



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

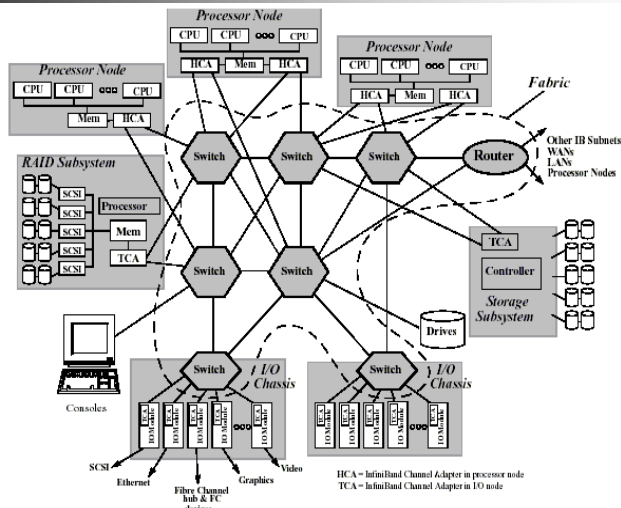
Liran Liss, Yitzhak Birk, Assaf Schuster
Technion, Israel.



Outline

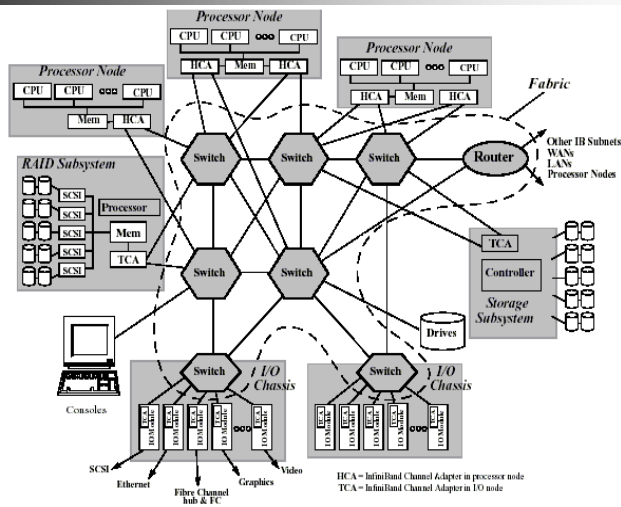
- The Infiniband access dilemma
- Case study: Distributed Shared Memory
- Implementation
- Results
- Conclusions

