In-Kernel Integration of Operating System and Infiniband Functions for High Performance Computing Clusters: A DSM Example

Liran Liss, *Student Member*, *IEEE*, Yitzhak Birk, *Senior Member*, *IEEE Computer Society*, and Assaf Schuster, *Senior Member*, *IEEE*

Abstract—The Infiniband (IB) System Area Network (SAN) enables applications to access hardware directly from user level, reducing the overhead of user-kernel crossings during data transfer. However, distributed applications that exhibit close coupling between network and OS services may benefit from accessing IB from the kernel through IB's native Verbs interface, which permits tight integration of these services. We assess this approach using a sequential-consistency Distributed Shared Memory (DSM) system as an example. We first develop primitives that abstract the low-level communication and kernel details, and efficiently serve the application's communication, memory, and scheduling needs. Next, we combine the primitives to form a kernel DSM protocol. The approach is evaluated using our full-fledged Linux kernel DSM implementation over Infiniband. We show that overheads are reduced substantially, and overall application performance is improved in terms of both absolute execution time and scalability relative to an entirely user level implementation.

Index Terms—Hardware/software interfaces, high-speed networks, distributed shared memory, parallel computing.

1 Introduction

Infiniband (IB) [1] is a high-performance system area network (SAN) architecture. IB SANs implement in hardware many legacy software protocol tasks, such as reliability and multiplexing among different connections. New hardware capabilities such as Remote Direct Memory Access (RDMA) are also supported. Consequently, applications can send and receive data at high rates when accessing IB through user-level networking interfaces, e.g., VIA [2]. However, since IB defines its basic primitives in the kernel, kernel subsystems and extensions can also exploit the new hardware. In this paper, we assess the benefits of transferring the communication related functionality of an application into the kernel for high performance IB clusters.

At first sight, this approach seems unnatural. A kernel implementation is harder to develop and debug; it is also less robust, since a single bug can crash the whole system. Past research has shown that integrating the conventional kernel network protocol stack (TCP) with high-level protocols [10] or with the file cache [12] can offer applications performance gains that offset these deficiencies. This is achieved by eliminating excessive memory copies and other overheads during protocol processing. However, there are no such apparent advantages for IB SANs because IB allows zero-copy communication directly from application buffers, and the network protocol is executed in hardware anyhow. Furthermore, direct data transfers from the user level have been demonstrated to substantially improve the performance of systems such as databases [13] and distributed file systems [14].

The above not withstanding, researchers have pointed out that additional specialized APIs would be needed in order to attain the full benefits of SANs [15]. Also, for applications that require close coupling between network functions and those of the operating system, it might be better to bundle these functions in the kernel, thereby reducing the overheads incurred by calling these functions individually. These observations have motivated us to evaluate the integration of SAN access with other OS functions in the kernel. We use a software Distributed Shared Memory (DSM) system as a context.

A DSM system is a runtime environment that emulates shared memory across a computing cluster. A common method for achieving transparent DSM in software entails the use of the operating system's page-protection mechanism to implement an invalidation-based protocol [3], [4]. Access rights to invalidated pages are revoked, and a page fault triggers a protocol action that updates the page.

Page-based DSM protocols vary widely: Some tolerate the coarse sharing granularity induced by the OS/hardware (the system page size) by using relaxed consistency memory models (e.g., Lazy Release Consistency (LRC) [4]), while others employ fine-grain sharing and retain the intuitive Sequential Consistency (SC) memory model [5]. Nonetheless, several common observations hold for these protocols:

- *Each protocol invocation requires at least one system call.* These are usually multiple calls for changing page

- *L. Liss and Y. Birk are with the Electrical Engineering Department, Technion—Israel Institute of Technology, Technion 32000, Haifa, Israel. E-mail: liranl@tx.technion.ac.il, birk@ee.technion.ac.il.*
- *A. Schuster is with the Computer Science Department, Technion—Israel Institute of Technology, Technion 32000, Haifa, Israel. E-mail: assaf@cs.technion.ac.il.*

Manuscript received 7 Aug. 2003; revised 17 June 2004; accepted 22 Oct. 2004; published online 21 July 2005.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number tpds-0134-0803.

protection or for synchronizing with application or communication threads (using semaphores, mutexes, etc.).
- *The communication is inherently asynchronous.* Various request messages (Pages, Locks, Diff applications, Barriers) arrive unexpectedly.
- *Latency is important.* A DSM system is intended for parallel, computation-bound applications. An application thread waiting for a remote response can severely affect the parallel computation. Also, the communication workload comprises mostly small packets, so high bandwidth does not suffice.
- *Application data is frequently transferred among nodes.* This data is not processed by the DSM protocol, and its destination address is known in advance.

Therefore, reducing expensive system calls and user-kernel crossings, high responsiveness to asynchronous events, and efficient data transfer in terms of buffer copies and associated OS protocol processing are all required for high performance.

The introduction of high-performance user-level SANs to DSM systems [6], [7] eliminated OS protocol processing, and reduced extra memory copying through remote memory operations. Responsiveness, however, remains a problem. Constant polling is the most responsive method, but wastes valuable CPU cycles; a separate communication thread requires a context switch to and from it; and catching a signal depends on the receiving task being scheduled. Also, memory-protection system calls are reported to constitute substantial overhead in user-level implementations [8], [9]. Accordingly, DSM systems appear well suited for evaluating the kernel/IB platform.

A particularly interesting related work is SoftFlash [10], which shows how an aggressive kernel implementation can be used to construct an efficient DSM over a system comprising a small number of wide SMP machines. While we share several ideas with SoftFlash (we refer to it where relevant), the network used by SoftFlash does not support the hardware capabilities and standard interfaces that we consider.

We designed and implemented a set of primitives, and used them to construct a highly efficient Linux kernel/IB platform. We then adapted Multiview [5], a fine-grain SC DSM protocol, to this environment, and carried out an extensive comparative performance evaluation of our prototype implementation. We found that common DSM overheads were substantially reduced using our kernel/IB platform: Response latency for asynchronous events improved by 33 percent relative to a user-level implementation, and changing page protections for large page groups performed an order of magnitude better than conventional system calls. These improvements enabled our kernel/IB DSM system to reduce application execution time by up to 23 percent relative to a corresponding VIA/IB implementation. In addition, our system scales better than the same DSM protocol implemented over a dedicated hardware VIA platform (ServerNet-II).

