

# Design Tradeoffs for User-level I/O Architectures

Lambert Schalicke, *Member, IEEE*, Alan L. Davis, *Member, IEEE*

**Abstract**—To address the growing I/O bottleneck, next-generation distributed I/O architectures employ scalable point-to-point interconnects and minimize operating system overhead by providing user-level access to the I/O subsystem. Reduced I/O overhead allows I/O intensive applications to efficiently employ latency hiding techniques for improved throughput. This paper presents the design of a novel scalable user-level I/O architecture and evaluates the impact of various architectural mechanisms in terms of overall performance improvement. Results demonstrate that eliminating data movement across protection domains is the dominant contributor to improved scalability. Eliminating system call and interrupt overhead has only a small additional benefit that may not justify the additional hardware support required. While this evaluation is based on one specific design, the conclusions can be generalized to other user-level I/O architectures.

**Index Terms**—architecture, user-level, input/output devices, performance analysis, simulation

## I. INTRODUCTION

**T**he efficiency of the input/output (I/O) subsystem plays a large role in the overall performance of many applications. This importance is increasing since I/O intensive commercial applications are a fast growing market segment [1]. As network connectivity improves and as various kinds of audible and visual media become common data types, data volume escalates rapidly. This evolution has placed significant pressure on the performance of the I/O subsystem.

The I/O subsystem is traditionally accessed directly only by the operating system. However, the relatively poor cache and TLB locality of operating system code, together with low degrees of available instruction level parallelism, make operating system code mostly memory performance bound [2]. As semiconductor technology improves there is a growing gap between the DRAM based memory system and the SRAM and logic based processor core. This causes a similar growth in the gap between application and operating system performance. As a result, many I/O intensive applications spend increasing amounts of execution time in operating system code.

For instance, the MySQL database server [3] spends about one third of its processor cycles executing operating system code related to I/O operations, most of this time is spent copying data between kernel and user space. This overhead limits an application's ability to overlap long-latency I/O operations with independent work in order to improve throughput.

Improved I/O architectures such as InfiniBand [4] address the I/O bottleneck by replacing the conventional I/O bus with a scalable network. This choice results in better bandwidth, scalability and expandability, and also can provide user-level access to the I/O subsystem [5] [6]. Building on user-level communication techniques, such I/O systems devote hardware resources to providing low-overhead I/O mechanisms while maintaining proper protection and security.

Using one particular user-level I/O architecture as an example, this paper analyzes the design tradeoffs involving hardware support for low-overhead user-level I/O systems. The paper first presents a user-level I/O architecture that extends several existing approaches and solutions into a new, highly scalable design. Dedicated hardware support for user-level messaging at the host processor and network interface, combined with a novel stateless communication protocol, result in an I/O system that avoids hardware-imposed scalability limitations and preserves existing I/O programming interfaces. Performance evaluation results subsequently demonstrate the benefits of this approach compared to conventional kernel-based I/O systems. The second main contribution of this paper is a detailed analysis of the design space for user-level I/O architectures that reveals the relative contributions of the key mechanisms. These results indicate that for disk I/O, user-space data transfers provide the most significant improvement, while avoiding system calls and interrupts has only a small additional benefit that may not justify the additional hardware and software complexity. While this analysis is based on one specific I/O architecture, the performance results and conclusions presented here are equally

applicable to other low-overhead I/O architectures.

Section II describes the key mechanisms of the user-level I/O architecture, discusses the design rationale and contrasts the approach with other solutions. Section III outlines the performance evaluation approach, and Section IV presents overall performance results and analyzes the contributions of the individual mechanisms. Finally, Section V summarizes related work, and Section VI outlines directions for future research in high-performance I/O systems.

## II. USER-LEVEL I/O DESIGN

Next-generation distributed I/O architectures are commonly based on a scalable I/O network that connects client systems via a network interface device to a number of network-attached autonomous I/O devices [4]. Remote I/O devices perform low-level management tasks, such as disk block management or access protection checks [7] locally on the device controller. Clients issue I/O requests directly to remote devices using encrypted capabilities provided by the I/O manager to identify themselves. To fully utilize the greater bandwidth and to take advantage of the enhanced scalability, user-level processes can directly initiate I/O operations without involving the operating system.

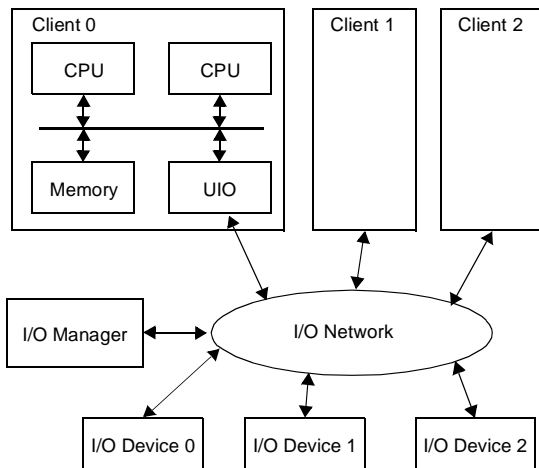


Fig. 1. Distributed I/O Architecture

Bypassing the operating system for I/O operations, while maintaining both the process-level protection and robustness of existing multi-user operating systems, requires architectural mechanisms to: initiate I/O operations, transfer data between user space and I/O devices, and to signal the completion of operations to the originating process.

User-level communication systems commonly found in clustered parallel architectures often use a connection-oriented approach to address these challenges. In such systems, connection endpoints include a set of pinned buffer pages which allow the network interface to know the physical addresses for buffer manipulation. To initiate communication, a message is assembled in the preallocated buffer and a descriptor is placed in a queue shared with the network interface. The network interface polls all such endpoints, reads the request descriptor and performs the necessary DMA operation using the previously established virtual-to-physical address mapping for that endpoint. On the receiving side, the network interface writes the message into a similar endpoint and then enqueues a descriptor in the receive queue shared with the destination process.

This connection-oriented solution eliminates the need for dynamic address translation by utilizing only preallocated and pinned buffers as source and destinations for DMA transfers. Requests are transmitted atomically to the device through per-process queues in memory. Overall, this approach has been shown to significantly reduce communication overhead, resulting in greater scalability. However, the need to preallocate buffers restricts the programming model and places the burden of buffer management on the application programmer. Pinning the communication buffers can also increase pressure on the physical memory management system and degrade overall performance if a large number of buffers or endpoints is required. Finally, keeping state information about all endpoints at the network interface significantly increases complexity which potentially limits scalability.

User-level I/O architectures share many requirements with low-overhead communication systems, and can thus use a similar approach. On the other hand, I/O systems have several unique characteristics that provide an opportunity to reduce hardware and programming complexity. I/O requests are always issued from a client to the device. Responses, while asynchronous, are expected by the application process, and data transfers are in many cases relatively large and involve sequential data. Building on these insights, the UIO architecture described here is a low-overhead I/O architecture that bypasses the operating system for common I/O requests, and does not impose the usual scalability limitations caused by per-connection state or preallocated buffers.

