

In-Kernel Integration of Operating System and Infiniband Primitives for High Performance Computing Clusters: a 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. We show that overheads are reduced substantially, and overall application performance is improved both in terms of absolute execution time and scalability.

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.

Software 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 [5,6]. However, several observations hold for DSM protocols in general:

- *Each protocol invocation requires at least one system call.* These are usually multiple calls for changing page 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. In addition, 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 [7,8] 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 [9,10]. Accordingly, DSM systems appear well suited for evaluating the kernel/IB platform.

Previous work demonstrated the advantages of integrating the kernel network protocol stack (TCP/IP) with high-level protocols [11] or with the file cache [12] in network servers. In this paper, we show that this approach is beneficial even for SANs, wherein the network protocol stack is implemented in hardware.

Systems such as databases [13] and distributed file systems [14] can benefit substantially from new hardware capabilities such as reliable data transfer and RDMA. However, researchers have pointed out that specialized APIs would be needed in order to attain the full benefits [15].

These observations have motivated us 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 [6], a fine-grain SC DSM protocol, to this environment, and carried out an extensive comparative performance evaluation of our prototype implementation.

Our main findings are as follows. Common DSM overheads were substantially reduced using our kernel/IB platform: response latency for asynchronous events improved by 33% relative to a user-level implementation, and changing page protections for large page groups performs an order of magnitude better than conventional system calls. These improvements enabled our kernel/IB DSM system to improve application execution time by up to 23% 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).

Our approach has broad applicability beyond DSM. The availability in the kernel of Infiniband's software primitives enabled us to integrate network and operating system resources 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 combined services can offer high performance to applications through an appropriate user-level API.

The remainder of the paper is organized as follows. In section 2, we briefly review Infiniband and Multiview. 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. Background

2.1 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 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 function 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 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.

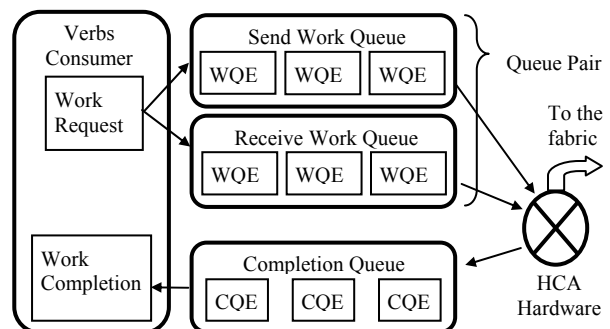


Fig. 1: Infiniband queuing model

IB defines two data transfer models:

- *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 sender specifies memory locations at both ends, and memory is either read or written according to the corresponding RDMA-WR (Read or Write).

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 sub-page sharing granularity. It was first implemented in the Millipage system [6]. 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 ‘mini-page’.

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

A request is triggered by a page fault and forwarded to the manager. After handling previous requests for the same mini-page, 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 mini-page 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.

From this point on, the term ‘protocol’ will refer to the DSM protocol mentioned above. However, in many cases the mechanisms that we describe are also applicable to other protocols / services.

3. Our primitives

We identified and implemented a set of primitives that serve the communication, memory, and scheduling needs of the protocol. Their implementation did not require any modifications to the operating system.¹ We next detail these primitives, their associated Infiniband abstractions, and the kernel mechanisms that we used.

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.

3.1 Buffer management and flow control

While the data integrity needs of our system map nicely to IB’s Reliable Connection service (we open such a connection between every two nodes in the cluster), WR processing and its associated buffer management are low-level and complex. Therefore, we decided to provide the protocol with simpler primitives for handling channel-semantic operations. Application data or protocol meta-data that are accessed in place by memory-semantic operations are better left to the control of the protocol.

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. Additionally, 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.

In many parallel systems, including our DSM, the number of in-flight messages is bounded. Moreover, unbalanced communication patterns are not uncommon in parallel applications, and this bound can be reached whenever all threads access new data following a sequential phase. Finally, our protocol uses Send semantics 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 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 [16]), 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 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.

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. 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 (while remaining in ISR context) 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.

Our protocol takes actions such as waking processes, sending responses to the network, state manipulation, and changing page protections. However, 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 rescheduling asynchronous-event handling in process context in the uncommon case of resource shortages (like a taken lock). 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, latency depends on the process' scheduling which can take considerable time and increases overhead.

Remark. For events whose handling requires longer processing, handling in the ISR is not adequate. In these cases, the Linux Task Queue mechanism [17] is a good solution. While most task queues execute in interrupt context² (and thus impose similar restrictions to ISRs), they take place at a "safer" time (interrupts enabled) than

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.