The availability in the kernel of Infiniband's software interface enabled us to integrate network and operating system functions efficiently, which resulted in fewer user-kernel crossings, less overhead in accessing OS functions, and better control over the scheduling of network related events. Such integration can be used to implement optimized high-performance APIs for additional application domains.

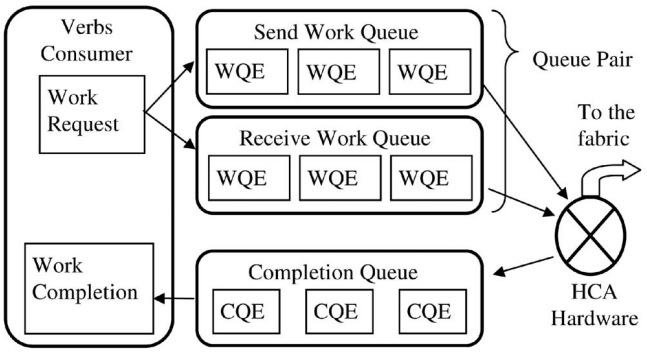

Fig. 1. Infiniband queue structure.

The remainder of the paper is organized as follows: In Section 2, we briefly review Infiniband and Multiview. Our kernel/IB platform is presented in Section 3. The DSM protocol adaptation is discussed in Section 4. Performance results are summarized in Section 5, and Section 6 presents a discussion and concluding remarks.

2 Background: Infiniband and Multiview

2.1 Infiniband

Infiniband is a switch-based serial I/O interconnect architecture that provides low-latency, high-speed communication. Among its main features are 2.5/10/30Gb/s link speeds, connection-based and connectionless communication modes, unreliable as well as reliable services, and support for provision of quality of service, all implemented directly in hardware. IB defines two classes of end-point devices:

- *Host Channel Adapters (HCAs)* are used for connecting computing nodes. HCAs must support the IB Verbs interface [1, vol. 1, ch. 11], which defines the functions provided to the node by the channel adapter.
- *Target Channel Adapters (TCAs)* are used for connecting I/O devices. The interface between the interconnect and the target device is not specified.

For computing clusters, we focus on HCAs.

The Verbs interface (which is not a formal API) defines the semantics for utilizing various HCA resources (Fig. 1). The basic communication end-point abstraction is the Queue Pair (QP), which consists of a Send Work Queue and a Receive Work Queue. Each queue must be associated with a Completion Queue (CQ). Multiple queues (even from different QPs) can be associated with a single CQ. A Verbs consumer (any entity that makes use of the Verbs abstraction) posts work requests (WR) to the work queues, which are then processed asynchronously by the HCA hardware.

When a QP is configured for Signaled Completions, completed WRs always insert a Completion Queue Element (CQE) into the appropriate CQ. Alternatively, a QP can be configured for Unsignaled Completions: In this case, a successfully completed WR that was posted to the **Send**

Work Queue does not generate a CQE unless it was explicitly requested to do so. A Verbs consumer can poll a CQ for completions, or request Completion Notification for a certain CQ when the next CQE is inserted.

IB defines two data transfer modes:

- *Message-passing (channel semantics)*. Data is sent using Send-WRs, and its destination in remote memory is determined at the receiver by posting in advance corresponding Receive-WRs.
- *Remote Direct Memory Access (memory semantics)*. The initiator specifies memory locations at both ends, and memory is either read or written according to the corresponding RDMA-WR (Read or Write). The target node's processor is not involved in the data transfer.

All communication buffers are referenced using virtual memory addresses. To guarantee direct, safe access by hardware, these buffers have to reside in registered virtual memory regions that are pinned to physical memory (fixed virtual-to-physical mappings).

Using the Verbs, operating systems can implement software interfaces that enable applications to use IB directly. The Verbs can also form the basis for kernel primitives that expose IB to operating-system subsystems and extensions.

2.2 The Multiview DSM Protocol

Multiview is a technique for achieving subpage sharing granularity in order to mitigate the "false sharing" problem. It was first implemented in the Millipage system [5]. Consider two variables that reside in the same physical page. By mapping two virtual pages to the same physical page, each variable can be accessed through a different virtual page, enabling hardware protection for a shared variable that is smaller than the system page size. If access is attempted only to the variables associated with such a virtual page, we get in effect a smaller page to which we refer as a "minipage."

Our Multiview DSM implements a thin sequential consistency protocol that consists of three entities: the requestor (retrieves the required minipage on behalf of a faulting process), the manager (holds the state information of all minipages in the system and manages page requests), and the server (responds to manager requests for protection changes and minipage transfers). The manager is statically distributed (with respect to minipages) in a round-robin fashion (based on minipage identifiers).

A request is triggered by a page fault and forwarded to the manager. After handling previous requests for the same minipage, the manager sends invalidation and page transfer notices to one or more servers (on nodes currently holding a valid copy of the page), which notify the requestor once they complete their handling. After the requestor receives all notifications and a possible minipage update, it sends an acknowledgement to the manager and resumes the faulting process. Page faults can take two or three hops (excluding the final acknowledgement), depending on whether the manager node is also the requestor, the server, or neither one of them. The protocol is single-writer.

3 Our Kernel/IB Platform

One of the benefits of kernel code is complete control of the timing and execution contexts of various tasks in the system. Since responsiveness was recognized as a major remaining source of overhead in software DSMs, we decided to provide to the protocol a kernel platform[1] tailored for fast dispatching of asynchronous events in interrupt context [10]. The platform consists of three groups of primitives: asynchronous-event-handling primitives that enable a registered handler to be invoked in interrupt context; memory primitives that allow interrupt-context page-protection changes; and communication primitives that abstract the low-level IB Verbs interface. While the last group does not relate directly to a kernel implementation, it offers efficient buffer management and flow control. Next, we detail these primitives, their associated Infiniband abstractions, and the kernel mechanisms that we used. Note that, although these primitives were developed with a DSM in mind, they can also serve as building blocks in implementations of efficient operations for other applications.

3.1 Asynchronous-Event Handling

DSM control messages arrive from the network unexpectedly, and must be handled with minimal latency. Furthermore, the protocol may want to be notified whenever operations such as RDMA data transfers complete. IB addresses the notification of such asynchronous events by enabling a Verbs Consumer to register a handler function and request completion notification for each CQ. Once such notification is requested, the next CQE inserted into that CQ triggers the registered handler.

