support in the NIC.

The most general form of copy avoidance for file services uses header splitting and page flipping, variants of which have been used with TCP/IP protocols for more than a decade (e.g., [5, 8, 11, 35]). To illustrate, we briefly describe FreeBSD enhancements to extend copy avoidance to *read* and *write* operations in NFS. Most NFS implementations send data directly from the kernel file cache without making a copy, so a client initiating a *write* and a server responding to a *read* can avoid copies. We focus on the case of a client receiving a *read* response containing a block of data to be placed in the file cache. The key challenge is to arrange for the NIC to deposit the data payload—the file block—page-aligned in one or more physical page frames. These pages may then be inserted into the file cache by reference, rather than by copying. It is then straightforward to deliver the data to a user process by remapping pages rather than by physical copy, but only if the user's buffers are page-grained and suitably aligned. This also assumes that the file system block size is an integral multiple of the page size.

To do this, the NIC first strips off any transport headers and the NFS header from each message and places the data in a separate page-aligned buffer (*header splitting*). Note that if the network MTU is smaller than the hardware page size, then the transfer of a page of data is spread across multiple packets, which can arrive at the receiver out-of-order and/or interspersed with packets from other flows. In order to pack the data contiguously into pages, the NIC must do significant protocol processing for NFS and its transport to decode the incoming packets. NFS complicates this processing with variable-length headers.

We modified the firmware for Alteon Tigon-II Gigabit Ethernet adapters to perform header splitting for NFS *read response* messages. This is sufficient to implement a zero-copy NFS client. Our modifications apply only when the transport is UDP/IP and the network is configured for Jumbo Frames, which allow NFS to exchange data in units of pages. To allow larger block sizes, we altered IP fragmentation code in the kernel to avoid splitting page buffers across fragments of large UDP packets. Together with other associated kernel support in the file cache and VM system, this allows zero-copy data exchange with NFS block-transfer sizes up to 32KB. Large NFS transfer sizes can reduce overhead for bulk data transfer by limiting the number of trips through the NFS protocol stack; this also reduces transport overheads on networks that allow large packets.

While this is not a general solution, it allows us to assess the performance potential of optimizing a kernel-based file system rather than adopting a direct-access user-level file system architecture like DAFS. It also approximates the performance achievable with a kernel-based DAFS client, or an NFS implementation over VI or some other RDMA-capable network interface. As a practical matter, the RDMA approach embraced in DAFS is a more promising alternative to low-overhead NFS. Note that page flipping NFS is much more difficult over TCP, because the NIC must buffer and reassemble the TCP stream to locate NFS headers appearing at arbitrary offsets in the stream. This is possible in NICs implementing a TCP offload engine, but impractical in conventional NICs such as the Tigon-II.

7 Experimental Results

This section presents performance results from a range of benchmarks over our DAFS reference implementation and two kernel-based NFS configurations. The goal of the analysis is to quantify the effects of the various architectural features we have discussed and understand how they interact with properties of the workload.

Our system configuration consists of Pentium III 800MHz clients and servers with the ServerWorks LE chipset, equipped with 256MB-1GB of SDRAM on a 133 MHz memory bus. Disks are 9GB 10000 RPM Seagate Cheetahs on a 64-bit/33 MHz PCI bus. All systems run patched versions of FreeBSD 4.3-RELEASE. DAFS uses VI over Giganet cLAN 1000 adapters. NFS uses UDP/IP over Gigabit Ethernet, with Alteon Tigon-II adapters.

Table 1: **Baseline Network Performance.**

	VI/cLAN	UDP/Tigon-II
Latency	30 μs	132 μs
Bandwidth	113 MB/s	120 MB/s

Table 1 shows the raw one-byte roundtrip latency and bandwidth characteristics of these networks. The Tigon-II has a higher latency partly due to the datapath crossing the kernel UDP/IP stack. The bandwidths are comparable, but not identical. In order to best compare the systems, we present performance results in Sections 7.1 and 7.2 normalized to the maximum bandwidth achievable on the particular technology.

