The Case Against User-Level Networking

Kostas Magoutis
IBM T. J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532

Margo Seltzer
Harvard University
33 Oxford Street
Cambridge, MA 02138

Eran Gabber
Google, Inc.
1440 Broadway, 21st Floor
New York, NY 10018

Abstract— **Extensive research on system support for enabling I/O-intensive applications to achieve performance close to the limits imposed by the hardware suggests two main approaches: Low overhead I/O protocols and the flexibility to customize I/O policies to the needs of applications. One way to achieve both is by supporting user-level access to I/O devices, enabling user-level implementations of I/O protocols.** *User-level networking* **is an example of this approach, specific to network interface controllers (NICs). In this paper, we argue that the real key to high-performance in I/O-intensive applications is user-level file caching and user-level network buffering, both of which can be achieved without user-level access to NICs.**

Avoiding the need to support user-level networking carries two important benefits for overall system design: First, a NIC exporting a privileged kernel interface is simpler to design and implement than one exporting a user-level interface. Second, the kernel is re-instated as a global system resource controller and arbitrator. We develop an analytical model of network storage applications and use it to show that their performance is not affected by the use of a kernel-based API to NICs.

I. Introduction

The need to reduce networking overhead in system-area networks in the early 1990's motivated a flurry of research on user-level networking protocols. Projects such as SHRIMP [2], Hamlyn [4], and U-Net [36] proposed user-level access to a network interface controller (NIC) as an approach that offered two primary benefits: First, it enabled host-based implementations of new, lightweight networking protocols with lower overhead compared to kernel-based TCP/IP protocol stacks. Second, for applications requiring use of the TCP/IP protocol stack, there is a potential for application-specific customization of user-level libraries. In recent years, the need for scalable and more manageable services, such as HTTP servers and network storage systems, leads to increased amounts of data transferred through the network. This trend links overall I/O performance to the efficiency of the network subsystem and makes network interface controllers (NICs) a key system I/O device. The large-scale deployment of high-speed (1 Gb/s and soon 10 Gb/s) Ethernet networks stimulated interest in the design of systems that offload TCP/IP to a new generation of NICs [27] and can transfer data directly between the network and application buffers. Many such TCP-offload NICs are currently being designed to export user-level interfaces to host applications.

In this paper, we show that a user-level interface to NICs offering transport offload is not necessary. A kernel host interface to the NIC in combination with NIC support for *remote direct data placement* (RDDP) [16] enables an operating system (OS) structure that supports user-level file caching [25] and user-level network buffering [3]. User-level caching offers full control over I/O policies, an essential requirement for resource-intensive applications, such as databases. This paper describes such an OS structure that blends ideas from traditional and novel/radical OS architectures: In accordance with standard OS principles, resource abstractions, such as files and network sockets, and global system policies, are implemented in the kernel. However, similarly in spirit to more radical system designs, such as the Exokernel [20], full control over application-specific I/O policies is possible with caching implemented in user space. This caching, however, is performed over the file or network socket abstractions rather than over the raw hardware. The key advantages of this approach over Exokernel's are improved security and safety due to a larger common kernel code-base, support for global resource policies, and improved portability.

In summary, the main arguments presented in this paper are: First, NICs exporting a user-level host interface pose implementation challenges not present in NICs exporting an interface to a trusted host entity such as the kernel. Second, with the transport protocol offloaded to the NIC, the key to lowering networking overhead is remote direct data placement. RDDP enables a new OS structure that supports user-level file caching and user-level network buffering. The performance of a user-level API to file and network services should be similar to the performace of a kernel-based API, since the difference is only in user-kernel boundary crossing, which is not a dominant cost. Third, kernel involvement is necessary in order to enforce global policies. Systems that advocate implementing all system policies in user-space libraries [20] do not offer a solution to the problem of implementing global policies without involving a privileged server. In addition, security and safety are other benefits of our proposed design: For example, common kernel-based networking code may be more robust against attacks than a multitude of user-level networking stacks. A benefit of user-level networking is better portability due to bypassing the OS. However, a kernel interface that is simple to implement should be rapidly incorporated into most mainstream operating systems.

The layout of this paper is as follows: In Section III we compare the benefits of user-level networking to those of a kernel interface to a NIC offering transport offload and RDDP. In Section IV we describe an OS structure that takes advantage of RDDP to support user-level file caching and user-level network buffering. In Section V we develop an analytical model of network storage applications to study the effect of the user-kernel protection boundary crossing on performance. Our motivation to use an analytical model backed by experimental measurements of key parameters, instead of a full experimental system comparison, is based on the fact that the latter depends significantly on the quality of the implementations and focuses on a particular point of the design space. An analytical model enables the examination of the entire design space, provides significant qualitative results, and points to the key parameters affecting system performance. Earlier studies of operating system structure, such as the classical work by Lauer and Needham [21], have also used analytical modeling to reason about system performance. Finally, in Section VI we present experimental results.

II. Related Work

Support for user-level networking was first offered in supercomputers such as the Thinking Machines CM-5 [22], Meiko CS-2 [17], and IBM SP-2 [35], and later introduced in commodity networks by U-Net [36], Application Device Channels (ADC) [13], Hamlyn [4], SHRIMP [2], DART [31] and others. User-level networking was recently standardized with the Virtual Interface [9] (VI) and InfiniBand [18] architectures. In addition to the ability for user-level access to the NIC, which is the defining feature of user-level NICs, most of these projects also advocated a new host programming interface to the NIC. This programming interface is based on a queue-pair abstraction and requires pre-posting of receive buffers [3, 9]. It is important to note that this new programming interface is *not* a defining feature of user-level NICs and can be implemented without special NIC support [5]. In contrast, user-level access to the NIC necessarily requires special NIC support, increasing the complexity of NIC design, as explained in Section III-A.