In our platform, the completion notification is delivered as part of an interrupt service routine (ISR). Therefore, we have three possible execution contexts for calling the protocol handler. The first is within the ISR itself, which would provide the lowest possible latency. However, this context imposes several limitations on the called code: It cannot sleep (or call any OS service that may block), spinlock, or access user space. Furthermore, processing should be extremely fast because other pending interrupts that need immediate attention may be disabled. The second is a software interrupt, such as the Linux Task Queue mechanism [17]. While fast task-queues also execute in interrupt context[2] (and, thus, impose similar restrictions to ISRs), they allow longer processing because they take place at a "safer" time (interrupts enabled) than ISRs. The last is a process, which does not impose any of the aforementioned restrictions. However, scheduling a process can take considerable time and increases overhead.

To achieve our goal of minimizing response time while maintaining correctness and system stability, we follow the well-known principle of optimizing for the common case: We initially attempt to process the CQ in the ISR; if the handler indicates that processing cannot proceed in this context (because of a taken resource or many accumulated

1. All our kernel extensions were implemented as loadable driver modules. For convenience, we also customized the kernel to export additional symbols. Otherwise, the kernel is unchanged.

2. In Linux, "Interrupt-Context" refers to any execution context that is not related to a process. Examples include ISRs, Bottom-Halves, certain Task Queues, and Tasklets.

messages), we continue CQ processing in a process context. Note that in contrast with traditional OS design, we do not defer interrupt-context processing to a software interrupt. The sensibility of this approach will be discussed in the next section that describes the protocol adaptation.

3.2 Efficient Page Protection

In our DSM, page-protection changes are a common operation in asynchronous entry points of the protocol. However, they involve acquisition of semaphores and locks, expensive data structure manipulation and often flushing the TLB. As a result, changing page protections cannot be done in interrupt context using the normal system call implementation. Furthermore, page-protection system calls have been reported as a major source of overhead for DSM systems. Therefore, we decided to implement a unique kernel manager for virtual memory areas dedicated to DSM memory, which allows changing page protections in interrupt context (in the common case), and reduces much of the overhead incurred by the system-call implementation.

Our memory manager presents DSM memory to the Linux OS as a single Virtual Memory Area (VMA) [17]. Unlike the Linux memory manager, however, we do not maintain homogenous page protections for all pages in the VMA. Rather, we mark the VMA as inaccessible to the application, and allow arbitrary protections for each page in the VMA page tables. This scheme has several desirable properties: It eliminates the expensive VMA management during protection changes; it prevents VMA "explosion" due to consecutive pages with different protections; the application can access any page with matching protections (in the page table) as required; and, access to a page with conflicting protections will always generate a segmentation-violation signal, which can trigger a corresponding DSM protocol action.

Eliminating VMA maintenance reduces all the blocking operations required to change the page tables down to a single lock. So, manipulating the page tables is possible in interrupt context whenever this lock is not taken. TLB flushing requires acquisition of a lock, and optionally flushing the TLBs of other processors by sending them an interprocessor interrupt (IPI). Because the completion of these IPIs is detected by short-term polling, careful inspection reveals that TLB flushing is also possible in interrupt context if a few simple conditions hold and the lock is not taken. Therefore, by providing to the protocol a proper predicate, we allow it to change page protections during interrupt processing in a safe manner.

As a final optimization, our memory manager also supports changing any group of pages to any set of protections. Here, the TLB is flushed only once after all modifications to the page tables were applied. This can offer substantial improvements in SMP machines [10].

Remark. In our implementation, DSM memory is always pinned to physical memory (see Section 4.2). However, the forgoing page-protection scheme is not necessarily limited to pinned memory because it only involves changing the page permission bits, rather than the "page present" bit or the OS swapping policy. Furthermore, the implementation does not affect other memory areas, nor does it require any modifications to the kernel.

3.3 IB Abstraction Layer

While the data-integrity needs of our system map nicely to IB's Reliable Connection service, WR processing and its associated buffer management are low-level and complex. Therefore, we decided to provide the protocol with an abstraction layer that provides simple and efficient point-to-point communications.

Upon initialization, we open a reliable connection between every two nodes in the cluster. Since all connections are symmetric in our system, and an asynchronous message can arrive from any node at any time, we chose to serve all WQs with a single CQ. We allow the protocol to register a single completion handler, and handle general CQE processing (dequeuing CQEs, requesting notification, and polling remaining CQEs) in a centralized manner (at each node). Moreover, the use of a single CQ and at most one outstanding completion notification request jointly provide for atomic handling of events, so less locking is needed when accessing shared data. This setting has good scalability because both the connection QPs and the aggregation of all completion events into a single CQ are implemented in dedicated hardware (modern HCAs can support up to 16 million QPs).

Send buffers are allocated on behalf of the protocol in response to a buffer reservation request. After the protocol signals that the buffer can be sent, a corresponding Send WR is enqueued, and the buffer is reclaimed upon completion. To ensure resource reuse while maintaining acceptable performance, we provide an efficient scheme for fast completion detection as follows: We configure the QPs for Unsignaled Completions to prevent completion-processing overhead for every posted WR. In addition, we decouple the detection of completed WRs from explicit signaling requested by the protocol: When the protocol requests a signaled completion, a notification is passed as soon as the corresponding CQE is dequeued; also, a signaled completion is requested occasionally for cleanup purposes as necessary, but the protocol is not notified. For RDMA operations, we handle WR processing and notification similarly, but buffers are better left to the control of the protocol.

In many parallel systems that do not follow a client-server paradigm, such as our DSM, the number of in-flight messages can be bounded. Moreover, this bound is often reached in our system because unbalanced communication is common in certain application phases. (For example, whenever all threads access new data following a sequential phase.) Finally, our protocol uses message passing only for short control messages, so the maximum buffer space for in-flight messages cannot be very large. Therefore, we decided to allocate the maximum number of receive buffers to every receive queue, thereby eliminating the need for application-level flow control and achieving efficient delivery for every message. (While the flow control mechanism itself does not add much overhead, a window size that is not matched to the application's bursty traffic pattern could pause the sender often, wasting valuable CPU cycles for polling or responding to an asynchronous event to complete the send operation.) The scalability of our approach is limited only by the physical resources in each node (memory and WQ

