DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY
Office for Scholarly Communication

# Application Performance on the Direct Access File System

**Citation**
Fedorova, Alexandra, Margo Seltzer, Kostas Magoutis, and Salimah Addetia. 2003. Application Performance on the Direct Access File System. Harvard Computer Science Group Technical Report TR-01-03.

**Permanent link**
http://nrs.harvard.edu/urn-3:HUL.InstRepos:23017124

**Terms of Use**
This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

# Share Your Story

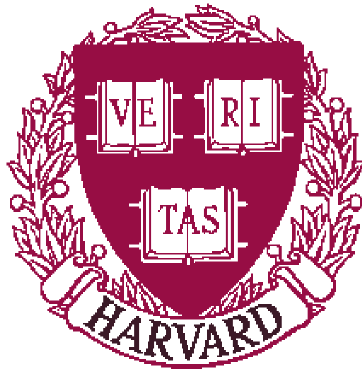
The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story .

Accessibility

# Application Performance on the Direct Access File System

Alexandra Fedorova
Margo Seltzer
Kostas Magoutis
and
Salimah Addetia

TR-01-03




Computer Science Group
Harvard University
Cambridge, Massachusetts

# Application Performance on the Direct Access File System

Alexandra Fedorova, Margo Seltzer, Kostas Magoutis and Salimah Addetia

*Harvard University*

## Abstract

The Direct Access File System (DAFS) is a distributed file system built on top of direct-access transports (DAT). Direct-access transports are characterized by using remote direct memory access (RDMA) for data transfer and user-level networking. The motivation behind the DAT-enabled distributed file system architecture is the reduction of the CPU overhead on the I/O data path.

In collaboration with Duke University we have created and made available an open-source implementation of DAFS for the FreeBSD platform. In this paper we describe a performance evaluation study of DAFS that was performed with this software. The goal of this work is to determine whether the architecture of DAFS brings any fundamental performance benefits to applications compared to traditional distributed file systems. In our study we compare DAFS to a version of NFS optimized to reduce the I/O overhead.

We conclude that DAFS can accomplish superior performance for latency-sensitive applications, outperforming NFS by up to a factor of 2. Bandwidth-sensitive applications do equally well on both systems, unless they are CPU-intensive, in which case they perform better on DAFS. We also found that RDMA is a less restrictive mechanism to achieve copy avoidance than that used by the optimized NFS.

## 1. Introduction

The Direct Access File System (DAFS) [1] is a new distributed file system designed to take advantage of *direct-access transports* (DAT) [13]. Direct-access transports allow for efficient and lightweight data transfer between the nodes in a distributed system through the use of *remote direct memory access* (RDMA) and *user-level networking*. By utilizing direct-access transports DAFS aims to increase the performance and efficiency of network-attached storage systems. The DAFS protocol is based on the Network File System (NFS) protocol version 4 [4], with built-in support for RDMA.

While DAFS uses new technology to build a distributed file system, conventional distributed file systems (such as NFS) have evolved as well. The research community has continuously addressed performance problems associated with conventional network storage systems, and, as a result, these systems have improved [6, 9, 10, 11, 12]. In the face of recent improvements in the performance of these systems, it is of interest to determine whether the DAFS architecture provides any fundamental performance benefits to applications compared to conventional network storage systems. This is the research question that we address in the current work.

In collaboration with Duke University we have created and made available an open-source implementation of the DAFS kernel server and the user-level client for the FreeBSD platform. In this paper we describe a performance evaluation study of DAFS that we performed with this software. As open source platforms become more common for supporting multi-tier applications, understanding the performance characteristics of the underlying file system alternatives will allow systems designers to construct high-performance platforms.

Our study makes the following contributions: We use microbenchmarks to understand the fundamental performance characteristics of DAFS. We conclude that the degree to which an application can benefit from DAFS largely depends on the characteristics of the application. We establish these characteristics and develop a simple framework that helps reason about an application's performance on DAFS without actually running the application. We evaluate the predictive power of our framework on a TPC-C database benchmark. We compare the performance of DAFS to an implementation of NFS modified to avoid data copies in the kernel (NFS-nocopy).

The rest of the paper is organized as follows: In section 2 we discuss related work. In sections 3 and 4 we give an overview of DAFS and of NFS-nocopy. In section 5 we present the microbenchmarks. In section 6 we discuss the application characteristics that determine the performance on DAFS, and derive the performance-predicting framework. In section 7 we evaluate our framework on the TPC-C benchmark. We conclude in section 8.

## 2. Related work

The current work is an extension of a performance evaluation study of DAFS performed at Harvard and Duke universities [7]. It expands this study by experimenting with a wider range of application workloads and by developing the framework for reasoning about an application's performance on DAFS. Some of the experiments demonstrated in this paper have also appeared in a paper presented at the 2002 USENIX conference [7]. We present these experiments here again for completeness. We include a reference to the original paper next to the figures.

Recent work has explored performance of database systems on direct-access transports. Zhou studied performance of Microsoft SQL Server that communicated with a storage system over a DAT network [17]. Scott performed a similar study with DB2 [18]. This work presents a general study of the file system built on top of direct-access transports.

## 3. DAFS

We first give an overview of remote direct memory access (RDMA) and user-level networking – the enabling technologies behind DAFS. We then proceed to describe their respective roles in the DAFS architecture.

### 3.1. RDMA

RDMA is a direct transfer of data between memory buffers on two hosts. It avoids the copying of data that is normally required when sending data over conventional mechanisms such as remote procedure call (RPC). RDMA also implies offloading the execution of the transport protocol code to the network device. RDMA, therefore, *decreases host CPU overhead involved in I/O*.