NFS clients and servers exchange data in units of 4KB, 8KB, 16KB or 32KB (the NFS block I/O transfer size is set at mount time), sent in fragmented UDP packets over 9000-byte MTU Jumbo Ethernet frames to minimize data transfer overheads. Checksum offloading on the Tigon-II is enabled, minimizing checksum overheads. Interrupt coalescing on the Tigon-II was set as high as possible without degrading the minimum one-way latency of about 66μs for one-byte messages. NFS client (*nfsiod*) and server (*nfsd*) concurrency was tuned for best performance in all cases.

For NFS experiments with copy avoidance (*NFS-nocopy*), we modified the Tigon-II firmware, IP fragmentation code, file cache code, VM system and Tigon-II driver for NFS/UDP header splitting and page remapping as described in Section 6. This configuration is the state of the art for low-overhead data transfer over NFS. Experiments with the standard NFS implementation (*NFS*) use the standard Tigon-II driver and vendor firmware.

7.1 Bandwidth and Overhead

We first explore the bandwidth and client CPU overhead for reads with and without read-ahead. These experiments use a 1GB server cache prewarmed with a 768MB dataset. For this experiment, we factor out client caching as the NFS client cache is too small to be effective and the DAFS client does not cache. Thus these experiments are designed to stress the network data transfer. These results are representative of workloads with sequential I/O on large disk arrays or asynchronous random-access loads on servers with sufficient disk arms to deliver data to the client at network speed. None of the architectural features discussed yield much benefit for workloads and servers that are disk-bound, as they have no effect on the remote file system or disk system performance, as shown in Section 7.4.

The application's request size for each file I/O request is denoted *block size* in the figures. The ideal block size depends on the nature of the application; large blocks are preferable for applications that do long sequential reads, such as streaming multimedia, and smaller blocks are useful for nonsequential applications, such as databases.

In the read-ahead (sequential) configurations, the DAFS client uses the asynchronous I/O API (Section 3.3), and NFS has kernel-based sequential read-ahead enabled. For the experiments without read-ahead (random access), we tuned NFS for best-case performance at each request size (block size). For request sizes up to 32K, NFS is configured for a matching NFS transfer size, with read-ahead disabled. This avoids unnecessary data transfer or false prefetching. For larger request sizes we used an NFS transfer size of 32K and implicit read-ahead up to the block size. One benefit of the user-level file system structure is that it allows clients to select transfer size and read-ahead policy on a per-application basis. All DAFS experiments use a transfer size equal to the request block size.

Random access reads with read-ahead disabled. The left graphs in Figure 3 and Figure 4 reports bandwidth and CPU utilization, respectively, for random block reads with read-ahead disabled. All configurations achieve progressively higher bandwidths with increasing block size, since the wire is idle between requests. The key observation here is that with small request sizes the DAFS configuration outperforms both NFS implementations by a factor of two to three. This is due to lower operation latency, stemming primarily from the lower network latency (see Table 1) but also from the lower per-I/O overhead. This lower overhead results from the transport protocol offload to the direct-access NIC, including reduced system-call and interrupt overhead possible with user-level networking.

With large request sizes, transfer time dominates. NFS peaks at less than one-half the link bandwidth (about 60 MB/s), limited by memory copying overhead that saturates the client CPU. DAFS achieves wire speed using RDMA with low client CPU utilization, since the CPU is not involved in the RDMA transfers. NFS-nocopy eliminates the copy overhead with page flipping; it also

