The Optimistic Direct Access File System: Design and Network Interface Support

Kostas Magoutis

Abstract— **The emergence of commercially-available network interface controllers (NICs) with remote direct memory access (RDMA) capability and the prospect of their tighter integration with the host memory system motivate the design of distributed systems based on an RDMA paradigm. A recent example is the Direct Access File System (DAFS). DAFS clients communicate requests to servers using lightweight Remote Procedure Calls (RPC) based on low-overhead message passing. Data can be transmitted either in-line with the messages, or via server-initiated RDMA independent of the messages. DAFS clients do not initiate RDMA, despite the potentially lower latency of this mechanism for short transfers. With current NICs, servers that export their entire file cache would have to resort to excessive page wiring to guarantee success of client-initiated RDMA operations.**

In this paper we present the design of the Optimistic Direct Access File System, a distributed filesystem that enhances DAFS by enabling clients to directly access remote memory pages of cached files exported by servers. Using our proposed NIC support, exported pages are not permanently wired in physical memory except during RDMA operations referencing them. Client RDMA attempts for pages that are no longer memory-resident result in exceptions thrown by the RDMA target (server) and caught by the initiator (client), prompting a switch to an alternative access method. We present simulation results showing that optimistic client-initiated RDMA seldom fails when client working sets fit in server physical memory. Microbenchmarks with a prototype RDMA-capable NIC designed to support the Optimistic Direct Access File System show that client-initiated RDMA can achieve lower file access latency when compared to file access using RPC. The applicability of optimistic client-initiated RDMA extends to other domains, such as Distributed Shared Memory systems.

I. Introduction

System-area networks (SANs) offer low-latency, high-bandwidth data transfer over switched interconnects. Applications over SAN are often constrained by host overhead in copying data in kernel and application buffers, as well as overhead in accessing the network interface controller (NIC). Excessive copying is caused by insufficient integration between buffering subsystems [1], [2], [3], e.g., in network, filesystem and applications buffers. High overhead NIC access results from going through the kernel, as is the case when using sockets. Remote direct memory access (RDMA) and user-level networking are two mechanisms requiring NIC support that can address the above problems. RDMA can be used to transfer data directly between source and destination buffers over the network, avoiding any intermediate copies. User-level networking is used to reduce the overhead of NIC access by the application using memory-mapped access to the device and bypassing the kernel.

Research on system-area networks with user-level networking and RDMA capabilities [4], [5], [6], [7] has motivated the design of distributed systems based on an RDMA paradigm. Recent commercially-available SAN interconnects such as InfiniBand [8] and protocols such as VI [9] offer advanced RDMA capabilities for data transfer and atomic operations. Network interfaces for these interconnects are expected to become increasingly tightly integrated with the host memory system [10] (e.g., as is anticipated in the case of InfiniBand), further lowering the latency of remote memory access. A recent example of a network file access protocol for RDMA-capable SAN is the Direct Access File System (DAFS) [11]. DAFS client requests are communicated to the server using lightweight RPC. Reads, writes and some metadata operations (e.g., *getattr*, *readdir*) use server-initiated RDMA when doing long transfers. Short transfers do not use RDMA – they are instead inlined in the RPC request or response. An alternative way (not used by DAFS) to access data and metadata is using client-initiated RDMA, provided the client has direct remote memory references to data or metadata in the server file cache.

There are two main reasons why DAFS does not use client-initiated RDMA, despite the lower latency of this mechanism for short transfers: First, DAFS is not targeted for workloads sensitive to I/O latency, such as engineering and other workloads that are dominated by short transfers [12]. Second, current commercially-available RDMA-capable NICs lack some of the capabilities necessary to enable its use.

DAFS is optimized for file access workloads using long transfers. Most of the benefit using RDMA in these workloads comes out of the reduction of the per-byte cost of data transfer due to avoiding memory copies [13]. DAFS file servers are expected to be able to use and potentially export their entire available physical memory space as file cache. With clients allowed to directly access any part of that cache at any time using RDMA and given that current NICs are unable to wire/unwire memory pages on-demand (for the duration of an RDMA), the server would have to resort to excessive page wiring. Even with the NIC's ability to wire/unwire on-demand, client RDMA attempts for pages that had been previously exported but are no longer memory-resident at the server require a remote access exception notification mechanism. There is presently no integration between NICs and mainstream operating systems necessary to deal with virtual memory page locking. There

Kostas Magoutis is with the Division of Engineering and Applied Sciences, Harvard University. E-mail: magoutis@eecs.harvard.edu . This paper appears in Proceedings of the First Workshop on Novel Uses of System Area Networks (held in conjunction with HPCA-8), Cambridge, MA, February 2, 2002.

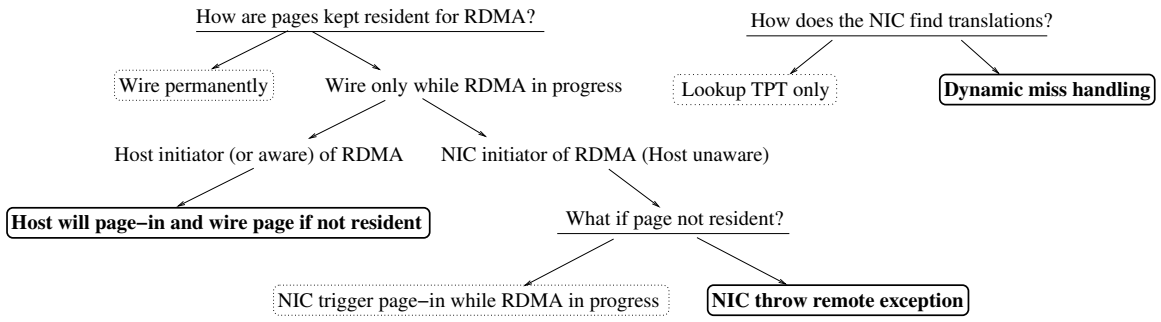

Fig. 1. **Design Decisions for RDMA Page Access and Address Translation. Optimistic DAFS Choices are in Boldface.**

is also no support for exception handling on behalf of the NIC. Finally, most current NICs provide only weak safety when incoming RDMA requests use *stale* or *stray* memory references. As a result, servers would rather initiate the RDMA themselves to make sure only their intended exported buffers are accessed. This paper argues that NIC support for solving these problems will enable protocols and applications with better performance, particularly in file access workloads dominated by short transfers. It will also offer a higher degree of safety than possible under current RDMA-capable SANs.