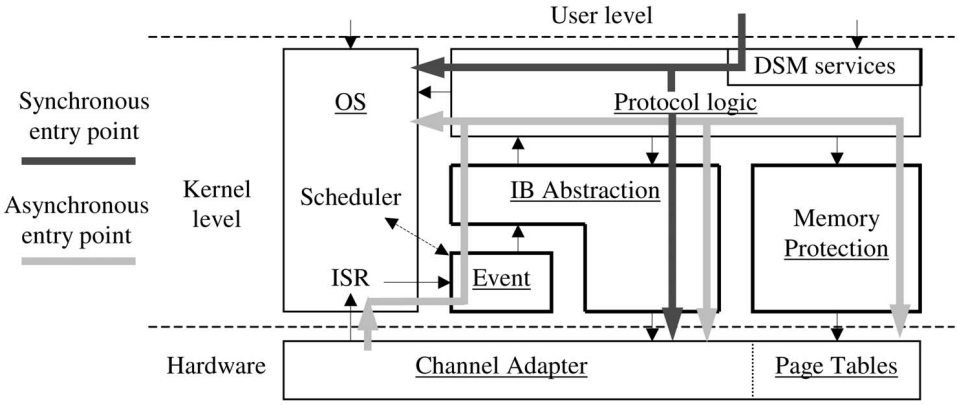

Fig. 2. Our Kernel-IB system control path.

sizes). Consequently, flow control can be avoided altogether when the bound is reasonable, or used with a window size that is sufficiently large to capture common-case traffic. The protocol is given access to receive buffers only during a handler call (as in FM [16]), allowing the buffers to be consumed and freed in a simple round-robin fashion.

4 DSM Protocol Adaptation

In order to fully utilize the kernel/IB platform, we decided to implement the entire protocol in kernel code. This reduces user-kernel crossings to a minimum, as a user process issues a system call only when it has to block (e.g., after suffering a DSM page fault). Furthermore, all the protocol's asynchronous entry points can be implemented in interrupt context based on our asynchronous-event handling and memory primitives, which cuts latency and eliminates context switching due to network events. Finally, various components of the protocol can be synchronized efficiently without requiring expensive system calls. We next detail the control flow of the protocol, followed by implementation notes.

4.1 Control Flow

In order to implement asynchronous entry points in interrupt context, we defined a clean separation between tasks performed by the synchronous and asynchronous portions of the coherence protocol:

- Synchronous entry points (application threads) handle all request bookkeeping tasks. These tasks access coherence metadata (an efficient internal representation of minipage protections) only for reading.
- Asynchronous entry points (message and WQ completion handlers) handle only page protection tasks and coherence metadata manipulation. Protections are granted when a reply to a page request arrives, and are revoked when serving invalidation requests.

Based on this separation, we adjust the protocol to our platform (see Fig. 2). Synchronous entry points are executed in the context of application threads. Asynchronous entry points are message handlers. Initially, the event-handling primitives forward an event indicating a new message to the IB abstraction layer in interrupt-context. The IB abstraction layer, in turn, commences CQ processing and calls the appropriate message handlers. If CQ processing cannot proceed in this context, it is resumed in process context. The control flow of the protocol is described below.

After suffering a page-fault, an application thread enters the kernel as a requestor, and competes for exclusive access to bookkeeping information. After access is granted, it inspects coherence metadata in order to determine whether a new page request message needs to be generated. If the page is already available, the requestor just returns. If an outstanding request will also satisfy the new one, the requestor is added to a proper OS wait queue after incrementing a usage count. Otherwise, a new message is sent to the appropriate manager, and the requestor is added to the OS wait queue assigned for this request.

Both the manager and server nodes receive incoming asynchronous messages, and respond by sending a new message during the execution of the message handler. The server may also change page protections, update coherence metadata, and initiate a minipage transfer.

When a reply signals the completion of the request at the requestor node, the message handler performs necessary page-protection changes, updates coherence metadata, and signals the corresponding wait queue. After reacquiring exclusive access, a woken up requestor decrements the request usage count and returns. Resources can be released and reused once the usage count drops to zero.

This flow achieves two goals. First, asynchronous entry points are indeed suitable for execution in interrupt-context: Blocking operations are not required; sending a short control message or initiating a zero-copy data transfer using RDMA (see implementation notes below) amounts to signaling the IB hardware that a new WR is available; changing page protections is possible using our memory primitives; and finally, waking a process is a main function of interrupt handlers. Moreover, hardly any computation is involved, so asynchronous entry points can actually be implemented during the ISR itself without disabling interrupts for too long. Second, no severe data races between interrupt and process contexts will occur. Since the synchronous entry points closely follow the monitor synchronization paradigm, and asynchronous entry points

TABLE 1
Basic OS/Infiniband Operation Latencies

Operation	Latency [μs]
Interrupt delivery	10
Page fault cost (fault to signal handler)	5
System call invocation	0.7
DSM page protection primitive (single page)	0.9
Post Work Request (software overhead)	2
RDMA-W one-way latency	8-9
SEND one-way latency	22-23
RDMA-R (completion detected by polling memory)	9
RDMA-R + CQ update	30
Poll (empty) completion queue	7

are executed atomically, the only feasible data race is a read-write data race, whereby a process reads coherence metadata while an interrupt handler updates it. However, this does not affect the correctness of the protocol: When an interrupt signals that a page is available, we prevent a new requestor from joining the corresponding wait queue by using Linux's `wait_event` primitive (which checks the sleep condition after the process is put "half to sleep" [17]); when a page is "stolen" by an interrupt handler while a requestor is released, the requestor will simply generate another page fault (the normal behavior).

4.2 Implementation Notes

Application data movement in DSM systems is well matched to IB's memory semantics because data is transferred to well-known virtual addresses in memory. Furthermore, memory semantics eliminate data copies between the application's address space and dedicated communication buffers. (This has been shown to improve DSM performance by up to 15 percent [7].) Protocol control messages such as page requests and lock acquisitions, which generally require processing on the remote node, are better matched to channel semantics. Therefore, we decided to implement data movement and control messages by RDMA-W and Send WRs, respectively. Since IB requires all virtual memory regions that participate in communication to be pinned in physical memory, this decision implies that the application problem size is limited to the amount of physical memory. While some applications can achieve good speedups on DSMs only if the problem fits in physical memory, others can clearly benefit from lifting this restriction, by exploiting locality of reference. However, for the purpose of this research, static pinning suffices. (If the problem size exceeds that of physical memory, a policy for dynamic memory registration can be employed, or communication buffers can be used instead [7].)

Our DSM also supports barriers and locks. These are simple operations in a sequential consistency DSM, which do not involve the coherence protocol. We implemented them in a straightforward manner using channel semantics, with a similar control flow.

