Design And Implementation of a Direct Access File System (DAFS) Kernel Server for FreeBSD[†]

Kostas Magoutis
Division of Engineering and Applied Sciences, Harvard University
magoutis@eecs.harvard.edu

Abstract

The Direct Access File System (DAFS) is an emerging commercial standard for network-attached storage on server cluster interconnects. The DAFS architecture and protocol leverage network interface controller (NIC) support for user-level networking, remote direct memory access, efficient event notification, and reliable communication. This paper describes the design of the first implementation of a DAFS kernel server for FreeBSD, using existing interfaces with minor kernel modifications. We experimentally demonstrate that the current server structure can attain read throughput of more than 100 MB/s over a 1.25 Gb/s network even for small (i.e. 4K) block sizes when prefetching using an asynchronous client API. To reduce multithreading overhead and integrate the NIC with the host virtual memory system, our forthcoming system will incorporate new FreeBSD kernel support for asynchronous *vnode* I/O interfaces, integrating network and disk event notification and handling, and VM support for remote direct memory access. We believe our proposed kernel support is necessary to scale event-driven file servers to multi-gigabit network speeds.

1 Introduction

The emergence of network transports enabling low-overhead access to the network interface from user or kernel address space, remote direct memory access (RDMA), transport protocol offloading, and hardware support for event notification and delivery, has given rise to new applications and services that take advantage of these capabilities. The Direct Access File System [6] (DAFS) is a new commercial standard for file access over this new class of networks. DAFS grew out of the *DAFS Collaborative*, an industrial and academic consortium led by Network Appliance and Intel. DAFS file clients are usually applications that link with user-level libraries that implement the file access API. DAFS file servers are implemented in the kernel.

This paper describes the first implementation of a DAFS kernel server for FreeBSD. It is also as far as we know the first implementation in any general-purpose, demand-paged operating system. Section 2 summarizes the characteristics of network transports DAFS is designed to work with. Section 3.1 describes our prototype DAFS implementation using existing kernel interfaces. Section 3.2 introduces Optimistic DAFS, a new design we propose with potentially better performance but requiring additional kernel support. Section 4 outlines the kernel structure issues one is faced with in order to efficiently support DAFS kernel servers in FreeBSD. These include the need for asynchronous *vnode* interfaces to map and lock file pages in the buffer cache, integrating network and disk I/O event delivery, virtual memory support for network interface controller (NIC) memory management hardware, the need to revisit buffer cache locking assumptions, and a new BSD device driver model. Section 5 presents the performance of the current DAFS kernel prototype implementation and Section 6 summarizes our conclusions.

2 Memory-to-Memory Transports

DAFS [6] is a file access protocol specification deriving from NFS version 4 [22]. It is tailored for network transports (often referred to as *memory-to-memory* networks) providing user-level access to the network interface, remote direct memory access, efficient asynchronous event delivery mechanisms, and reliable communication semantics. Examples of memory-to-memory transports are Virtual Interface (VI) [5] and InfiniBand [11]. Current commercially available memory-to-memory

[†]Appears in proceedings of USENIX BSDCon 2002 Conference, San Franscisco, CA, February 11-14, 2002.

network intercace have a long research heritage behind them [4, 23, 15]. The potential of advanced memory management features has also been considered [21, 24].

In this section we describe the characteristics of commercially available memory-to-memory networks that are relevant to a DAFS kernel server implementation.

Remote direct memory access (RDMA). Memory-to-memory networks are capable of data transfer between virtually addressed buffers in user process or kernel address space over the network. Hosts have to register virtual address mappings of buffers with the NIC prior to RDMA but are not involved in the actual data transfer. The programming interface to RDMA (except for buffer registration which is handled by the device driver) is usually through access to a memory-mapped data structure of transfer descriptors. Read, write (and sometimes atomic) remote memory access is allowed.

Registration of memory buffers with the NIC. The NIC includes a memory management unit in order to translate host virtual addresses to physical (bus) addresses to use in setting up DMA transfers. Most current commercially available NIC do not handle translation miss faults. The host needs to register (i.e. fill in mappings) with the NIC for all virtual memory regions the NIC is expected to access.

VM pages that have their mappings registered with the NIC have to be prevented from pageout at least while RDMA with them is in progress. Kernel interfaces that lock pages for the duration of an I/O suffice to prevent pageout when RDMA is locally initiated. With remotely initiated RDMA transfers that may happen at any time (as described in Section 3.2), only the NIC knows exactly when these transfers take place. To avoid excessive page locking by the host CPU, the NIC should have the ability to trigger or carry out page locking when needed. Support for integrating the NIC with the VM system is described in Section 4.3. Such support will enable a server to export large buffers (i.e. the entire VM cache) without underutilizing physical memory.

Efficient asynchronous event delivery mechanism. Memory-to-memory networks offer the *completion group* abstraction for scalable event notification and delivery. Completion groups simplify the task of simultaneously polling a large set of connections by aggregating their event notification and delivery into a single structure. Events such as receipt of a client request, or completion of a data transfer, can be efficiently detected and handled.