To transfer data with RDMA, a client sends to the server an RPC request, telling it what data it needs and the memory address where the data should be placed on the client. The server then tells its network interface controller (NIC) to initiate the RDMA. The NIC takes the data from the server memory and puts it on the wire (note no copying or host CPU involvement). When the data arrives at the other side of the wire, the NIC on the client machine deposits the data directly into the memory buffer that had been allocated by the client (again, there is no data copying or host CPU involvement). Figure 1 illustrates the difference between the DAFS client that uses RDMA and a conventional file system client that uses RPC for data transfer.

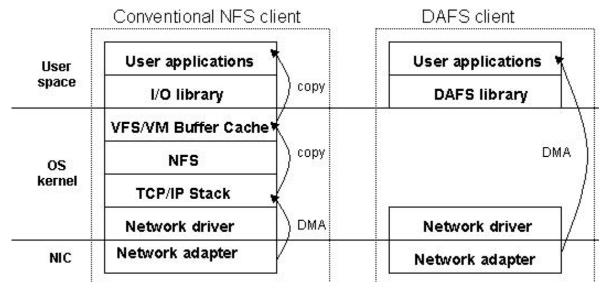



**Figure 1.** NFS client vs. DAFS client

Note that to make RDMA available to a file system service, some protocol provisions are necessary. In particular, a service that is using RDMA must be able to pass the address of a memory buffer where the RDMA data should be placed to the remote host. Unlike the NFS protocol, the DAFS protocol has such support for RDMA. It is this protocol support that makes DAFS DAT-ready, and makes the DAFS architecture fundamentally different from that of traditional distributed file systems.

### 3.2 User-level networking

Another characteristic of direct-access transports is *user-level networking*. User-level networking allows the user program to interact directly with the network interface controller

(NIC), bypassing the kernel. The NIC exposes an array of connection descriptors to the system's physical address space. At connection setup time, the kernel NIC driver maps a connection descriptor into the process virtual address space, giving the process a direct way to communicate with the NIC by simply writing and reading the descriptor memory.

To perform a data transfer using such a NIC, the user program must *register* with the NIC a memory buffer, which will serve as the destination for the incoming data. During registration the kernel pins the buffer in physical memory, and the NIC sets up a virtual-to-physical translation of the buffer's address in its internal page table. Once the buffer is registered, the RDMA transfer can proceed into the buffer without the kernel involvement.

User-level networking *reduces CPU overhead* for applications, by allowing an application to initiate I/O without system calls.

### 3.3. The DAFS architecture

DAFS has been envisioned and specified by a group of more than 85 companies led by Network Appliance. Network Appliance has released a commercial implementation of DAFS; several non-commercial implementations have been developed at universities [7]. The DAFS client and server implementations that we use have been developed by research groups at Harvard and Duke Universities [7, 23].

Although DAFS could be implemented in the kernel, just like traditional file systems, RDMA and user-level networking enable a user-level file system structure for DAFS. The DAFS client that we use is implemented at user-level, and the server is in the kernel. The client has an asynchronous event-driven design and implements the full DAFS client API [2], which is similar to the POSIX API. Although the client has been extended to support caching [8], the version used for the experiments in this paper does not include any caching or pre-fetching.

RDMA and user-level networking require special support on the NIC. The Virtual Interface (VI) architecture [5] defines a host interface and API for NICs supporting such features. Our DAFS implementation runs on top of a VI-capable NIC (Giganet cLAN 1000).

The DAFS server is currently a self-contained kernel module that does not require any core kernel changes. It will eventually be a part of the FreeBSD kernel distribution and is already available in source form as a FreeBSD 4.6 kernel module. It works with Myrinet GM 2.0 (alpha release) and VI-GM 1.0, which is also open-source software.

## 4. NFS-nocopy

NFS-nocopy[1] is an implementation of the standard NFS protocol modified to reduce the overhead on the I/O data path by avoiding data copies.

In the traditional kernel NFS client, there are two data copies that have to be made on the incoming I/O data path: the copy between the network stack and the kernel buffer cache, and then the copy between the buffer cache and the application buffers (see Figure 1). NFS-nocopy avoids these copies by using two techniques: *header splitting* and *page flipping*. Header splitting and page flipping are the traditional methods of copy avoidance, variants of which have been used with TCP/IP protocols in the past [6, 10, 11, 12]. The idea behind these techniques 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 can then be inserted into the kernel buffer cache by reference (page flipping). To do this, the NIC first strips off any transport headers and the NFS header from each message and places the data into a separate page-aligned buffer (header splitting).

We use an Alteon Tigon II Gigabit Ethernet NIC whose firmware has been modified to perform header splitting for the incoming NFS read response messages. Once the data payload has been deposited into page-aligned buffers in the kernel buffer cache, the data is delivered to the application buffers without copying, by simply re-mapping the physical pages into the application's address space (page flipping). The copy into the application buffers can only be avoided if the application has provided page-aligned buffers for the data.

[1] The NFS-nocopy system that we use in our experiments has been implemented at Duke University.

While this approach does not reduce system call overhead, it does not require changing or re-linking the applications. It does, however, require kernel modification and proper NIC support. We picked NFS-nocopy as a system to compare to DAFS because it is representative of a conventional network file system with overhead-reducing optimizations.

## 5. Microbenchmarks

In this section we describe how we used microbenchmarks to understand the performance characteristics of DAFS. We begin with simple experiments, and then gradually increase the complexity of the benchmarks in order to improve our understanding of the system. We compare the performance of the benchmarks on DAFS and NFS-nocopy. Since the compared systems are targeted at improving performance on the client side, we focus on benchmarking the clients.