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) can 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% [8].) 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, 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 [8], or a hybrid approach can be taken. For example, accessing part of the application data directly using RDMA, and part of it using communication buffers. Such partitioning can change dynamically at certain phases of the application execution

(such as barriers). However, these solutions come at a cost of additional data copying or expensive system calls when buffers are re-registered. Note that 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, the protocol's asynchronous entry points are all 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 eliminate severe data races between interrupt and process contexts, we defined a clean separation between tasks performed by the synchronous and asynchronous portions of the coherence protocol:

- Synchronous entry points (requestor threads) handle all request bookkeeping tasks. These tasks access coherence meta-data only for reading.
- Asynchronous entry points (message and WQ completion handlers) handle only page protection tasks and coherence meta-data manipulation. Protections are granted when a reply for a page request arrives, and are revoked when serving invalidation requests.

The control flow of our request model is as follows. A requestor competes for exclusive access to bookkeeping information. After access is granted, it checks 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 the proper wait queue after incrementing a usage count. Otherwise, a new message is sent to the appropriate manager, and the requestor is added to the wait queue assigned for this request. Page availability is determined by inspecting coherence meta-data.

When an asynchronous reply signals the completion of the request, necessary protection changes are performed, coherence meta-data is updated, and the corresponding wait queue is signaled. After reacquiring exclusive access, a woken requestor decrements the request usage count and returns. Resources can be reused once the usage count drops to zero.

Since the synchronous entry points closely follow the monitor synchronization paradigm, and asynchronous entry points are executed atomically, the only feasible data race 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: 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).

Fig. 2 shows the control path among the system components. In the common case, an asynchronous operation that involves coherence meta-data updates, protection changes, sending a response and waking up processes, is executed to completion by the ISR itself.

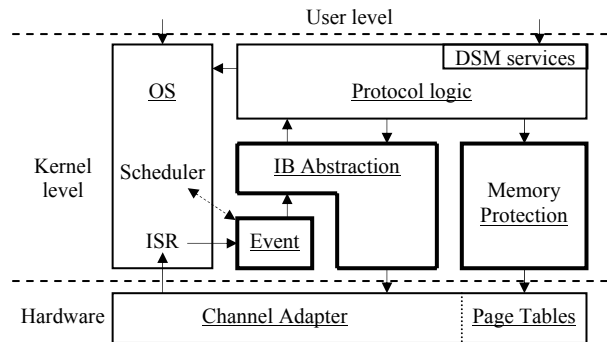


Fig. 2: Our Kernel-IB system control path

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 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.

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 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 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. (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.

Table 1: Basic OS/Infiniband operation latencies.

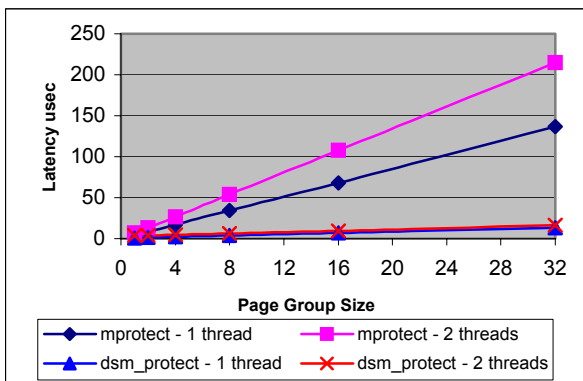
Operation	Latency [μ s]
Interrupt delivery	10
Page fault cost (fault to signal handler)	5
System call invocation	0.7
DSM-Protect (single page)	0.9
Post Work Request (software overhead)	2
RDMA-W one-way latency	8-9
SEND one-way latency	22-23
RDMA-R (completion detected by polling memory)	9
RDMA-R + CQ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

5.1 Applications

Our application suite comprises eight applications: Water-nsquad (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 in the appendix for the input datasets used for each application.

5.2 Micro-benchmark results

Fig. 3 depicts the dramatic performance advantage of our memory subsystem for page protection over user-level calls to Linux's `mprotect` function. The reported latencies correspond to the time it takes to change an arbitrary page group (not necessarily consecutive) to arbitrary permissions (not necessarily the same). 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 only handles single page faults, we utilize our memory primitives mainly for supporting ISR protocol handling.


Fig. 3: Page protection latencies

To evaluate the handling of asynchronous events inside interrupt handlers, we 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, with some advantage to ISR over Task Queues.