Finally, in order to reduce latency further, we experimented both with selective polling (replacing interrupts with polling whenever a process is expecting a response and has nothing else to do) and fetching data with RDMA-R when the remote processor need not be disturbed. This situation arises during Read page-fault handling, when the requested page is currently shared and not available in the manager node. Thus, the requestor can pull the page from a server node containing a valid copy without changing its protections.

TABLE 2
Round-Trip Time for Different Receive Contexts

Polling	ISR	Task Queue	Process
45μs	60μs	70μs	90μs

5 Performance Evaluation

In this section, we evaluate the performance of our implementation. All experiments were performed on a cluster of twelve SMP PCs, running the Linux 2.4.18 operating system. Each machine has two 733 MHz P-III processors, 512 KB L2 cache, 512 MB memory, and a 32-bit, 33MHz PCI bus. Every node employed a first-generation multiport IB card (Mellanox MT21108 [18]), which provides IB switch and TCA[3] functionality. The device also has limited HCA support in the form of a dedicated DMA engine. (We implemented a subset of the HCA Verbs interface, achieving full hardware performance for data transfers.) Basic OS/IB operation latencies are reported in Table 1. Node to node bandwidth varies from 52 MB/s for 256 byte WRs up to 103 MB/s for 4 KB WRs.

5.1 Applications

Our application suite comprises eight applications: Water-nsquared (Water), LU-contiguous (LU), and Barnes-Hut (Barnes) from SPLASH-2 [19]; Integer-Sort (IS) from the NAS parallel benchmarks [20]; Successive Over-Relaxation (SOR) and the Traveling Salesperson Problem (TSP) from the Treadmarks [21] benchmark applications; N-Body (NBody) and N-Body-Write (NBodyW) are computation kernels that imitate N-body applications [22]. See Table 3 for the input data sets used for each application.

5.2 Kernel/IB DSM Performance

Initially, we tested some aspects out our platform in isolation, because they can be useful to other DSM systems or application domains. First, we evaluated the handling of asynchronous events inside interrupt handlers, and compared it with task queue handling and with passing a signal to a user-level handler (resembles VIA implementations) using a simple ping-pong test. Polling is added for reference. As shown in Table 2, kernel handling performs substantially better than user context (it reduces latency by 33 percent), with some advantage to ISR over Task Queues.

Next, we compared the performance of our page-protection primitives with that of the `mprotect` system call. For changing the protection of a single page, our memory primitives achieve roughly half the latency of the OS implementation. Although our DSM changes the protections of one page at a time, we also evaluated the time it takes to change the protections of an arbitrary page-group (this can be

3. As a TCA, the device can be used to transparently translate PCI cycles to IB packets and vice-versa.

TABLE 3
Benchmark Application Data Sets and Runtime Statistics

Application	Input Set	Shared Memory Size	Page Faults / sec	Barrier / sec	Lock / sec	RDMA Write Bandwidth	RDMA Read Bandwidth	Send Bandwidth
Barnes	16K bodies	3.2 MB	4256	1.8	0	3.9 MB/s	174 KB/s	228 KB/s
IS	2^{24} numbers x 10	2 KB	138	76	0	18 KB/s	0	19 KB/s
LU	1024x1024	8.3 MB	129	35	0	0.8 MB/s	625 KB/s	21 KB/s
Nbody	8K bodies	0.52 MB	1089	4.3	0	1.8 MB/s	44 KB/s	81 KB/s
NBodyW	8K bodies	0.52 MB	826	2.3	0	1.2 MB/s	23 KB/s	50 KB/s
Ocean	1026x1026	238 MB	647	88	76	0.5 MB/s	0	84 KB/s
SOR	4096x4096x10	67 MB	21	11	0	83 KB/s	0	4.3 KB/s
TSP	19 Cities Tour	1.4 MB	313	0	28	1.3 MB/s	26 KB/s	30 KB/s
Water	512 Molecules	0.3 MB	882	19	857	2.7 MB/s	482 KB/s	157 KB/s

useful for prefetching or LRC DSMs). Even for small page groups of eight pages, our primitives outperform the required multiple `mprotect` system calls by an order of magnitude, mainly due to the single TLB flush required in our implementation.

In order to evaluate the overall contribution of our scheme, we compared our implementation (Kernel-ISR) with a simulated VIA implementation (VIA-sim) on a cluster of eight nodes, utilizing two threads per node. The simulation was conducted by incorporating the following changes in our implementation:

- Whenever a completion notification is issued, the interrupt handler pushes a signal to the application, which in turn passes control to the driver for receive processing.
- Before each protocol action that would require a system call, we insert a 1μs delay (slightly longer than the simplest system-call latency).
- We perform memory protection changes by calling the OS implementation (`sys_mprotect`) rather than using our memory primitives.

Otherwise, the system is unchanged. This comparison is quite accurate because it essentially captures the differences between kernel and user-level implementations, while leaving the hardware, IB software, and DSM protocol strictly identical. In addition, we also evaluated an additional kernel implementation that executes asynchronous events in task queues (Kernel-TQ) for reference.

Compared with VIA-sim, Kernel-ISR demonstrated considerable performance gains for some applications. For example, execution time was reduced by 23 percent in TSP, 20 percent in Barnes, and 7 percent in NBody-W. Other applications presented smaller improvements.[4] Kernel-TQ demonstrated roughly half of these improvements over VIA-sim, with one exception. In TSP, execution time almost doubled in Kernel-TQ with respect to VIA-sim.

A detailed examination of TSP execution time revealed that page faults in Kernel-TQ cost twice as much as in VIA-sim, and as much as four times more than in Kernel-ISR. Combined with the race for shared locks in this application, lock acquisitions result in 50ms wait times, which dominate the total execution time. We explain this phenomenon by the nature of task queue invocations: The *Immediate* task queue (on which we based the Kernel-TQ implementation) is run either after system calls or after scheduler invocations [17]. In TSP, synchronization is maintained using several shared locks, and local computation is relatively uninterrupted by page faults or system calls. Consequently, the Task Queues are examined infrequently, resulting in poor responsiveness to asynchronous requests and contention for the shared locks. Note that the user process handles this situation better because of the high responsiveness of the Linux signal-handling mechanism.

We also tested the effects of the system load (an additional load of a single CPU-intensive process was run on each node to simulate occasional interference by other users of a cluster). In this configuration, the gap between Kernel-ISR and VIA-Sim increased considerably in all applications, indicating that the responsiveness of user-process message handling is much more sensitive to load.

5.3 Optimizations

In addition to the evaluation of our baseline kernel implementation, we also quantified the effects of two optimizations to our platform mentioned in Section 4.2, namely, selective polling and using RDMA reads. (These optimizations are not limited to a kernel implementation, but are a natural extension of our system.)