Our system configuration consists of two Pentium III 800 MHz client and server machines. The client and the server are equipped with 256 MB and 1GB of RAM respectively, on a 133 MHz memory bus. All systems run patched versions of FreeBSD 4.3. DAFS uses VI over Giganet cLAN 1000 adapters. NFS uses UDP/IP over Gigabit Ethernet, with Alteon Tigon-II adapters. In some cases we also compare the systems to regular NFS. Experiments with the standard NFS implementation use the standard Tigon-II driver and vendor firmware. UDP checksum computation is offloaded to the NIC.

Table 1 shows the raw one-byte roundtrip latency of these networks. The Tigon-II has a higher latency. The bandwidths are comparable, but not identical. Disparity of the interface characteristics sometimes makes it difficult to compare the results of the experiments. Therefore, whenever appropriate we report the results normalized to the maximum bandwidth achievable by the underlying interface. In some cases we analytically derive the numbers that we would receive if identical networks were used. It would have been desirable to perform the measurements with identical networks. This was not possible, because both DAFS and NFS-nocopy needed to have special feature support on the NIC. DAFS needed a NIC that supported RDMA and user-level networking. NFS-nocopy needed a NIC capable of performing header splitting. We could not get a single NIC that would provide all of these features.

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

**Table 1.** Baseline network performance

### 5.1. Simple file access

The key motivation behind the architecture of DAFS is to reduce CPU overhead on the I/O data path. This is likely to decrease latency of I/O operations. Our first goal, therefore, was to test this by experimenting with a *latency-sensitive* workload. A latency-sensitive workload is a workload whose running time is dominated by the latency of individual I/O operations, rather than by the throughput achievable by the link. An example of a latency-sensitive workload is an application that reads small chunks at random offsets in a file. The significance of using small chunks is that the latency of issuing and responding to I/O, rather than the time that the data spends on the wire, dominates the execution latency of this application. The significance of random access is to make sure that the client file system does not perform read-ahead, which could make the workload sensitive to the link throughput.

Some applications, however, are able to hide the latency of individual I/Os by performing aggressive read-ahead and using large transfer size. Such workloads have a potential to saturate the underlying link and become limited by its bandwidth. These are *bandwidth-sensitive* workloads. To determine whether or not such workloads would benefit from running on top of DAFS, we also evaluate the performance of a bandwidth-sensitive workload.

Our first set of experiments involves reading a large file from the file server. The entire file fits into the server memory, and we read the file into the server memory prior to running the benchmark. Therefore, this experiment measures only the network transfer speed that can be achieved using the compared systems. We also include results for the non-optimized NFS client.

### 5.1.1. The latency-sensitive workload

To construct a latency-sensitive scenario we configure a benchmark that reads randomly chosen blocks from the file. We vary the transfer block size. When the block size becomes large, the workload effectively becomes bandwidth-sensitive. We configure the NFS client for maximum performance (the block size matches the application block size up to 32 KB, and the read-ahead is disabled).

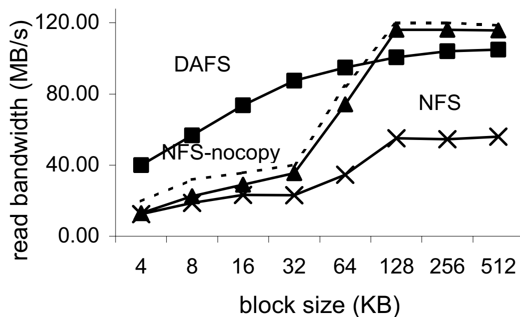



**Figure 2.** Read throughput. No read-ahead. [7]

Figure 2 shows the throughput in MB/s achieved by the systems. DAFS outperforms both NFS systems for small block sizes. This is due to the lower network latency (see Table 1) and lower protocol overhead. Note that since the application provides page-aligned read buffers, NFS-nocopy is able to avoid copies between the kernel and user space as well as the network to buffer cache copy (see discussion in section 4). The dashed curve above the NFS-nocopy curve was derived analytically to demonstrate the results that would be achieved if the NIC used with the NFS-nocopy system had the same latency as the NIC used with DAFS. Additionally, recent work has confirmed that DAFS outperforms the optimized NFS[2] in the latency-sensitive scenario when both systems are run on top of identical NICs [22].

When the block size becomes large, the application is able to fill the network pipe with data and saturate the link. At this point DAFS and NFS-nocopy become limited by the maximum throughput achievable by the underlying network interfaces (113 MB/s for cLAN and 120 MB/s for Tigon II, see Table 1). Regular NFS delivers lower performance because it saturates the local CPU due to copying overhead.

Figure 3 shows the CPU usage reported as the number of milliseconds used per MB of transferred data. The CPU usage for non-optimized NFS remains constantly high, saturating the client CPU. With DAFS, the CPU usage falls as the block size increases, because fewer network requests are issued. The interesting observation here is that for NFS-nocopy the CPU usage remains constant with increasing block size. This is due to the page-flipping cost, which is a function of the number of pages and is independent of the block size.

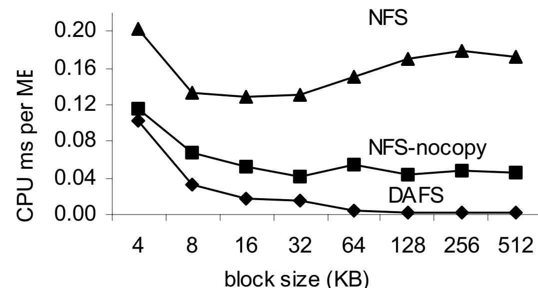



**Figure 3.** CPU ms per MB. No read-ahead.

### 5.1.2. The bandwidth-sensitive workload

The bandwidth-sensitive scenario involves issuing read-ahead for the blocks in the file. For NFS, we cause the read-ahead to happen in the kernel by requesting sequential file access. With DAFS, the read-ahead is done by the application, using the DAFS asynchronous API. We configure the NFS client for maximum performance (the read block size is 32 KB, maximum read-ahead is enabled). Figure 4 shows the throughput achieved by the systems. The throughput numbers are normalized to the maximum throughput achievable by the underlying interface.