A key motivation for user-level networking is the reduction of I/O overhead. This reduction, however, is the result of three *orthogonal* factors: (a) new lightweight network protocols, (b) new host programming interfaces to the NIC offering support for remote direct data placement, and (c) avoidance of the user-kernel protection domain crossing to perform I/O. Recent experimental evidence points to (a) and (b) as the key factors affecting performance. For example, Zhou and her colleagues [37] found that a kernel-based implementation of a storage client offering block access over a VI network interface performs comparably to a user-level implementation of the same client in microbenchmarks and TPC-C database benchmark configurations. This evidence supports our argument that it is primarily the API to the NIC rather than user-level access to it that are key to achieving high performance.

Frequent crossing of the user-kernel protection boundary is often associated with high cache and TLB miss overheads. For example, early microkernels enforcing protection using hardware address spaces were observed to exhibit such high overheads [7]. This behavior, however, was mainly attributed to their larger memory footprints, which is an implementation artifact, rather than to the use of hardware address space protection. Research on systems such as L4 [23] and Pebble [14] showed that the overhead of switching protection domains could be as low as 150 machine cycles, allowing for efficient modular designs using hardware address spaces. In addition, we believe that increased cache and TLB sizes in modern CPUs will help to alleviate cache and TLB miss overheads.

Another significant factor contributing to network overhead is hardware interrupts, as shown in a recent study of network protocol performance over high speed networks [6]. Detecting I/O completions by polling is an alternative that is offered by many user-level networking protocols [9, 36]. This mechanism can reduce network overhead in network storage applications as shown by Zhou and her colleagues [37]. Polling I/O completions, however, does not necessarily depend on user-level access to the NIC and is equally applicable within the kernel.

User-level NICs enable the construction of user-level operating system libraries[1] (libOSes) such as proposed by Exokernel [15, 20]. In the same spirit, the Direct Access File System [25] (DAFS) is an example of a network attached storage system that proposes a user-level file system client. Just like in Exokernel, application-specific extensibility was one of the main motivations behind user-level networking. For example, application-specific customization of the networking stack in a user-level networking system was shown to be beneficial by enabling application-controlled flow control and feedback [36]. As the debate on extensibility technologies eventually pointed out, however, the key is often in finding the right API to kernel services rather than supporting user-level implementations of kernel subsystems. In many cases, parametrized kernel interfaces are sufficient from a performance standpoint [12].

Another motivation for user-level networking was the observation that kernel implementations of legacy protocols are constrained by having to rely on general-purpose or buggy kernel components. For example, a user-level implementation of UDP and TCP/IP over U-Net [36] achieved better performance than the standard kernel implementation, as the latter was constrained by its reliance on the BSD *mbuf* buffering system, which is not suited for high-speed networks [36]. In this case, however, the alternative of enhancing or replacing the underperforming kernel implementations promises similar performance benefits.

Finally, another recent trend in networking has been that of offloading TCP/IP processing to the NIC [32]. While the networking community has resisted such attempts in the past, it appears that use of TCP offload as an enabling technology for remote direct memory access (RDMA) finally makes it worthwhile [27]. One difference of a TCP offload NIC from other high-performance NICs offering RDMA capability (such as Myrinet [28] and InfiniBand [18]) is that TCP offload NICs are optimized for throughput rather than latency. In addition, offloaded TCP/IP implementations are harder to customize.

[1]At least for the network and network-storage subsystems.

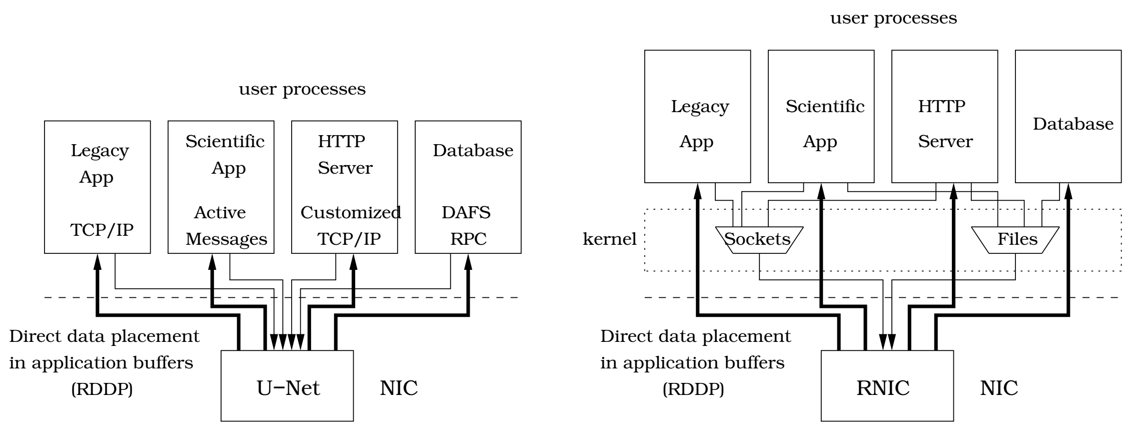

Fig. 1. **(a) User-level networking vs. (b) Kernel API to an RDDP-capable NIC (RNIC).** Both models enable direct data transfers between the network and application address space. In user-level networking, the NIC multiplexes the hardware between multiple processes. Narrow arrows indicate control fbw, and wide arrows indicate data fbw.