As an example of an RDMA-based distributed system that takes advantage of our proposed support, we introduce a file access protocol that enhances DAFS by allowing direct access to data stored in a distributed file cache. This enhanced filesystem aims to lower the access latency in short transfers by using client-initiated RDMA as often as possible. To enable client-initiated RDMA, servers export to clients direct memory references to server buffers and clients store those references in a local directory. These references can become stale when the server no longer keeps the corresponding data in the buffers. This is likely to happen when the page daemon invalidates page mappings and writes pages to disk. In our design, servers do not need to send explicit invalidations when page mappings are invalidated. Clients *optimistically* use remote memory references hoping that accesses will succeed, but expecting to catch an exception if they fail.

The outline of this paper is as follows: Section II examines design decisions involved in address translation and access of memory pages with an RDMA mechanism. Section III describes Optimistic DAFS, the distributed filesystem that aggresively uses client-initiated RDMA and *optimistic* remote memory reference consistency without explicit invalidations. We present a cost-benefit analysis that can be used in an adaptive client policy to decide when the use of client-initiated RDMA pays off (intuitively, when few client accesses result in exceptions due to stale memory references), or when switching to standard RPC-based communication is preferable. Section IV describes our proposed NIC support for Optimistic DAFS. This support aims to a) integrate programmable RDMA-capable NICs in the host virtual memory system, b) enable a server NIC to report an RDMA exception back to the client NIC when client-initiated RDMA fails, and c) strengthen safety against invalid remote memory references. Section V discusses shortcomings of I/O buses (such as the widely used PCI) when used for direct memory access by RDMA-capable NIC bus masters. Section VI presents our preliminary performance results.

II. RDMA Design Issues

RDMA is a generalization of Direct Memory Access (DMA), a data transfer mechanism used in I/O and system buses to achieve data transfer between devices and main memory, reducing host involvement to the setup phase of the DMA operation. RDMA extends the data transfer path over the SAN, between memory buffers in remote hosts. The main characteristic of RDMA is that host CPUs are not involved in actual data transfer. User-level networking and the need to avoid all memory copies requires that RDMA use virtually-addressed buffers. NICs with such capabilities use a Translation and Protection Table (TPT) to translate virtual addresses carried on RDMA requests to physical (bus) addresses.

Distributed systems using RDMA have to make sure that the NIC can always find address translations (virtual to physical) of exported pages referenced in RDMA requests and that memory pages used for RDMA are kept resident (e.g., by wiring them) in physical memory while the transfer takes place. The issues of address translation and page wiring are decoupled in the case of a host CPU initiating DMA to and from devices. Success in translation either through the TLB or the page table means that the page is in physical memory. However, explicit page wiring is necessary to ensure a page stays resident for the duration of the DMA. Except for the case of NICs that are integrated with the host CPU sharing a TLB [14], NICs on the I/O bus usually choose to couple address translation and page wiring to avoid having to wire/unwire pages on each RDMA operation accessing them.

In summary, in this paper we are interested in the following design issues (shown in Figure 1):

1. How does the NIC find page translations? One possibility is to restrict the NIC to storing all accessible translations in its TPT. This approach treats a TPT miss as a failure of the RDMA operation. Although this a simple solution, it is not going to work in the case of servers exporting large amounts of physical memory due to the finite

capacity of the TPT. Another possibility is to use the TPT as a cache of translations and allow dynamic handling of NIC translation miss exceptions [15], [16].

2. How are pages kept memory-resident for RDMA? The simplest solution, but one that leads to underutilization of memory, is wiring pages for long periods of time. The alternative of wiring on-demand (i.e., on each RDMA operation) by the host, only works if the host is the initiator of RDMA or is aware of the RDMA and its duration (e.g., by exchanging messages with the remote host). This particular kind of wiring on-demand is the method used by DAFS servers (RDMA initiators) and clients (using request/response RPC messages). Another possibility (which is used by Optimistic DAFS, described in Section III) is to allow exported VM pages on RDMA targets to be candidates for pageout and require the NIC to do the wiring on each RDMA operation. This scheme requires that NICs that succeed in finding a translation for a page to explicitly wire the page (to avoid subsequent pageout) before the translation can be used for DMA.

3. What if a page is found not to be resident? If a page referenced in RDMA is found to have been paged out by the host, the NIC can either trigger page-in or simply abort the RDMA and report an exception to the remote host. The former method introduces disk latencies in remote memory operations that the client may not be prepared to tolerate. The latter is the method used in Optimistic DAFS, as described in Section III.

Previous research [15] has explored the design space for NIC TPT translation miss handling. U-Net/MM [16] provides for dynamic miss handling by the host but demands that pages with TPT translations be wired in physical memory. Although this approach avoids having to wire/unwire pages on each RDMA operation referencing them, it has the drawback of excessive memory wiring in cases of large (multi-GB) TPT sizes.

III. Optimistic DAFS

As mentioned earlier, DAFS uses lightweight RPC for all file requests, followed by server-initiated RDMA for data transfer in *direct* read and write operations. This reliance on lightweight RPC for all file requests comes at a cost. For small data or metadata requests from server memory, the overall latency of the RPC-based file access is composed of (a) the cost of communicating the file access RPC to the server, and (b) the cost of going through the server *vnode* interface to map and lock file pages in the buffer cache [17]. RPC processing imposes an overhead in terms of interrupt handling and thread scheduling. A large part of the latency in having to interrupt the server, schedule and process the RPC and go through the *vnode* interface can be avoided if a direct memory access is used instead, i.e., if the client can maintain and use valid (with high probability) direct memory references to server file data.

Optimistic DAFS aims to lower the latency of small accesses to server memory. The idea behind Optimistic DAFS is that directory information about cached data is distributed (either eagerly or lazily/on-demand) among all nodes in the system but is not eagerly updated when cached data is invalidated or otherwise becomes stale (as a result of the actions of the pageout daemon). As a result, a memory access can fail if it uses stale directory information. In case of memory access failure, the client can use a file access RPC to take the slower path to the file data.

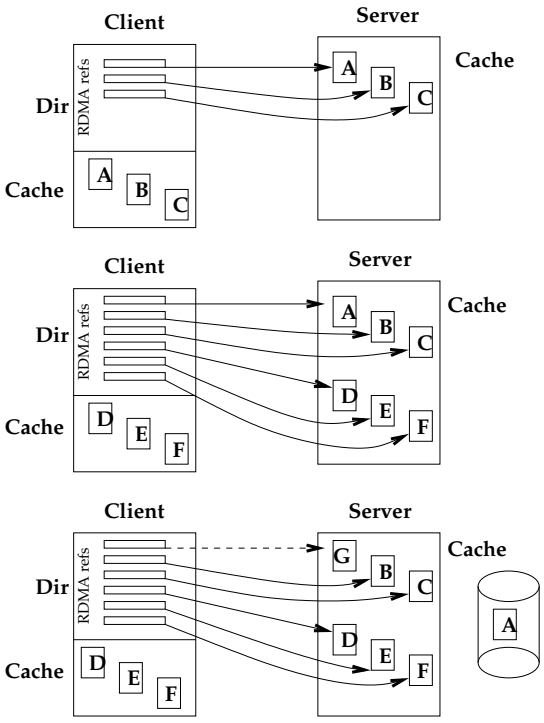

Fig. 2. **Optimistic Client-Initiated RDMA.**

A client node is expected to use part of its memory for caching data and another part for keeping directory information about the data. The directory can be thought of as a cache of remote references that is managed by some replacement policy. As data are replaced in the client file cache, directory information about the data is expected to live longer and be used when that data is requested again. In a sense, the cache directory is an extension of the node's cache where data occupy less space but take longer to access.