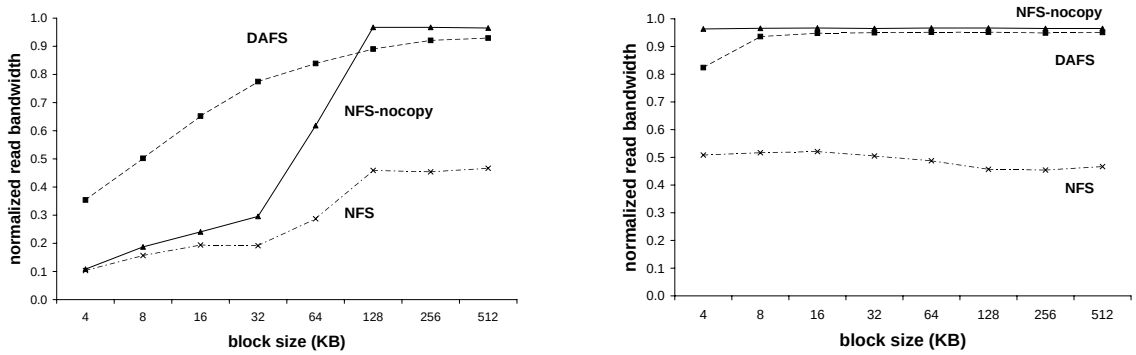

Figure 3: **Read bandwidth** without (left) and with (right) read-ahead.

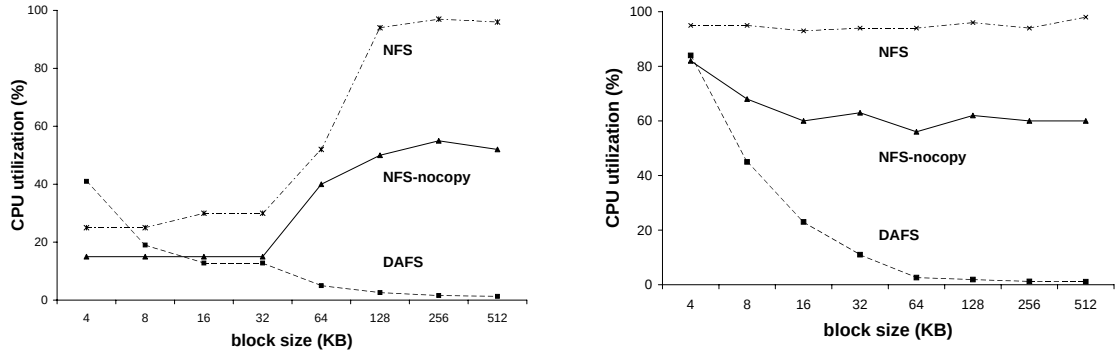

Figure 4: **Read client CPU utilization (%)** without (left) and with (right) read-ahead.

approaches wire speed for large block sizes, but the overhead for page flipping and kernel protocol code consumes 50% of the client CPU at peak bandwidth.

Sequential reads with read-ahead. The right graphs in Figure 3 and Figure 4 report bandwidths and CPU utilization for a similar experiment using sequential reads with read-ahead enabled. In this case, all configurations reach their peak bandwidth even with small block sizes. Since the bandwidths are roughly constant, the CPU utilization figures highlight the differences in the protocols. Again, when NFS peaks, the client CPU is saturated; the overhead is relatively insensitive to block size because it is dominated by copying overhead.

Both NFS-nocopy and DAFS avoid the byte copy overhead and exhibit lower CPU utilization than NFS. However, while CPU utilization drops off with increasing block size, this drop is significant for DAFS, but noticeably less so for NFS-nocopy. The NFS-nocopy overhead is dominated by page flipping and transport protocol costs, both of which are insensitive to block size changes beyond the page size and network MTU size. In contrast, the DAFS overhead is dominated by request initiation and response handling costs in the file system client code, since the NIC handles data transport using RDMA. Therefore, as the number of requests drops with the increasing block size, the client CPU utilization drops as well.

The following experiments illustrate the importance of these factors for application performance.

7.2 TPIE Merge

This experiment combines raw sequential I/O performance, including writes, with varying amounts of application processing. As in the previous experiment, we configure the system to stress client data-transfer overheads, which dominate when the server has adequate CPU and I/O bandwidth for the application. In this case the I/O load is spread across two servers using 600MB memory-based file systems. Performance is limited by the client CPU rather than the server CPU, network, or disk I/O.