For these reasons, this paper's argument against user-level NICs is particularly applicable to TCP offload NICs.

III. User-level vs. Kernel NICs

Traditional NICs are designed to be directly accessible by a single trusted entity, typically the kernel. User-level applications perform I/O by interacting with higher-level abstractions, such as files and network sockets. When these abstractions are implemented in the kernel, the latter ensures that NICs and other devices can be safely shared by multiple processes. Devices exporting a user-level host interface, as shown in Figure 1 (a), cannot rely on the kernel for safe multiplexing between user-level processes. For this reason, a user-level NIC has to implement and export higher-level abstractions, such as virtual connection endpoints, and to maintain per-process and per-connection state. Such a NIC should be provisioned with sufficient resources to manage a large number of connections, which is a factor increasing NIC complexity as will be explained in Section III-A. In addition, the NIC should maintain a table to store translations of user-space virtual addresses to use in memory transactions on the system bus.

A NIC that implements the network transport and exports a remote direct data placement [16] mechanism through a kernel host programming interface can match two important benefits of user-level networking:

- Copy avoidance through direct transfers between the network and application address space buffers
- Control over I/O policy, which can be achieved by performing file caching and network buffering in application address space.

It can also offer two other benefits of user-level networking:

- Low per-I/O latency of interaction with the NIC. A user-level access to the NIC bypasses the kernel. Crossing the user-kernel interface, however, need not be an expensive operation, as shown in Section VI-A.
- Customization of the networking protocol, possible through a parametrized interface, e.g., socket options.

In practice, the only performance difference between a user-level and a kernel interface to a NIC is the user-kernel protection boundary crossing inherent in a kernel-based system-call API. In Section V we show that this performance difference does not significantly affect application performance in most network storage applications. It is important to note here that although user-level NICs bypass the kernel for regular I/O operations (e.g., send, receive), they cannot avoid involving the kernel for connection management (e.g., setup, teardown). For this reason, user-level NICs are not expected to offer any advantage in workloads with short-lived network connections.

Direct data transfers between the network and application-space buffers through either a user-level or a kernel interface require that the applications buffers are registered with the NIC, i.e., pinned in physical memory and their VM translations known to the NIC for the duration of the I/O operation. Registration involves the kernel and can be performed either per-I/O or less frequently, by caching registrations. Pre-registering large amounts of memory is sometimes possible but results in underutilization of physical memory in multiprogramming workloads.

A. *Complexity of NIC Design*

The requirements for user-level networking increase the complexity of NIC design. In particular, each virtual connection endpoint is associated with at least the following NIC resources [36]:

- A NIC memory region to store the context of virtual connection endpoints.
- A NIC memory region for NIC-host communication (e.g., doorbells and other events). This region is mapped in process address space on the host.
- One or more memory regions for queues (e.g., send and free queues as in U-Net [36]), also mapped in process address space on the host.
- Virtual to physical address translations for user-space addresses of I/O descriptors and other metadata.

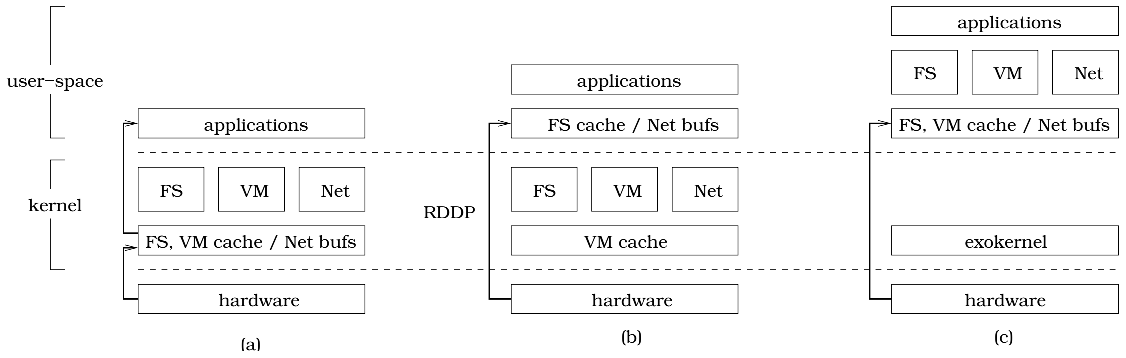

Fig. 2. **OS Structures.** (a) Mainstream (UNIX) and extensible systems (SPIN, VINO), (b) Hybrid (described in Section IV), (c) Exokernel.

The aggregate size of these resources increases linearly with the number of simultaneously open connections, which may be quite high for certain server applications (e.g., Web servers). Achieving scalable access to the user-level NIC has been recognized as a significant challenge in the past [36, Section 4.2.4]. One possible solution proposed by von Eicken and his colleagues [36] is to provide a memory resource management mechanism that enables the NIC to cooperate with the kernel in order to dynamically allocate NIC memory and DMA address space. Implementation of such a mechanism, however, is likely to compromise the efficiency and simplicity of user-level NICs.

A significant advantage of a kernel-based API to a NIC is that the NIC need not store virtual to physical address translations of user I/O buffers used in messaging operations (i.e., send and receive [3]). This is because the kernel is always on the I/O path and loads only physical buffer addresses on the NIC. In addition, a kernel NIC does not need to traverse data structures mapped in user-space, avoiding the need to store virtual to physical address translations for such structures. On the other hand, RDMA operations pose the same requirements to both user-level and kernel NICs: They require that NICs store virtual address translations for exported user I/O buffers, because these user-level addresses are used by the RDMA protocol to target remote memory buffers[2].