Figure 2 depicts a client with a cache that fits three file blocks but with a directory that can hold many more remote references. Initially the client cache is filled with blocks A, B, C. As the client references the next three blocks (D, E, F), the three initial blocks are replaced but their remote memory references retained. On the event of a server pageout (block A is written to disk with A's frame replaced by block G), the client's reference to A is no longer valid (depicted by the dashed arrow line). The client will only find out when it tries to use the reference to A and fails.

TABLE I
DESIGN ISSUES

How are cache directories built?	Cache directories can be built either *lazily* (i.e., add an entry to a block only after an access to that block), or *eagerly* (i.e., add entries proactively in anticipation of future access).
How is file access done?	Optimistic DAFS supports the standard file access DAFS RPC interface (the *slow* path). File access can also be done with client-initiated RDMA operations alone (the *fast* path), when possible.
How are cache directories maintained?	When an RDMA using a directory reference fails, the reference is invalidated. The client then issues an RPC. When an RPC response is received, a new directory entry is added.

A. Design

The main issues in the Optimistic DAFS architecture are summarized in Table I. Cache directories can index the entire cache space or just part of it. For each file access, the client looks up the file blocks needed in the local directory and initiates a remote operation accordingly.

Next we describe two schemes that can simultaneously be used to build cache directories in order to index pages in the distributed file cache.

Lazy scheme. The server piggybacks the memory locations of the buffers containing the data the client asked for, along with the data on each file request RPC (e.g., server responds "*here's the data you asked for, and by the way, these are their memory locations that you can directly use in the future*").

Eager scheme. The client explicitly asks for remote memory references of files that it expects to use in the near future. These requests take the form of queries for an entire file or a list of files (e.g., client asks "*give me the locations of all your resident pages associated with file foo*").

The eager scheme enables prefetching of remote memory references and poses an interesting space/time tradeoff. Since buffer descriptions occupy only a small amount of space as opposed to the actual buffer contents, we can afford to prefetch more aggressively without fear that too much prefetching will evict useful cache data. The penalty one pays in prefetching remote memory references rather than actual data is that access latency is that of remote rather than local memory.

The design of Optimistic DAFS raises a number of research questions:

1. What is the best replacement policy for the client directory? Clearly, a good policy is one that replaces stale references first.

2. Both the file cache and the directory are dynamically sized and expected to compete for physical memory. What is the right balance between the two?

3. How can the directory index file metadata (e.g., directories, in-core inodes, etc.) in server memory, in addition to data blocks?

4. Servers could explicitly invalidate stale references in addition to throwing exceptions. What is the effect of explicit server invalidations?

5. What is the benefit of agressive prefetching of remote memory references? What is the right amount of such prefetching?

A good directory replacement policy requires knowledge of stale references. This information could be obtained if the server sends invalidation messages to clients. Without explicit invalidations, a good approximation is to use a directory replacement policy that coincides with the policy the server uses for VM page replacement, assuming that (a) client access is reflected on server VM state, and (b) there is only a single file client. With many non-cooperating file clients, server feedback will be necessary to come up with an effective directory replacement policy. A problem with direct client access to in-core copies of file metadata on the server is that updates of such structures often need to be followed by disk operations to reflect the updates on stable storage, except when metadata is backed by non-volatile RAM. Given that engineering workloads are metadata-intensive [18], enabling direct access to in-core metadata structures in server memory is expected to be beneficial.

Besides the NIC support described in Section IV, Optimistic DAFS requires VM system support at the server operating system. For example, the host VM system needs to enter and remove NIC mappings when exported VM regions are paged in or out. Such support in the context of the FreeBSD operating system has been presented elsewhere [17] and is beyond the scope of this paper.

B. Cost-Benefit Analysis

Optimistic DAFS reduces interaction with the server processor when a direct path to server memory is known and can be established. It is expected to be beneficial in environments with large server memory caches experiencing high hit rates. The relative benefit from Optimistic DAFS

compared to standard DAFS is expected to come out of lower latency in fine-grain file access and will be proportional to:

1. The fraction of client accesses that miss in the client cache and need to go to the server,

2. The fraction of directory lookups for remote references that hit in the client directory, and

3. The fraction of direct accesses to server memory that are successful (i.e., using valid references).

The benefit of Optimistic DAFS is realized for all file access operations that hit in the client directory and the subsequent RDMA succeeds. The cost of Optimistic DAFS is realized for all file operations that hit in the directory but the subsequent RDMA operation fails. That cost is the time spent from the issue of the RDMA to the time the RDMA exception is caught and handled.

The cost-benefit ratio for a file access workload using Optimistic DAFS compared to standard DAFS is

$$\frac{\text{benefit}}{\text{cost}} = \frac{N_{DirHit} * (T_{1,direct|inline} - T_2)}{N_{DirHitStale} * T_{StaleRDMA}}$$

where $T_{1,direct|inline}$ is the latency of DAFS file access using *inline* or *direct* (defined in Section VI-B) requests, T_2 is the latency of file access using client-initiated RDMA, N_{DirHit} is the client directory hit rate, $N_{DirHitStale}$ is the client directory stale hit rate, and $T_{StaleRDMA}$ is the cost of handling an unsuccessfull RDMA operation. $T_{StaleRDMA}$ includes the roundtrip time to the remote NIC adaptor and the time to receive and handle the RDMA exception.

The above formula can be the basis of an adaptive client policy to decide when use of client-initiated RDMA pays off or when switching to standard RPC-based communication is preferable. Intuitively, client-initiated RDMA is beneficial when relatively few client accesses result in exceptions due to stale memory references. After switching to standard RPC-based communication this formula is no longer applicable and some other way is needed to decide when to switch back to client-initiated RDMA. One possibility is to spontaneously switch back to client-initiated RDMA after a certain time interval and restart monitoring performance using the cost-benefit formula. Another possibility is to switch back to client-initiated RDMA based on server feedback of its cache efficiency.

