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

Proceedings of the 11 th Symposium on High Performance Interconnects (HOTI'03)
0-7695-2012-X/03 $17.00 © 2003 IEEE
Authorized licensed use limited to: Tara Casaletto. Downloaded on April 29,2024 at 13:53:44 UTC from IEEE Xplore. Restrictions apply.

IEEE COMPUTER SOCIETY

Google Exhibit 1048
Google v. VirtaMove
Oracle Ex. 1048
Oracle v. VirtaMove, IPR2025-00982

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)* for connecting computing nodes. HCAs must support the IB Verbs interface [1-vol.1, ch.11], which defines the function provided to the host by the channel adapter.
- *Target Channel Adapters (TCAs)* 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.

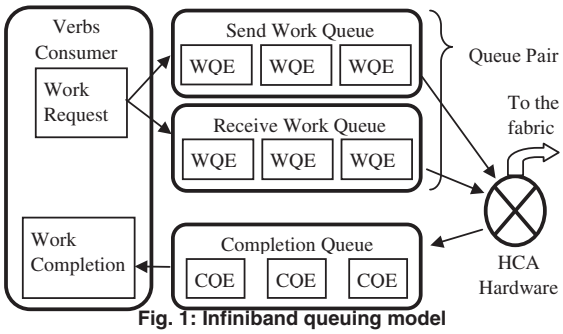

Fig. 1: Infiniband queuing model

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 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.

3. Our primitives

We identified and implemented a set of primitives that serve the communication, memory, and scheduling needs of the protocols. 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.

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 to protocols simpler primitives for handling channel-semantics operations. Application data or protocol meta-data that are accessed in place by memory-semantics 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 read data from a single 'centralizing' thread. 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, thus eliminating the need for application-level flow control and achieving efficient delivery for every message. (While the flow control

[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.

Proceedings of the 11 th Symposium on High Performance Interconnects (HOTI'03)
0-7695-2012-X/03 $17.00 © 2003 IEEE
Authorized licensed use limited to: Tara Casaletto. Downloaded on April 29,2024 at 13:53:44 UTC from IEEE Xplore. Restrictions apply.

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[2] (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

[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.

Proceedings of the 11 th Symposium on High Performance Interconnects (HOTI'03)
0-7695-2012-X/03 $17.00 © 2003 IEEE
Authorized licensed use limited to: Tara Casaletto. Downloaded on April 29,2024 at 13:53:44 UTC from IEEE Xplore. Restrictions apply.

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.

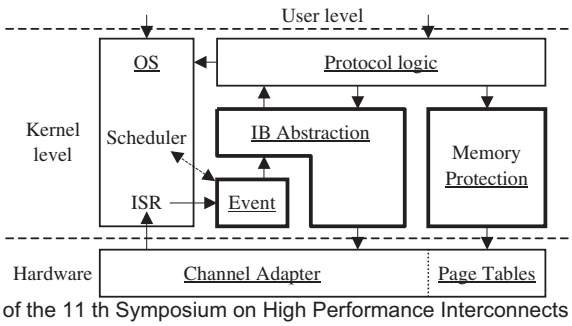

Fig. 2: Our Kernel-IB System Control Path

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 singe 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
45µs	60µs	70µs	90µs

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.

Proceedings of the 11 th Symposium on High Performance Interconnects (HOTI'03)
0-7695-2012-X/03 $17.00 © 2003 IEEE
Authorized licensed use limited to: Tara Casaletto. Downloaded on April 29,2024 at 13:53:44 UTC from IEEE Xplore. Restrictions apply.

- Before each protocol action that would require a system call, we insert a 1µs delay.
- We perform memory protection changes by calling the OS implementation (`sys_mprotect`) rather than using our memory primitive.

Otherwise, the system is unchanged. Under normal operation (two application threads per host on an otherwise idle system), application execution time in the kernel implementation improved by up to 23% relative to the user-level simulation. Furthermore, when we tested the effects of load on the system (an additional load of a single CPU-intensive process was run on each host to simulate occasional interference by other users of a cluster), the gap increased considerably in all applications, indicating that the responsiveness of user-process message handling is much more sensitive to load. We also evaluated a kernel/Task-Queue implementation, which exhibited only mild improvements over the user-level simulation, and was even inferior to it for some applications [17].