IV. OS Structure for Efficient I/O

Mainstream operating systems, such as UNIX, implement abstractions such as files (FS), address spaces (VM), and network connections (Net) in the kernel, as shown in Figure 2 (a). Applications in these systems have limited control over I/O policies and often experience high communication overhead due to data movement between software layers. Some extensible operating systems, such as VINO and SPIN, preserve this structure and address these issues by enabling safe execution of kernel extensions. An alternative to extensible operating systems, the Exokernel, structured as shown in Figure 2 (c), enables construction of all OS abstractions in user space. This paper describes an alternative OS structure, Hybrid OS[3], which is depicted in Figure 2 (b). Hybrid OS combines features from (a) and (c). Like mainstream OS structures, Hybrid OS places all OS abstractions in the kernel. Following trends towards scalable network storage, it assumes that applications access storage using file access over a network attached storage protocol, such as NFS. Hybrid OS exposes all file I/O policies to user-space applications by moving file caching to user space using RDDP for file data transfers. Network transfers are also performed using direct transfers between the network and user-space buffers. In this model, the kernel is still responsible for VM management, CPU scheduling, and most important, global resource allocation decisions.

The need for memory registration with the NIC requires involvement of the kernel. This is true for Hybrid OS, as well as for user-level networking systems. Some amount of registration caching is desirable to reduce the frequency of interaction with the NIC and to avoid VM page faults on the network data path. This implies that some fraction of the system physical memory would be pinned on behalf of the application and be unavailable to other system tasks. To ensure that there is enough unpinned physical memory to be shared by new system tasks, some amount of per-I/O registration is necessary. This is a cost that was not taken into account by user-level networking research [36], which assumed a limited buffer pool. Per-I/O registration opens up the possibility that user-level file accesses result in page faults in the absence of any coordination between the application and the VM system. Some form of application control over virtual memory management over the region used by the user-level cache or adaptation [1] to changing memory demands is necessary to avoid or reduce such page faults.

V. Analytical Performance Modeling

In this section, we examine the effect of protection domain crossing on application performance in kernel-based file system implementations with a system-call API. This study is based on an analytical model that draws from earlier work [10, 34] and assumes that (a) applications saturate the host CPU when performing I/O using relatively small block sizes (e.g., 4KB-64KB), and (b) the NIC supports transport protocol offload and an RDDP mechanism, which eliminates the network protocol, memory copy and checksum

[2]They are transported on the wire and are translated on-the-fly by the NIC.
[3]Exemplified by the Direct Access File System [25] (DAFS)

Source of Host Overhead	Without Offload/ RDDP	With Offload/ RDDP
System Call	Per-I/O	Per-I/O
File System	Per-I/O	Per-I/O
Interrupts	Per-Packet	Per-I/O
Device Interaction	Per-Packet	Per-I/O
Network Protocol	Per-Packet	-
Memory Copies	Per-Byte	-
Checksums	Per-Byte	-

TABLE I

Sources of file system client overhead and their type. Most of the sources of overhead are eliminated with a NIC offering transport offload and RDDP.

overheads listed in Table I. We assume that the NIC itself never becomes a performance bottleneck[4], except for adding a latency component in the I/O data path. With the assumption of transport protocol offload and the availability of an RDDP mechanism, the only host overheads remaining are the system-call, file system, device interaction, and interrupt processing costs, all incurred on a per-I/O basis. As will be shown later, the real difference between a user-level and a kernel-based implementation of a file client is the cost of the system-call protection domain crossing. Next, we examine the effect of this cost on I/O throughput and latency, based on analysis of the operations to issue and complete I/Os (Table III).

A. Throughput

For the purpose of this section we focus on throughput-intensive applications that perform I/O using asynchronous operations at the speed of the host CPU. The reason that we focus on this domain is because the effect of additional overhead due to the system calls is expected to be maximal when the CPU is saturated. In Table III we estimate the per-I/O overhead, assuming that each I/O involves two operations, one to issue the I/O and another to complete it. The difference between the "Best Case" and "Worst Case" columns is that the latter assumes that the check for I/O completion is unsuccessful and the application has to block waiting on the I/O. Blocking on I/O typically involves a system call and an interrupt for notification. Interrupts can be avoided in most cases in throughput-intensive workloads. This is possible either because polling in user or kernel space is always successful, as is the case when small blocks are used on a fast network and with a sufficient amount of asynchronous I/Os, or because the cost of per-I/O interrupts is amortized, as is the case when large blocks are used.

Looking at the "Best Case" column in Table III, it becomes evident that the only difference between the user-level and the kernel-based API, assuming the same file system implementation, is the overhead of two system calls in each I/O. Next, we estimate the degradation in application performance caused by the system-call overhead in a kernel-based API, compared to the performance achievable with the same implementation and a user-level API. Using a simple model with the parameters described in Table II we can express the throughput (in terms of number of I/O operations per second) achievable for a particular block size b as follows:

Parameter	**Description**
a	Application CPU consumption per record of data (s). The inverse of a is the application's record processing rate.
G	Gap per byte (s/MB). The inverse of G is the peak network bandwidth for large messages.
$o(b)$	Network file system overhead (s) per record, when record size is b.