IV. Network Interface Support

As explained in Section II, NICs designed to support optimistic client-initiated RDMA need to be able to

1. Synchronize with other system agents (such as the host CPU) over access to VM state of memory pages,

2. Dynamically handle TPT misses, and

3. Report RDMA exceptions to remote initiators.

In addition, NICs need to be able to update VM page counters and flags to reflect network memory access, so that the host can take it into account when making paging decisions.

An RDMA capability requires reliable transfer semantics that can be implemented using either hardware-supported mechanisms at the link layer (e.g., as in the case of Fibre Channel), or end-to-end software protocols such as TCP (e.g., as in the case of Ethernet). Both host and NIC implementations of RDMA are possible. However, NIC implementations are preferable as they can offload the server CPU and permit protocol optimizations [19], [20] that are either not possible or not easily deployed on host operating system network stacks. NICs with programmable processors require a lightweight kernel to support a network stack and an RDMA protocol.

Enabling RDMA-capable NICs to interoperate with *any* host operating system can be done by defining a NIC module that can be customized for this task. Separating a software module that enables interoperation with the host OS from the rest of the NIC firmware opens the possibility of decomposing the NIC architecture between a hardware state machine (e.g., an ASIC) and a programmable processor executing this module. Inserting a customized software module into a NIC can be done by either flashing (i.e., overlaying) the NIC with a new control program that contains the module, or by being able to dynamically download a software module into existing (possibly running) NIC firmware. The solution to flash the NIC with host OS-specific firmware is impractical as it affects the entire firmware and not just the part that concerns the interaction with the host VM system. When the entire NIC firmware is implemented in software, dynamically downloading a module is a better solution as it leaves the core of the firmware unaffected.

There are two main issues with designing such a module: First, what is the interface between this and the rest of the NIC firmware? Second, how does this module access host VM state? To address the first issue, we define an interface that encapsulates all VM actions that need to be triggered by the NIC. Access to host VM state can happen either by a) direct memory access to shared VM state, using memory read/write instructions and possibly bus support for mutual exclusion, or b) exceptions thrown by the NIC and handled by the CPU. In either case, NIC action is targeted at VM page metadata (i.e., VM structures in main memory keeping information about VM pages). To facilitate interaction, the NIC stores references (and translations) to such metadata in its TPT. When the NIC can find such information in its TPT, either direct access or indirect access through the host CPU is expected to be fast since it avoids page lookup in an address space or a backing object.

An exception notification mechanism is necessary for RDMA operations using stale memory references (e.g., due to changes in the target memory maps, as happens with pageouts), and RDMA operations using unauthorized

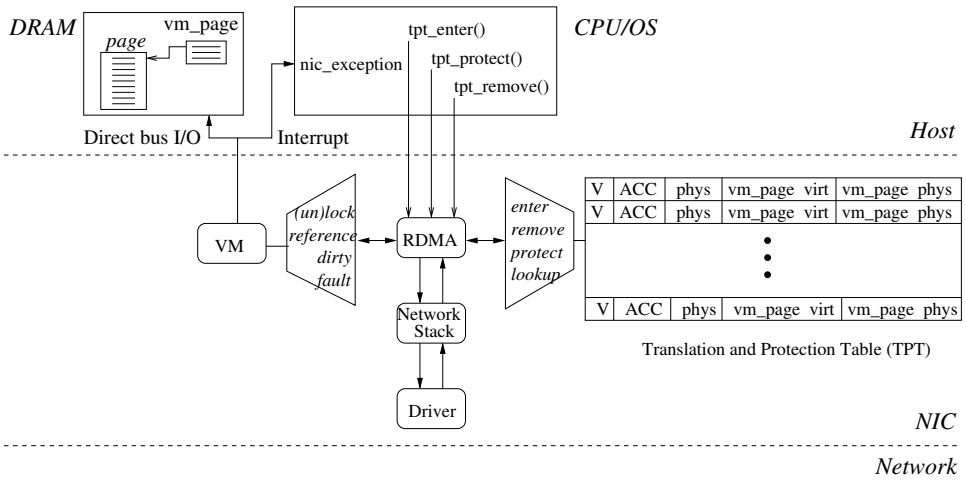

Fig. 3. **NIC Firmware Structure. The NIC interacts with the host OS either by direct memory access or using interrupts.**

or stray remote memory references (e.g., due to program bugs). Such a notification mechanism can be built as part of an RDMA protocol as described in Section IV-B. A capability-based safety mechanism that guards against invalid or unauthorized RDMA access is described in Section IV-C.

A. NIC Firmware Structure

Figure 3 shows the NIC firmware structure proposed in this paper. It includes a network stack, an RDMA module that implements the remote direct memory access protocol, a VM module that interacts with the host VM system, and a Translation and Protection Table. One way to realize this structure and enable dynamic downloading of modules on a programmable NIC is by using a lightweight extensible kernel such as eVINO, an adaptation of the VINO [21] operating system for NIC platforms. The module implementing the RDMA protocol can be layered on top of TCP in case of an unreliable transport such as Ethernet, or over a reliable link-layer protocol such as Fibre Channel.

The TPT (shown in Figure 3) includes both page address translations *and* page metadata address translations. In what follows we refer to metadata info about a VM page as its *vm_page* structure. The host OS triggers TPT updates (passing translations for both pages and their *vm_page*s) via a simple interface exported by the NIC. For interaction with the host VM system, the RDMA module invokes the VM module. The TPT entry format is modeled after standard RDMA protocols such as VI and contains *valid* (V), *access control* (ACC), and *physical address* (phys) fields. We extend this standard TPT entry format to store translations of page metadata in *vm_page virtual* and *vm_page physical* address fields. We note here that the TPT has no information as to how to interpret page metadata. This is the task of the downloadable OS-specific VM module. Additional TPT fields are required for use by safety mechanisms such as the one described Section IV-C.

A.1 VM Module

The VM module is responsible for interacting with the host VM system. The interface between the VM module and the other NIC subsystems (such as the RDMA module) includes the following functions:

1. *lock()* sets the busy flag for a page or region of memory. This is not equivalent to wiring memory in that the page(s) marked busy are still in the paging queues, just temporarily protected from pageout by the paging daemon.

2. *unlock()* resets the busy flag for a page or region of memory. It may also need to wake up anyone who is waiting for it.

3. *reference()* marks a page or region of memory as referenced. Such an action may require moving page(s) between paging queues, e.g., making a transition from an inactive to an active state.

4. *dirty()* marks a page or region of memory as having been modified.

5. *nic_vm_fault()* triggers dynamic translation miss handling. Due to the complexity of this operation it is best handled entirely by the CPU.