Both DAFS and NFS-nocopy achieve the wire speed bandwidth. Standard NFS delivers lower throughput, because of the copying overhead that saturates the local CPU. Figure 5 shows the CPU usage in milliseconds per MB of transferred data. With DAFS, the CPU usage falls

[2] The system used in this work [22] employs different (and likely more efficient) copy-avoidance mechanism than NFS-nocopy.

for large block sizes; with NFS-nocopy it stays constant because of the page-flipping overhead.

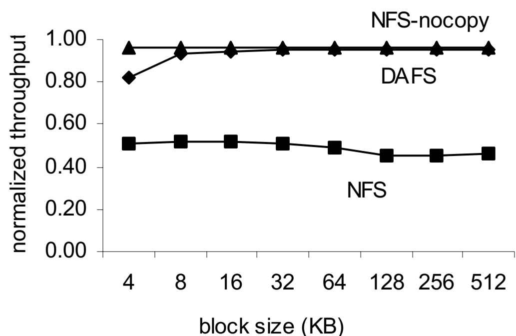



**Figure 4.** Normalized read throughput. Read-ahead. [7]

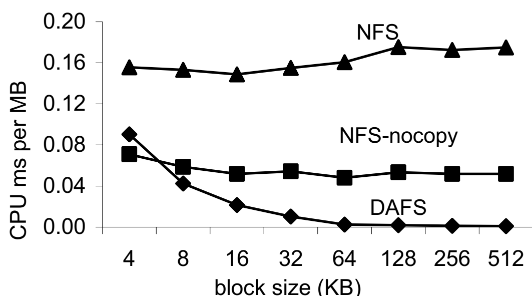



**Figure 5.** CPU ms per MB. Read-ahead.

### 5.1.3. Summary

In this section we showed that both DAFS and NFS-nocopy are able to perform at the wire speed in the bandwidth-sensitive scenario, although DAFS uses less CPU in doing so. Therefore, we conclude that bandwidth-sensitive applications can do equally well on both systems, unless they are CPU-bound. For the latency-sensitive scenario DAFS outperforms NFS-nocopy due to lower per-I/O overhead.

The applications used in this section were quite simple: they did not even touch the data that they read. In the next section we attempt to repeat the experiments of this section with more complex applications. We set up a latency-sensitive and a bandwidth-sensitive application, expecting to get similar results. We find that higher complexity of the applications affects the behavior of the experiments, leading us to get the results that we did not expect.

### 5.2. Berkeley DB

All of the experiments that we describe in subsequent sections use Berkeley DB [3]. Therefore we take a moment to describe it here.

Berkeley DB (*db*) is an open-source embedded database library that provides support for transactional concurrent storage and retrieval of key/value pairs. *Db* manages its own buffering and caching, independent of caching in the underlying file system buffer cache. *Db* can be configured to use a specific *page size,* a unit of caching, locking and I/O (usually 8 KB), and buffer pool size.

In our experiments, *db* acts as a user application that reads files from a remote server either through DAFS or NFS-nocopy. We chose *db* as a workload generator for the experiments because it can be easily configured to produce various application workloads.

### 5.3. Latency-sensitive scenario: a real application

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

We vary the size of the database in order to change the bottleneck from local memory, to remote memory and then to remote disk I/O. We compare DAFS and NFS-nocopy clients each running on a machine with 265 MB of RAM. In both cases the server is configured 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 16 KB. The *db* user-level cache size is set to the amount of physical memory expected to be available for allocation by the user process (190 MB). The DAFS client uses about 36 MB for communication buffers and statically sized structures leaving about 190 MB for the *db* cache. To facilitate comparison between the systems, we configure the cache identically for NFS-nocopy.

Figure 6 reports throughput in transactions per second. For database sizes up to the size of the *db* cache (190 MB), the performance is determined by local memory access as *db* satisfies the requests entirely from the local cache. Therefore, for this segment of the graph both systems achieve identical performance.

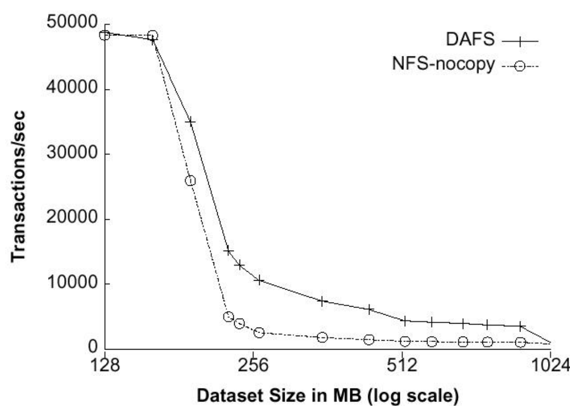



**Figure 6.** Berkeley DB. The effect of double caching and remote memory access. [7]

Once the database size exceeds the size of the client cache (the 3rd and subsequent data points), performance degrades as both systems start accessing remote memory. We expected that DAFS would perform slightly better than NFS-nocopy since this is what we saw in the latency-sensitive experiment of section 5.1.

Contrary to what we expected, the throughput achieved with DAFS is several orders of magnitude higher. The reason lies in the structure of the NFS kernel client. With NFS-nocopy, reading through the file system cache creates competition for physical memory between the user-level and file system caches, which happens because in 4.3 FreeBSD the VM cache and buffer cache are unified, meaning that the VM system and the buffer cache draw physical pages from the single memory pool. As a result, the file system cache grows and the user-level cache is paged out to disk causing future page faults. We call this the *double caching* effect[3]. The DAFS client avoids this effect by maintaining a single cache.