TABLE II

Summary of model parameters.

$$\text{Throughput(b)} = \min(\frac{1}{Gb}, \frac{1}{a + o(b)}) \qquad \frac{\text{I/O}}{s}. \qquad (1)$$

For the purpose of this section we consider block sizes b for which the host can achieve its peak throughput $1/G$ when performing raw communication[5]. Note that the network throughput for very small blocks (e.g., of the order of one to a few tens of bytes) may be limited by the NIC to less than the asymptotic bandwidth $1/G$ of the network[6]. The application processing a and overhead $o(b)$ are expressed in units of time-per-I/O operation. The overhead incurred by the host CPU per unit of time increases with decreasing block sizes. In the case of a kernel-based API, the overhead of the two system calls is an additional factor that contributes to the load on the host CPU. Next, we focus on the following two questions:

- Under what conditions does the performance difference between the user-level and the kernel-based API become noticeable in throughput-intensive applications that have small computational requirements (i.e., small a)?
- What is the effect of the system-call overhead when the application involves significant data processing (i.e., large a) and is therefore heavily CPU-bound?

The performance degradation (PD) using a kernel-based API compared to a user-level API can be expressed as:

[4]This assumption does not always hold. For example, the NIC can limit I/O throughput when its processing speed significantly lags behind the host CPU [33, 34] or when using very small message sizes.

[5]In the Myrinet (Lanai9.2) network we use in this paper, this is possible for 4KB or larger blocks as can be shown with a simple benchmark performing raw communication using Myrinet's native GM commnuication library.

[6]This limit stems from the latency incurred in setting up the NIC DMA engines for sending and receiving very small messages. This limitation is captured in the *gap (g)* parameter of the LogP model [10].

API	Issue I/O	Complete I/O	Issue & Complete I/O	
			Best Case	Worst Case
User-level	Interaction with the NIC	User-level NIC polling, *or* System Call + *(possible)* interrupt	Two NIC interactions	System call + two NIC interactions + interrupt
Kernel-based	System-call + Interaction with the NIC	System call + NIC polling + *(possible)* interrupt	Two system calls + two NIC interactions	Two system calls + two NIC interactions + interrupt

TABLE III

Estimating the total per-I/O overhead for a user-level and for a kernel-based API. The file system overhead is the same in all cases and therefore omitted here. The kernel-based API impementation incurs an additional overhead for implementing global policies, which is also not shown here. The "Best Case" is based on the assumption that polling for I/O completion is always successful. The NIC supports transport protocol offload and RDDP in both cases.

$$\begin{aligned} \text{PD}(\%) &= \frac{\text{Throughput}_{user} - \text{Throughput}_{kernel}}{\text{Throughput}_{user}} \\ &= \frac{\min(\frac{1}{G}, \frac{1}{a+o(b)_{user}}) - \min(\frac{1}{G}, \frac{1}{a+o(b)_{kernel}})}{\min(\frac{1}{G}, \frac{1}{a+o(b)_{user}})} \\ &= \frac{\frac{1}{a+o(b)_{user}} - \frac{1}{a+o(b)_{kernel}}}{\frac{1}{a+o(b)_{user}}} \\ &= \frac{o(b)_{kernel} - o(b)_{user}}{a+o(b)_{kernel}} \\ &= \frac{\delta}{a+o(b)_{kernel}}. \end{aligned} \quad (2)$$

Where the overhead parameter δ is defined as:

$$\begin{aligned} \delta &= \text{User-kernel Crossing} + \text{Parameter Checking} \\ &\quad + \text{ Global Policy.} \end{aligned} \quad (3)$$

This parameter captures the fact that besides the overhead of the user-kernel crossing, a kernel-based API incurs the overhead of parameter checking, which may involve the cost of traversing the page table in case of validating memory addresses, and that of implementing global policies. The effect of the system-call overhead becomes maximal when the host CPU is close to its saturation point. Equation 2 therefore becomes most interesting in the region of I/O block sizes for which the host is CPU-bound. This excludes block sizes for which performance may be limited (a) by the NIC, or (b) by the network link. A close look at Equation 2 points to answers to the questions posed earlier:

For throughput-intensive applications with small computational requirements (i.e., small a), the degradation depends on the ratio:

$$\frac{\delta}{o(b)_{kernel}} = \frac{\delta}{\delta + \omega} \quad (4)$$

where the overhead parameter ω is defined as:

$$\omega = \text{File System and Device Interaction Overhead.} \quad (5)$$

The decomposition of the kernel API overhead to user-kernel crossing cost (δ) and file system and device interaction overhead (ω), is based on the assumptions about NIC support outlined earlier, i.e., transport protocol offload and RDDP. It is also based on the fact that the cost of interrupts can be made negligible by successful polling.

For file system and device interaction overhead much larger than the user-kernel crossing cost ($\omega \gg \delta$), Equation 4 becomes

$$\text{PD}(\%) = \frac{\delta}{\omega} \quad (6)$$

With a $0.5\mu s$ system-call cost on the Pentium III (measured in Section VI-A) and with file system and device interaction overhead in the tens of μs, this ratio is expected to be roughly somewhere between 1-5%. In practice, the 5% upper bound should be a conservative estimate if one accounts for the additional non-negligible overhead of application-level I/O libraries such as Berkeley DB [30]. In the future, improvements in processor techology are expected to reduce the cost of the user-kernel crossing and file system processing but will not affect the overhead of interaction with the NIC, which is a function of the I/O bus technology.

For applications performing significant data processing (i.e., large a), there is practically negligible performance degradation, since Equation 2 is no longer sensitive to the user-kernel crossing cost.