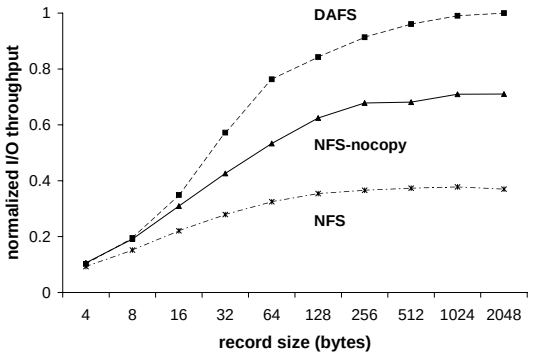

Figure 5: **TPIE Merge** throughput for $n = 8$.

The benchmark is a TPIE-based sequential record *Merge* program, which combines n sorted input files of x y-byte records, each with a fixed-size key (an integer), into one sorted output file. Performance is reported as total throughput:

$$\frac{2 \cdot n \cdot x \cdot y}{t} \quad \frac{\text{(bytes)}}{\text{(sec)}}$$

This experiment shows the effect of low-overhead network I/O on real application performance, since the merge processing competes with I/O overhead for client CPU cycles. Varying the merge order n and/or record size y allows us to control the amount of CPU work the application performs per block of data. CPU cost per record (key comparisons) increases logarithmically with n; CPU cost per byte decreases linearly with record size. This is because larger records amortize the comparison cost across a larger number of bytes, and there are fewer records per block.

We ran the *Merge* program over two variants of the TPIE library, as described in Section 5. One variant (TPIE/DAFS) is linked with the user-level DAFS client and accesses the servers using DAFS. In this variant, TPIE manages the streaming using asynchronous I/O, with zero-copy reads using RDMA and zero-copy writes using inline DAFS writes over the cLAN's scatter/gather messaging (cLAN does not support DAFS writes using RDMA). The second variant (TPIE/NFS) is configured to use the standard kernel file system interface to access the servers over NFS. For TPIE/NFS, we ran experiments using both standard NFS and NFS-nocopy configurations, as in the previous section. For the NFS configurations, we tuned read-ahead and write-behind concurrency (*nfsiods*) for the best performance in all cases. The TPIE I/O request size was fixed to 32KB.

Figure 5 shows normalized merge throughput results; these are averages over ten runs with a variance of less than 2% of the average. The merge order is n=8. The record size varies on the x-axis, showing the effect of changing the CPU demands of the application. The results show that the lower overhead of the DAFS client (noted in the previous section) leaves more CPU and memory cycles free for the application at a given I/O rate, resulting in higher merge throughputs. Note that the presence of application processing accentuates the gap relative to the raw-bandwidth tests in the previous subsection. It is easy to see that when the client CPU is saturated, the merge throughput is inversely proportional to the total processing time per block, i.e., the sum of the total per-block I/O overhead and application processing time.

For example, on the right side of the graph, where the application has the highest I/O demand (due to larger records) and hence the highest I/O overhead, DAFS outperforms NFS-nocopy by as much as 40%, as the NFS-nocopy client consumes up to 60% of its CPU in the kernel executing protocol code and managing I/O. Performance of the NFS configuration is further limited by memory copying through the system-call interface in writes.

An important point from Figure 5 is that the relative benefit of the low-overhead DAFS configuration is insignificant when the application is compute-bound. As application processing time per block diminishes (from left to right in Figure 5), reducing I/O overhead yields progressively higher returns because the I/O overhead is a progressively larger share of total CPU time.

7.3 PostMark

PostMark [22] 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 typical of mail and netnews servers used by Internet Service Providers. PostMark workloads are characterized by a mix of metadata-intensive operations. The benchmark begins by creating a pool of files with random sizes within a specified range. The number of files, as well as upper and lower bounds for file sizes, are configurable. After creating the files, a sequence of transactions is performed. These transactions are chosen randomly from a file creation or deletion operation paired with a file read or write. A file creation operation creates and writes random text to a file. File deletion removes a random file from the active set. File read reads a random file in its entirety

and file write appends a random amount of data to a randomly chosen file. In this section, we consider DAFS as a high-performance alternative to NFS for deployment in mail and netnews servers [9, 10].