For database sizes larger than 1GB that cannot fit in the server cache, both systems are disk I/O bound on the server.

We were also interested in measuring the client CPU usage of each system. In Figure 7 we report the CPU seconds used per 1000 transactions. As expected, DAFS uses many fewer CPU cycles per transaction than NFS-nocopy.

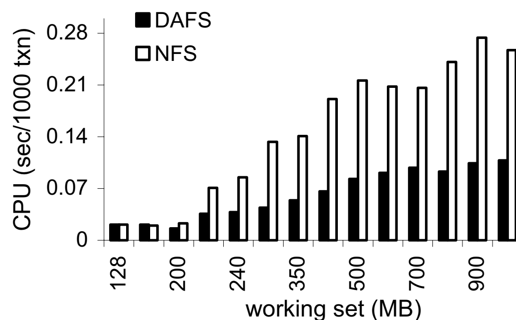



**Figure 7.** Berkeley DB. CPU seconds per 1000 transactions.

This experiment has unexpectedly demonstrated to us a benefit of the user-level file system architecture: applications running on top of DAFS can disable file system caching when it is not needed. While it is possible to disable caching in the NFS kernel client by making a small change to the kernel, this is not a general solution. Disabling caching for the NFS client would affect all processes that use that system, some of which may want caching enabled. While this experiment has demonstrated to us the benefits of the user-level file system architecture, it has also directed us to think about its limitations. Consider a scenario of a naïve application that does not perform its own caching but could benefit from the caching traditionally

[3] Double caching could be avoided if the *db* cache were pinned in physical memory using the mlock() system call. However, most operating systems limit how much memory can be pinned.

performed by the file system clients. Such an application can suffer from poor performance when run on top of a simple DAFS client that does not support caching. We demonstrate this scenario in the following section.

### 5.4. Flipping the coin: Kernel caching

In this section we simulate a naïve user application that does not perform its own caching, but can potentially benefit from the caching performed by the kernel file system client. The application performs a sequence of read-only transactions, each of which retrieves a small record from a B-tree selected uniformly at random. *Db*'s logging and locking are disabled. This time we set the *db* cache to a small size of 50 MB. The client machine has 256 MB of physical RAM. We use a database of size 191 MB. The server has 1 GB of RAM. We perform several runs, each with an increasing number of transactions.

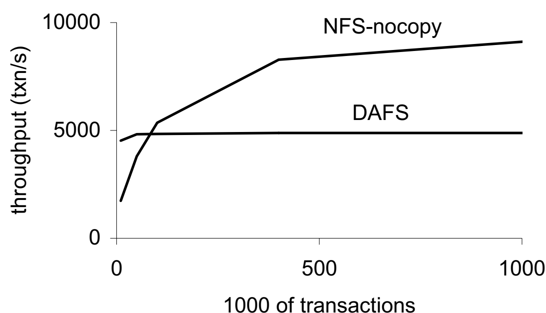



**Figure 8.** Berkeley DB. The effect of kernel caching.

Figure 8 shows the results. On the x-axis we report the number of transactions performed in the run, on the y-axis we report the throughput in transactions per second. We start with a warm server cache and, a cold *db* cache, and, in case of NFS, a cold client file cache.

When a small number of transactions is performed, both systems have to read data from the server memory. DAFS outperforms NFS due to lower per-I/O latency. Since the client machine has plenty of RAM and there is no double caching, we are able to repeat the result of the latency-sensitive benchmark in section 5.1 where DAFS achieved about twice the throughput of NFS-nocopy for 16 KB transfer size.

As the number of transactions in the run increases, the application starts accessing the records that have already been accessed. Since the *db* cache is small in comparison to the working set, and the access pattern is random, DAFS performance remains the same. However, the NFS-nocopy client benefits from the kernel caching on the client, and eventually its performance exceeds DAFS as all file requests of the application are satisfied from the local buffer cache.

The DAFS user-level client can certainly implement its own caching [8], however it is difficult to design a user-level cache that is shared among separate untrusting processes.

### 5.5. Bandwidth-sensitive scenario: a real application

Getting back to our original plan to repeat simple experiments of section 5.1 with more complex applications, we construct a bandwidth-sensitive application scenario. We modified Berkeley DB to perform read-ahead when the application access patterns are known. By issuing read-ahead the *db* application is able to keep a high number of outstanding I/O requests, creating a bandwidth-sensitive workload.

In this experiment an application uses *db* to compute a simple equality join. The result of the join produces a list of keys, whose corresponding data records are fetched from the database. We have modified *db* to perform asynchronous pre-fetching on this list of keys using DAFS asynchronous API or POSIX *aio* API in the case of NFS-nocopy. In Figure 9 we compare the throughput achieved with DAFS and NFS-nocopy without pre-fetching (NO_PREFETCH) and with pre-fetching (PREFETCH).

The results displayed in Figure 9 demonstrate that, just like in the experiment of section 5.2, there is a benefit from application-initiated pre-fetching when using either of the compared systems. In section 5.2, we also saw that when pre-fetching both systems achieve similar throughput and achieve wire speed performance. This is not the case here. Neither system saturates

the link because the application we run here uses more CPU than the simple bandwidth benchmark.