The key components of the UIO architecture are shown in Fig. 2. The conditional store buffer (CSB) implements non-blocking user-level synchronization and flow control. It is used to atomically transfer the request arguments from applications to the UIO device. User-space data transfers are facilitated by a device TLB, which performs virtual-to-physical address translation and protection checks. Upon request completion, user-level notifications are delivered directly to the application by the host operating system based on information provided by the UIO device.

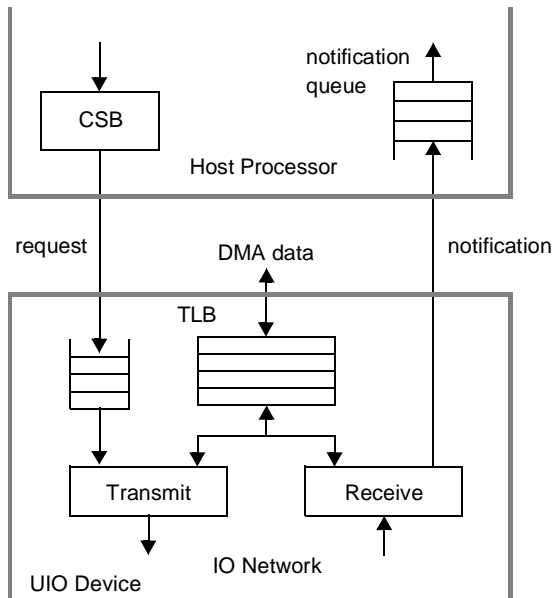


Fig. 2. Interaction of Host Processor and UIO Device

Notably absent from the UIO device design are data structures and state machines managing existing connection endpoints. The following sections describe the stateless communication protocol and the supporting hardware mechanisms in more detail.

### A. Stateless Communication

The fundamental structure used to communicate between application software, the UIO device and the remote devices is a request packet (or request structure). This structure contains all information needed to handle a request, both locally at the client and at the remote device. The request structure is passed from the application to the UIO network interface device and then forwarded to the remote I/O device over the I/O network. When the I/O request completes, the structure is returned to the

TABLE I  
UIO REQUEST STRUCTURE

Name	Description
Remote Information	
capability	identifies remote object and clients access rights
command	desired operation
arguments and flags	additional operation specific arguments
return status	completion status of operation, returned to client
Local Information	
context ID	identifies process to device
buffer and length	buffer address and size for data transfers
notification buffer	buffer address for request structure during notification
notification handler	address of user-level routine called during notification
request argument	application-defined value to identify request

originating UIO device and eventually reaches the application in the form of a notification.

Having software provide all relevant information with every request facilitates a simple and scalable UIO device design since no state information needs to be maintained by the hardware. In addition, the stateless device design simplifies failure recovery as no state needs to be reconstructed or synchronized in the event of a transient failure. The slight increase in per-request data transferred between software and the UIO device does not affect performance as long as the entire request structure does not exceed the size of a cache block, since that is the natural transfer size on the system bus. Table I lists the components of the UIO request structure, separated into remote and local information. In the prototype implementation described here, the UIO request structure fits into a 64 byte block.

Remote information such as the capability, command and arguments is used only by the remote device; the UIO device is simply a forwarding agent. Local arguments, on the other hand, are relevant only to the UIO device. These arguments include a unique process identifier that enables the UIO device to perform local protection checks on the buffer addresses and to deliver completion notifications to the correct process. A buffer address and length is used by requests that transfer data to or from the remote device. The notification buffer address is a location in the application address space where the request structure is deposited when it is returned

from the remote device, so that the application can inspect the return value. The notification handler is a user-level routine that is executed when the request complete. This handler performs the requisite synchronization operations within the application process.

To initiate a request, the application software assembles all required arguments into a request structure and sends it to the local UIO device via the CSB. After initiating the I/O request, the application continues executing since no kernel scheduling operation was involved. When the request completes, the system invokes the user-level notification handler routine which performs the necessary synchronization operations. This low-overhead non-blocking I/O approach enables applications to optimally overlap long-latency I/O operations with independent computations. A user-level thread library, for instance, could initiate a thread switch and thus preserve the traditional blocking I/O programming model for individual threads, while avoiding the overhead of kernel support for non-blocking I/O.

### B. Atomic Transfer Initiation

When initiating an I/O request, software must ensure that the individual arguments and parameters are communicated to the network interface device atomically since the device may be accessed by multiple sources. Eliminating system calls from the I/O request implies that multiple applications may be competing for device access without global synchronization. Since user-level processes can be preempted at any time and cannot be trusted to perform the necessary synchronization and flow control, a low-overhead hardware mechanism is needed to provide atomic user-level device access.

The conditional store buffer (CSB) is a software controlled hardware buffer located in the processor bus interface that combines a sequence of I/O stores into a single bus transaction up to the maximum size of one cache line [8]. The CSB permits user-level code to explicitly control which stores will be combined and when the combined set of stores will be issued on the system bus. In addition, it provides a non-blocking flow control mechanism to reliably inform applications if requests are issued at a faster rate than the I/O device can process them.

Fig. 3 shows the structure of the conditional store buffer. When the CSB receives a combining store, it

compares the destination address with the value that has been saved from the previous store instruction. On a match, it stores the data in the appropriate data buffer slot and increments the hit counter. If the comparison fails, the data buffer is cleared, the hit counter is set to one, and the new data is stored. To achieve interprocess atomicity, the conditional store buffer is cleared and the hit counter is reset to zero under any condition that may lead to a process context switch, notably traps and external interrupts.

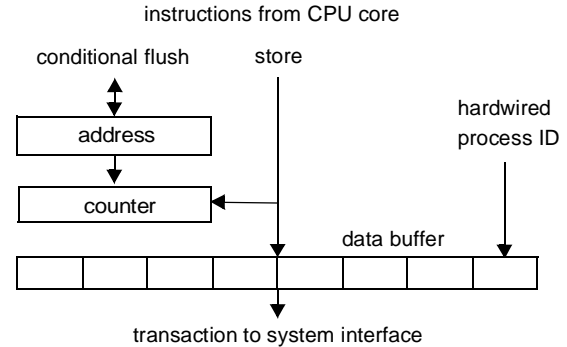


Fig. 3. Conditional Store Buffer Structure

At the end of the uncached store sequence, when all request arguments have been written, the application issues a conditional flush instruction to indicate that the sequence is complete and to check the atomicity and flow control status. The conditional flush instruction communicates the expected value of the hit counter to the CSB, and returns both the atomicity and flow control status to the application. If the counter value is equal to the expected value provided by the instruction, and the destination address matches the value present in the CSB, then the data is sent to the system interface as a single burst transaction on the system bus, and the buffer and hit counter are cleared. Upon receiving the burst transaction, the local UIO network interface returns the flow control status to the processor, which becomes the return value of the conditional flush instruction. If either the addresses or counter values do not match, then the data register and counter are cleared, nothing is issued to the system interface, and the conditional flush instruction returns a negative result to the application.