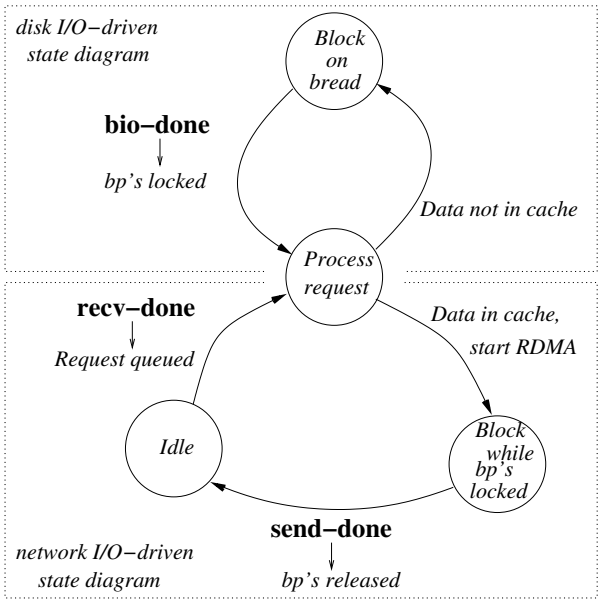

Figure 1: **Event-Driven DAFS Server.** Blocking is possible with existing interfaces.

Connection-oriented and reliable transport. Data transfer is usually over peer-to-peer transport connections (or channels). Reliable, exactly-once transport semantics are expected to be offered. Such semantics are usually implemented with hardware support in the network (as in the case of Fibre Channel [3]) or with end-to-end protocols implemented on the NIC (as in the case of VI/IP [9]). In either case, the host is not involved.

3 Direct Access File Systems

Direct access file systems are providing network file service over memory-to-memory transports. Section 3.1 introduces DAFS [6], an emerging commercial standard, and describes its prototype server implementation for FreeBSD. Section 3.2 introduces Optimistic DAFS, an improved design that relies more on RDMA and less on RPC for communication but requires special kernel support, particularly in the virtual memory subsystem.

3.1 DAFS

DAFS clients use lightweight RPC to communicate file requests to servers. In *direct* read or write operations the client provides virtual addresses of its source or target memory buffers and data transfer is done using RDMA operations. RDMA operations are always issued by the server. In this paper we focus on server structure and I/O performance.

3.1.1 Server Design and Implementation

This section describes our current DAFS server design and implementation using existing FreeBSD kernel interfaces with minor kernel modifications. Our prototype DAFS kernel server follows the event-driven state transition diagram of Figure 1. Events are shown in boldface letters. The arrow under an event points to the action taken when the event occurs. The main events triggering state transitions are *recv-done* (a client-initiated transfer is done), *send-done* (a server-initiated transfer is done) and *bio-done* (a block I/O request from disk is done). Important design characteristics of the DAFS server in the current implementation are:

1. The server uses the buffer cache interface to do disk I/O (i.e. *bread()*, *bwrite()*, etc.). This is a zero-copy interface that can be used to lock buffers (pages and their mappings) for the duration of an RDMA transfer. RDMA transfers take place directly from or to the buffer cache.

2. RDMA transfers are initiated in the context of RPC handlers but proceed asynchronously. It is possible that an RDMA completes long after the RPC that initiated it has exited. Buffers involved in RDMA need to remain locked for the duration of the transfer. RDMA completion event handlers unlock those buffers and send an RPC reply if needed.

3. The kernel buffer cache manager is modified to register/deregister buffer mappings with the NIC on-the-fly, as physical pages are added or removed from buffers. This ensures that the NIC never takes translation miss faults and pages are wired only for the duration of the RDMA.

Each of the network and disk events has a corresponding event handler that executes in the context of a kernel thread.

1. *recv-done* is raised by the NIC and triggers processing of an incoming RPC request. For example, in the case of read or write operations the handler may initiate block I/O with the file system using *bread()*. After data is locked in buffers (hereafter referred to as *bp*'s) in the buffer cache, RDMA is initiated and the *bp*'s remain locked for the duration of the transfer.

2. *send-done* is raised by the NIC to notify of completion of a server-initiated (read or write) RDMA operation. The handler releases locks (using *brelse()*) on *bp*'s involved in the transfer and sends out an RPC response.

3. *bio-done* is raised by the disk controller and wakes up a thread that was blocking on disk I/O previously initiated by *bread()*. This event is currently handled by the kernel buffer cache manager in *biodone()*.

The server forks multiple threads to create concurrency in order to deal with blocking conditions. Kernel threads are created using an internal *rfork()* operation. One of the threads is responsible for listening for new transport connections while the rest are workers involved in data transfer. All transport connections are bound to the same completion group. Message arrivals on any transport connection generate *recv-done* interrupts which are routed to a single interrupt handler associated with the completion group. When the handler is invoked, it queues the incoming RPC request, notes the transport that was the source of the interrupt, and wakes up a worker thread to start processing. After parsing the request, a thread locks the necessary file pages in the buffer cache using *bread()*, prepares the RDMA descriptors and issues the RDMA operations. The RPC does not wait for RDMA completion. A later *send-done* interrupt (or successfull poll) on a completed RDMA transfer descriptor starts clean up and release of resources that the transfer was tying up (i.e. *bp* locks held on file buffers for the duration of the transfer), and sends out the RPC response. Threads blocking on those resources are awakened.

Event-driven design requires that event handlers be quick and not block between events. Our current server design deviates from this requirement due to the possibility of blocking under certain conditions:

1. Need to wait on disk I/O initiated by *bread()*. It is possible to avoid using the blocking *bread()* interface by initiating asynchronous I/O with

the disk using *getblk()* followed by *strategy()*. We opted against this solution in our early design since disk event delivery is currently disjoint from network event delivery, complicating event handling. Integrating network and disk I/O event delivery in the kernel is discussed in Sections 4.1 and 4.2.