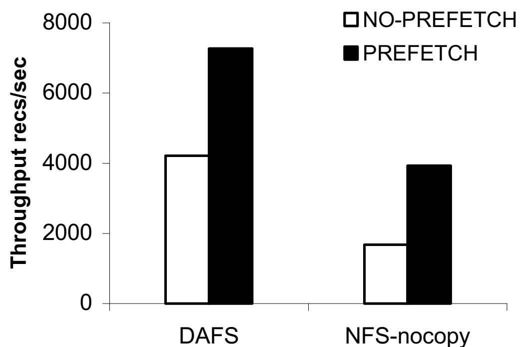



**Figure 9.** Bandwidth-sensitive application.

Another notable difference is that NFS-nocopy achieves lower throughput than DAFS. There are two reasons for this. The first is that NFS-nocopy uses more CPU cycles than DAFS (see section 5.1) and, given the additional CPU requirements of the application, CPU becomes the bottleneck, preventing efficient link utilization. The second reason, further contributing to the CPU bottleneck, is that the application running on top of NFS-nocopy does not enjoy the full benefits of copy avoidance in this experiment, since *db* reads data into unaligned buffers (see section 4), and it would require non-trivial amount of work to modify *db* to use page-aligned buffers. This demonstrates that RDMA is a more general copy avoidance method for applications than that employed by NFS-nocopy.

In this section we presented an experiment with a sophisticated application that was modified to do its own pre-fetching. It is also interesting to look at a scenario of a naïve application does not perform its own pre-fetching, but that could benefit from the pre-fetching usually provided by the kernel file system clients. Such an application could suffer from poor performance when running on top of a simple DAFS client that does not do read-ahead. We demonstrate such a scenario in the next section.

### 5.6. Flipping the coin: kernel pre-fetching

The following experiment demonstrates a scenario in which the NFS kernel client delivers higher performance than DAFS, in this case, due to pre-fetching initiated by the NFS client when the sequential file access pattern is detected. The DAFS client does no pre-fetching.

To trigger kernel pre-fetching, we configure the application to use the "queue" data structure available in the *db* library and perform a sequential traversal of the queue, which requires sequentially reading the file containing the data structure. The client host is configured with 256 MB of RAM, and the *db* cache is set to 50 KB. Since we perform a single traversal, touching each page in the file exactly once, there is no need for a large cache. We vary the size of the records populating the database, in order to vary the amount of computation performed by the application.

Figure 10 shows the throughput in MB/s achieved by the application depending on the record size. For all record sizes the NFS-nocopy delivers higher throughput due to kernel pre-fetching, but the effect of pre-fetching is small when the application is CPU-bound due to the overhead of processing of small records.

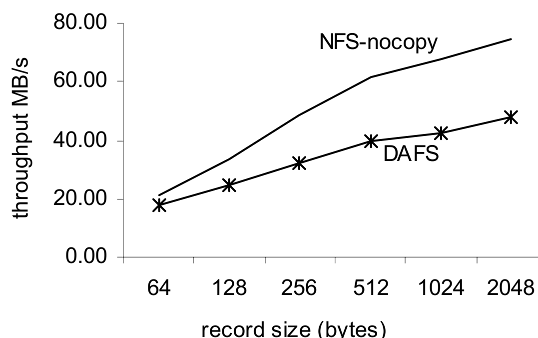



**Figure 10.** The effect of kernel pre-fetching.

While a DAFS client could be easily extended to perform pre-fetching like the kernel client, this result demonstrates the point that when migrating applications to DAFS, special care must be taken to find out exactly what assumptions the application makes about the underlying file system and whether or not these assumptions are satisfied by the DAFS client.

## 6. Reasoning About Application Performance

In the previous section we presented several experiments that helped us understand what factors affect the performance of applications

running on top of DAFS. In this section we summarize the results of the experiments and describe a framework that helps us reason about the performance of an application on DAFS.

Previous work has looked into building models for predicting the exact latency of application execution. Smith tackled the problem of predicting application performance on local file systems in his Ph.D. thesis [14]. He used a combination of vector-based and trace-based techniques developed by M. Seltzer and her colleagues [16]. Zhang used similar techniques to predict the performance of Java Virtual machines [15]. We believe that it would be difficult to directly apply these techniques here because, as we will see in the next section, performance of an application may depend on the degree of concurrency in the file system access exhibited by the application. Therefore, instead of building a model for predicting the exact latency, we simply relate the results of the microbenchmarks to the application characteristics in a way that helps us reason about how an application would perform on DAFS.

As we saw from the results presented in the previous section, the performance of an application on DAFS largely depends on the characteristics of the application. Therefore, in order to reason about an application's performance we need to categorize applications based on their characteristics. To facilitate such categorization in the infinite application space we point out three groups of application characteristics are key in determining the performance on DAFS:

1) I/O characteristics: Is the application latency-sensitive or bandwidth-sensitive?

In the experiments of section 5.2 we saw that a latency-sensitive application is likely to see a performance benefit from the low per-I/O overhead of DAFS. Bandwidth-sensitive applications, on the other hand, can achieve comparable performance on DAFS and NFS-nocopy, unless they are CPU-bound.

2) CPU characteristics. Is the application CPU bound?

DAFS uses fewer CPU cycles per I/O operation. This is an important consideration for CPU-bound applications, as we saw in section 5.5 with the Berkeley DB join application. If an application is subject to the CPU bottleneck on the client machine it may achieve better performance with DAFS, because DAFS uses fewer CPU cycles for I/O, leaving more CPU cycles for the application.

3) Application structure: Is the application structured in a way that allows it to reap the benefits offered by the compared file systems?

Application design aspects that may affect the performance include the use of page-aligned buffers, implementation of custom caching or lack thereof, and reliance on pre-fetching provided by the file system.

By describing an application's characteristics with respect to the three characteristic groups defined above, we can predict how the application will perform on DAFS compared to NFS-nocopy.