Infiniband

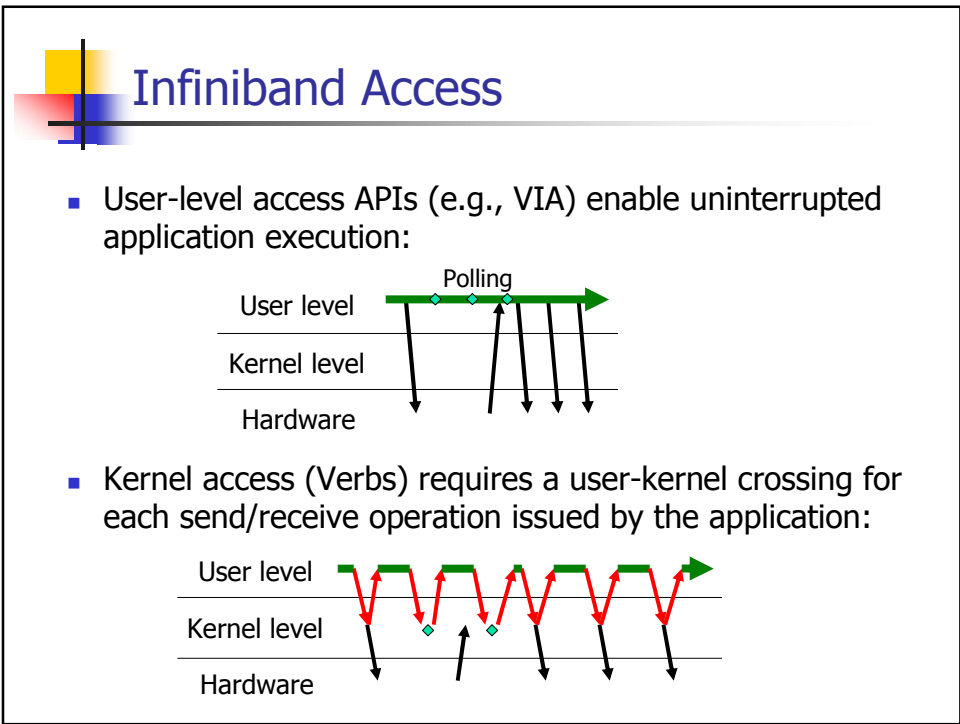
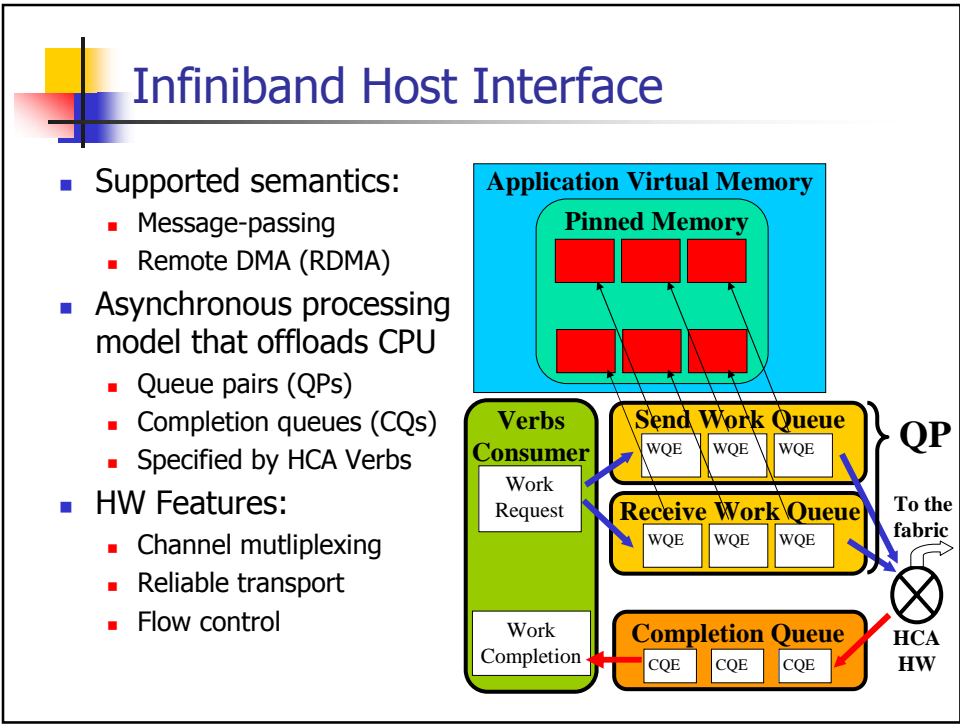


- An interconnect architecture intended to connect both computing nodes and I/O devices in a unified fabric

Infiniband (cont.)



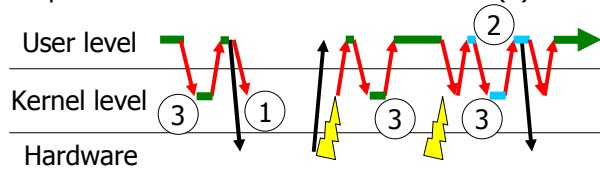
- Switched-based serial I/O with 2.5-30Gb/s bandwidth
- End nodes are connected through channel adapters



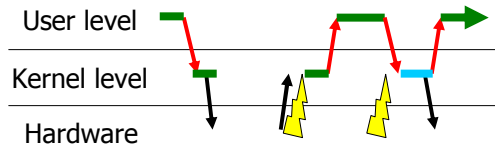


Is User-level Access Clearly Better?