The introduction of selective polling reduced page fault latencies by 3-7 percent. Note that in a typical 3-hop page fault, only the final receiver polls, unlike the ping-pong test summarized in Table 2. (Although other nodes could be polling at the same time, this does not occur frequently.) Overall, application performance improved by up to 6 percent. However, when the number of application threads per node was increased beyond the number of CPUs in each machine, polling only degraded performance.

Using RDMA reads whenever possible in read faults actually **increased** read-fault latencies by 2-3 percent on average, mainly due to the relatively slow CQ update for RDMA-Rs in our architecture. However, the total execution time of most applications improved slightly. The contribution of using RDMA reads in our system is thus mitigation of the interference of remote read requests with the computation of the node providing the data (recall that all nodes play both roles at different times).

4. The measurements for these were conducted on four nodes.

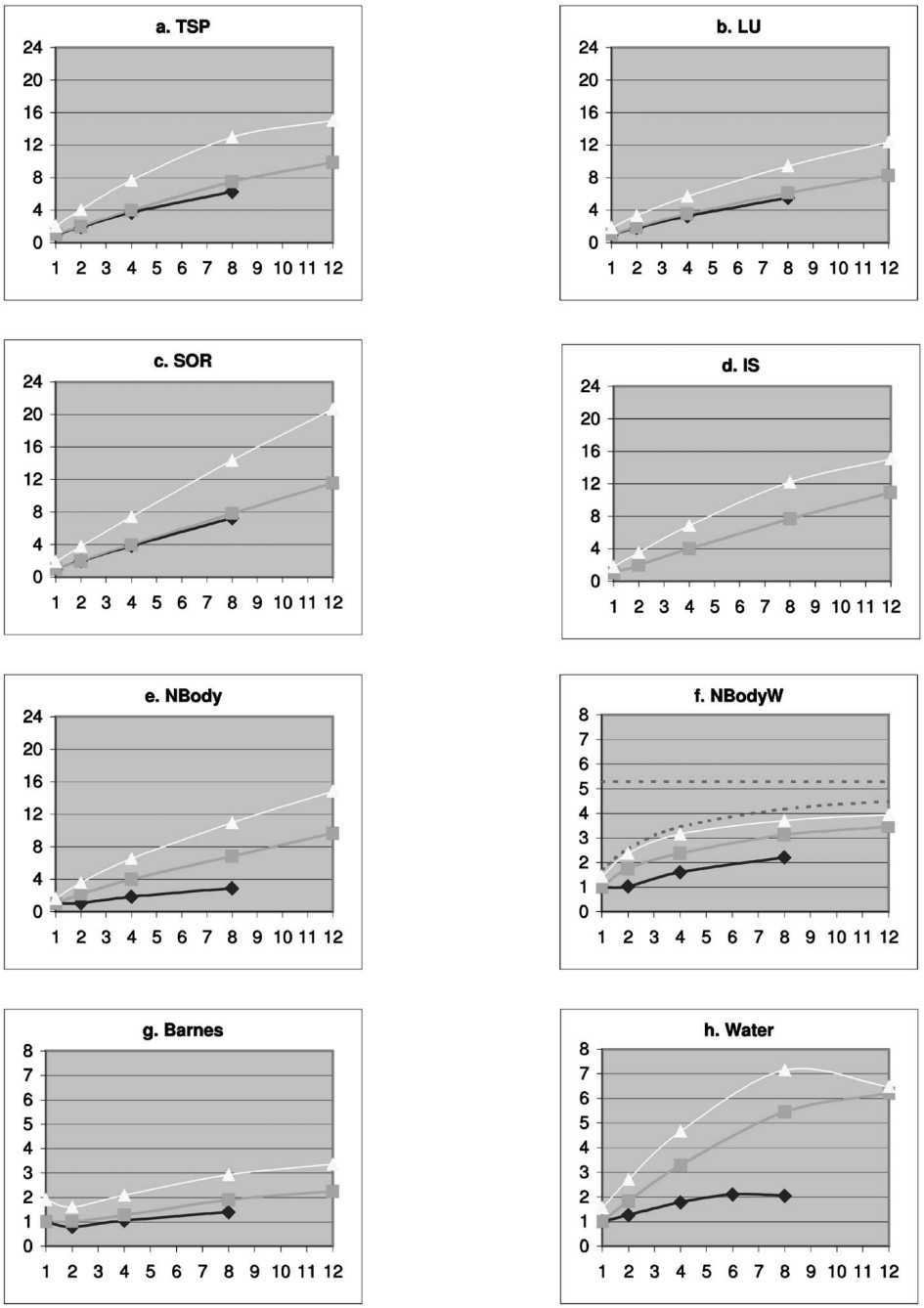

Fig. 3. Application speedup versus number of nodes. (A single node with one processor is used as the baseline.) Legend: Diamond—VIA/ServerNet, single thread per node; Square—Kernel/IB, single thread per node; Triangle—Kernel/IB, two threads per node; Dashed line—NBodyW Theoretical curve and limit.

5.4 Application Scalability

We evaluated the scalability of our implementation using eight benchmark applications. We also compared the speedup with our implementation to that of a true VIA implementation on the same computing nodes, identical benchmark code, and a similar DSM protocol. The VIA implementation ran over Windows NT with the ServerNet-II VIA interconnect, whose performance exceeds that of our hardware (13μs send latency, 180MB/sec data rate). Despite the similarities, the VIA/ServerNet speedups are provided mainly in support of a scalability comparison. Nonetheless, the results do provide a strong indication regarding the relative execution times and overheads of the two implementations.

The speedups relative to a sequential execution are reported for all applications in Figs. 3a, 3b, 3c, 3d, 3e, 3g, 3g, and 3h. Recall that our nodes are dual-SMP machines, so an execution with two threads per node utilizes twice as many CPUs as an execution with a single thread per node. See Table 3, Fig. 4, and the Appendix for runtime statistics and execution-time breakdown for each application.