The application software is responsible for checking the return value of a conditional flush instruction. The application may recover from a failed flush for instance by branching back to the beginning of the sequence of combining stores. To reduce the

likelihood of livelock, applications should implement some form of retry damping mechanism such as exponential backoff. The CSB approach is similar to the load-linked store-conditional instructions found in many modern processor architectures, but the CSB combines it with atomic data transfer and flow-control.

Although the UIO request structure is assembled and issued by user-level software, the context identifier is privileged and must be managed by the operating system, since it is used for address translation and protection checks by the UIO device. Many microprocessors store some form of process ID in a privileged register to detect TLB and cache aliasing and accelerate TLB miss handler traps. Hardwiring this register to a known CSB entry allows the operating system to manage this component of each UIO request without incurring the cost of a system call for every request.

Several alternative implementations of the basic CSB approach are possible. Ideally, the conditional store buffer should be located in the processor bus interface. This ensures the lowest possible system bus utilization and efficient conflict detection with few hardware modifications. Alternatively, the CSB may be implemented in the UIO device. This approach avoids the difficulty of modifying the processor bus interface, but it incurs a bus transaction for each individual argument and it requires a mechanism to forward interrupts and traps to the device to detect atomicity violations. Third, assuming the remainder of the architecture is unchanged, a relatively simple system call can be used to provide protected device access. This system call employs conventional kernel-level synchronization techniques to ensure exclusive access, copies the I/O arguments into device control registers and checks the device status to avoid overflowing the request buffer. In the simulated prototype architecture, such a system call is more than two times as expensive as a hardware CSB that is integrated with the CPU, but it does not require any hardware modifications.

Connection-oriented communication architectures address the problem of atomic device access by providing process-private memory regions to store request descriptors. It is the responsibility of the network interface to poll connection endpoints and to multiplex requests from multiple processes. The cost of initiating a transfer in such a system is comparable to that

incurred by the use of a CSB. In both cases, a number of arguments are assembled and stored at a specific address, and the success of the operation is checked by application software. The main benefit of the CSB is the reduced hardware complexity of the I/O network interface, since it is not required to poll a number of dedicated process queues. Furthermore, the lack of polling may reduce overall latency, and there is no architectural limit on the number of concurrent connections or requests. On the other hand, the number of I/O request arguments is limited by the size of the CSB, which may restrict the number of elements in a scatter/gather DMA list.

### C. Virtual Address DMA

Copying data between kernel and user space is often the largest component of I/O overhead. This operation not only occupies the host CPU for many cycles, it also has significant cache and TLB pollution effects. In addition, host CPU copy bandwidth is usually lower than that of a dedicated DMA engine.

Existing solutions for high-performance communication networks [9] [10][11][12] often require the application to set up the communication buffers in advance, allowing the kernel to pin the pages in physical memory and to communicate the address mapping to the I/O device. This scheme places the burden of managing the limited buffer space on the programmer, and often forces the application to copy data between the buffer and other locations where the data is used. Furthermore, pinning pages reduces the general pool of available memory, possibly affecting the performance of the entire system due to increased memory pressure which also limits scalability.

The UIO architecture enables the network local interface to perform virtual to physical address translations autonomously using a TLB, thus minimizing host CPU occupancy for data transfers. For every bus transaction, the DMA engine presents the virtual address and a context identifier to the TLB, which returns the corresponding physical address and flags access violations. The context identifier is provided with each request packet and is used to distinguish address mappings for different processes with the same virtual address. On a TLB miss, the network interface device either performs

the necessary page table lookup independently, or invokes the kernel for assistance. As a cache of address translations, the TLB participates in the TLB coherency protocol like any other CPU in a shared memory multiprocessor. When unmapping a page, the operating system informs the device TLB which in turn invalidates any matching entries. Since the device TLB entries are tagged with process context identifiers, this invalidation is only required for relatively infrequent physical memory allocations and deallocations and does not impact process context switch performance. In the simulated prototype system described in the following section, the UIO device driver registers a callback routine with the memory management subsystem that invalidates UIO device TLB entries when a page table entry is modified.

Unlike a microprocessor TLB, the device TLB does not need to perform an address translation every cycle, since it is only accessed once per bus transaction. Furthermore, due to the relatively long latency of most I/O transactions, single-cycle access to the TLB is not as critical as in a processor. In addition, most DMA operations access memory sequentially, thus reducing the need for a highly associative TLB design to achieve acceptable hit rates. These relaxed requirements enable TLB designs that use more dense and less expensive commodity SRAM structures with lower associativity.

Enabling the network interface device to handle TLB misses independently minimizes CPU occupancy for data transfers. The device TLB uses the same page table structures as the host processor to avoid the overhead and complexity of maintaining separate I/O page tables in the kernel. Sharing page tables with the kernel also allows the device to detect page faults in the same way as processor TLBs do, in which case the host operating system is invoked through a conventional interrupt.

Several processor architectures [13][14] specify page table structures that are intended to be traversed by a hardware engine. For such organizations, a programmable page table walk engine is able to perform page table lookups independently from the host CPU, while being flexible enough to adapt to a variety of page table and operating system organizations. The instruction set of this table walk engine consists of memory access instructions, simple integer arithmetic such as add, compare and logic operations. This instruction set is

sufficient to realize a wide variety of page table walk algorithms, yet it can be implemented with simple hardware resources. A prototype implementation of a programmable table walk engine using a standard cell library for a 0.25 micrometer bulk CMOS process as a synthesis target operates at 66 MHz and uses 0.388 squaremillimeter. The hardware costs of the TLB itself are dependent on its size, but are expected to be minor considering that the device TLB can be constructed from commodity SRAM.

Alternatively, device TLB misses can be handled by interrupting the host operating system. In this case the device needs to provide both the virtual address and the address space identifier to the interrupt handler via control registers. The interrupt handler, after reading the device status register to determine the cause of the interrupt, reads the virtual address and address space identifier causing the TLB miss, performs the address translation by calling a routine in the memory management subsystem and writes the resulting physical address into a control register on the device. This scheme further reduces the hardware cost by eliminating the device TLB miss handler, at the expense of greater host processor occupancy.

The device TLB is a key component of the stateless communication protocol. It enables applications to use arbitrary buffers for I/O operations without having to invoke an operating system service to pin the pages and communicate address translations to the device. While adding a TLB and the associated miss handling hardware to the device consumes additional chip resources, it eliminates the need to maintain per-connection state by the UIO device and avoids the associated scalability limitations. Other user-level communication architectures also support device TLBs, but usually avoid the cost of hardware support for TLB miss handling. This work shows that such hardware support can be implemented inexpensively, while being able to adapt to a variety of page table and kernel organizations.