- Consider applications that:
 - Use blocking receive operations (1)
 - Respond to asynchronous events (2)
 - Require related OS and network services (3)

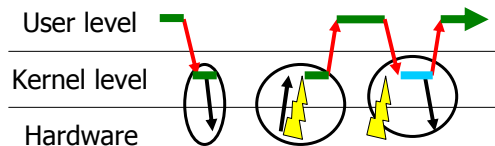


- Here, kernel access can be more efficient:



Our Approach

- Observation: For applications the exhibit close coupling between OS and network services, common APIs can be inefficient
- Key idea: OS and IB primitives can be efficiently re-bundled/packaged together in the kernel, in order to improve the performance

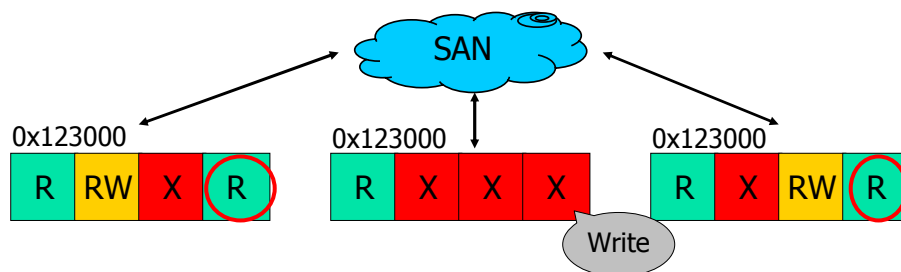


Methodology

- Demonstrate this approach using a working system (DSM)
- Phases:
 - Identify common DSM overheads and useful OS/IB primitives
 - Implement a prototype based on these primitives
 - Evaluate the implementation

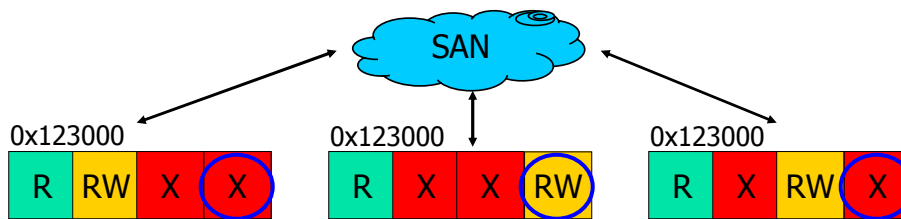
Software DSM

- Emulates shared memory across a computing cluster
- Software DSMs implement an invalidation based protocol using OS page protection
- Execution example...



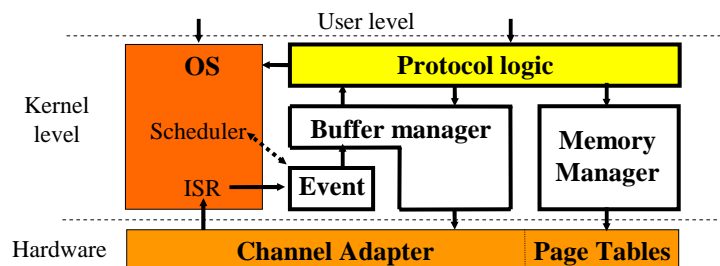
DSM Overheads

- Common DSM overheads:
 - Response to asynchronous events
 - OS calls – mainly page protection changes
 - Data transfer latency
- User-level SANs reduced data transfer latency considerably
- A kernel implementation of basic protocol operations can also mitigate the other overheads!



Our Kernel/Infiniband DSM

- We first implemented useful primitives to build upon:
 - IB Buffer management
 - Interrupt context asynchronous-event handling mechanism
 - DSM memory manager
- We then adapted the DSM protocol to this platform
- Finally, we exposed the protocol entry points to a user-level DSM library



Buffer Management Primitives

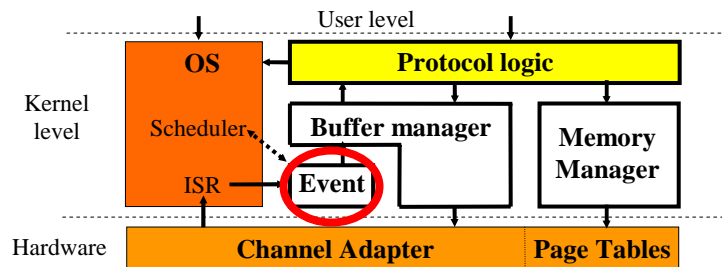
- WR processing semantics are low-level and complex
- We provide simple and efficient messaging primitives for handling buffer requirements
- For sends:
 - Buffers are reserved and sent upon request, and reclaimed automatically after WR completion
 - Completion detection is decoupled from explicit signaling requests

Buffer Management Primitives (cont.)