Relatively "well behaved" applications (SOR, LU, IS, and TSP) achieve good speedups with both implementations. Nevertheless, our kernel/IB platform consistently exhibits

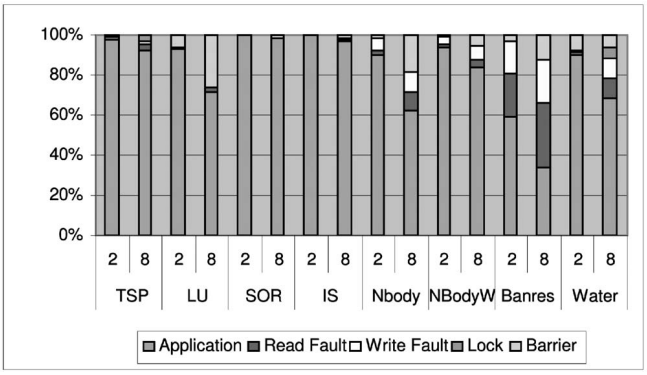

Fig. 4. Normalized execution time breakdown.

better scalability, which is most noticeable in TSP. In more demanding applications such as Water, Nbody, NbodyW and Barnes, the scalability advantage of our kernel/IB implementation over VIA is even more pronounced.

The combination of a relatively large number of page faults and extremely frequent synchronizations limits the scalability of the Water benchmark. The VIA implementation exhibits poor speedups and does not scale beyond six nodes. The kernel/IB implementation, in contrast, still achieves acceptable speedups on a cluster of 12 nodes and a single thread per node. However, with two threads per node, our system does not scale from eight to 12 nodes. This is because of a computation imbalance that results in long barrier times.

Despite a high page-fault rate, the NBody application manages to get a speedup of 15 on 24 processors on our architecture. NbodyW, in contrast, performs much worse due to a sequential phase that exhibits a mismatch in sharing granularity (a single thread reads and writes all bodies): As the number of processors increases, this phase dominates the execution time. We added for reference a theoretical curve (for two threads per node) based on the execution time on a single node and perfect speedup of the parallel phases, as well as the upper speedup limit. For both of these applications, the VIA implementation demonstrates inferior scalability.

Barnes is the most demanding application in terms of page faults because of the high degree of true sharing. This, in turn, introduces imbalances that result in long barrier times that affect both implementations.

For most applications, our system scales similarly while running one or two threads per node. This can be attributed to the small footprint of asynchronous-event handling in our system, which does not involve thread-switching overhead within the same CPU.

Remark. The speedup differences are more noticeable than those observed relative to our VIA simulation in the previous section. This points to the conservative approach taken in the simulation, and strengthens the confidence in our findings.

6 Discussion and Conclusions

In this section, we elaborate on some of the general lessons learned from our implementation, discuss topics for future research, and point out insights that may be applicable beyond DSM systems.

6.1 DSM Conclusions and Opportunities

Our kernel/IB platform substantially reduced common DSM overheads. ISR event handling reduces the response time for asynchronous messages by 33 percent relative to user-level signal handlers, and our memory primitives outperform the corresponding system calls for changing the protection of page groups by an order of magnitude. While the full benefits of our memory services were not realized in our protocol (only single-page groups were used), we expect them to substantially improve the performance of DSM protocols that require multiple instantaneous page-protection changes (e.g., RC protocols and adaptive-granularity SC protocols [22]). We have shown how a high-level protocol can be split between interrupt and process contexts without introducing harmful data races or compromising other OS activity.

Our kernel/IB DSM system performs up to 23 percent better than a corresponding VIA/IB implementation. Our system also scales better than the same DSM protocol implemented over a dedicated hardware VIA platform (ServerNet-II). As anticipated, applications that exhibit a high computation-to-communication ratio and already achieve good performance on DSM systems benefit only marginally from our platform. Likewise, the performance of applications with poor locality and fine-grain access patterns (such as FFT computations) will remain low. However, there remains a large class of applications that exhibit fine-grain sharing, which may benefit substantially from the kernel/IB platform. For example, the NBody and Water applications more than doubled their scalability compared to the VIA/ServerNet implementation mentioned in Section 5.4.

Infiniband is well matched to the communication needs of DSM systems. Its built-in flow control, reliability, and RDMA capabilities eliminate the need for processing in the majority of the data transfers. We found the main contribution of RDMA reads to be reduced interference with remote nodes, and expect it to be more noticeable for larger clusters, especially for unbalanced page requests among nodes. Furthermore, atomic operations (which were not supported by our first generation IB hardware) can drastically reduce the number of remote CPUs that must be interrupted to process a protocol action. With fully functional second-generation HCAs, better hardware performance will increase the detrimental effects of software overhead. Therefore, we expect our findings to be even more relevant to future hardware.

Finally, our approach can be extended to implement a completely synchronous sequential consistency system on hardware platforms that can trigger TLB invalidations from I/O devices: Necessary locking could be achieved by atomic operations, and page protections could be changed by manipulating the page tables using RDMA and flushing the TLB remotely. (A DSM that eliminates asynchronous protocol processing using special support in the network interface card has been demonstrated in [8], but it presents a Release Consistency model.) We believe that such an implementation can reduce all overheads in the system

dramatically because it replaces the distributed processing on behalf of a page request with pipelined IB requests.

6.2 Beyond DSM

The mechanisms developed in this work have broad applicability. Our IB abstraction layer provides send-receive semantics that are both easy to use and efficient (0-copy and minimal processing overhead). By separating an application's communication traffic to data and control, application-level flow-control can be avoided for control information, as long as its size is inherently bounded by the application semantics. As for data, performing bulk transfers using RDMA relieves software overhead. However, statically pinning all memory for these transfers is clearly not a solution for multitasking environments. Implementing efficient kernel primitives for dynamic memory registration is a topic for future work.

High performance communication alone does not suffice for low-latency message handling—the responsiveness of the receiving context plays an important role as well. For systems that demand predictable low-latency responses, the ability to generate a response during interrupt handling offers a good solution. For applications that require more processing time, the commonly used Linux Task Queue mechanism offers comparable average responsiveness. However, it has less predictable response times and is more sensitive to load—in some runs we measured an average response time of more than 300μs.

The availability in the kernel of Infiniband's software primitives enabled us to integrate network and operating system functions efficiently. This approach resulted in fewer user-kernel crossings, less overhead in accessing OS resources, and better control over the scheduling of network related events. For certain functions, these gains can offset the deficiencies of a kernel implementation, even when using SANs with rich hardware features and user-level capabilities such as Infiniband.

Finally, note that our approach does not necessitate applications to be implemented in the kernel. Rather, integration of OS and network functions in the kernel can provide high performance to applications through an appropriate user-level API. For example, SAN functions can be combined with file-cache management to implement storage APIs (for Web and File Servers), integrated with the scheduler (for Remote-execution/Process-migration facilities), and more. Such APIs can close remaining gaps between software and hardware interfaces, and enable systems to attain the full benefits of today's high-performance hardware.