Before we proceed with evaluation of our framework, we take a moment to point out several limitations imposed on applications by the architecture of DAFS. Although these limitations do not directly affect performance, they may turn out to be important to consider when making decisions about a file system choice:

a) When migrating applications to DAFS, they must be modified or re-linked. There is no such need when migrating to an optimized NFS client.
b) Since DAFS has a user-level architecture[4] it is difficult to design a client cache for DAFS that can be shared by multiple untrusting processes. Many high performance systems, however, have a single-process architecture that does not depend on sharing a file system cache with other processes. For such systems the limitations of the user-level cache architecture do not present a serious problem.

---

[4] Although DAFS can also be implemented in the kernel, we consider the user-level architecture here, as explained in section 2.

c) Since the DAFS user-level library is embedded into the application address space, a buggy application can inadvertently overwrite the memory belonging to the file system client.
d) DAFS requires registering the application buffers used for I/O with the NIC (see section 3). Registration of the buffers means that the buffers are pinned in the physical memory. This can result in an unfair use of operating system resources by a DAFS application in a multi-process environment.

On a positive note, the DAFS user-level structure offers opportunities for the customization of the file client for the application-specific needs.

## 7. TPC-C

In this section we test the effectiveness of our framework in predicting performance of a complex application. The application we use is the TPC-C benchmark.

TPC-C is a standard database benchmark specified by the Transaction Processing Performance Council [19]. It is an online transaction-processing workload that involves a mixture of read-only and update intensive transactions that simulate the activities of order-processing warehouses. A conventional TPC-C set-up is a complex multi-tier system that consists of *transaction clients* that issue transaction requests by sending messages to a *message server*, that eventually forwards these requests to a *database server* (see Figure 11). In our configuration, the database server stores its database files on a network-attached storage system running either DAFS or NFS *file server*. Therefore, the database server uses either the DAFS or the NFS-nocopy file system client to access the files. We were interested in comparing the performance of the database server on these two file systems.

Sun Microsystems graciously shared with us their database-benchmarking tool, which implements the database-independent part of TPC-C [20]. We ported the Sun's implementation of TPC-C for Informix to run on top of Berkeley DB. We implemented the database server as a collection of one or more *server processes* that receive transaction requests from the message server and execute queries against Berkeley DB. Although we have ported the tool in its entirety, in the experiment described here, the transaction workload is not generated by transaction clients – it is generated locally on the machine running database servers. We do this in order to simplify our understanding of the experiment.

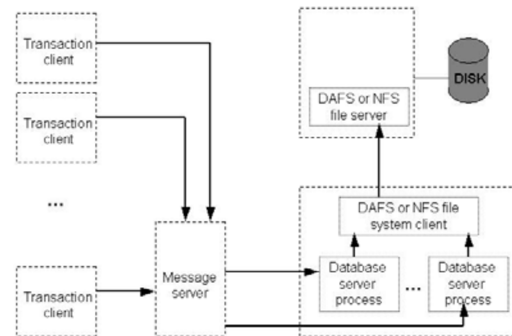



**Figure 11.** Our TPC-C configuration.

Implementing the TPC-C benchmark required careful tuning of the database schema and transactions in order to minimize database lock contention [21]. In the process of tuning we also discovered that we were limited by the inadequate disk I/O system that was at our disposal. We only had regular 10000-RPM SCSI disks available to us, while conventionally, the TPC-C benchmark is run with powerful RAID storage systems on the back-end that achieve hundreds of megabytes in disk throughput [19]. We simulate the availability of a more powerful disk I/O system by disabling synchronous writes to disk and by configuring the file server machine with 1 GB of physical RAM in order to fit the entire database in memory. The client machine hosting the database server has enough physical RAM to accommodate the *db* cache, configured to 100 MB (1/7 of the database size). We pinned the *db* cache in memory in order to avoid the undesirable double caching effect we described in section 5.3.

To apply our framework for predicting the behavior of TPC-C, we first need to establish the I/O, the CPU, and the application structure

characteristics of the benchmark that will be important in predicting its behavior.

The I/O characteristic of the TPC-C benchmark depends on how many concurrent database server processes are running. When only one process is running, the application is latency-sensitive so we expect it to have better performance on DAFS. When multiple servers processes are running simultaneously, they can issue a number of simultaneously outstanding I/O requests. This creates a potential for saturating the link and making the workload bandwidth-sensitive. As we showed in section 5.2 bandwidth-sensitive applications do equally well on both systems unless they are CPU-intensive. Each TPC-C transaction involves traversing several database indices. We, therefore, anticipated that the benchmark would be CPU-intensive and expected that it would do better on DAFS. We also expected that TPC-C over DAFS would scale better as the number of concurrent server processes increased, because of lower CPU requirements of the DAFS client.

The application structure of the benchmark is poorly suited for NFS-nocopy because Berkeley DB uses unaligned buffers for I/O transfer, preventing NFS-nocopy from completely avoiding data copies. It is, however, well suited for the simple DAFS client that we used: since Berkeley DB implements its own caching the benchmark's performance would not suffer from the lack of caching in the DAFS client. Since the access pattern is random, we did not expect pre-fetching to be helpful.

Figure 12 shows the combined throughput (in transactions per second) achieved by all the database server processes as the function of the number of processes.

As we expected, DAFS outperform NFS-nocopy in all cases. When two database server processes are running concurrently, the systems deliver better throughput compared to the case when a single server process is used, because of the improved utilization of the system resources. When more server processes are added, both systems experience negative scaling because of the adverse effect of database lock contention.

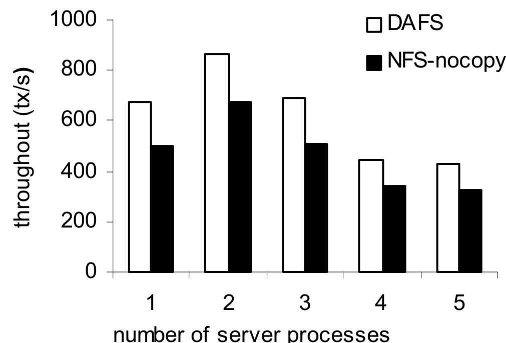



**Figure 12.** TPC-C: total throughput as a function of a number of concurrent server processes.

DAFS uses CPU more efficiently: both systems achieve the same CPU utilization of about 80%, but since DAFS offers higher throughput, it uses fewer CPU cycles per transaction.