Finally, we present the effects of polling and utilizing RDMA-R WRs on the whole system. The introduction of selective polling reduced page fault latencies by 3-7%. Overall application performance improved by up to 6%. However, when the number of application threads per host machine was greater than the number of CPUs in each machine, polling only degraded performance. RDMA-R WRs (rather than RDMA-W) were used 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).

5.2 Application Performance

We evaluated the performance and scalability of our implementation using eight benchmark applications [17]. 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. Because of the differences, the NT/VIA 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.

Table 2: Application speedups (relative to a sequential run on a single CPU) taken on 8 hosts for 1,2 threads per host.

Application	User/VIA 1 thread	Kernel/IB 1 thread	Kernel/IB 2 threads
SOR	7.2	7.8	14.3
LU	5.5	6.1	9.4
IS	-	7.7	12.2
TSP	6.3	7.5	13
Water	2	5.5	7.2
Nbody	2.9	6.8	11
NbodyW	2.2	3.1	3.7
Barnes	1.3	1.9	2.9

As depicted in Table 2, 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. Note that the 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. We have shown how a high-level protocol can be split between interrupt and process contexts, and employ our primitives to reduce complexity. Our kernel/IB DSM system performs up to 23% better than a corresponding user-level/VIA implementation, and 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 allready 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 5.2.

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

Proceedings of the 11 th Symposium on High Performance Interconnects (HOTI'03)
0-7695-2012-X/03 $17.00 © 2003 IEEE
Authorized licensed use limited to: Tara Casaletto. Downloaded on April 29,2024 at 13:53:44 UTC from IEEE Xplore. Restrictions apply.

the majority of the data transfers. We found the main contribution of RDMA reads to be reduced interference with remote hosts, 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 [6].

Finally, our approach can be extended to implement a completely synchronous sequential consistency system on hardware platforms that can trigger TLB invalidations from IO 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 [7], 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 300µs.

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. However, it has additional applications: combining the network with the file cache (for Web and File Servers, etc.); task management (for Remote-execution/Process-migration facilities), and more.

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. www.infinibandta.com/.
2. Virtual Interface Architecture Specification. 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. Zwaenepol, "Lazy Consistency for Software Distributed Shared Memory". In Proc. of the 19th Annual Symposium on Comp. Arch., p. 13-21, May 1992.
5. 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". Int'l. Conf. on Par. Proc. (ICPP), 2001.
6. M. Rangarajan and L. Iftode, "Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance". Proc. 4th Annual Linux Showcase and Conf., 2000.
7. 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. on Comp. Arch., 1999.
8. R. Samanta, A. Bilas, L. Iftode and J.P. Singh, "Home-based SVM protocols for SMP clusters: Design and Performance". Proc. 4th Int'l. Symp. on High-Perf. Comp. Arch. (HPCA), 1998.
9. 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 Annual Tech. Conf., 2001.
10. V. S. Pai, P. Druschel and W. Zwaenepoel, "IO-lite: A unified I/O buffering and caching system", OS Design and Impl. (OSDI), 1999.
11. Oracle, "Oracle Net VI Protocol Support", a technical white paper. www.vidf.org/Documents/whitepapers/Oracle_VI.pdf, 2001.
12. 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 Annual Tech. Conf., 2002.
13. Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J.F. Philbin and K. Li, "Experiences with VI Communication for Database Storage", Proc. 29th Intl. Symp. on Comp. Arch.(ISCA), 2002.
14. A. Itzkovitz and A. Schuster, "MultiView and Millipage: Fine-Grain Sharing in Page-Based DSMs", Proc. Conf. on OS Design and Implementation, 1999.
15. S. Pakin, V. Karamacheti and A. Chien, "Fast Messages: Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors", *IEEE Concurency*, 5(2): 60-73, 1997.
16. A. Rubini and J. Corbet, *Linux Device Drivers*, 2nd Edition. O'reilly books. Online version: www.xml.com/ldd/chapter/book/.
17. L. Liss, Y. Birk and A. Schuster, "In-kernel Integration of Operating-System and Infiniband Primitives for High-Performance Computing Clusters: a DSM Example", CCIT Rep. #428 (also EE Pub #1367), Elec. Engr. Dept, Technion, June 2003.
18. Mellanox Technologies: www.mellanox.co.il/.

Proceedings of the 11 th Symposium on High Performance Interconnects (HOTI'03)
0-7695-2012-X/03 $17.00 © 2003 IEEE
Authorized licensed use limited to: Tara Casaletto. Downloaded on April 29,2024 at 13:53:44 UTC from IEEE Xplore. Restrictions apply.