We compare PostMark performance over DAFS to NFS-nocopy. We tune NFS-nocopy to exhibit best-case performance for the average file size used each time. For file sizes less than or equal to 32K, NFS uses a block size equal to the file size (to avoid read-ahead). For larger file sizes, NFS uses 32K blocks, and read-ahead is adjusted according to the file size. Read buffers are page-aligned to enable page remapping in delivering data to the user process. In all cases an FFS file system is exported using soft updates [17] to eliminate synchronous disk I/O for metadata updates.

Our NFS-nocopy implementation is based on NFS Version 3 [30]. It uses write-behind for file data: writes are delayed or asynchronous depending on the size of the data written. On close, the NFS client flushes dirty buffers to server memory and waits for flushing to complete but does not commit them to server disk. NFS Version 3 *open-to-close consistency* dictates that cached file blocks be re-validated with the server each time the file is opened[1].

With DAFS, the client does not cache and all I/O requests go to the server. Writes are not synchronously committed to disk, offering a data reliability similar to that provided by NFS. DAFS inlines data with the write RPC request (since the cLAN cannot support server-initiated RDMA read operations required for direct file writes) but uses direct I/O for file reads. In both cases, the client waits for the RPC response.

A key factor that determines PostMark performance is the cost of metadata operations (e.g., open, close, create, delete). Enabling soft updates on the server-exported filesystem decouples metadata updates from the server disk I/O system. This makes the client metadata operation cost sensitive to the client-server RPC as well as to the metadata update on the server filesystem. As the number of files per directory increases, the update time on the server dominates because FFS performs a linear time lookup. Other factors affecting performance are client caching and network I/O.

In order to better understand the PostMark performance, we measured the latency of the micro-operations involved in each PostMark transaction. As we saw in Table 1, the Tigon-II has significantly higher roundtrip latency than the cLAN. As a result, there is a similar difference in the null-RPC cost which makes the cLAN RPC three times faster than Tigon-II RPC (47μs vs. 154μs). In addition, all DAFS file operations translate into a single RPC to the server. With NFS, a file create or remove RPC is preceded by a second RPC to get the attributes of the directory that contains the file. Similarly, a file open requires an RPC to validate the file's cached blocks. The combination of more expensive and more frequent RPCs introduces a significant performance differential between the two systems.

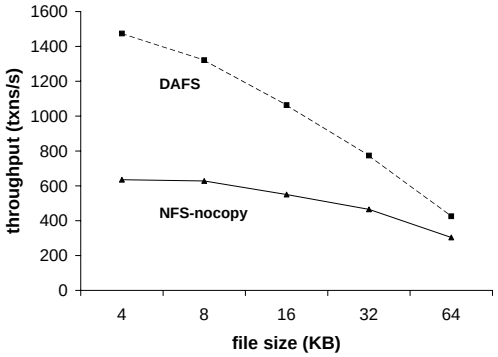

Figure 6: **PostMark.** Effect of I/O boundedness.

This experiment measures the effect of increasing I/O boundedness on performance. We increase the average file size from 4KB to 64KB, maintaining a small number of files (about 150) to minimize the server lookup time. Each run performs 30,000 transactions, each of which is a create or delete paired with a read operation. We report the average of ten runs, which have a variance under 10% of the average. The client and server have 256MB and 1GB of memory, respectively. At small file sizes, the increased cost of metadata operations for NFS-nocopy dominates its performance. As the I/O boundedness of the workload increases, DAFS performance becomes dominated by network I/O transfers and drops linearly. For large file sizes, reads under NFS-nocopy benefit from caching, but writes still have to go to the server to retain open-to-close semantics.

