

# Efficient Exploitation of Kernel Access to Infiniband: a Software DSM Example

Liran Liss, Yitzhak Birk and Assaf Schuster  
*Technion – Israel Institute of Technology*  
{liranl@tx, birk@ee, assaf@cs}.technion.ac.il

## 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.*

## 1. Introduction

Infiniband (IB) [1] is a high-performance SAN architecture that implements in hardware 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. 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 accessing IB through the kernel for applications that exhibit close coupling between network services and those of the operating system. We use a software Distributed Shared Memory (DSM) system as a context.

DSM is a runtime system that emulates shared memory across a computing cluster [3,4]. Software DSMs implement an invalidation-based protocol using the operating system's page protection mechanism. Access rights to invalidated pages are revoked, while a page fault triggers a protocol action that updates the page. Several observations hold for DSM protocols: each protocol invocation requires at least one system call (page-protection change, synchronization); communication is inherently asynchronous; both the source and destination addresses are known in advance when transferring application data between nodes; and latency is crucial for

the parallel computation, so high bandwidth does not suffice. Therefore, achieving high performance requires 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.

The introduction of high-performance user-level SANs to DSM systems [5,6] 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; 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 [7,8]. Accordingly, DSM systems appear well suited for evaluating the kernel/IB platform. Previous work demonstrated the advantages of integrating the kernel network stack (TCP/IP) and high-level protocols [9] or the file cache [10] in network servers. In this paper, we show that this approach is beneficial even when the network protocol stack is implemented in hardware.

Systems such as databases [11] and distributed file systems [12] can benefit substantially from the new hardware capabilities. However, researchers have pointed out that specialized APIs would be needed in order to attain the full benefits [13]. This establishes the motivation to evaluate the integration of SAN access with other OS functions in the kernel.

We designed and implemented a set of primitives, and used them to construct a highly efficient Linux kernel/IB platform. We then adapted Multiview [14], a fine-grain sequential consistency (SC) DSM protocol, to this environment, and carried out an extensive comparative performance evaluation of our prototype implementation.

In section 2, we briefly review Infiniband. Our communication and memory management primitives are presented in section 3. The DSM protocol adaptation is discussed in section 4. Performance results are summarized in section 5, and Section 6 presents discussion and concluding remarks.

## 2. Infiniband

Infiniband is a switch-based serial I/O interconnect architecture that provides low latency, high bandwidth communication. Among its main features are 2.5/10/30Gb/s link speeds, Connection-based and



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 this approach is limited only by the physical resources in each node (memory and WQ sizes). Therefore, 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 [15]), allowing the buffers to be consumed and freed in a simple round-robin fashion.

### 3.2 Asynchronous-event handling

Request messages arrive from the fabric unexpectedly, and must be handled with minimal latency. Furthermore, the protocol may want to be notified whenever an asynchronous operation such as RDMA Read completes. IB addresses these issues by enabling the 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.

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 CQE processing internally. 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.

A remaining question is where to perform the associated protocol processing whenever the asynchronous notification handler is called. In our platform, the completion notification is delivered as part of an interrupt service routine (ISR), so calling the protocol handler would provide superb latency. However, such a scheme does not allow the called code to sleep (or to call any OS service that may block), spin-lock or access user space, and implies that processing should be extremely fast because other interrupts may be disabled in this context. Although our protocol takes actions such as waking processes, sending responses to the network, state manipulation, and changing page protections, careful examination reveals that executing asynchronous entry points of our protocol inside ISR context is permissible. Waking processes is a main function of ISRs, and posting a WR to the network during interrupt context is supported by our architecture. We address synchronization and locking issues by a unique design of the protocol (section 4), and by using a two-stage approach to asynchronous-event handling: in the uncommon case of resource shortages (like a taken lock), we reschedule the handling of the event in process context. Changing page protections

inside ISRs is discussed below. Finally, our lightweight SC protocol satisfies the requirement for fast processing.

Note that, although performing the associated protocol processing inside a process context does not impose any of the aforementioned restrictions, a process depends on its scheduling which can take considerable time and increases overhead.

**Remark.** For longer event processing, handling in the ISR is not adequate. In these cases, the Linux Task Queue mechanism [16] is a good solution. While most task queues execute in interrupt context<sup>2</sup> (and thus impose similar restrictions as ISRs), they take place at a "safer" time (interrupts enabled) than ISRs. In addition, they are a fairly fast mechanism and do not rely on process scheduling. Although we do without task queues, the Immediate Task Queue (the fastest queue in the system) should be considered for other protocols and applications.