2. Locking a *bp* twice by the same kernel thread or releasing a *bp* from a thread other than the lock owner causes a kernel panic (Section 4.4). Solutions are to (a) defer any request processing by a thread while past transfers it issued are still in progress, to ensure that a *bp* is always released by the lock owner and a thread never locks the same *bp* twice, or (b) modify the buffer cache so that these conditions no longer cause a kernel panic. To avoid wider kernel changes in the current implementation, we do (a). (b) is addressed in Section 4.4.

An important concern when designing an RDMA-based server is to minimize response latency for short transfers and maximize throughput for long transfers. In the current design, notification of incoming messages is done via interrupts and notification of server-initiated transfer completions via polling. Short transfers using RDMA are expected to complete within the context of their RPC request. In this way, the RPC response immediately follows RDMA completion, minimizing latency. Throughput is maximized for longer transfers by pipelining them as their RDMA operations can be concurrently progressing.

The low cost of DAFS RPC, the efficient event notification and delivery mechanism, and the absence of copies due to RDMA help towards low response latency. Maximum throughput is achievable even for small block sizes (as shown in Section 5) assuming the client is throwing requests at the server at a sufficiently high rate (i.e. doing prefetching using asynchronous I/O). The DAFS kernel server presently runs over the Emulex cLAN [8] and GN 9000 VI/IP [9] transports and is currently being ported to Mellanox InfiniBand [10].

3.2 Optimistic DAFS

In DAFS *direct* read and write operations, the client always uses an RPC to communicate the file access request along with memory references to client buffers that will be the source or target of a server-issued RDMA transfer. The cost associated with always having to do a file access RPC is manifested as unnecessarily high latency for small accesses from server memory. A way to reduce this latency is to allow clients to access the server file and VM cache directly rather than having to go each time through the server *vnode* interface via a file access RPC.

Optimistic DAFS [14] improves on the existing DAFS specification by reducing the number of file access RPC operations needed to initiate file I/O and replacing them with memory accesses using client-issued RDMA. Memory references to server buffers are given out to clients or other servers that maintain cache directories, and they are allowed to use those references to directly issue RDMA operations with server memory. To build cache directories, the server returns to the client a description of buffer locations in its VM cache (we assume a unified VM and file cache, as in FreeBSD). These buffer descriptions are returned either as a response to specific queries (i.e. client asks: *"give me the locations of all your resident pages associated with file foo"*), or piggybacked in the response to a read or write request (i.e. 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"*).

In Optimistic DAFS, clients use remote memory references found in their cache directories but accesses succeed only when directory entries have not become stale, for example as a result of actions of the server pageout daemon. There is no explicit notification to invalidate remote memory references previously given out on the network. Instead, remote memory access exceptions [14] thrown by the target NIC and caught by the initiator NIC can be used to discover invalid references and switch to the slower access path using file access RPC.

Maintaining the NIC memory management unit in the case where RDMA can be remotely initiated by a client at any time is tricky and needs special NIC and OS support. Section 4.3 describes the design of our forthcoming implementation that views the NIC as another processor in an asymmetric multiprocessor system and is based on the following design choices:

1. To make sure that exported pages have valid NIC mappings for as long as they are resident in physical memory and that these mappings are invalidated when pages are swaped to disk, paging activity on-the-fly adds or invalidates NIC mappings.

2. Being able to initiate DMA to and from main

memory, the NIC (or the driver, in the absence of NIC support) has to synchronize and integrate with the VM system. To do that, it has to be able to manipulate *lock*, *reference*, and *dirty* bits of `vm_page`s in main memory.

3. To manage NIC mappings in servers with enormous physical memory sizes, the NIC address translation table is viewed as a cache of translations (i.e. a TLB). Translation misses are handled by the NIC (or the driver, in the absence of NIC support) and require access to page tables in main memory.

Previous research [21, 24] has looked at memory management of network interfaces but has not focused on kernel modifications or virtual memory system support. In Section 4.3 we address such support for the FreeBSD VM system. Finally, Optimistic DAFS requires maintainance of a directory on file clients (in user-space) and on other servers (in the kernel).

4 Kernel Support for DAFS Servers

Special capabilities and requirements of networking transports used by DAFS servers expose a number of kernel design and structure issues. In general, a DAFS file server needs to be able to

1. Do asynchronous file I/O
2. Integrate network and disk I/O event delivery
3. Lock file buffers while RDMA is in progress
4. Avoid memory copies

In what follows we describe our proposals for new kernel support in the FreeBSD kernel. Work on implementing these proposals is currently in progress. Section 4.1 argues for kernel asynchronous file I/O interfaces presently lacking in FreeBSD, and integrating network and file event notification and delivery. Section 4.2 presents a *vnode* interface designed to address these needs. Section 4.3 examines kernel support for memory management of the asymmetric multiprocessor system that consists of the NIC and the host CPU. Finally, Section 4.4 argues for modifications to buffer cache locking and Section 4.5 outlines device driver requirements of memory-to-memory NIC.

4.1 Event-Driven Design Support

An area of considerable interest in recent years [2, 18, 19] has been that of event-driven application design. Event-driven servers avoid much of the overhead associated with multithreaded designs but require truly asynchronous interfaces coupled with efficient event notification and delivery mechanisms integrating all types of events. The DAFS server requires such support in the kernel.