Table 2: Round-trip time for different receive contexts

Polling	ISR	Task Queue	Process
45 μ s	60 μ s	70 μ s	90 μ s

5.3 Primitives' contributions and optimizations

We next evaluate the combined contribution of our primitives and two optimizations to the whole system, on a cluster of eight nodes utilizing two threads per node.

Fig. 4 compares the execution time of three benchmark applications, between a kernel implementation based on our primitives (Kernel-ISR), and a simulation of a VIA/IB implementation (VIA-sim). An additional kernel implementation that executes asynchronous events in task queues (Kernel-TQ) is added for reference. The VIA 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.
- Before each protocol action that would require a system call, we insert a 1 μ s delay.
- We perform memory protection changes by calling the OS implementation (`sys_mprotect`) rather than using our memory primitive.

Otherwise, the system is unchanged.

The application execution time in Kernel-ISR was 23% shorter than with VIA-Sim. While Kernel-TQ performed better than VIA-Sim in Barnes and NBodyW, it is substantially inferior to VIA-sim in TSP. Detailed execution time examination revealed that page faults in Kernel-TQ cost twice more than VIA-Sim, and as much as four times more than 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 Task-Queue 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.

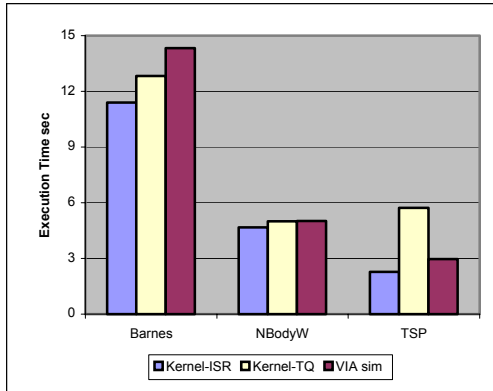


Fig. 4: Execution time vs. protocol execution context

We also tested the effects of load on the system (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.

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

Finally, we evaluated the use of RDMA-R WRs (rather than RDMA-W) in read-faults whenever data retrieval did not require interrupting the remote processor. While read-fault latencies **increased** by 2-3% on average (mostly due to the relatively slow CQ update for RDMA-Rs in our architecture), the total execution time of most applications improved slightly. The main contribution of using RDMA reads in our system is thus to mitigate 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).

5.4 Application performance

We evaluated the performance and 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 is comparable to our hardware (13 μ s send

latency, 180MB/sec bandwidth). Because of the differences, 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 Fig. 5(a-h). 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 the appendix for runtime statistics (Table 3) and an execution time breakdown (Fig. 6) for each application.

Relatively “well behaved” applications (SOR, LU, IS and TSP) achieve good speedups on both implementations. Nevertheless, our kernel/IB platform consistently exhibits 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 high synchronization rate 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 twelve nodes for a single thread per node. However, for two threads per node, our system does not scale from eight to twelve 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 high barrier times, which 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.