B. *Response Time*

The effect of the system-call interface in I/O latency is minimal. For example, the additional latency imposed by two system calls, one to issue the I/O and one to complete it, is about one microsecond on the 1GHz Pentium III platform used in our experiments (as measured in Section VI-A). This is less than 1% of the latency to fetch a 4KB file block from server memory, as measured on the same setup in a related study [26, Table 3]

Applications sensitive to messaging latency, such as distributed scientific applications with frequent synchronous communication using small messages, may still see a benefit from

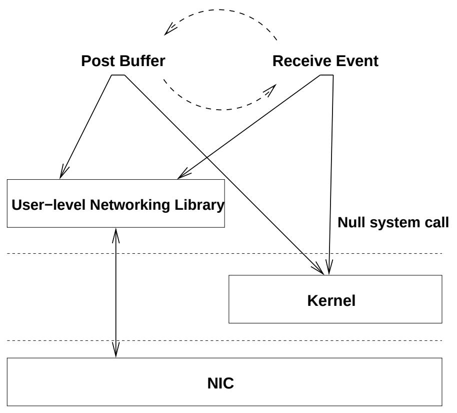

Fig. 3. **Emulation of a kernel-based networking library.** Each call into the user-level library is followed by a null system call with the same argument list. The calls shown here (*receive event, post buffer*) are typical of a receiver loop.

using a user-level NIC, as indicated by the latency measurements in Section VI-B. However, recent trends towards optimizing networks for throughput rather than latency suggest that the user-kernel protection boundary crossing will still be a small fraction of the overall I/O latency in these cases as well.

VI. Experimental Results

In this section, we report measurements of (a) system-call cost on the Pentium architecture, (b) latency and throughput for small messages, and (c) network file access performance. Some network file system and I/O library overhead estimates that were used earlier in the paper were drawn from our earlier work [25, 26].

A. System Call Cost

To quantify the cost of crossing the user-kernel boundary, we measured the system-call cost on a Pentium III CPU clocked at 1GHz with FreeBSD 4.6. Using the *perfmon(4)* interface to the Pentium cycle counters we found that a call to the *getpid* system service takes about 500 cycles (500ns). This result is corrected for counter manipulation overhead.

Earlier work measuring system-call invocation overhead under different operating systems on the Pentium found that low overhead access to system functionality is possible with mainstream operating systems such as BSD UNIX and Windows [8]. In addition, new low-overhead system call mechanisms, such as the Pentium ”Fast System Call” facility, part of which are the SYSENTER/SYSEXIT instructions [19], have recently been incorporated into mainstream operating systems (e.g., the Linux *vsyscall* API), resulting in significant reduction of the system-call overhead. In the following experiments, however, system calls are implemented using the standard `int 0x80` mechanism [19] and none of the above optimizations.

B. Latency and Small Message Throughput

Previous work on user-level networking underlined the importance of low per-message communication overhead in achieving low end-to-end latency and high throughput for small messages over high-speed system-area networks [36]. Myrinet [28] is an example of a network infrastructure that offers a user-level NIC programming interface via its GM communication library. In this section, we use experimental measurements from a Myrinet platform to show that similar benefits are possible with a kernel implementation of the GM programming interface.

We used two single-processor PCs with 1GHz Pentium III CPUs, 66MHz/64-bit PCI buses, and the ServerWorks LE chipset, connected via a 2Gb/s Myrinet switch using LANai 9.2 NICs. The user-level networking library we used was Myricom’s GM version 2.0.7 over FreeBSD 4.9. We measured the latency and unidirectional bandwidth achieved between our two hosts using the standard *gm_allsize* benchmark, which is part of the GM software distribution. The latency measurement involves roundtrip of a one-byte message without any process blocking at either end. The bandwidth measurement involves a sender that tries to keep the network fully utilized by asynchronously transmitting messages at a certain rate and a receiver pulling messages out of the network without blocking. Absence of blocking at both ends implies nearly zero interrupt overhead as we verified by examining the breakdown of the CPU time collected by a kernel profiler during the experimental runs.

We used two configurations of the *gm_allsize* benchmark: (a) the standard (unmodified) form, which uses the user-level GM interface, and (b) a configuration that emulates use of a kernel-based GM interface. The latter is based on (a) with the following addition: Each function call into the GM library is followed by a null system call as shown in Figure 3. These system calls have the same number and type of arguments as the equivalent GM calls and typically include a memory pointer to a user-level buffer. Besides the cost of *copying-in* the argument list, each system call performs an additional *copy-in* of 16 bytes from the user-level buffer and returns. To avoid (unfairly) benefiting from caching effects, different buffers were used at each invocation of the system calls. This way we ensure that the overall cost of the system calls approximated or exceeded their cost in a true kernel-based GM library.

A true kernel-based GM library is expected to perform at least as good as our approximation for two reasons: First, the only difference between a true kernel-based GM library and our approximation is that the former would execute code and access data in the kernel instead of a user-level address space. This, however, should not impede performance in any way. Actual kernel-based performance should be better due to privileges enjoyed by the kernel address space, such as wired page mappings. Second, our approximation may perform more than one system calls when checking for I/O completion if it eventually needs to block: one system call to check for I/O completion and another to block the process. A true kernel-based implementation would perform both within a single system call.