FreeBSD presently lacks an internal asynchronous interface to the buffer cache although it does provide an asynchronous I/O (AIO) system call API. AIO is an implementation of the POSIX 1003.1B standard and can be found in other systems such as Solaris [16]. It is implemented as a multithreaded interface to the filesystem over regular files, or using asynchronous device I/O over device-special files. The latter mechanism is more efficient as it avoids overhead associated with multithreading but can only be used for applications using raw device access, such as relational databases.

To integrate disk events with memory-to-memory NIC event handling, a generalization of the network-specific *completion group* abstraction is needed. Events from network and disk sources can be uniformly handled using *kqueue* [13], a recently introduced FreeBSD kernel abstraction for scalable event handling. *Kqueue* can be used to aggregate event sources such as a) completed network I/O, b) completed disk I/O, and c) process synchronization signals. A *kqueue* structure can handle all event types a DAFS server is interested in. Posting asynchronous operations requires registering *kevent*s with the *kqueue*. Network and disk event handlers notify the server *kqueue* using appropriate event filters (i.e. EVFILT_KAIO for disk, and EVFILT_RDMA for network events). Notification can either be via polling (using *kqueue_scan()*) or, with a minor addition to the *kqueue* implementation, via a direct or delayed kernel upcall to a handler routine.

4.2 Vnode Interface Support

Vnode/VFS is a kernel interface that separates generic filesystem operations from specific filesystem implementations [20]. It was conceived to provide applications with transparent access to kernel filesystems, including network filesystem clients such as NFS. The *vnode*/VFS interface consists of two parts: VFS defines the operations that can be done on a filesystem. *Vnode* defines the operations that can be done on a file within a filesys-

Table 1: **Vnode ops (Sandberg et al.[20]).**

Vnode operation	**Description**
VOP_ACCESS	Check access permission
VOP_BMAP	Map block number
VOP_BREAD	Read a block
VOP_BRELSE	Release a block buffer
VOP_CLOSE	Mark file closed
VOP_CREATE	Create a file
VOP_FSYNC	Flush dirty blocks of a file
VOP_GETATTR	Return file attributes
VOP_INACTIVE	Mark *vnode* inactive
VOP_IOCTL	Do I/O control operation
VOP_LINK	Link to a file
VOP_LOOKUP	Lookup file name
VOP_MKDIR	Create a directory
VOP_OPEN	Mark file open
VOP_RDWR	Read or write a file
VOP_REMOVE	Remove a file
VOP_READLINK	Read symbolic link
VOP_RENAME	Rename a file
VOP_READDIR	Read directory entries
VOP_RMDIR	Remove directory
VOP_STRATEGY	Read/write fs blocks
VOP_SYMLINK	Create symbolic link
VOP_SELECT	Do select
VOP_SETATTR	Set file attributes
VOP_GETPAGES	Read and map pages in VM
VOP_PUTPAGES	Write mapped pages to disk

tem. Table 1 lists the *vnode* operations originally defined [20] to support NFS along with a number of local filesystems as well as later additions introduced into BSD systems with a unified file and VM cache to transfer data directly between the VM cache and the disk.

Existing *vnode* I/O interfaces are all synchronous. VOP_READ and VOP_WRITE take as an argument a `struct uio` buffer description and have copy semantics. VOP_GETPAGES and VOP_PUTPAGES are zero-copy interfaces transfering data directly between the VM cache and the disk. VM pages returned from VOP_GETPAGES need to be explicitly wired in physical memory to be used for device I/O. An interface for staging I/O should be designed to return buffers in a locked state. We believe that a *vnode* interface modeled after the low-level buffer cache interface with new support for asynchronous operation naturally fits the requirements of a DAFS server as outlined earlier. Such an asynchronous interface is easier to implement than an asynchronous version of VOP_GETPAGES, VOP_PUTPAGES, while being functionally equivalent to it in FreeBSD's unified VM and buffer cache.

Central to this new interface (summarized in Table 2) is a VOP_AREAD call which can be used to issue disk read requests and return without blocking. VOP_AREAD internally uses a new *aread()* buffer cache interface (described below) integrated with the *kqueue* mechanism. It takes as one of its arguments an asynchronous I/O control block (*kaiocb*) used to keep track of progress of the request.

```
aread(struct vnode *vp, struct kaiocb *cb)
{
    derive block I/O request from cb;
    bp = getblk(vp, block request);
    if (bp not found in the buffer cache) {
        register kevent using EVFILT_KAIO;
        register kaio_biodone handler with bp;
        VOP_STRATEGY(vp, bp);
    }
}
```

On completion of a request issued by *aread()*, the data is in a *bp*, in a locked state, and *kaio_biodone()* is called to deliver the event:

```
kaio_biodone(struct buf *bp)
{
    get kaiocb from bp;
    deliver event to knote in klist of kaiocb;
}
```

To unlock buffers and update filesystem state if necessary, VOP_BRELSE is used. Local filesystems would implement the interface of Table 2 in order to be efficiently exported by a DAFS server. For lack of this or another suitable interface, a local filesystem could always be exported by a DAFS server using existing interfaces, albeit with higher overhead mainly due to multithreading.