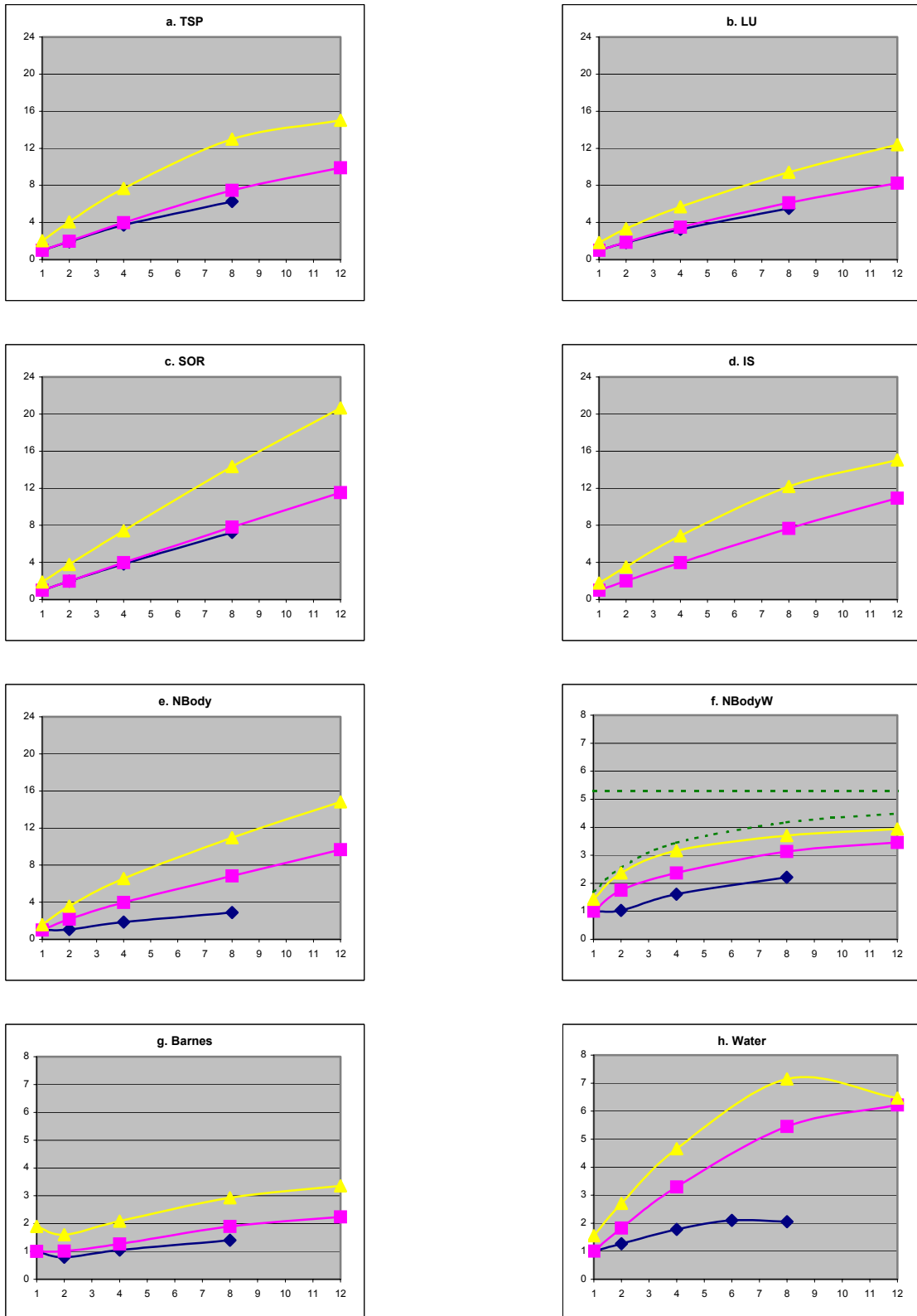


Fig. 5: Application speedups vs. number of nodes. (A single node with a single processor is used as the baseline.)
Legend: Diamond (blue) – VIA/NT, single thread per node; Square (pink) – Kernel/IB, single thread per node;
 Triangle (yellow) – Kernel/IB, two thread per node; Dashed line (green) – NBodyW Theoretical curve and limit.

Remark. The speedup differences are more noticeable than those observed relative to our VIA simulation in the previous subsection. 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. We detail some of the Infiniband features/aspects that were not exploited, discuss topics for future research, and point out insights that go beyond DSM systems.

6.1 DSM conclusions and opportunities

Our communication, memory, and event-handling primitives substantially reduce common DSM overheads. ISR event-handling reduces the response time for asynchronous messages by 33% relative to user-level signal handlers, and our memory services 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, adaptive granularity SC protocols [22]). We have shown how a high-level protocol can be split between interrupt and process contexts, and employ these primitives to reduce complexity. Our kernel/IB DSM system performs up to 23% better than a simulated VIA/IB implementation. (In view of the way in which the simulation was carried out, the comparison is quite accurate.) 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 hardware) can drastically

reduce the number of remote CPUs interrupted to process a protocol action. Relaxed consistency DSMs can benefit greatly from IB's broadcast support [8].

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 [9], 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 communication primitives, which abstract the low-level WR processing model, enable a dramatic complexity reduction. They provide protocols with send-receive semantics that are both easy to use and efficient (0-copy and minimal processing overhead).

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 over 300 μ s.

The availability in the kernel of Infiniband's software primitives enabled us to integrate network and operating system resources efficiently. This approach resulted in fewer user-kernel crossings, less overhead in accessing OS functions, and better control over the scheduling of network related events. Note that applications do not need to be implemented in the kernel in order to take full advantage of the platform: integration of OS and network services in the kernel can provide high performance to applications through an appropriate user-level API.

In this paper, network and memory operations served to demonstrate our approach through a DSM API. However, it has additional applications: combining the SAN with the file cache (for Web and File Servers), task management (for Remote-execution/Process-migration facilities) and more.