This experiment shows that DAFS outperforms NFS-nocopy by more than a factor of two for small files due largely to the lower latency of metadata operations on the cLAN network. Part of this difference could be alleviated by improvements in networking technology. However, the difference

[1] In the NFS implementation we used, writing to a file before closing it originally resulted in an invalidation of cached blocks in the next open of that file, as the client could not tell who was responsible for the last write. We modified our implementation to avoid this problem.

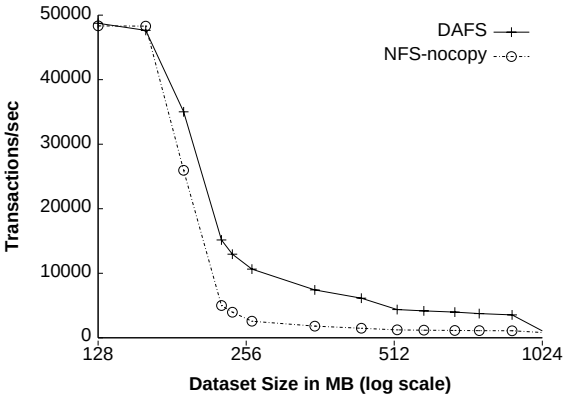

Figure 7: **Berkeley DB.** Effect of double caching and remote memory access performance.

in the number of RPCs between DAFS and NFS is fundamental to the protocols. For larger files, the NFS-nocopy benefit from client caching is limited by its consistency model that requires waiting on outstanding asynchronous writes on file close.

7.4 Berkeley DB

In this experiment, we use a synthetic workload composed of read-only transactions, each accessing one small record uniformly at random from a B-tree to compare *db* performance over DAFS to NFS-nocopy. The workload is single-threaded and read-only, so there is no logging or locking. In all experiments, after warming the *db* cache we performed a sequence of read transactions long enough to ensure that each record in the database is touched twice on average. The results report throughput in transactions per second. The unit of I/O is a 16KB block.

We vary the size of the *db* working set in order to change the bottleneck from local memory, to remote memory, to remote disk I/O. We compare a DAFS *db* client to an NFS-nocopy *db* client both running on a machine with 256MB of physical memory. In both cases the server runs on a machine with 1GB of memory. Since we did not expect read-ahead to help in the random access pattern considered here, we disable read-ahead for NFS-nocopy and use a transfer size of 16K. The *db* user-level cache size is set to the amount of physical memory expected to be available for allocation by the user process. The DAFS client uses about 36MB for communication buffers and statically-sized structures leaving about 190MB to the *db* cache. To facilitiate comparison between the systems, we configure the cache identically for NFS-nocopy.

Figure 7 reports throughput with warm *db* and server cache. In NFS-nocopy, reading through the file system cache creates competition for physical memory between the user-level and file system caches (Section 5). For database sizes up to the size of the *db* cache, the user-level cache is able to progressively use more physical memory during the warming period, as network I/O diminishes. Performance is determined by local memory access as *db* eventually satisfies requests entirely from the local cache.

Once the database size exceeds the client cache, performance degrades as both systems start accessing remote memory. NFS-nocopy performance degrades more sharply due to two effects. First, the double caching effect creating competition for physical memory between the user-level and file system caches is now persistent due to increased network I/O demands. As a result, the filesystem cache grows and user-level cache memory is paged out to disk causing future page faults. Second, since the *db* API issues unaligned page reads from the file system, NFS-nocopy cannot use page remapping to deliver data to the user process. The DAFS client avoids these effects by maintaining a single client cache and doing direct block reads into *db* buffers. For database sizes larger than 1GB that cannot fit in the server cache, both systems are disk I/O bound on the server.

8 Conclusions

This paper explores the key architectural features of the Direct Access File System (DAFS), a new architecture and protocol for network-attached storage over direct-access transport networks. DAFS or other approaches that exploit such networks and user-level file systems have the potential to close the performance gap between full-featured network file services and network storage based on block access models.

The contribution of our work is to characterize the issues that determine the effects of DAFS on application performance, and to quantify these effects using experimental results from a public DAFS reference implementation for an open-source Unix system (FreeBSD). For comparison, we report results from an experimental zero-copy NFS implementation. This allows us to evaluate the sources of the performance improvements from DAFS (e.g., copy overhead vs. protocol overhead) and the alternatives for achieving those benefits without DAFS.