Network event delivery can be integrated with that of disk I/O as described earlier through the *kevent* mechanism. Each time an RDMA descriptor is issued, a *kevent* is registered using the EVFILT_RDMA filter and recorded in the completion group (CG) structure. Completion group handlers need to deal with *kqueue* event delivery:

```
send_event(CG *cq, Transport *vi)
{
    deliver event to knote in klist of CG;
}
```

Table 2: **Vnode Interface to Buffer Cache.**

Vnode operation	Description
VOP_BREAD	Lock all buffers needed for I/O; read from *vp*.
VOP_AREAD	Lock all buffers needed for I/O; read from *vp*; don't block .
VOP_BDWRITE	Mark dirty entries; delayed write to *vp*; update state if requested.
VOP_BWRITE	Block writing to *vp*; update state if requested.
VOP_BAWRITE	Mark dirty entries; async write to *vp*; update state if requested.
VOP_BRELSE	Unlock buffers; update file state if requested.

The DAFS server is notified of new events by periodically polling the *kqueue*. Alternatively, a common handler is invoked each time a network or disk event occurs.

We illustrate the use of the proposed *vnode* interface to the buffer cache by breaking down and describing the steps in *read* and *write* operations implemented by a DAFS server. For comparison with existing interfaces, we describe the same steps implemented by NFS. Without loss of generality we assume an FFS underlying filesystem at the server.

Read. With DAFS, a client (direct) read request carries the remote memory address of the client buffers. The DAFS server issues a VOP_AREAD to read and lock all necessary file blocks in the buffer cache. VOP_AREAD starts disk operations and returns without blocking, after registering with *kqueue*. When pages are resident and locked and the server notified via *kqueue*, it issues RDMA Write operations to client memory for all requested file blocks. When the transfers are done, the server does VOP_BRELSE to unlock the file buffer cache blocks.

With NFS, on a client read operation the server issues a VOP_READ to the underlying filesystem with a `uio` parameter pointing to a gather/scatter list of mbufs that will eventually form the response to the read request RPC. In the FFS implementation of VOP_READ and without applying any optimizations, a loop reads and locks file blocks into the buffer cache using *bread()*, subsequently copying the data into the mbufs pointed to by `uio`. For page-aligned, page-sized buffers, page-flipping techniques can be applied to save the copy into the mbufs.

Write. With DAFS, a client (direct) write request carries only client memory addresses of data buffers. The DAFS server uses VOP_AREAD to read and lock in the buffer cache all necessary file blocks. When pages are resident and locked, it issues RDMA Read requests to fetch the data from the client buffers directly into the buffer cache blocks. When the transfer is done, the server uses one of VOP_BWRITE, VOP_BDWRITE, VOP_BAWRITE, depending on whether this is a stable write request or not, to issue a disk write I/O and unlock the buffers. Additionally, a metadata update is effected if requested.

With NFS, a client write operation carries the data to be written inline with the RPC request. The NFS server prepares a `uio` with a gather/scatter list of all the data mbufs and calls VOP_WRITE. Apart from the `uio` parameter that describes the transfer, an `ioflags` parameter is passed signifying whether the write to disk should happen synchronously. With NFS version 2 all writes and metadata updates are synchronous. NFS versions 3 and 4 allow asynchronous writes. In the FFS implementation of VOP_WRITE, a loop reads and locks the file blocks to be written into the buffer cache using *bread()*, copies into them the data described by `uio`, then uses one of *bwrite* (synchronous), *bdwrite* (delayed), or *bawrite* (asynchronous) writes depending on whether this is a stable write request (see `ioflags`) or not. Finally, a metadata update is effected if requested.

An interesting note on the ability of the DAFS server to implement file writes using RDMA Read (instead of client-initiated RPC or RDMA Write) is that this enables it to read data from client memory no faster than dirty buffers can be written to disk. This *bandwidth matching* capability becomes very important in multi-gigabit networks when network bandwidth is often greater than disk bandwidth.

4.3 VM System Support

In systems deriving from 4.4BSD [17], maintaing virtual/physical address mappings and page access rights used by the main CPU memory-management hardware is done by the machine-dependent *physical mapping (pmap)* module. Low level machine-independent kernel code such as the

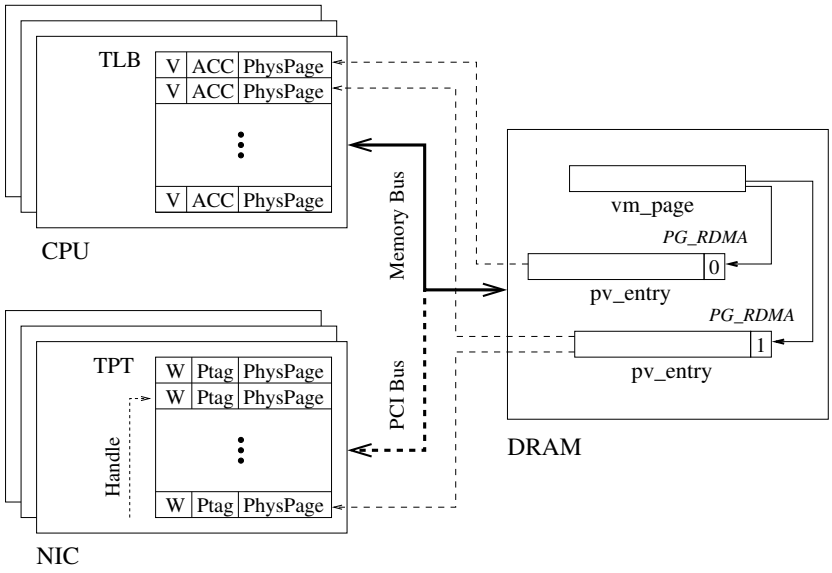

Figure 2: **CPU and NIC Memory Management Hardware.**

buffer cache, kernel malloc and the rest of the VM system are using *pmap* to add or remove address mappings and alter page access rights. Symmetric multiprocessor (SMP) systems sharing main memory can use a single *pmap* module as long as translation lookaside buffers (TLB) on each CPU are kept consistent. *Pmap* operations apply to page tables shared by all CPU. TLB miss exceptions thrown by a CPU result in a lookup for mappings in the shared page tables. Invalidations of mappings are applied to all CPU.