Appendix

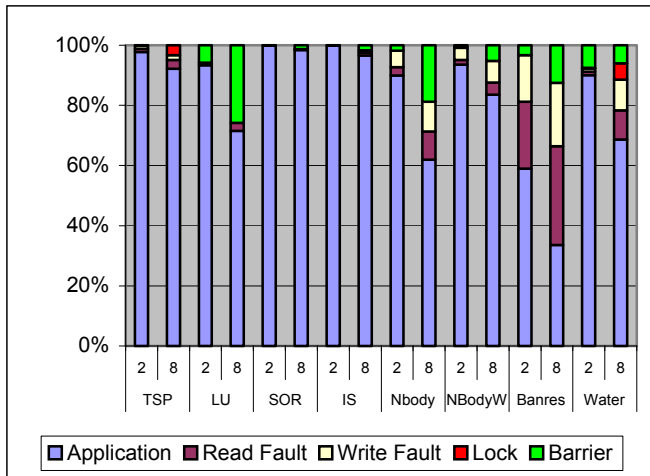
Table 3 presents the input set size and runtime statistics for each benchmark application.

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

The statistics were gathered from a single node in a parallel computation consisting of eight nodes.

The normalized execution time break down for all applications in our suite is shown in Fig. 6. (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.


Fig. 6: Normalized execution time breakdown

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

References

1. Infiniband Trade Assoc. – Infiniband Spec. <http://www.infinibandta.com/>.
2. Virtual Interface Architecture Specification. <http://www.viaarch.org/>.
3. K. Li and P. Hudak, Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comp. Sys.*, 7(4):321-359, Nov. 1989.

4. P. Keleher, A.L. Cox, and W. Zwaenepoel, Lazy Consistency for Software Distributed Shared Memory. In *Proc. of the 19th Annual Symposium on Comp. Arch.*, p. 13-21, May 1992.
5. D.J. Scales and K. Gharachorloo, Shasta: a system for supporting fine-grain shared memory across clusters. In *Proc. of the 8th SIAM Conference on Parallel Proc. for Sci. Comp.*, Mar. 1997.
6. A. Itzkovitz and A. Schuster, MultiView and Millipage: Fine-Grain Sharing in Page-Based DSMs. In *Proc. Conf. on OS Design and Implementation*, 1999.
7. 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. In *Int'l. Conf. on Parallel Processing (ICPP)*, 2001.
8. M. Rangarajan and L. Ifode, Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance. In *Proc. 4th Annual Linux Showcase and Conf.*, 2000.
9. A. Bilas, C. Liao, and J.P. Singh, Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems. In *Proc. 26th Int'l. Symp. on Comp. Arch.*, 1999.
10. R. Samanta, A. Bilas, L. Ifode and J.P. Singh Home-based SVM protocols for SMP clusters: Design and Performance. In *Proc. 4th Int'l. Symp. on High-Perf. Comp. Arch. (HPCA)*, 1998.
11. P. Joubert, R.B. King, R. Neves, M. Russinovich and J.M. Tracy. High-Performance Memory-Based Web Servers: kernel and User-Space Performance. In *Proc. USENIX Annual Tech. Conf.*, 2001.
12. V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-lite: A unified I/O buffering and caching system. In *OS Design and Impl. (OSDI)*, 1999.
13. Oracle, Oracle Net VI Protocol Support, a technical white paper. http://www.vifd.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. In *Proc. USENIX Annual Tech. Conf.*, 2002.
15. Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J.F. Philbin and K. Li, Experiences with VI Communication for Database Storage. In *Proc. 29th Intl. Symp. on Comp. Arch. (ISCA)*, 2002.
16. S. Pakin, V. Karamacheti and A. Chien, Fast Messages: Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 5(2): 60-73, 1997.
17. A. Rubini and J. Corbet. *Linux Device Drivers*, 2nd Edition. O'reilly books. Online version: <http://www.xml.com/ldd/chapter/book/>.
18. Mellanox Technologies. <http://www.mellanox.co.il/>.
19. S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Annual Int'l. Symp. on Comp. Arch. (ISCA'95)*, 1995.
20. D. Bailey, J. Barton, T. Lasinski, and H. Simon, The NAS parallel benchmarks. *Tech. Rep. 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. In *Proc. of the USENIX Conference*, pages 115-131, 1994.
22. N. Niv and A. Schuster, Transparent Adaptation of Sharing Granularity in Multiview-Based DSM Systems. In *Int'l. Conf. on Par. and Distr. Proc. Symp.*, Apr. 2001.