### 3.3 Efficient page protection

Page-protection system calls are used extensively by DSM systems, and are reported as a major source of overhead. Beyond the overhead of the system call itself, changing page protections involves acquisition of semaphores and locks, expensive data structure manipulation and often flushing the TLB. (In SMP machines, this can require interruption of other processors to flush their TLB and polling for completion.) Therefore, we decided to implement a unique kernel manager for virtual memory areas dedicated to DSM memory. Our implementation achieves the following goals:

- No data structures are changed except the ones necessary for the hardware (page tables).
- A single call can change any group of pages to any set of protections.
- There are no sleeping operations. Locking is reduced to acquisition of a single lock, which is nearly always free.
- Page protection changes can be attempted in interrupt context. In the rare case that the lock is already being held, the operation fails and should be retried by the protocol.

A complete description of our memory manager will be reported elsewhere.

## 4. DSM Protocol Adaptation

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% [6].) Protocol control messages such as page requests and lock acquisitions, which generally require processing on the remote host, are

<sup>2</sup> 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.

better matched to channel semantics. Therefore, we decided to implement data movement and control messages by RDMA-W and Send WRs, respectively, using our communication and buffering primitives. 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. If the problem size exceeds that of physical memory, communication buffers can be used instead [6], or a hybrid approach can be taken. (For large problems, virtual memory page thrashing due to insufficient physical memory is likely to limit execution speed regardless of the data transfer semantics.)

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 of the protocol's asynchronous entry points are implemented in interrupt context based on our asynchronous event handling and memory primitives, which cuts latency and eliminates context switching due to network events. To achieve this, we defined a clean separation between tasks performed by the synchronous and asynchronous portions of the coherence protocol. Request bookkeeping tasks are assigned to synchronous entry points. These tasks access coherence meta-data only for reading, and follow closely the monitor synchronization paradigm. Page protection and coherence meta-data manipulation tasks are performed exclusively by asynchronous entry points, which are executed atomically. Asynchronous entry points do not access request bookkeeping meta-data. When a reply indicates that a certain request is complete, the only action performed is to wake up the associated processes. Thus, the only feasible data race between synchronous and asynchronous entry points is a read-write data race, whereby a process reads coherence meta-data while an interrupt handler updates it. However, this does not affect the correctness of the protocol. (For details, see [17].) Figure 2 shows the control path between the system components. In the common case, an asynchronous operation that involves coherence meta-data updates, protection changes, sending a response and waking processes, is executed to completion by the ISR itself.

Note that, in a sequential consistency DSM, barrier and lock requests are simple actions that do not involve any coherence information. We implemented them using a similar approach. In order to reduce latency further, we experimented both with selective polling (replacing interrupts with polling when 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.

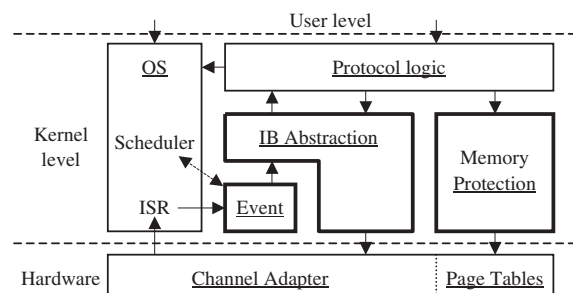


Fig. 2: Our Kernel-IB System Control Path

## 5. Performance Evaluation

In this section we evaluate the performance of our implementation. Results are reported for micro-benchmarks as well as for complete applications. All experiments were performed on a cluster of eight SMP PCs, running the Linux 2.4.18 operating system. Each machine has two 733MHz P-III processors, 512KB L2 cache, 512MB memory and a 32-bit, 33MHz PCI bus. Every node employed a multi-port Mellanox MT21108 card [18], which provides IB switch and TCA (targets the PCI bus) functionality. The device also has limited HCA support in the form of a dedicated DMA engine. (The interested reader may contact the authors for information on our implementation of the Verbs interface.) See [17] for basic OS/IB operation latencies.

### 5.1 Results

For a single page, our memory primitives enable a change of page protections in roughly half the time of the OS implementation. For groups of 16-32 pages, they perform more than an order of magnitude better than the required multiple `mprotect` system calls. As our protocol currently handles single page faults, we utilize our memory primitives mainly for supporting ISR protocol handling.

To evaluate the handling of asynchronous events inside interrupt handlers, we compared it to task queue handling and 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 1, kernel handling performs substantially better than user context, with some advantage to ISR over Task Queues.

Table 1: Round-trip time for different receive contexts

Polling	ISR	Task Queue	Process
45 $\mu$ s	60 $\mu$ s	70 $\mu$ s	90 $\mu$ s

We next perform a comparison of the whole system, between our kernel implementation that handles all protocol asynchronous entry points during the ISR, and a simulation of a user-level/VIA implementation. The simulation was done by incorporating the following changes into the system:

- 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.