Memory-to-memory NIC store virtual-to-physical address translations and access rights for all user and kernel memory regions directly addressable and accessible by the NIC. Figure 2 shows a system combining both CPU and NIC memory management hardware: Main CPU use their on-chip translation lookaside buffer (TLB) to translate virtual to physical addresses. A typical TLB page entry includes a number of bits such as V and ACC signifying whether the page translation is valid, and what the access rights to the page are, along with the physical page number. A miss on a TLB lookup requires a page table lookup in main memory.

NIC on the PCI (or other I/O) bus have their own translation and protection (TPT) [5] tables. Each entry in the TPT includes bits enabling RDMA Read or Write (i.e. the W bit in the diagram) operations on the page, the physical page number, and a Ptag value identifying the process that owns the pages (or the kernel). Whereas the TLB is a high-speed associative memory, the TPT is usually implemented as a DRAM module on the NIC board. To accelerate lookups on the TPT, remote memory access requests carry a Handle index that helps the NIC find the right TPT entry.

This section focuses on operating system support to integrate NIC memory management units in the FreeBSD VM system. The main benefits of this integration are

1. VM pages exported for RDMA are wired in physical memory only for as long as RDMA transfers from or to them are in progress, resulting in better utilization of physical memory.

2. The entire VM cache (i.e. potentially all file VM objects) can be safely exported in the face of paging activity.

We consider the NIC as a processor (with special I/O capabilities [12]) in the asymmetric multiprocessor system of Figure 2 and allow sharing of kernel VM structures in main memory between NIC and main processors. We assume that access to VM structures on behalf of the NIC is done by the CPU, executing driver handlers in response to interrupts. Direct access by the NIC to VM structures in main memory is considered later in this section.

In FreeBSD (and other systems deriving from 4.4BSD), a physical page is represented by a `vm_page` structure and an address space by a `vm_map` structure. A page may be mapped on one or

Table 3: **Low-level NIC VM primitives.**

Function	Description
tpt_init()	Initialize
tpt_enter()	Insert mapping
tpt_remove()	Remove mapping
tpt_protect()	Protect mapping

Table 4: **VM interface used by the NIC.**

Function	Description
vm_page_io_start()	Lock page
vm_page_io_finish()	Unlock page
vm_nic_fault()	Handle NIC fault
vm_page_reference()	Reference page
vm_page_dirty()	Dirty page

more `vm_map`s with each mapping represented by a `pv_entry` structure. Figure 2 shows a `vm_page` with two associated `pv_entry` structures in main memory.

In our design, the VM system maintains the NIC MMU via the OS-NIC interface. The NIC accesses VM structures via the NIC-OS interface.

OS-NIC Interface. The OS needs to interact with the NIC to add, remove and modify VM mappings stored on its TPT. A mapping of a VM page expected to be used in RDMA has to be registered with the NIC. Registering the mapping with the NIC happens in *pmap*, right after the CPU mapping with a `vm_map` is established. The NIC exports low-level VM primitives (Table 3) for use by *pmap* to add, remove and modify TPT entries. NIC mappings may later be deregistered (when the original mapping is removed/invalidated), or have their protection changed.

To keep an account of VM mappings that have been registered with the NIC, we add a PG_RDMA bit in the `pv_entry` structure to be set whenever a `pv_entry` has a NIC as well as a CPU mapping. In Figure 2, the `pv_entry` with the PG_RDMA bit set has both a CPU and a NIC mapping. In all *pmap* operations on VM pages, the *pmap* module interacts with the NIC only if the PG_RDMA flag is set on the `pv_entry`.

Higher-level code can trigger registration of a virtual memory region with the NIC by propagating appropriate flags from higher to lower-level interfaces and eventually to the *pmap*. For example, the DAFS server sets an IO_RDMA bit in the `ioflags` parameter of the *vnode* interface (Table 2) when planning to use the buffer for RDMA. This eventually translates into a VM_RDMA flag in the *pmap_enter()* interface that results in mappings being registered with the NIC.

A problem with invalidating `pv_entry` mappings that have also been registered with the NIC is that NIC invalidations may need to be delayed for as long as RDMA transfers using the mappings are in progress. *Pmap_remove_all()* is complicated by this fact as (for atomicity) it has to try to remove `pv_entry` structures with NIC mappings first and may eventually fail if NIC invalidations are not possible within a reasonable amount of time.