Our latency measurements show that the kernel-based approximation increases the one-byte roundtrip latency from

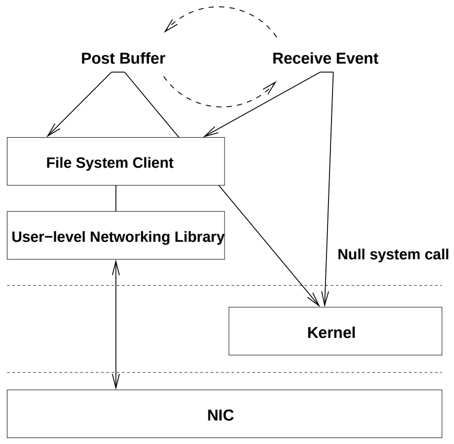

Fig. 4. **Emulation of kernel-based network file system client.** Each call into the file system client (e.g., *dap_read, dap_io_done* [11]) is followed by a null system call.

15.2μs with the user-level configuration, to 24.3μs. This increase can be attributed to the use of expensive system calls in the critical communication path. Improvements in system call overhead with the techniques mentioned in Section VI-A are expected to reduce this latency.

The throughput measurements show that both the user-level and the kernel-based configurations perform identically for message sizes ranging from a few bytes up to a few tens of KBs (e.g. 32KB). For very small message sizes, performance is limited by the NIC and is independent of the user or kernel implementation. For larger message sizes (e.g., $>$ 4KB), the limiting factor is the network link, which is saturated by both configurations. This is possible because there are spare CPU cycles to absorb the additional system call overhead of the kernel-based configuration without reducing the achievable bandwidth. Note here that a significant factor in this experiment is the absence of interrupts, which if present, may have biased the results in favor of the user-level implementation, as can be seen in the next experiment.

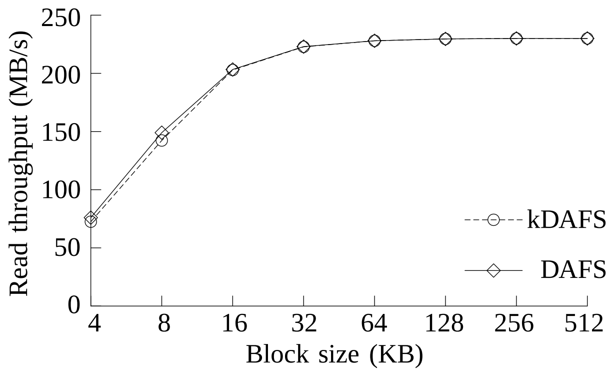

Fig. 5. **User-level vs. emulated kernel-based DAFS client performance in streaming file access.** The kernel structure is about 5% slower for small (4KB and 8KB) blocks and performs similar to the user-level structure for large blocks.

C. *Network File Access Performance*

In this section, we show that a network file system client can be implemented equally efficiently in user or kernel address space. We compare a user-level DAFS client (which we refer to as *DAFS*) to an implementation emulating a kernel-based DAFS client (*kDAFS*). The latter is derived by modifying the user-level DAFS client to perform a null system call[7] at each function entry (e.g., at I/O issue and completion), as shown in Figure 4. A true kernel-based DAFS client is expected to perform similar to kDAFS for the reasons mentioned in the previous section.

Two factors differentiate this experiment from the simpler throughput experiment of Section VI-B: (a) the increased overhead of a file system client compared to the overhead of the networking library alone; and (b) the additional overhead of interrupts for small transfers (even though the DAFS client is configured to poll when possible), which are due to the VI-GM implementation described below.

In this experiment we used the same setup as in the previous experiment except that the GM and FreeBSD software were based on versions 2.0-alpha1 and 4.6, respectively. The DAFS client [25] used in this experiment is implemented over VI-GM [29], Myricom's host-based VI implementation over GM. The DAFS server [24] is implemented in the FreeBSD kernel and uses our kernel version of VI-GM. One important implementation issue with VI-GM is that its event notification mechanism uses a dedicated thread (internal to VI-GM) to translate GM events to VI events. One problem with this implementation is the possibility of that thread blocking to wait for GM events (and thus incurring an interrupt on wakeup) even though the application thread intends to avoid interrupts by polling on the VI descriptor. This behavior results in additional interrupt overhead in certain cases.

Figure 5 shows that DAFS and kDAFS are equally effective in reducing communication overhead in a simple streaming workload where a client reads a 1GB file that is cached in server memory, using block sizes varying from 4KB to 512KB. The only performance difference occurs for small (4KB and 8KB) blocks, where communication overhead is highest and the effect of the system-call overhead becomes maximal. Even in this case, however, kDAFS is within 5% of the performance of the fully user-level client.

VII. CONCLUSIONS

In this paper, we argued that flexibility and good performance can be achieved with application-level file caching and buffering, both of which can be achieved without user-level access to NICs, simplifying NIC design and implementation. There is a commonly-held belief, however, that kernel APIs hurt performance due to the overhead of protection domain crossing. In this paper, we show that this overhead is rarely a problem for I/O-intensive network storage applications. To demonstrate this, we use an analytical model backed by

[7]Note that this system call has an empty argument list and is therefore not as heavyweight as the system calls used in the previous experiment.

experimental measurements and show that performance in such applications is dominated by other overheads, such as those of the file system and the host interaction with the NIC, which are incurred on a per-I/O basis.

REFERENCES

[1] A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, N.C. Burnett, T.E. Denehey, T.J. Engle, H.S. Gunawi, J.A. Nugent, and F.I.Popovici. Transforming Policies into Mechanisms with Infokernel. In *Proc. of 19th Symposium on Operating Systems Principles (SOSP-19)*, Bolton Landing, NY, October 2003.

[2] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, and E. Felten. A Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proc. of 21st Annual International Symposium on Computer Architecture (ISCA)*, Chicago, IL, April 1994.