This interface can be implemented simply with the NIC triggering interrupts and letting the host CPU take the appropriate action in each case. Such an implementation requires minimal host operating system kernel changes apart from the NIC device driver. A potentially more efficient way would be to implement this interface (except *nic_vm_fault()*) entirely at the NIC using only atomic memory (single word) read and write operations. The only impediment to such an implementation is that activity such as waking up waiting processes and moving pages between paging queues requires host CPU assistance. A solution could be to delay such bookkeeping so that work can be batched and processed by the CPU with fewer interrupts.

A.2 RDMA Module

The RDMA module implements a protocol that provides for message passing, remote direct memory access, and remote atomic operations. The NIC communicates with the host over shared *command descriptor* and *doorbell* queues using special messaging hardware support [22]. The RDMA module is layered over a network protocol offering a reliable transfer mode.

For locally-initiated operations, the RDMA module interacts with the local host via the shared queues and reports exceptions by triggering interrupts. Incoming remotely-initiated RDMA requests carry virtual address references to host main memory. The NIC is expected to lookup these virtual addresses into its TPT and take one of the following actions:

1. In case of a TPT hit, lock the relevant memory pages before the data transfer, unlock them after the transfer is over and update the pages' reference and dirty bits as appropriate.

2. In case of a TPT miss, try to find a translation via a slower path by asking the VM module (Section IV-A.1) to trigger lookup into VM system structures in main memory.

3. If the VM module cannot find a translation, the RDMA module reports an exception (Section IV-B) to the remote initiator.

The RDMA module exports the following interface to the host to be used in order to register and deregister page mappings with the NIC.

1. *tpt_enter()* is used to add a page virtual-to-physical address translation as well as a translation for the page's *vm_page* structure. Mappings can be added in the TPT by the host when it expects network I/O to take place in a memory region, or implicitly by the NIC when it succeeds in finding a translation after a previous TPT miss fault.

2. *tpt_remove()* is used to invalidate a page translation. Mappings are usually removed from the TPT by the pageout daemon when pages are swapped out to disk, or implicitly by the NIC when it needs to replace an entry.

3. *tpt_protect()* is used to change the protection of a page entry.

B. RDMA Exceptions and Remote Memory Reference Consistency

It is possible that client-initiated RDMA accesses miss on both the TPT lookup and the subsequent VM lookup (via *nic_vm_fault()*) at the server NIC, either because a requested memory page is swapped out to disk, or because an invalid memory reference is used (i.e., causing a "segmentation fault"). Such a miss should be reported to the RDMA initiator (client) in order to prompt a switch to an alternative access method. In this paper, we propose the constructive use of remote memory access exceptions caused by missing or invalid NIC mappings as the main tool of a remote memory reference consistency scheme. To support such a scheme, we require that translation errors in RDMA operations initiated by a local NIC and detected by a remote NIC be reported back to the initiator NIC. Existing RDMA-capable SAN implementations often choose to treat such exceptions as catastrophic (i.e., they tear down the connection) and for simplicity opt not to report them to the initiator of the RDMA operation. However, specifications that allow non-catastrophic reporting of remote exceptions to local initiators do exist (e.g., as part of the Reliable Reception mode in the VI [23] protocol) and can be implemented using message exchanges. Besides missing or invalid mappings, other reasons to report RDMA exceptions include failure of the NIC to lock a page and protection violations.

Our NIC support strives to minimize the amount of TPT and VM misses taken by RDMA accesses. First, an appropriate TPT entry replacement policy ensures that frequently accessed pages mostly hit in the TPT. Similarly, frequently accessed pages are expected with high probability to be found resident in physical memory as the NIC reflects network access to VM state using the *reference()* (Section IV-A.1) interface.

C. RDMA Safety

Two important concerns with using RDMA on a SAN are to avoid accidental or malicious buffer corruption, as well as unauthorized remote memory access. We have devised a capability [24] mechanism based on cryptographically-strong hashing that guards against invalid or unauthorized accesses to remote memory. In this scheme, a server exporting a memory segment associates a random k-bit value (*AccessKey*, a long integer) with the registered memory segment and stores this value at its NIC translation and protection table. The server exports a memory reference (which is a triple <*Address, Handle, AccessKey*>) only to a client with which it has a trust relationship. A trust relationship means that the client and the server share a secret key K. The value *AccessKey* has to accompany each client RDMA request and has to match the value stored at the server NIC. Each client RDMA operation carries a tuple of the form <*Address, Handle, AccessKey, Timestamp, KMAC*> where

1. *Address* is the virtual address of the remote segment.

2. *Handle* is an index into the translation table of the NIC memory management unit.

3. *AccessKey* is the capability that the server has assigned to the memory.

4. *Timestamp* is a monotonically increasing value at the client, guarding against replays from clients that snoop RDMA requests.

5. *K-MAC* (message authentication code) is a keyed hash function computed on all the above quantities, verifying their integrity and authenticity.

The server makes sure the *AccessKey* value of an incoming request matches that in its local table and that the *Timestamp* is greater than the last received timestamp from that client. Finally, it makes sure that *K-MAC* is correct by recomputing it. Note that only the client and the server can compute *K-MAC* since they are the only ones who know K. A secret key exchange protocol (e.g., based on a public key protocol such as RSA, at connection setup time) is required.

Important properties of this capability mechanism are:

1. The server can revoke a capability at any time by simply invalidating an entry at its NIC translation and protection table. The server does not need to keep track of all clients who have a copy of that capability and does not need to send explicit invalidation messages.

2. The access rights granted with capabilities are stored at the NIC tables and are not given out to clients. Therefore clients cannot forge these rights.

V. Mutual Exclusion Between CPU and NIC

Mutual exclusion between NICs and main processors is necessary when they simultaneously access shared memory structures such as VM page state. Mutual exclusion between multiple processors in a symmetric multiprocessor (SMP) with a shared bus is often implemented with bus support for locking primitives and atomic operations. However, current RDMA-capable NICs (including InfiniBand prototype host-bus adapters) use the PCI bus to interface with host processors and memory which provides limited support for mutual exclusion.

One way to support atomic memory access on system buses is by applying bus interlocks using a LOCK# signal. For example the Intel Architecture [25] provides a *lock* instruction prefix that is used to assert the LOCK# bus signal during critical memory operations. This is sufficient to implement locking primitives in the face of simultaneous memory accesses from multiple processors. However, this mechanism does not easily extend to NIC on the PCI bus even though PCI provides a LOCK# signal. This is because there are limitations as to whether NIC processors can drive this PCI signal, and whether multiple NIC can use this mechanism in a scalable, deadlock-free manner to synchronize access to host memory.

In the absence of bus locking support, the NIC and main CPU can synchronize using special mutual exclusion algorithms [26] that rely solely on atomic (single word) read and write operations.