#### *D. Light-weight Notifications*

Interrupt or notification handling is an important part of I/O transactions. In most systems, interrupts are handled entirely by the kernel. Applications block when initiating an I/O operation and are unblocked by the interrupt handler. This approach simplifies the application interface to I/O, but it can

incur significant overheads of 15 ms or more for each interrupt [15]. This overhead is a result of the generality of the interrupt handler, which requires it to save and restore significant process context and requires a complex code structure to handle all possible interrupt causes. Furthermore, blocking I/O does not allow applications to overlap long-latency I/O operations with independent work.

The user-level I/O architecture provides a flexible, lightweight mechanism to invoke arbitrary user routines for I/O notifications with minimal kernel involvement. The user-level notification mechanism consists of a lightweight kernel interrupt handler that closely cooperates with applications to asynchronously execute user-level notification routines, and a notification queue located in the host processor bus interface. The queue reduces the kernel interrupt handler overhead by enabling the UIO device to send all required information to the host processor at the time of the interrupt. Each entry in this hardware queue holds information about a unique I/O interrupt, including the target process identifier, notification buffer address, notification handler pointer and a user-defined argument. The required depth of the queue depends on the number of outstanding notifications for each UIO device, and the number of devices. The head of the queue is available to software via memory-mapped control registers inside the processor. The interrupt handler can thus obtain all pertinent information locally without the cost of multiple uncached reads from the UIO network interface. The flexibility of the notification mechanism allows applications to use any number of specifically tailored routines to handle notifications, thus reducing the execution cost of each routine. Handling notifications almost exclusively in user space reduces overhead since no heavyweight context switch is necessary. This approach also minimizes cache and TLB pollution effects. The notification mechanism exploits the fact that the originating application process is likely to be currently executing, in which case heavyweight scheduling and context switching is not necessary. Note that these lightweight notifications are used only for the common case where an I/O request completes without exception. Critical exceptions are handled by the kernel via conventional interrupts.

When initiating an I/O request, the application specifies a buffer location where the original request structure with the return status will be deposited by

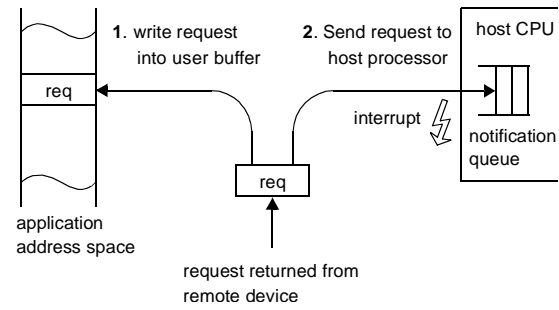


Fig. 4. User-level Notification Mechanism

the UIO device. This buffer allows the application to inspect the return value upon request completion. In addition, each request structure specifies a user-level notification routine that will be executed when the I/O request completes. The process identifier inserted into the request structure by the CSB is used to locate the target process for a notification. After writing the request structure into user space, the UIO device writes the same structure to the notification queue in the host processor bus interface to trigger an interrupt, similar to the way normal external interrupts are delivered. However, the bus transaction causing the interrupt carries the complete request structure to the CPU, thus making all pertinent information available locally in the CPU. When receiving the notification, the host processor enters a low-priority kernel interrupt handler that accesses the head of the notification queue via control registers to process the notification. Using the process identifier in the request structure, it determines if the currently running process is the target process for the notification. If it is, the kernel interrupt handler arranges for the notification handler to be executed next, while passing the address of the request structure in user space as well as the address of the interrupted instruction as arguments. Upon completion, the user space notification handler returns to the interrupted instruction by executing a jump to the address provided by the argument.

If the current process is not the target process for the notification, or the current process is not executing in user mode (e.g., as would be the case during a system call), the notification is queued for later delivery to ensure that each individual notification is forwarded to the application. To implement this queue, the kernel maintains a list of pointers to request structures in the non-pageable part of the

process structure. When receiving a notification for an inactive process, the kernel interrupt handler appends the virtual address of the current request structure (that was written to a user buffer) to this list. The maximum size of the list determines the maximum number of outstanding requests for a process and is a system-specific constant. On a context switch, the kernel checks to see whether the new process has any pending notifications. If this is the case, the kernel uses the pointer array to create a linked list of request structures in user space. The kernel then schedules the first notification handler to run prior to the application. This handler recursively calls the next handler, thus traversing the list of notifications. The last notification handler returns to the instruction that the application was executing before the last context switch, using a jump instruction.

The advantage of this scheme over a conventional signal handling mechanism is fourfold. First, the initial kernel interrupt handler saves only a minimal amount of state and returns to user-code almost immediately. This minimizes execution time and secondary cache pollution effects. Second, all information pertinent to a notification is pushed to the host processor as part of the notification, thus avoiding the latency of reading device status registers. Third, specifying a notification handler with each request allows applications to use a multitude of specifically tailored and optimized routines. Finally, neither the operating system nor the device hardware need to maintain a table of notification handlers for applications, which would limit the scalability of the approach. Instead, this information is part of each UIO request.

User-level trap handling has been proposed previously to allow applications to handle synchronous traps [16]. This work is based on the same concept and uses similar mechanisms, but extends the scope to asynchronous I/O interrupts and replaces the static trap vector table with a more flexible method. The idea is to invoke user-level routines upon I/O request completion for application synchronization is based on Active Messages [17] and was applied here to the I/O domain.

### III. EVALUATION

#### A. Simulation Environment

A prototype of the user-level I/O architecture described here has been implemented in the ML-RSIM simulation environment [18]. Based on RSIM

[19], this simulation system combines detailed models of a dynamically scheduled CPU with caches, a memory controller and several I/O devices with a fully-functional Unix-like operating system. The operating system running on the simulated hardware implements multiprogramming, signal handling, virtual memory, and a comprehensive Solaris-compatible application programming interface. The I/O path follows standard Unix conventions and consists of a Unix FFS file system, a buffer cache, SCSI disk and controller device drivers and disk interrupt handlers [20]. Most of the kernel code including the file system and device drivers is ported from NetBSD to the simulated hardware. The overall accuracy of the simulation infrastructure, including the hardware microarchitecture and operating system overhead, has previously been validated against an SGI Octane workstation [21].

The prototype UIO implementation consists of models of the CSB, the UIO device and the notification mechanism, along with a device driver that facilitates kernel management of the UIO device. A user-level thread library hides details of the request structure and notification handling from applications and integrates the I/O facilities with the thread scheduling system to provide per-thread blocking I/O semantics consistent with conventional I/O interfaces. This library can be linked to existing applications without any other modifications. In order to compare the CSB approach to the conventional system call method, the simulated operating system also implements I/O system calls. Furthermore, UIO device TLB misses can be handled either by the dedicated device table walk engine described here, or via conventional host processor interrupts. The simulated kernel has also been modified to support user-level interrupts. Changes include adding a 64-entry notification structure to the process context and implements the kernel-level interrupt handler that forwards notifications to the applications. This variety of mechanisms, combined with the level of detail provided by ML-RSIM make this environment well suited to evaluate user-level I/O design tradeoffs.