DAFS offers significant overhead reductions for high-speed data transfer. These improvements result primarily from direct access and RDMA, the cornerstones of the DAFS design, and secondarily from the effect of transport offload to the network adapter. These benefits can yield significant application improvements. In particular, DAFS delivers the strongest benefit for balanced workloads in which application processing saturates the CPU when I/O occurs at network speed. In such a scenario, DAFS improves application performance by up to 40% over NFS-nocopy for TPIE *Merge*. However, many workload factors can undermine these benefits. Direct-access transfer yields little benefit with servers that are disk-limited, or with workloads that are heavily compute-bound.

Our results also show that I/O adaptation libraries can obtain benefits from the DAFS user-level client architecture, without the need to port applications to a new (DAFS) API. Most importantly, adaptation libraries can leverage the additional control over concurrency (asynchrony), data movement, buffering, prefetching and caching in application-specific ways, without burdening applications. This creates an opportunity to address longstanding problems related to the integration of the application and file system for high-performance applications.

9 Acknowledgments

This research was supported by the National Science Foundation (through grants CCR-00-82912, EIA-9972879, and EIA-9870728), Network Appliance, and IBM. We thank our shepherd Greg Ganger and the anonymous reviewers for their valuable comments.

10 Software Availability

The DAFS and NFS-nocopy implementations used in this paper are available from `http://www.eecs.harvard.edu/vino/fs-perf/dafs` and `http://www.cs.duke.edu/ari/dafs`.

References

[1] S. Addetia. User-level Client-side Caching for DAFS. Technical report, Harvard University TR-14-01, March 2002.

[2] D. Anderson, J. S. Chase, S. Gadde, A. Gallatin, and K. Yocum. Cheating the I/O Bottleneck: Network Storage with Trapeze/Myrinet. In *Proc. of Usenix Technical Conference, New Orleans, LA*, June 1998.

[3] L. Arge, J. S. Chase, L. Toma, J. S. Vitter, R. Wickremesinghe, P. N. Halpin, and D. Urban. Flow computation on massive grids. In *Proc. of ACM-GIS, ACM Symposium on Advances in Geographic Information Systems*, November 2001.

[4] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, and E. Felten. A Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proc. of 21st Annual Symposium on Computer Architecture*, pages 142–153, April 1994.

[5] J. Brustoloni. Interoperation of Copy Avoidance in Network and File I/O. In *Proc. of 18th IEEE Conference on Computer Communications (INFOCOM'99). New York, NY*, March 1999.

[6] G. Buzzard, D. Jacobson, S. Marovich, and J. Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *Proc. of Second Symposium on Operating Systems Design and Implementation*, October 1996.

[7] B. Callaghan. NFS over RDMA. Work-in-Progress Presentation, *USENIX File Access and Storage Symposium, Monterey, CA*, January 2002.

[8] J. S. Chase, A. J. Gallatin, and K. G. Yocum. End system optimizations for high-speed TCP. *IEEE Communications, Special Issue on TCP Performance in Future Networking Environments*, 39(4):68–74, April 2001.

[9] N. Christenson, D. Beckermeyer, and T. Baker. A Scalable News Architecture on a Single Pool. *;login*, 22(3):41–45, December 1997.

[10] N. Christenson, T. Bosserman, and D. Beckermeyer. A Highly Scalable Electronic Mail Service Using Open Systems. In *Proc. of USENIX Symposium on Internet Technologies and Systems*, December 1997.

[11] H. J. Chu. Zero-Copy TCP in Solaris. In *Proc. of USENIX Technical Conference, San Diego, CA*, January 1996.

[12] Compaq, Intel, Microsoft. *Virtual Interface Architecture Specification, Version 1.0*, December 1997.

[13] DAFS Collaborative. *DAFS API, Version 1.0*, November 2001.