VI. Performance

This section describes a simulation method that aims to collect statistics that characterize system behavior with Optimistic DAFS. To compare file access latency using RPC and client-initiated RDMA we analyze the part of the file access latency that involves NIC interaction with the host and present microbenchmark measurements for both cases.

A. Simulation Method

We simulate an Optimistic DAFS system by modifying our existing DAFS server [17] and client implementations as follows: The DAFS server is modified to piggyback on RPC responses the memory references of accessed data, as described in Section III. New RPC procedures have been added to enable clients to access the data pointed to by these memory references. When references are found to be invalid, the server reports an exception in the status field of the RPC response.

User-level client caching with DAFS is possible but subject to limits on the amount of memory a process can allocate for the cache. To make sure the client file cache can extend up to the amount of available physical memory we decided to use a kernel file client and the kernel buffer cache. Client applications use a modified MFS [27] kernel file system. Instead of using pageable memory, MFS is modified to use the DAFS file client (built inside the kernel) for block access to a *logical disk* implemented as a single large file exported by the DAFS server. Application file I/O requests enter the kernel via standard system calls and are first looked up in the kernel VFS cache. Cache misses are then looked up into a client directory to find out whether remote references to this data have been previously cached. A directory hit is the result of a reference found and succesfully used by the DAFS file client to directly access the requested blocks in server memory using the new RPC interface. A stale directory hit is the result of a reference found but whose use triggers an exception at the server. After being notified of a stale reference, the client removes it from the directory and issues a standard file-access RPC. This simulation method allows us to collect the following statistics:

1. Client cache hit rate ($N_{CacheHit}$)

2. Client directory hit rate (N_{DirHit})

3. Client directory stale hit rate ($N_{DirHitStale}$)

For our simulations we use two Pentium III PC at 800MHz connected via an Emulex cLAN [28] VI network. The client has 256MB and the server 1GB of DRAM. All systems run patched versions of FreeBSD 4.3.

We generate a workload using a simple application that randomly reads 8KB blocks from a large file, the only file in the logical disk. With this setup, the file accessed by the application via MFS essentially coincides with the server file simulating the logical disk exported by DAFS. We varied the file size in order to put progressively more strain on the server virtual memory system. Our results show that:

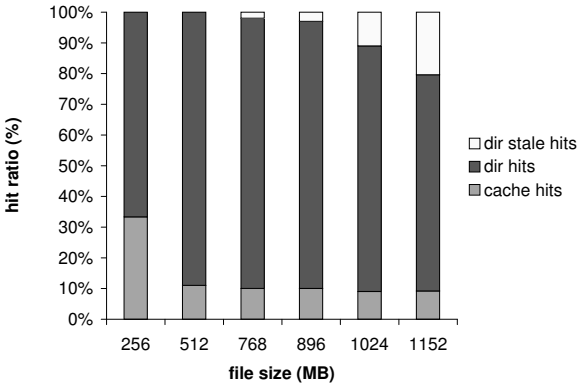

Fig. 4. **Distribution of cache hits, directory hits and stale directory hits, over a long stream of uniformly random requests. The client cache grows up to 200MB of physical memory.**

1. Optimistic client-initiated RDMA operations seldom fail when the file fits in server physical memory, and

2. The rate of stale RDMA increases slowly as the working set exceeds server physical memory, suggesting a graceful performance degradation.

Figure 4 shows the distribution of client cache hits, directory hits, and stale directory hits for a long stream of random file accesses and for file size ranging from 256MB to 1.15GB. Before each run, the client directory is pre-loaded with references to the accessed file. Due to the compact directory entry encoding used in this simulation, the client directory can index the entire server physical memory space using less than 2MB of client physical memory. For realistic directory entry sizes (e.g., 64 bytes), memory requirements of the directory are expected to interfere with data caching. Limiting the size of the directory will reduce its efficiency but leave more memory available for data caching. Determining the right balance is a goal of our on-going research.

B. File Access Latency

In this section we compare the latency of the following two methods of file access:

1. Access using file request RPC (e.g., using file handle, offset, length). This is the method used by DAFS clients. Based on whether the client request is for *inline* or *direct* transfer, the data transfer happens either inline in the RPC request or response message, or using server-issued RDMA operations separately from the RPC request/response.

2. Access using client-initiated RDMA to server memory. This method assumes that the client maintains a directory of references to server memory. The host CPU can assist the NIC at the server by doing exception-level processing in response to actions triggered by the NIC VM module (Section IV-A.1).

Section VI-B.1 analyzes latency in the case of file access using RPC. Section VI-B.2 does a similar analysis for the case of client-initiated RDMA. Both access methods have similar internal NIC latency given that we assume the same NIC architecture. Under this assumption, we compare the access methods based on microbenchmarks of NIC interaction with the host CPU and main memory. We note here that although our experiments involve a NIC with a programmable processor running a light embedded kernel, the NIC support proposed in Section IV (except for the customizable VM module) can be implemented in hardware.

For our experiments we use a Pentium III host at 800MHz running FreeBSD 4.3. Our prototype RDMA-capable NIC implementation has an on-board Intel i960 RD microprocessor [22] at 66 MHz running the eVINO extensible kernel. The NIC has a 32-bit, 33MHz internal bus and interfaces with the host over 33MHz, 32-bit PCI. Communication between the host and the NIC is implemented by message exchanges over a shared circular queue. The NIC interrupts the host by enqueuing a message describing a request. The host dequeues the request, carries out the requested action and responds by enqueueing a response message. The NIC is continously polling for host messages. Both sides can access each other's memory via DMA or via memory instructions using appropriate bus space addresses.

B.1 RPC Latency

The latency experienced by a client in doing a file access RPC depends on whether the client request is for *inline* or *direct* transfer. Without loss of generality, we consider the steps in the case of a read request.

1. In an incoming RPC request, the NIC writes the message contents into a pre-posted buffer in main memory and signals completion. The server is notified of the incoming RPC request by either polling a descriptor, or by receiving an interrupt. Polling is expected to work well on busy servers.

2. The server parses the file or memory request, looks up the requested pages in the file or VM cache, and locks them in order to avoid modification or pageout. We assume that the request hits in the server cache.

3. The server prepares to respond in a way that depends on the type of access which the client requested:

(a) In a *direct* read request, the server issues an RDMA Write out of its file cache and into the client buffers. The server needs to wait until this operation is done before sending the RPC response, either polling for completion or sleeping until woken up by an event handler.

(b) In an *inline* read request, the server copies the data out of its file cache into the response message, unlocks its buffers and prepares the RPC response message. After posting a send on the RPC response, the server does not have to wait for completion.