Appendix

Table 3 presents the input set size and runtime statistics for each benchmark application. The statistics were gathered from a single node in a parallel computation consisting of eight nodes.

The normalized execution-time breakdown for all applications in our suite is shown in Fig. 4. (The times reported are measured from user level and do not take into account asynchronous handling time.)

The measurements were taken on node 0 only, for two and eight-node configurations utilizing a single thread per node. Note that in Barnes and NBodyW, node 0 executes a sequential phase. Therefore, average barrier times for other nodes will be substantially longer.

Acknowledgments

The authors are grateful to Mellanox Technologies Inc. for providing the required Infiniband hardware and related technical support.

References

[1] Infiniband Trade Assoc.—Infiniband Specification, http://www.infinibandta.com/, 2005.
[2] Virtual Interface Architecture Specification, http://www.viaarch.org/, 2005.
[3] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. Computer Systems,* vol. 7, no. 4, pp. 321-359, Nov. 1989.
[4] P. Keleher, A.L. Cox, and W. Zwaenepol, "Lazy Consistency for Software Distributed Shared Memory," *Proc. 19th Ann. Symp. Computer Architecture,* pp. 13-21, May 1992.
[5] A. Itzkovitz and A. Schuster, "MultiView and Millipage: Fine-Grain Sharing in Page-Based DSMs," *Proc. Conf. OS Design and Implementation,* 1999.
[6] M. Banikazemi, J. Liu, D.K. Panda, and P. Sadayappan, "Implementing TreadMarks over Virtual Interface Architecture on Myrinet and Gigabit Ethernet: Challenges, Design Experience, and Performance Evaluation," *Proc. Int'l. Conf. Parallel Processing (ICPP),* 2001.
[7] M. Rangarajan and L. Iftode, "Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance," *Proc. Fourth Ann. Linux Showcase and Conf.,* 2000.
[8] A. Bilas, C. Liao, and J.P. Singh, "Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems," *Proc. 26th Int'l Symp. Computer Architecture,* 1999.
[9] R. Samanta, A. Bilas, L. Iftode, and J.P. Singh, "Home-Based SVM Protocols for SMP Clusters: Design and Performance," *Proc. Fourth Int'l Symp. High-Performance Computer Architecure (HPCA),* 1998.
[10] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy, "SoftFLASH: Analzying the Performance of Clustered Distributed Virtual Shared Memory," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* 1996.
[11] P. Joubert, R.B. King, R. Neves, M Russinovich, and J.M. Tracy, "High-Performance Memory-BasedWeb Servers: Kernel and User-Space Performance," *Proc. USENIX Ann. Technical Conf.,* 2001.
[12] V.S. Pai, P. Druschel, and W. Zwaenepoel, "IO-Lite: A Unified I/O Buffering and Caching System," *Proc. Conf. OS Design and Implementation (OSDI),* 1999.
[13] Oracle, Oracle Net VI Protocol Support, a technical white paper, http://www.vidf.org/Documents/whitepapers/Oracle_VI.pdf, 2001.
[14] K. Magoutis, S. Addetia, A. Fedorova, M.I. Seltzer, J.S. Chase, A.J. Gallatin, R. Kisley, R.G. Wickremesinghe, and E. Gabber, "Structure and Performance of the Direct Access File System," *Proc. USENIX Ann. Technical Conf.,* 2002.
[15] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J.F. Philbin, and K. Li, "Experiences with VI Communication for Database Storage," *Proc. 29th Int'l Symp. Computer Architecture (ISCA),* 2002.
[16] S. Pakin, V. Karamacheti, and A. Chien, "Fast Messages: Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors," *IEEE Concurency,* vol. 5, no. 2, pp. 60-73, 1997.
[17] A. Rubini and J. Corbet, *Linux Device Drivers,* second ed. O'reilly Books, http://www.xml.com/ldd/chapter/book/, 2005.
[18] Mellanox Technologies, http://www.mellanox.co.il/, 2005.
[19] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Ann. Int'l Symp. Computer Architecture (ISCA '95),* 1995.

[20] D. Bailey, J. Barton, T. Lasinski, and H. Simon, "The NAS Parallel Benchmarks," Technical Report RNR-91-002, NASA Ames, Aug. 1991.
[21] P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel, "Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems," *Proc. USENIX Conf.*, pp. 115-131, 1994.
[22] N. Niv and A. Schuster, "Transparent Adaptation of Sharing Granularity in Multiview-Based DSM Systems," *Proc. Int'l Parallel and Distributed Processing Symp.*, Apr. 2001.

Liran Liss received the BSc degree (summa cum laude) in computer engineering from the Technion—Israel Institute of Technology, in 2001. He is currently pursuing the PhD degree in the direct PhD program at the Technion. During his studies, he worked as a developer at the Microsoft Research and Development center in Haifa, Israel, and at Mellanox, Inc. His research interests include hardware-software interfaces and large-scale distributed computing. He is a student member of the IEEE.

Yitzhak Birk (M '82 SM02) received the BSc (cum laude) and MSc degrees from the Technion in 1975 and 1982, respectively, and the PhD degree from Stanford University in 1987, all in electrical engineering. He has been on the faculty of the Electrical Engineering Department at the Technion since October 1991, and heads its Parallel Systems Laboratory. From 1976 to 1981, he was a project engineer in the Israel Defense Forces. From 1986 to 1991, he was a research staff member at IBM's Almaden Research Center, where he worked on parallel architectures, computer subsystems, and passive fiber-optic interconnection networks. From 1993 to 1997, he also served as a consultant to Hewlett Packard Labs in the areas of storage systems and video servers, and was later involved with several companies. Dr. Birk's research interests include computer systems and subsystems, as well as communication networks. He is particularly interested in parallel and distributed architectures for information systems, including communication-intensive storage systems (e.g., multimedia servers) and satellite-based systems, with special attention to the true application requirements in each case. The judicious exploitation of redundancy for performance enhancement in these contexts has been the subject of much of his recent work. He is a senior member of the IEEE Computer Society.

Assaf Schuster received the BSc, MSc, and PhD degrees in mathematics and computer science from the Hebrew University of Jerusalem. Since receiving the PhD degree in 1991, he has been with the Computer Science Department at the Technion—Israel Institute of Technology. His interests include all aspects of parallel and distributed computing. He is a senior member of the IEEE. More information on Dr. Schuster can be found at http://www.cs.technion.ac.il/~assaf.