We incorrectly anticipated that the benchmark would be CPU-intensive and would therefore scale better with DAFS than with NFS-nocopy. Contrary to what we expected, the benchmark does not create CPU pressure. Even though we eliminated the disk I/O bottleneck and heavy lock contention points, there was still some amount of database lock contention that prevented the benchmark from saturating the CPU. Such lock contention is expected in the face of concurrent database access.

## 8. Conclusions

In this work we conducted a performance evaluation study of DAFS using our open-source implementation for FreeBSD. We addressed the question whether the DAFS architecture provides any fundamental performance benefits for applications compared to conventional network storage systems. We concluded that the DAFS architecture does provide fundamental performance benefits, but whether or not and application can enjoy these benefits largely depends on the structure of the application. Using microbenchmarks we understood the application characteristics that determine its performance on DAFS. We summarized the important architectural limitations of DAFS and compared

DAFS to a competing system: NFS-nocopy. We concluded that DAFS delivers better performance for latency-sensitive applications and for bandwidth-intensive CPU-intensive applications.

We developed a framework for reasoning about an application's performance on DAFS. Even though our framework proved useful for reasoning about the performance of simple applications, we found that for complex scenarios we were not able to correctly describe the application characteristics, which is the pre-requisite for the effective use of our framework. This pre-requisite is difficult to satisfy when dealing with a complex distributed system.

## 10. Software availability

The DAFS and NFS-nocopy software evaluated in this paper is freely available at http://eecs.harvard.edu/vino/fs-perf/dafs.html and http://www.cs.duke.edu/ari/dafs.

## 11. References


[1] DAFS Collaborative. *Direct Access File System Protocol, Version 0.90*, June 2001, *http://www.dafscollaborative.org*.

[2] DAFS Collaborative. *DAFS API, Version 0.6,* June 2001, *http://www.dafscollaborative.org*.

[3] Olson, M., Bostic, K., Seltzer, M. "Berkeley DB", *In Proceedings of USENIX 1999 Annual Technical Conference*, June 1999.

[4] Shepler, S. et al. NFS Version 4 Protocol. RFC 3010, December 2000.

[5] Compaq, Intel, Microsoft, *Virtual Interface Specification, Version 1.0*, December 16, 1997.

[6] J. S. Chase, K. G. Yocum, and A. J. Gallatin. "End-System Optimizations for High-Speed TCP", *IEEE Communications Special Issue on TCP Performance in Future Networking Environments*, 39(4):68-74, April 2001.

[7] Magoutis, K., Addetia, S., Fedorova, A., Seltzer, M., Chase, J., Kisley, R., Gallatin, A., Wickremesinghe, R., Gabber, E. "Structure And Performance Of Direct Access File System", *In Proceedings of USENIX 2002 Annual Technical Conference*, Monterey, CA, June 2002.

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

[9] Anderson, D., Chase, J., Gadde, S., Gallatin, A., Yocum, K. "Cheating the I/O Bottleneck: Network Storage With Trapeze/Myrinet", *Proceedings of USENIX 1998 Annual Technical Conference,* June 1998.

[10] Brustoloni, J., "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.

[11] Chu, J. "Zero-copy TCP in Solaris", *In Proceedings of USENIX 1996 Annual Technical Conference*, San Diego, CA, January 1996.

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

[13] DAT Collaborative, *http://www.datcollaborative.org.*

[14] Smith, K. "Workload-Specific File System Benchmarks", Ph.D. Thesis*, Harvard University,* Cambridge, MA, January 2001.

[15] Zhang, X., Seltzer, M. "Hbench:Java: An Application-Specific Benchmarking Framework for Java Virtual Machines", *In Proceedings of the ACM Java Grande 2000 Conference,* pp. 62-70, San Francisco, CA, June 2000.

[16] Seltzer, M., Krinsky, D., Smith, K., Zhang, X. "The Case for Application-Specific Benchmarking", *In Proceedings of the 7th Workshop on Hop Topics in Operating Systems (HOTOS-VII),* pp. 102-107, Rico Rio, AZ, March 1999.

[17] Zhou, Y., Bilas, A., Jagannathan, S., Dubnicki, C., Philbin, J., Li. K. "Experiences with VI Communication for Database Storage", *In Proceedings of the 29th International Symposium on Computer Architecture* (ISCA), May 2002.

[18] Scott, H., "User-level I/O for Database Management Systems", Master's Thesis, *Queen's University, Kingston, Ontario, Canada,* March 2001.

[19] Transaction Processing Performance Council, "TPC Benchmark C", Standard Specification, Revision 5.0, Februrary 26, 2001, *http://www.tpc.org.*

[20] Sun Microsystems, "Dbbench – Database Benchmarking Tool – Version 2.4 Manual", 1994.

[21] Fedorova, A., Seltzer, M. Personal Communication on Optimizing Transactions and Data Structures in Berkeley DB, *Harvard University*, October 2002.

[22] Magoutis, K, Addetia, S., Fedorova, A., Seltzer, M., "Making the Most out of Direct Access Network-Attached Storage", *To appear In Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03),* San Francisco, CA, 2003.

[23] Magoutis, K., "Design and Implementation of a Direct Access File System (DAFS) Kernel Server for FreeBSD", I*n Proceedings of USENIX BSDCon 2002 Conference*, San Franscisco, CA, February 11-14, 2002.