[14] DAFS Collaborative. *Direct Access File System Protocol, Version 1.0*, September 2001. `http://www.dafscollaborative.org`.

[15] C. Dalton, G. Watson, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner: A Network-Independent Card Provides Architectural Support for High-Performance Protocols. *IEEE Network*, pages 36–43, July 1993.

[16] G. Ganger, D. Engler, M. F. Kaashoek, H. Briceno, R. Hunt, and T. Pinckney. Fast and Flexible

Application-Level Networking on Exokernel Systems. *ACM Transactions on Computer Systems*, 20(1):49–83, February 2002.

[17] G. Ganger and Y. Patt. Metadata Update Performance in File Systems. In *Proc. of USENIX Symposium on Operating System Design and Implementation*, pages 49–60, November 1994.

[18] G. Gibson, D. Nagle, K. Amiri, J. Buttler, F. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proc. of 8th Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[19] P. Joubert, R. King, R. Neves, M. Russinovich, and J. Tracey. High-Performance Memory-Based Web Servers: Kernel and User-Space Performance. In *Proc. of USENIX Technical Conference, Boston, MA*, July 2001.

[20] C. Jurgens. Fibre Channel: A Connection to the Future. *IEEE Computer*, 28(8):88–90, August 1995.

[21] M. F. Kaashoek, D. Engler, G. Ganger, H. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application Performance and Flexibility in Exokernel Systems. In *Proc. of 16th Symposium on Operating System Principles*, October 1997.

[22] J. Katcher. PostMark: A New File System Benchmark. Technical report, Network Appliance TR-3022, October 1997.

[23] E. Lee and C. Thekkath. Petal: Distributed Virtual Disks. In *Proc. of 7th International Conference on Architectural Support for Programming Languages and Operating Systems (Cambridge, Mass., Oct). ACM Press, New York*, pages 84–92, October 1996.

[24] C. Maeda and B. Bershad. Protocol Service Decomposition for High-Performance Networking. In *Proc. of Symposium on Operating Systems Principles*, pages 244–255, 1993.

[25] K. Magoutis. Design and Implementation of a Direct Access File System (DAFS) Kernel Server for FreeBSD. In *Proc. of USENIX BSDCon Conference, San Francisco, CA*, February 2002.

[26] M. K. McKusick, K. Bostic, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996.

[27] D. Nagle, G. Ganger, J. Butler, G. Goodson, and C. Sabol. Network Support for Network-Attached Storage. In *Proc. of Hot Interconnects, Stanford, CA*, August 1999.

[28] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proc. of USENIX Technical Conference, FREENIX Track, Monterey, CA*, June 1999.

[29] V. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching Scheme. In *Proc. of Third USENIX Symposium on Operating System Design and Implementation, New Orleans, LA*, February 1999.

[30] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3 Design and Implementation. In *Proc. of USENIX Technical Conference, Boston, MA*, June 1994.

[31] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proc. of Symposium on Operating System Design and Implementation, Seattle, WA*, October 1996.

[32] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS Version 4 Protocol. RFC 3010, December 2000.

[33] A. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *Communications of the ACM, 25(4)*, pages 246–260, April 1982.

[34] M. Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981.

[35] M. Thadani and Y. Khalidi. An Efficient Zero-copy I/O Framework for UNIX. Technical report, SMLI TR95-39, Sun Microsystems Lab, Inc., May 1995.

[36] C. Thekkath, T. Mann, and E. Lee. Frangipani: A Scalable Distributed File System. In *Proc. of 16th ACM Symposium on Operating Systems Principles*, pages 224–237, October 1997.

[37] D. E. Vengroff and J. S. Vitter. I/O-efficient computation: The TPIE approach. In *Proc. of the Goddard Conference on Mass Storage Systems and Technologies*, NASA Conference Publication 3340, Volume II, pages 553–570, College Park, MD, September 1996.

[38] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[39] B. Welch. The File System Belongs in the Kernel. In *Proc. of Second USENIX Mach Symposium*, November 1991.