#### B. Benchmark

The experiment described in this paper simulates concurrent queries on the MySQL database server [3] to demonstrate the impact of I/O overhead on application throughput scaling. MySQL

is a public-domain database server that supports the SQL database query language. It is internally multithreaded and can take advantage of nonblocking I/O operations. Database tables are represented as regular directories and files, making the server very portable. However, performing disk accesses through the buffer cache is not necessarily representative of commercial databases, which often access the raw disk device. In this experiment, the MySQL database system serves as a representative of a broader class of I/O intensive applications that are characterized by data sizes on disk that exceed main memory caching capacities, and by high I/O throughput requirements. Such applications include file servers, mail and news servers, web servers and databases.

The database tables are set up following the Wisconsin database benchmark. Each record contains 13 integers and three strings. The first two fields are unique numbers that also serve as primary keys. The remaining fields are various modulo values of the second primary key. The string elements are textual representations of the primary keys and one of four random 6-character strings. While the Wisconsin benchmark is generally considered too simplistic for the evaluation of modern database servers, the well-understood and regular table setup provides easy and effective control over the data access patterns of each SQL query. In these experiments, the database server executes the following query:

```
select t.*, t.unique1 from tenk1 t where t.tenthous = 1000
```

This SQL statement returns all fields of each record where the second primary key modulo 10,000 is equal to 1,000. For a table with 60,000 records, six records are returned. Since the query does not refer to a primary key, the database server scans the table sequentially for a matching record. Consequently, the benchmark is more representative of a decision support system, data mining workloads, or other I/O intensive applications that operate sequentially on a large volume of data. A single query executes in approximately 11 seconds of real-time on the simulated architecture.

The size of the database table is 100 Mbytes, while the simulated buffer cache is configured with 20 Mbytes for all experiments. Since the cache is initially empty and even a single table is too large to fit in the cache, buffer cache effects do not influence observed I/O performance.

TABLE II  
UIO REQUEST STRUCTURE

Parameter	Value
Processor	800 Mhz, dynamically scheduled 4-way fetch/dispatch/issue/graduate 64 entry reorder buffer
Caches	32 kbyte 2-way set-associative level-1 2 Mbyte 2-way set-associative level-2
System Bus	125 Mhz 64-bit multiplexed address/data
Memory	SDRAM, 4 banks
UIO Device	4 concurrent DMA engines, 32-entry 4-way set-associative TLB
SCSI Bus	40 Mhz, 32-bit wide
SCSI Disk	9 Gbyte, 15000 rpm, 10 heads, 3.4 ms average seek time, 1 Mbyte segmented cache
Operating System	Lamix, based on NetBSD

In the following experiment, the database server is concurrently executing identical queries on separate tables that are stored on independent disks. The multi-threaded design of the server allows it to overlap I/O latency with processing of independent queries, thus improving throughput. Consequently, throughput scalability is the most important performance metric. Perfect scalability implies that  $N$  queries complete in the same amount of time as one query, resulting in an  $N$ -fold throughput improvement. Realistically, sequential application processing, thread scheduling and operating system overhead limit scalability.

Previous work has shown that low-overhead access to I/O devices leads to significant performance improvements by reducing or eliminating operating system overhead. The purpose of the following experiments is not only to strengthen these results but also to show the magnitude of the operating system overhead, and to quantify the contribution of each architectural mechanism in the UIO approach with respect to the overall benefit.

### C. System Configuration

The simulated system corresponds to a modern server-class architecture, consisting of a dynamically-scheduled processor running at 800 Mhz with 15,000 rpm SCSI disks and a Unix-like operating system. Table II summarizes the key architectural parameters.

Note that the modeled SCSI disk is relatively small to keep simulation memory and storage re-

quirements manageable. However, seek and transfer speeds are representative of modern high-performance disks. To facilitate concurrent queries, database tables are replicated over 16 disks with independent SCSI controllers for the kernel-based I/O experiments, or over 16 network-attached storage devices. The distributed I/O architecture follows the general outline of a proposed network-attached secure disk architecture [7]. The simulator uses a collection of workstation nodes running a kernel-level server process to model intelligent disks. The I/O network is modeled with fixed latencies and a finite aggregate bandwidth of 400 MByte/s to eliminate the I/O network as a bottleneck, thus focusing the experiments on the host processor's overhead.

#### IV. RESULTS

This section presents throughput scalability for a spectrum of I/O architectures using the execution-driven simulation system described previously. In this context, scalability is the execution time of  $N$  concurrent queries over that of a single query. In a system configuration with multiple disks and a single host processor, CPU utilization due to application and operating system processing is one of the bottlenecks for scalability. The goal of a user-level I/O architecture is to minimize processing costs not directly related to the application, thus improving scalability.

##### A. Overall Benefit

Fig. 5 shows throughput scaling of the MySQL database server when running between one and 16 identical queries. A conventional kernel-based I/O system saturates at approximately nine concurrent queries, while the user-level I/O architecture scales to over 13 queries, resulting in a 40 percent improvement.

This difference can be directly attributed to the reduction in operating system overhead. To demonstrate the magnitude of the OS overhead, Fig. 6 plots host processor utilization for varying numbers of concurrent queries when using conventional kernel-based I/O. As processor utilization approaches 100 percent, throughput saturates. The utilization graph also breaks out I/O-related overhead due to data copying, and other activities. Copy operations are required to move data between the file cache and

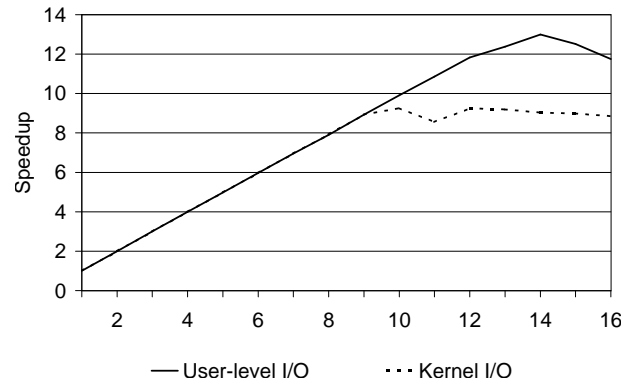


Fig. 5. Scalability of User-level and Kernel I/O

user buffer. For larger transfers, this operation not only incurs CPU overhead and increases overall I/O latency, but also introduces cache and TLB pollution effects. In this case, copy overhead is responsible for the majority of the I/O cost. Additional I/O overheads such as system call and interrupt handling account for less than 10 percent of the total utilization.

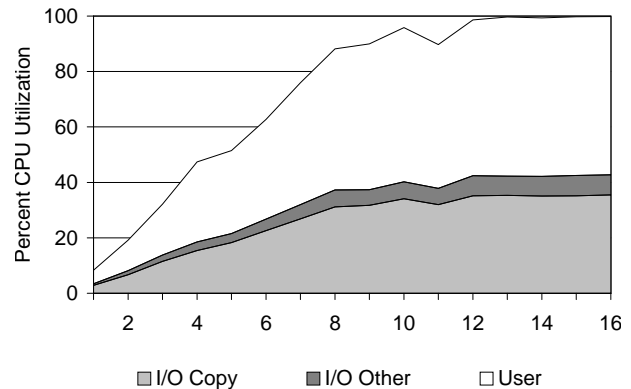


Fig. 6. Kernel I/O Processor Utilization

Overall, these results confirm the significant performance advantage attainable from user-level I/O systems.