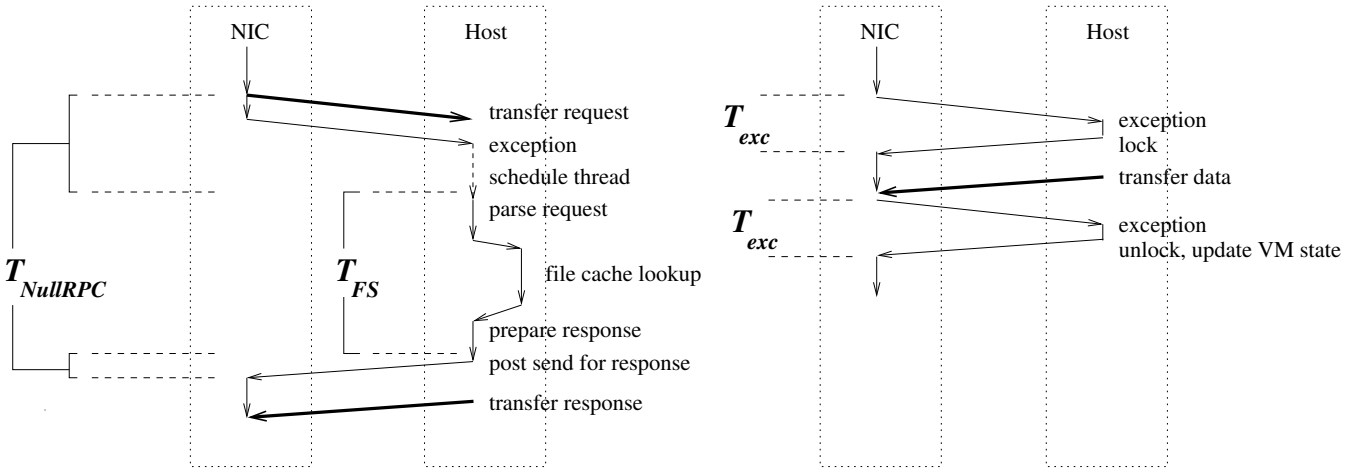

Fig. 5. **NIC/Host interaction in inline RPC (left) and client-initiated RDMA for a read request. We assume that the response data transfer time is about the same in both cases and therefore not taken into account in $T_{1,inline}$, T_2.**

The following formula describes the latency of the interaction between the NIC and the host CPU in an *inline* file request RPC:

$$T_{1,inline} = T_{NullRPC} + T_{FS}$$

$T_{NullRPC}$ (see Figure 5) is the sum of: (1) the time it takes for the NIC to transfer the request message to main memory and notify the host of a message arrival, up to when the host CPU starts executing the RPC handler routine in the context of a kernel thread, and (2) the time it takes from the host CPU posting a send command for the RPC response message, to just before the NIC starts to DMA the message into its buffers. T_{FS} is the sum of: (1) the time it takes to parse the file request, and (2) the time it takes to lookup, lock and unlock cached file pages using the *vnode* interface, and (3) the time to prepare the response descriptor.

$T_{1,direct}$ describes the case of a *direct* file RPC request:

$$T_{1,direct} = T_{NullRPC} + T_{FS} + T_{RDMA}$$

T_{RDMA} is the latency between the time the thread posts the RDMA operation to the time the RDMA operation is completed and the RPC response message is ready to be posted. We note that T_{RDMA} differs for read and write requests. A server-issued RDMA Read operation is expected to be more expensive than an RDMA Write operation as it includes additional processing at the client NIC and at least one additional message. For small requests, clients are more likely to ask for *inline* rather than *direct* transfer in order to avoid the overhead of issuing an RDMA at the server.

We set up a microbenchmark to measure $T_{1,inline}$ for a read request for 64 bytes using a DAFS server. We assume that both the request and response message are 64 byte long. Results reported in Table II are averages over ten measurements with small standard deviation.

$T_{1,inline}$ is 70μs when a worker thread can be immediately scheduled to handle the client request. $T_{NullRPC}$ is 45μs (measured on the NIC) and T_{FS} is 25μs (measured on the host). We note here that 12μs of $T_{NullRPC}$ are spent in the DMA transfer of the 64 byte request from the NIC to main memory. About 10μs of $T_{NullRPC}$ are spent context switching to the server thread after notification from the NIC. All measurements were taken with a warm buffer cache.

B.2 RDMA Latency

We now focus on access method (2). The cost of incoming RDMA read and write operations can be broken down into the following components, assuming successful TPT lookups.

1. Lock pages about to be used in DMA.

2. Set up and carry out the data transfer between NIC buffers and host memory.

3. Mark pages just used in DMA as referenced.

4. Mark pages dirty if access was a write.

5. Unlock pages.

When locking pages during exception handling, no lookup is necessary when the NIC can provide the host with references to *vm_page*s. Since exceptions are non-schedulable entities and not allowed to block, use of non-blocking primitives is required. Failure to lock after a limited number of retries (e.g., due to contention for that page lock) may result to failure of the entire RDMA operation.

The aim is to process as much of the above operations as possible directly at the NIC, asking for host CPU assistance only when it is necessary to access VM structures in main memory. The following formula describes the latency of the interaction between NIC and the host CPU when handling an RDMA operation with host CPU assistance.

TABLE II
Microbenchmark Results

Operation	Cost (μs)
$T_{NullRPC}$	45
T_{FS}	25
T_{exc}	15

$$T_2 = k \times (T_{exc} + T_{VM})$$

where k is the number of interactions with the host, T_{exc} is the exception handling overhead when the CPU is involved, and T_{VM} is the time to access VM state. Interaction between the host and the NIC is over a set of circular queues as described earlier.

We estimate T_2 for a read request for 64 bytes. We assume that interaction is needed twice, once to lock pages and once more to unlock and update VM state. Since it takes only a few machines cycles to carry out bit or counter operations in fields of the *vm_page* structure, T_{VM} is small and can be ignored. Results are reported in Table II. Based on the T_{exc} measurement of 15μs, T_2 is estimated at about 30μs.

B.3 Discussion

Based on published data [5] for an ATM adapter (Fore SBA-200) with i960 firmware, we estimate file access latency (excluding the NIC/host interaction on the target side and assuming the client is polling) of between 50μs and 100μs. Based on these estimates, we find that overall file access latency using client-initiated RDMA is 20% to 40% lower than with using RPC. We note that this is a rough estimate based on measurements of a prototype with few optimizations. We plan for a more thorough evaluation in the future.

NICs that are tightly coupled with the host [29], [30], [31], [14], [32] aim at lowering the NIC overhead as well as the overhead of the NIC interaction with the host for control and data transfer. Previous research [31] has pointed to the importance of NIC design for low-latency RPC communication.