[3] P. Buonadonna and D. Culler. Queue-Pair IP: A Hybrid Architecture for System Area Networks. In *Proc. of 29th Annual International Symposium on Computer Architecture (ISCA)*, Anchorage, AK, May 2002.

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

[5] National Energy Research Scientifi c Computing Center. M-VIA: A High Performance Modular VIA for Linux. In *http://www.nersc.gov/research/FTG/via*.

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

[7] J.B. Chen and B. Bershad. The Impact of Operating System Structure on Memory System Performance. In *Proc. of 14th ACM Symposium on Operating Systems Principles (SOSP-14)*, Ashville, NC, December 1993.

[8] J.B. Chen, Y. Endo, K. Chan, D. Mazieres, A. Dias, M. Seltzer, and M. Smith. The Measured Performance of Personal Computer Operating Systems. In *Proc. of 15th ACM Symposium of Operating Systems Principles (SOSP-15)*, Cooper Mountain, CO, December 1995.

[9] Compaq, Intel, Microsoft. *Virtual Interface Architecture Specification, Version 1.0*, December 1997. http://developer.intel.com/design/servers/vi/the_spec/specification.htm%.

[10] D. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP:Towards a Realistic Model of Parallel Computation. In *Proc. of 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, July 1993.

[11] DAFS Collaborative. *DAFS API, Version 1.0*, November 2001. http://www.dafscollaborative.org/tools/dafs_api.pdf.

[12] P. Druschel, V. Pai, and W. Zewnopoel. Extensible Kernels are Leading OS Research Astray. In *Proc. of 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, Cape Cod, MA, May 1997.

[13] P. Druschel, L. Peterson, and B. Davie. Experiences with a High-Speed Network Adaptor: A Software Perspective. In *Proc. of ACM SIGCOMM'94 Conference*, London, UK, August 1994.

[14] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble Component-Based Operating System. In *Proc. of USENIX Annual Technical Conference 1999*, Monterey, CA, June 1999.

[15] G. Ganger, D. Engler, M. F. Kaashoek, H. Briceno, R. Hunt, and T. Pinckney. Fast and Flexible Application-Level Networking on Exokernel Systems. *ACM Transactions on Computer Systems*, 20(1):49–83, February 2002.

[16] IETF Remote Direct Data Placement (RDDP) Working Group. http://www.ietf.org/html.charters/rddp-charter.html.

[17] M. Homewood and M. McLaren. Meiko CS-2 Interconnect Elan-Elite Design. In *Proc. of Hot Interconnects*, August 1993.

[18] Infi niBand Trade Association. *InfiniBand Architecture Specification, Release 1.0*, October 2000.

[19] Intel Inc. *Intel Architecture Software Developer's Manual. Volume 2: Instruction Set Reference*, 1999.

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

[21] H.C. Lauer and R.M. Needham. On the Duality of Operating Systems Structures. *Operating Systems Review*, 13(2):3–19, 1979.

[22] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. Daniel Hillis, B. C. Kuszmaul, M. A. St Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak. The Network Architecture of the Connection Machine CM-5 (Extended Abstract). In *Proc. of Symposium on Parallel Algorithms and Architectures (SPAA-92)*, San Diego, CA, June 1992.

[23] J. Liedtke. On Microkernel Construction. In *Proc. of 15th ACM Symposium on Operating Systems Principles (SOSP-15)*, Cooper Mountain, CO, December 1995.

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

[25] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. S. Chase, D. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. Structure and Performance of the Direct Access File System. In *Proc. of USENIX Annual Technical Conference 2002, Monterey, CA*, June 2002.

[26] K. Magoutis, S. Addetia, A. Fedorova, and M. I. Seltzer. Making the Most out of Direct-Access Network Attached Storage. In *Proc. of Second USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, March 2003.

[27] J. Mogul. TCP Offbad is a Dumb Idea Whose Time has Come. In *Proc. of Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*, Lihue, Hawaii, May 2003.

[28] Myricom, Inc. *Myrinet LANai9.2 Network Processor*. http://www.myri.com/vlsi/LANai9.pdf.

[29] Myricom, Inc. *Myrinet VI-GM 1.0*. http://www.myri.com/scs/VI-GM/doc/pdf/refman.pdf.

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

[31] R. Osborne, Q. Zheng, J. Howard, R. Casley, and D. Hahn. DART – A Low Overhead ATM Network Interface Chip. In *Proc. of Hot Interconnects IV*, Stanford, CA, 1996.

[32] R. Recio, P. Culley, D. Garcia, and J. Hilland. An RDMA Protocol Specifi cation. Technical report, IETF Internet Draft (Expires April 2004). http://www.ietf.org/internet-drafts/draft-ietf-rddp-rdmap-01.txt.

[33] P. Sarkar, S. Uttamchandani, and K. Voruganti. Storage over IP: When does Hardware Support Help? In *Proc. of Second USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, March 2003.

[34] P. Shivam and J.S. Chase. On the Elusive Benefi ts of Protocol Offbad. In *Proc. of Workshop on Network-I/O Convergence: Experience, Lessons, Implications (NICELI), in conjunction with ACM SIGCOMM 2003*, Karlsruhe, Germany, September 2003.

[35] C. B. Stunkel, D. G. Shea, B. Abali, M. G. Atkins, C. A. Bender, D. G. Grice, P. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swetz, R. F. Stucke, M. Tsao, and P. R. Varker. The SP-2 High Performance Switch. *IBM Systems Journal*, 32(2):185–204, 1995.

[36] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. of 15th ACM Symposium on Operating Systems Principles (SOSP)*, Cooper Mountain, CO, December 1995.

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