Generally, operating system overhead is reduced or eliminated through three architectural mechanisms: 1) user-level initiation of transactions, including setting up the DMA transfer; 2) direct user-space data transfer, and 3) low-overhead notifications or interrupts. While implementation details, cost and performance differ between solutions, the relative contribution to the overall benefit is expected to be comparable between this UIO architecture and other, similar approaches. The following section tests a spectrum of architectural solu-

tions to provide a better understanding of the cost-performance tradeoffs.

### B. Request Initiation

As an alternative to the conditional store buffer, a relatively simple system call can be used to initiate an I/O request. Like the CSB, the system call takes a list of request arguments and atomically transfers them to the UIO device. However, instead of relying on hardware support to achieve atomicity, it acquires a kernel lock to ensure uninterrupted access. In the simulated operating system, this system call is implemented as an `ioctl()` call to the device driver.

The complexity and overhead of this call is not directly comparable to a conventional `read()` system call, since the latter traverses more layers in the operating system and also pins the pages involved in the DMA transfer. On the other hand, the costs of entering and exiting the kernel and of locking the device remain; no special hardware support in the processor, system bus or device is required. Thus, this system call is representative of a less aggressive I/O architecture.

On the experimental system, the cost of one such system call ranges from 1.75 to 2.67 microseconds. Generally, with a larger number of concurrent queries the system call overhead increases due to higher cache and TLB miss rates introduced by the greater processing load. A single query performs 6425 of these system calls, resulting in a total overhead of 11.29 milliseconds. For 16 concurrent queries, the total system call overhead increases to 217 milliseconds.

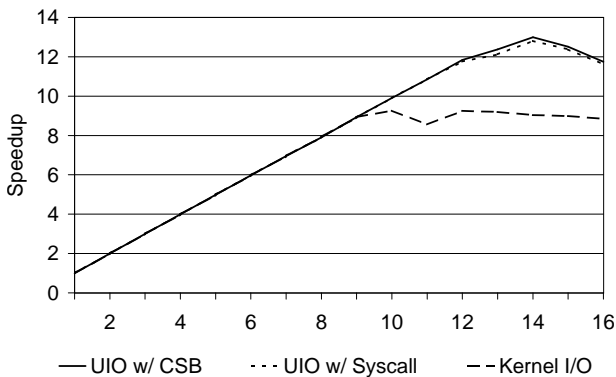


Fig. 7. Throughput Scaling for Alternative Request Initiation Schemes

Fig. 7 demonstrates that this additional overhead has minimal impact on scalability. This graph com-

pares the scalability of the UIO architecture with either the CSB or system call to initiate I/O transfers. For comparison, the scalability of the kernel-based I/O system is also included.

### C. Data Transfer

The DMA engine of the UIO device employs a TLB to facilitate the autonomous translation of virtual user-space addresses to physical addresses. Upon a DMA miss, a programmable table walk engine loads the required page table entry and restarts the bus transaction. As an alternative, device TLB misses can be satisfied via a more conventional kernel interrupt mechanism, in which the kernel interrupt handler performs the address translation. This approach avoids the additional complexity and cost of the table walk engine, and also simplifies the handling of page faults.

The disadvantage of kernel-level TLB miss handling is a higher miss latency due to the interrupt overhead, and the increased host processor utilization. For the two-level page table of the simulated system, the device table walk engine can handle a TLB miss on average in 770 to 840 nanoseconds, largely determined by the memory and I/O bridge latency. In the simulated system, the table walk engine overhead is fixed based on a hand-coded implementation of the code required for this particular page table organization, while the overall latency varies depending on the level of bus and memory contention.

TLB misses handled by the simulated kernel interrupt require between 2.5 and 3.1 microseconds, over three times longer. In this case, the UIO device driver interrupt handler normally reserved for error conditions also performs demand address translations. While this interrupt handler can perform the actual page table walk with a similar latency as a hardware implementation, it incurs a higher overall cost due to the interrupt entry and exit overhead. Furthermore, reading the device status, virtual address and context identifier from device control registers incurs additional latency. On the simulated system, a single TLB miss interrupt consumes between 2.9 and 4 microseconds of host CPU time, and also displaces valuable application data and code from the caches. Interestingly, at higher TLB miss rates, the cost of an individual interrupt decreases due to some code reuse across interrupt

invocations. With 16 concurrent queries, the UIO device incurs over 615,000 TLB misses, resulting in 1.77 seconds of CPU utilization. Offloading TLB miss interrupt processing to a separate CPU in an SMP would minimize the performance impact on the application, due to resource contention, but it would not significantly reduce the TLB miss handling latency. Furthermore, dedicating a general-purpose CPU exclusively to I/O processing is a very cost-intensive solution to the problem.

To illustrate the performance impact of kernel-level TLB miss handling, Fig. 8 shows the scalability of the UIO architecture with hardware and kernel miss handling. For comparison purposes, it also includes the results from a conventional I/O architecture.

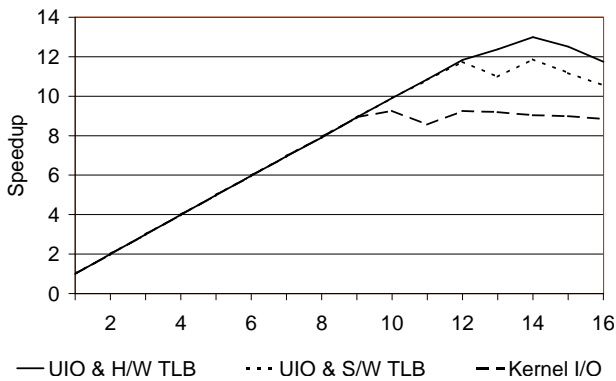


Fig. 8. Throughput Scalability of Alternative TLB Miss Handlers

Clearly, kernel-level miss handling has a distinct performance disadvantage over the faster but more costly device TLB table walk engine. With kernel-level miss handling, the execution times of larger numbers of queries are increased by amount that is approximately equal to the additional CPU overhead. On the other hand, this scheme still outperforms kernel-based I/O systems by over 30 percent.

Trends towards deeply-pipelined processors with high clock rates and faster system buses do not directly benefit the performance of a kernel TLB miss handler. Deep processor pipelines generally incur higher interrupt overheads despite higher clock frequencies, largely due to the high number of cache and TLB misses and the difficulties involved in branch prediction. Faster system buses support a higher aggregate bandwidth, but do not improve the access latency to main memory or I/O device registers.