Scheduling delays included in $T_{NullRPC}$ can be reduced using special RPC implementations, such as Optimistic RPC (ORPC) [33]. With ORPC, procedures are executed as interrupt handlers under the optimistic assumption that they will neither block nor run for too long. When handlers violate these assumptions, ORPC reverts to the slower method of creating and associating a thread with the procedure. ORPC is another point of comparison with optimistic client-initiated RDMA that we plan to explore in future work.

VII. Conclusions

We have presented the design of Optimistic DAFS, a system that enables clients to optimistically use RDMAs to directly access exported server cache pages without excessive page wiring at the server in the face of paging activity.

Our simulation results show that optimistic client-initiated RDMA exhibit low failure rates when working sets fit in server physical memory. We have taken microbenchmark measurements of optimistic RDMA and RPC for short messages. We estimate that optimistic RDMA can reduce the access latency by about 20% to 40% over file request RPC used by standard DAFS clients. Given that file servers with enormous physical memory configurations are becoming prevalent due to falling memory prices, we expect that performance benefits seen in microbenchmarks will carry on to end-to-end application performance for workloads dominated by short transfers.

New technologies such as InfiniBand set the trend of bringing the RDMA-capable NIC closer to the memory controller. This trend is expected to drastically lower the latency of RDMA and offer a strong motivation for designs based on optimistic client-initiated RDMA. To avoid any interaction with the host CPU, new VM algorithms will be needed to enable CPU/NIC interoperation based solely on shared memory access.

We are currently working on an implementation of an RDMA-capable NIC along the lines of the design presented in this paper. This implementation will allow us to measure the benefit of Optimistic DAFS (and of optimistic use of RDMA in general) in real workloads. We plan to report our results in a forthcoming paper.

VIII. Acknowledgments

The author would like to thank Margo Seltzer, Alexandra Fedorova, Eran Gabber and Nikos Hardavellas for their helpful feedback.

References

[1] P. Druschel and L. Peterson, "Fbufs: A High-Bandwidth Cross Domain Transfer Facility," in *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, 1993, pp. 189–202.

[2] V. Pai, P. Druschel, and W. Zwaenepoel, "IO-Lite: A Unified I/O Buffering and Caching Scheme," in *Proceedings of the 3rd USENIX Symposium on Operating System Design and Implementation (OSDI)*, 1999.

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

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

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

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

[7] C. Thekkath, H. Levy, and E. Lazowska, "Separating Data and Control Transfer in Distributed Operating Systems," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Sys tems, San Jose, CA*, October 1994, pp. 1–12.

[8] InfiniBand Trade Association, *InfiniBand Architecture Specification, Release 1.0*, October 2000.

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

[10] T. Noboru, Y. Junji, N. Hiroaki, K. Tomohiro, and Amano Hideharu H. Yoshihiro, Nakajo Hironori, "Low Latency High Bandwidth Message Transfer Mechanisms for a Network Interface Plugged into a Memory Slot," *Cluster Computing 2000*, vol. 5, January 2002.

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

[12] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout, "Measurements of a Distributed Filesystem," in *Proceedings of 13th ACM Symposium on Operating Systems Principles*, 1991, pp. 198–212.

[13] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, D. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber, "Structure and Performance of the Direct Access File System (DAFS)," in *Proceedings of the USENIX 2002 Technical Conference, Monterrey, CA (to appear)*, June 2002.

[14] B. S. Ang, D. Chiu, L. Rudolph, and Arvind, "Message Passing Support on StarT-Voyager," Tech. Rep., CSG Memo 387, MIT Laboratory for Computer Science, July 1996.

[15] I. Schoinas and M. D. Hill, "Address Translation Mechanisms in Network Interfaces," in *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA)*, February 1998.

[16] M. Welsh, A. Basu, and T. von Eicken, "Incorporating Memory Management into User-Level Network Interfaces," in *Proceedings of the 1997 Hot Interconnects Symposium*, August 1997.

[17] K. Magoutis, "Design and Implementation of a Direct Access File System (DAFS) Kernel Server for FreeBSD," in *Proceedings of the USENIX BSDCon 2002 Conference, San Francisco, CA*, February 2002.

[18] D. Robinson, "The Advancement of NFS Benchmarking: SFS 2.0," in *Proceedings of the XIII USENIX LISA Conference*, November 1999.

[19] T. Blackwell, "Speeding up Protocols for Small Messages," in *Proceedings of the ACM SIGCOMM '96*, September 1996, pp. 85–95.

[20] D. Mosberger, "Analysis of Techniques to Improve Protocol Latency," in *Proceedings of the ACM SIGCOMM '96*, September 1996, pp. 73–84.

[21] M. Seltzer, Y. Endo, C. Small, and K. Smith, "Dealing with Disaster: Surviving Misbehaving Kernel Extensions," in *Proceedings of the 1996 Symposium on Operating System Design and Implementation*, October 1996.

[22] Intel Inc., *i960 Rx I/O Microprocessor Developer's Manual*, April 1997.

[23] Intel Inc., *Virtual Interface Architecture Developer's Guide: Error Table Supplement*, September 1998.

[24] M. Wilkes, "Hardware Support for Memory Protection: Capability Implementations," in *Proceedings of the 1st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-I), Palo Alto, CA*, March 1982.

[25] Intel Inc., *Intel Architecture Software Developer's Manual. Volume 3: System Programming*, 1999.

[26] L. Lamport, "A Fast Mutual Exclusion Algorithm," Tech. Rep., DEC SRC, 1986.

[27] M. K. McKusick, M. J. Karels, and K. Bostic, "A Pageable Memory-Based Filesystem," in *Proceedings of the 1990 USENIX Annual Technical Conference*, 1990, pp. 137–144.

[28] Emulex Inc., "cLAN 1000 VI Host Bus Adapter," `http://wwwip.emulex.com/ip/products/clan1000.html`.

[29] D. S. Henry and C. F. Joerg, "A Tightly-Coupled Processor-Network Interface," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS), Boston, MA*, October 1992, pp. 111–121.

[30] J. Kubiatowicz and A. Agarwal, "Anatomy of a Message in the Alewife Multiprocessor," Tech. Rep. LCS/TM-498, MIT, 1993.

[31] C. Thekkath and H. L. Levy, "Low-Latency Communication in High-Speed Networks," *ACM Transactions on Computer Systems*, vol. 11, no. 2, May 1993.

[32] R. Osborne, Q. Zheng, J. Howard, R. Casley, and D. Hahn, "DART - A Low Overhead ATM Network Interface Chip," in *Proceedings of Hot Interconnects IV, Palo Alto, CA*, August 1996.

[33] S. Wallach, W. Hsieh, K. Johnson, M. Kaashoek, and W. Weihl, "Optimistic Active Messages: A Mechanism for Scheduling Communication and Computation," in *Proceedings of the Fifth Symposium on Principles and Practices of Parallel Programming*, April 1995, pp. 217–226.