Another problem is with VM system policies that are often based on architectural assumptions that do not hold with NIC characteristics. For example, the FreeBSD VM system unmaps pages from process page tables when moving them from an active to inactive or cached state. This is because the VM system is willing to take a reasonable number of reactivation faults to determine how active a page actually is, based on the assumption that reactivation faults are relatively inexpensive [7]. NIC reactivation faults are significantly more expensive compared to CPU faults due to lack of integration between the NIC and the host memory system. To reduce that cost, it would make sense to apply the deactivation policy only to CPU mappings, leaving NIC mappings intact for as long as VM pages are memory resident. However, full integration of the NIC memory management unit into the VM system argues for this policy to be equally applied to NIC page accesses.

NIC-OS Interface. The NIC initiates interaction with the VM system in the following occasions using the interface of Table 4.

1. The NIC can be the initiator of DMA from or to VM pages in main memory for incoming RDMA Read or Write requests, and thus has to be able to lock (busy) VM pages for the duration of the transfer. The interface is similar to the kernel interface used to busy pages during pagein or pageout activity.

2. On a translation miss, the NIC needs to do a page table lookup for the missed virtual ad-

dress. A new translation is loaded in the NIC TPT if the page is found to be resident in memory. If not, pagein may be initiated by the miss handler but the NIC may choose not to wait for it and report an RDMA exception instead.

3. The NIC updates *reference*, *dirty* bits whenever it accesses or writes to VM pages.

All this handling can be done by the host CPU in response to interrupts thrown by the NIC. For efficient access to `vm_page` bits in main memory without interrupting the host CPU, the NIC should share definitions of VM structures and store direct references to `vm_page`s. These references could either be physical (bus) addresses, or virtual addresses that are translated using the NIC TPT. In cases where a simple bit flip on a `vm_page` is needed, the NIC should be able to do that by a direct atomic memory access. Complicated page table lookups (i.e. in translation miss handling) are better handled by the host CPU.

4.4 Buffer Cache Locking

In our first implementation of a DAFS server, we chose to directly use the buffer cache for block I/O. In an RDMA-based data transfer, the server sets up the RDMA transfer in the context of the requesting RPC. Once issued, the RDMA proceeds asynchronously to the RPC. The latter does not wait for RDMA completion. To serialize concurrent access to shared files in the face of asynchrony, the *vnode* (*vp*) of a file needs to be locked for the duration of the RPC. However, the data buffers (*bp*'s) transfered need to be locked for the full duration of the RDMA. Locking the *vp* (i.e. the entire file) for the duration of the RDMA would also work but would limit performace in case of sharing since requests for non-overlapping regions of a file would have to serialize. Our decision to lock at a finer granularity than the *vp* for the duration of a transfer conflicts with current FreeBSD buffer cache locking assumptions:

1. Locking a buffer in the cache requires a process to acquire an exclusive lock on that buffer. A buffer lock can only be released by the same process that locked it or by the kernel.

2. Before an asynchronous disk I/O (i.e. an asynchronous write, or readahead), lock ownership has to be transfered to the kernel so that the block can later be released by the kernel (in *biodone()*).

A multithreaded event-driven kernel server that directly uses the buffer cache and does event processing in kernel process context faces problems in the following circumstances:

1. When a thread tries to lock a buffer it is already locking (because a transfer is in progress on that buffer) expecting to block until that lock is released by some other thread.

2. When a buffer is released from a different thread than the one that locked it.

Transfering lock ownership to the kernel during asynchronous network I/O does not help since lock release is done by some kernel process (whichever happens to have polled for that particular event) rather than by the kernel itself. The solution presently used is for the kernel process that issued an RDMA operation to wait until the transfer is done in order to release the lock. This also prohibits that process from trying to lock the same buffer again, thus causing a deadlock panic. A better solution is to enable recursive locking and allow lock release by any of the server threads.

4.5 Device Driver Support

Memory-to-memory network adapters virtualize the NIC hardware and are directly accessible from user space. One such example is VI [5] where the NIC implements a number of VI contexts. Each VI is the equivalent of a socket in traditional network protocols, except that a VI is directly supported by the NIC hardware and usually has a memory-mapped rather than a system call interface. The requirement to create multiple logical instances of a device, each with its own private state (separate from the usual device softcopy state) and to map those devices in user address spaces requires new support from BSD kernels.

Network driver model. Network drivers in BSD systems are traditionally accessed through sockets and do not appear in the filesystem name space (i.e. under /dev). User-level libraries for memory-to-memory network transports require these devices to be opened and closed multiple times with each opened instance appearing as a separate logical device maintaining private state, and be memory-mapped. FreeBSD until recently provided rudimentary support for this through the *devfs* file system which is being abandoned. *Devfs* allows logical instances of a device to be dynamically created but

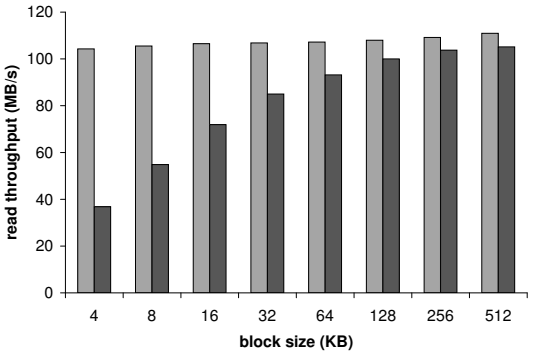

Figure 3: **DAFS Read Throughput.** From left to right: asynchronous API; synchronous API.

still associates a single *vnode* with that device. Our current drivers rely on a hack to associate a separate *vnode* and store private state with each logical instance of a device.