#### D. Design Tradeoffs

To compare the implementation options, Fig. 9 shows the scalability of all combinations of request initiation schemes and TLB miss handlers. In this graph, performance is normalized to the system with the best scalability, namely the CSB with a table walk engine. Data points lower than one indicate the relative disadvantage of the other combinations, including kernel-based I/O in the bottom. Note that the Y-coordinates range from 0.6 to 1, to improve readability of the figure.

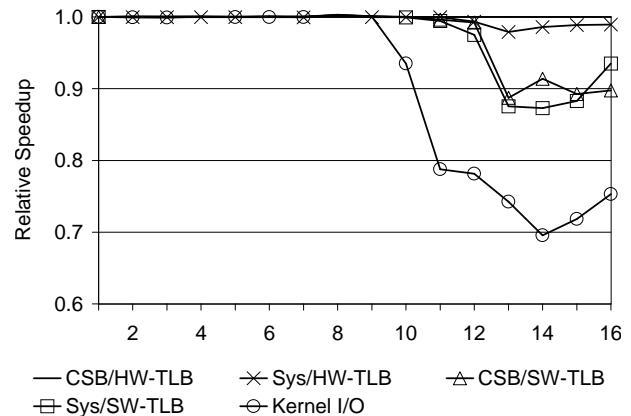


Fig. 9. Normalized Scalability

These graphs clearly indicate that the CSB or other aggressive schemes to support direct user-level access to I/O hardware have a negligible benefit when compared to a more conventional system call. The benchmark query used here results in a sequence of relatively large I/O requests of 16 kbytes each, with a long average latency. Small changes in the overhead to initiate requests have little impact on the overall throughput. As a result, the cost of adding support for atomic device access to the host processor is likely not justified, despite the small additional area requirements of the CSB structure. On the other hand, environments with lower average request latencies may still benefit from a CSB.

Utilizing the host processor to perform on-demand address translation degrades performance significantly. While this scheme avoids the cost of a hardware table walk engine at the device, it places excessive processing load on the host CPU. Interrupt handling is increasingly becoming the bottleneck in many high-performance applications, and current trends towards deeper pipeline and more aggressive speculation will not benefit interrupt performance [15]. Pinning I/O buffer pages in advance eliminates

the need to perform address translation at the device, but requires modifications to the application code to match the new I/O programming model, and can also introduce overhead if different buffers are to be used at different times. On the other hand, in cases where a preallocated buffer matches the application's usage of I/O buffers, this scheme can result in similarly low overheads.

Unfortunately, the impact of the notification scheme cannot be quantified as easily, since there is no kernel-based scheme with semantics equivalent to the user-level notification scheme proposed in this work. Unlike the UIO device with its autonomous address translation capabilities, kernel interrupt handlers cannot access arbitrary user memory due to the page fault risks that can not be handled in an interrupt context. As a first approximation, the cost of conventional interrupts and signaling schemes is at least that of a simple system call, the impact in scalability is comparable to that of the CSB versus system call tradeoff, or slightly greater. From a performance standpoint, such a minor degradation in scalability may not justify the complexity and cost of the user-level notification scheme. On the other hand, the scheme proposed here provides additional flexibility by allowing applications to specify different notification handlers for different requests. This flexibility cannot be easily implemented using existing kernel interrupt mechanisms.

## V. RELATED WORK

User-level networking as a means to reduce communication overhead and improve scalability of parallel and clustered architectures has been studied extensively [9][10][11] [12]. Many of these architectures achieve significant performance improvements by completely eliminating the operating system overhead. However, the commonly used connection-oriented approach restricts applications to preallocated message buffers and requires that the network interface maintain information about each active connection. Pinning pages can increase physical memory pressure and can limit system scalability. Moving the responsibility to maintain per-connection state information, and to multiplex application requests from the operating system to the hardware can lead to expensive implementations that do not scale to large numbers of processes. While the user-level I/O architecture applies the

same principle of bypassing the operating system for improved scalability to the I/O subsystem, it addresses the above shortcomings by not relying on a connection-oriented protocol.

User-level initiation of DMA transactions is one of the challenges in user-level communication and has been studied before [22]. Using shadow addresses to initiate DMA transfers requires that the DMA hardware maintains mappings of shadow to physical addresses, which means the hardware imposes a limit on the number of processes using this feature. Furthermore, all shadow mappings must be kept synchronized with the primary address map, thus increasing the memory management costs in the kernel. Block-transfer mechanisms that are now part of modern multimedia instruction sets [23] address the atomic transfer problem, but do not provide non-blocking flow control from the device to the application. Load-linked store-conditional instructions, on the other hand, provide non-blocking synchronization but do not facilitate data transfer. In addition, these instructions rely on voluntary synchronization by all participating processes.

Device TLBs that help to eliminate the restriction of preallocated communication buffers have also been proposed previously [24] [25]. The UIO device TLB design is most similar to UNet/MM, but adds the option that the UIO device performs page table walks independently.

The concept of network-attached, intelligent devices has been applied to storage devices in the network-attached secure disk architecture [7]. This work builds on these ideas by addressing the operating system overhead that can inhibit clients from taking full advantage of the available bandwidth and concurrency. The InfiniBand distributed I/O architecture combines intelligent, network-attached I/O devices with user-level access to the I/O system [4]. However, the mechanisms for initiating I/O transfers are based on existing user-level networking concepts, and as such inherits their limitations.

Active messages as a parallel programming model [17] have greatly influenced the design of the UIO application interface. However, in this context, message handlers are only invoked at the local node as part of a reply from the remote I/O device. User-level exception handling has been proposed to improve the interaction between the operating system and the application program for garbage collection and other memory-related traps

[16]. This work extends the concept to asynchronous I/O interrupts, and adds hardware support to reduce the cost determining the interrupt cause.

This study is one of the first to systematically evaluate I/O architecture design tradeoffs and to quantify the contribution of the various individual mechanisms. While most previous work provides quantitative results and evaluates design choices, this work benefits from the flexibility of a simulated prototype to investigate the hardware cost versus performance tradeoff.

## VI. CONCLUSIONS

Bypassing the operating system is an effective way to let I/O intensive applications take full advantage of the scalability and bandwidth of networked, distributed I/O subsystems. Architectural mechanisms to provide protected device access, user-space data transfers and light-weight notifications are required to maintain sufficient protection and system integrity. User-level communication mechanisms are commonly adapted to low-overhead I/O architectures. This paper makes two main contributions to the design of user-level I/O architectures. First, it proposes an alternative architecture that extends previous approaches to an inexpensive user-level I/O architecture that does not suffer from the same scalability limitations as previous solutions. Scalability experiments carried out on a simulated prototype confirm the significant performance advantages of user-level I/O architectures over conventional kernel-based I/O systems. The second contribution consists of a detailed design-space exploration. By evaluating individual architectural mechanisms this work provides insight into the significance and contribution of each mechanism. Results show that the majority of the performance improvement can be attributed to the direct user-space data transfers. Using system calls instead of user-level mechanisms to initiate requests has little impact on throughput. Consequently, the hardware cost of the conditional store buffer may not be justified in many cases. While the CSB consumes few silicon resources and does not affect the critical path of the processor core, it does require modifications of the processor chip. On the other hand, performing dynamic address translations via interrupts places a significant burden on the host processor and leads to significant slow downs. The impact of the notification handling

scheme is expected to be comparable to that of the CSB, again suggesting that adding hardware support to the processor core may not be justified.

## ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their helpful comments on an earlier draft of this paper. This work was in part supported by the Defense Advanced Research Projects Agency under agreement number N0003995C0018 and F306029810101, by an NSF Research Infrastructure Grant Number CDA9623614, and by a Graduate Research Fellowship from the University of Utah Graduate School for the academic year 2000/2001.

## REFERENCES

- [1] L.A. Barroso and K. Gharachorloo, and E. Bugnion, "Memory System Characterization of Commercial Workloads," in *25th International Symposium on Computer Architecture (ISCA-25)*. IEEE CS Press, Los Alamitos, Calif., 1998, pp. 3–14.
- [2] J.K. Osterhout, "Why Aren't Operating Systems Getting Faster as Fast as Hardware?" in *Usenix Summer Conference*. Usenix Assoc., Berkeley, Calif., 1990, pp. 247–256.
- [3] TcX AB and Detron HB and and Monty Program KB, *MySQL Reference Manual Version 3.2.1*. [Online]. Available: <http://www.mysql.com/documentation/mysql/>
- [4] *InfiniBand Architecture Specification Release 1.0*, InfiniBand Trade Association, Portland, Ore., 2000.
- [5] L. Schaelicke, "Architectural Support for User-level Input/Output," Ph.D. dissertation, Univ. of Utah, Utah, 2001.
- [6] Y. Zhou et al., "Experiences with VI Communication for Database Storage," in *29th International Symposium on Computer Architecture (ISCA-29)*. IEEE CS Press, Los Alamitos, Calif., 2002, pp. 257–268.
- [7] G. Gibson et al., "A Cost-Effective, High-Bandwidth Storage Architecture," in *8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*. ACM Press, New York, N.Y., 1998, pp. 92–103.
- [8] L. Schaelicke and A. Davis, "Improving I/O Performance with a Conditional Store Buffer," in *31st International Symposium on Microarchitecture (MICRO-31)*. IEEE CS Press, Los Alamitos, Calif., 1998, pp. 160–169.
- [9] M.A. Blumrich et al., "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," in *21st International Symposium on Computer Architecture (ISCA-21)*. IEEE CS Press, Los Alamitos, Calif., 1994, pp. 143–153.
- [10] T. von Eicken et al., "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," in *15th ACM Symposium on Operating Systems Principles (SOSP-15)*. ACM Press, New York, N.Y., 1995, pp. 40–53.
- [11] M. Fillo and R.B. Gillett, "Architecture and Implementation of Memory Channel 2," *DEC Technical Journal*, vol. 9, no. 1, 1997.
- [12] A.M. Mainwaring and D.E. Culler, "Design Challenges of Virtual Networks: Fast, General-Purpose Communication," in *7th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*. ACM Press, New York, N.Y., 1999, pp. 119–130.

- [13] *IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, Intel Corporation, 2004.
- [14] *PowerPC Microprocessor Family: The Programming Environment for 32-bit Microprocessors*, Motorola, Schaumburg, Ill, 1997.
- [15] B. Moore and T. Slabach and L. Schaelicke, "Profiling Interrupt Handler Performance through Kernel Instrumentation," in *International Conference on Computer Design (ICCD-2003)*. IEEE CS Press, Los Alamitos, Calif., 2003, pp. 156–163.
- [16] C.A. Thekkath and H.M. Levy, "Hardware and Software Support for Efficient Exception Handling," in *6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*. ACM Press, New York, N.Y., 1994, pp. 110–119.
- [17] T. von Eicken, "Active Messages: a Mechanism for Integrated Communication and Computation," in *19th International Symposium on Computer Architecture (ISCA-19)*. IEEE CS Press, Los Alamitos, Calif., 1992, pp. 256–266.
- [18] L. Schaelicke and M. Parker, "ML-RSIM Reference Manual," Dept. Computer Science Engineering, Univ. of Notre Dame, Notre Dame, In., Tech. Rep. TR 02-10, 2002.
- [19] V.S. Paj and P. Rangnathan and S.V. Adve, "RSIM Reference Manual, Version 1.0," Dept. Electrical and Computer Eng., Rice Univ., Houston, Tex, Tech. Rep. 9705, 1997.
- [20] M. McKusick and K. Bostic and M. Karels and J. Quarterman, *The Design and Implementation of the 4.4. BSD Operating System*. Addison Wesley Longman, Inc., 1996.
- [21] L. Schaelicke, "L-RSIM: A Simulation Environment for I/O Intensive Workloads," in *3rd Annual IEEE Workshop on Workload Characterization*. IEEE CS Press, Los Alamitos, Calif., 2000, pp. 83–89.
- [22] E.P. Markatos and M.G.H. Katevenis, "User-Level DMA without Operating System Kernel Modifications," in *3rd Symposium on High-Performance Computer Architecture (HPCA-3)*. IEEE CS Press, Los Alamitos, Calif., 1997, pp. 322–331.
- [23] D. Weaver and T. Germond, *The SPARC Architecture Manual Version 9*. PTR Prentice Hall Inc., 1994.
- [24] Y. Chen et al., "UTLB: A Mechanism for Address Translation on Network Interfaces," in *8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*. ACM Press, New York, N.Y., 1998, pp. 193–204.
- [25] M. Welsh and A. Basu and T. von Eicken, "Incorporating Memory Management into User-Level Network Interfaces," Dept. Computer Science, Cornell Univ., Ithaca, N.Y., Tech. Rep. TR97-1620, 1997.



**Alan L. Davis**'s current research interests include rapid system on chip development tools, high-performance low-power embedded system architectures, embedded perception, asynchronous circuits, and VLSI. He received a BS in electrical engineering from MIT, and a Ph.D. in electrical engineering from the University of Utah. He is currently a Professor of Computer Science at the University of Utah. He has also held industrial research positions at Burroughs, Hewlett-Packard, Fairchild Semiconductor, Schlumberger, and Intel.



**Lambert Schaelicke**'s research interests include high-performance computer architecture to system performance evaluation and modeling. As an assistant professor at the University of Notre Dame he directed a research project that developed and implemented a high-speed network intrusion detection system. He also participated in the design and evaluation of a next-generation high-end computer architecture based on merged logic-DRAM technology. He is one of the original developers and maintainers of the ML-RSIM system simulator. He is currently working in the Itanium performance group at Intel corporation. Dr. Schaelicke received a Diploma in Computer Science from the Technical University Berlin, Germany, and a PhD in Computer Science from the University of Utah.

He is one of the original developers and maintainers of the ML-RSIM system simulator. He is currently working in the Itanium performance group at Intel corporation. Dr. Schaelicke received a Diploma in Computer Science from the Technical University Berlin, Germany, and a PhD in Computer Science from the University of Utah.