Device driver models under other operating systems have different ways for logical device instances to maintain private state. Linux associates a *vnode* with each opened instance of a device file, and Solaris keeps private per-instance state via DDI [1].

5 Performance

We evaluate performance of the prototype DAFS server with a synthetic benchmark that involves a client fetching random blocks from warm server cache (no disk I/O). We measured performance for both the synchronous and asynchronous API. In the asynchronous case, the client issues read I/O maintaining up to 64 outstanding requests at any time. The server was configured to run with 64 kernel threads. In the synchronous case, the client was blocking waiting for completion on each request.

The experiment runs over two Pentium III 800 MHz systems with the ServerWorks LE chipset and 1GB RAM, connected via 1.25 Gb/s cLAN [8] VI cards on 64-bit/66MHz PCI over a cLAN switch. The network has been measured to yield 113 MB/s with a VI one-way throughput benchmark (using 32K packets and polling) provided by Giganet. We measured the effect of the block size used in read requests varying it from 4KB to 512KB and report results in Figure 3. We see that even for small block sizes, performance using the asynchronous interface is close to wire throughput by being able to pipeline server responses. Low DAFS overhead and the absence of copying reduces the memory bottleneck that would otherwise be a limiting factor. Performance using the synchronous interface converges to almost wire throughput for block sizes of about 128KB when the per-I/O overhead is fully amortized.

6 Conclusions

This paper focused on the kernel issues involved in building Direct Access File System servers. A range of issues was addressed drawing from our experience in building such a kernel server for FreeBSD. We described the current server structure using existing interfaces with minor kernel modifications. Performance results show that our current DAFS server prototype implementation can offer high performance file service for memory workloads over a 1.25 Gb/s network.

A problem with existing blocking kernel interfaces is that DAFS and other kernel servers using them experience the overhead of having to associate a process context with each I/O request. This overhead is expected to become more pronounced in multi-gigabit networks. Our proposed additions to the *vnode* interface offer support for asynchronous file I/O with integrated network and disk event notification and delivery.

We presented design possibilities for integrating a programmable RDMA-capable NIC with the FreeBSD VM system. This support will allow a DAFS server to export its entire VM cache over the network in the face of client-initiated RDMA operations and server paging activity. We are currently planning an implementation that embodies such a design and can be used to support Optimistic DAFS.

Finally, we have found that the existing BSD network driver model is inadequate to support the needs of memory-to-memory NIC devices and a new model is needed.

7 Acknowledgments

The author would like to thank Donn Seeley, the shepherd for this paper, and Alexandra Fedorova and Salimah Addetia for helpful feedback.

8 Software Status and Availability

The current prototype runs as a kernel loadable module for FreeBSD 4.3-RELEASE and implements a large subset of the DAFS specification including all basic file access, performance enhancements, locking, and client caching support opera-

tions. Besides the server, we have ported device drivers to FreeBSD for a number of VI NIC, including the Giganet/Emulex cLAN 1000 and GN 9000 VI/IP which use ATM and gigabit Ethernet respectively as their link layer transport. We anticipate to measure the performance benefits of Optimistic DAFS as soon as the transport support is implemented. Source code for the server and associated FreeBSD kernel patches can be obtained from `http://www.eecs.harvard.edu/vino/fs-perf/dafs`.

References

[1] *Writing Device Drivers*. Sun Microsystems, Inc., 2000.

[2] G. Banga et al. Better Operating System Features for Faster Network Servers. In *Proceedings of the Workshop on Internet Server Performance (WISP)*, June 1998.

[3] A. Benner. *Fibre Channel: Gigabit I/O and Communications for Computer Networks*. McGraw-Hill, 1996.

[4] M. Blumrich et al. A Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142–153, April 1994.

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

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

[7] M. Dillon. Design Elements of the FreeBSD VM System. `http://www.daemonnews.org/200001/freebsd_vm.html`.

[8] Emulex/Giganet Inc. cLAN 1000 VI adapter. `http://wwwip.emulex.com`.

[9] Emulex/Giganet Inc. GN 9000 VI adapter. `http://wwwip.emulex.com`.

[10] Mellanox Inc. Sleek 1X InfiniBand Channel Adapter. `http://www.mellanox.com`.

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

[12] Intel Inc. I/O Processors. `http://developer.intel.com/design/iio/80310.htm`.

[13] J. Lemon. Kqueue: A Generic and Scalable Event Notification Facility. In *2001 USENIX Technical Conference - FREENIX Track*, June 2001.

[14] K. Magoutis. The Optimistic Direct Access File System. *Submitted for publication at the Workshop on Novel Uses of System Area Networks (SAN 2001), Cambridge, MA*, February 2002.

[15] E.P. Markatos and M. G. H. Katevenis. Telegraphos: High-Performance Networking for Parallel Processing on Workstation Clusters. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture, San Jose, CA*, pages 144–153, February 1996.

[16] J. Mauro and R. McDougall. *Solaris Internals: Core Kernel Architecture*. Prentice Hall, 2000.

[17] M. Kirk McKusick et al. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996.

[18] J. Ousterhout. Why Threads are a Bad Idea (For Most Purposes). Invited Talk at the 1996 USENIX Technical Conference, January 1996.

[19] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *1999 USENIX Technical Conference*, June 1999.

[20] R. Sandberg et al. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Technical Conference, Portland, OR*, pages 119–130, June 1985.

[21] 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.

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

[23] 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.

[24] 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.