- For receives:
 - The number of in-flight messages is bounded in our system
 - We apply this bound for each QP, thus eliminating the need for application-level flow control
 - Protocol can access buffers only during CQ handler execution
- Scalability is limited by memory or WQ size

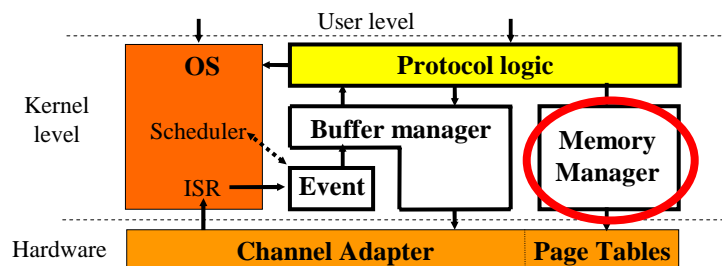
Event Handling Primitives

- **Asynchronous events must be handled with minimal latency**
- To reduce locking, we provide atomic CQ processing:
 - All WQs are associated with a single CQ
 - CQ processing is handled in a centralized manner
 - The protocol is allowed to register a single handler
- We employ a two-stage approach to minimize latency:
 - CQ processing is initially executed in ISR context
 - In case of resource shortages, we switch to a process context



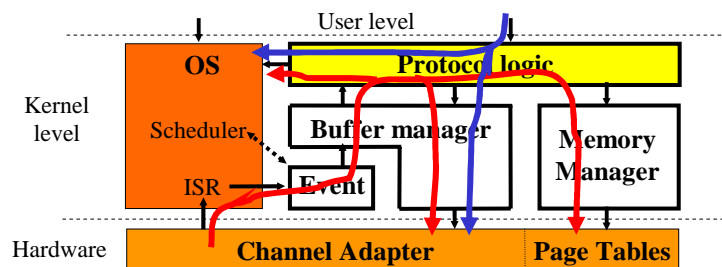
Memory Primitives

- **Changing page protection is an expensive system call**
- Our memory manager offers efficient primitives for changing page protection:
 - Any group of pages can be changed to any set of protections
 - Only the HW page tables are changed
- Protection change can be attempted in interrupt context



Protocol Adaptation

- Application data transferred using RDMA (pinned memory)
- Protocol messages use message-passing
- Clear separation between ISR- and process-context tasks
 - Request booking tasks are executed by synchronous entry points
 - Page protection and meta-data manipulation executed by asynchronous entry points
- No redundant user-kernel crossings in the common case



Performance Evaluation

- Host machines
 - 8 Dual processor SMP machines
 - 733MHz P-III CPU, 512KB L2 cache, 512MB main memory
 - 32-bit/33MHz PCI
 - Linux 2.4.18 OS
- Infiniband interconnect
 - Mellanox MT21108 TCA/Switch
 - Limited HCA support
- Measurements:
 - Micro-benchmarks
 - Overall performance
 - Application scalability

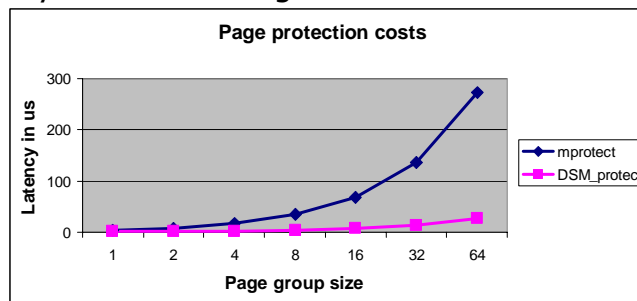


Micro-benchmark Results

- ISR handling cuts round-trip latency by 30% compared to user-level for short asynchronous messages:

Polling	ISR	Process
45 μ s	60 μ s	90 μ s

- Our memory manager outperforms conventional system calls by an order of magnitude:

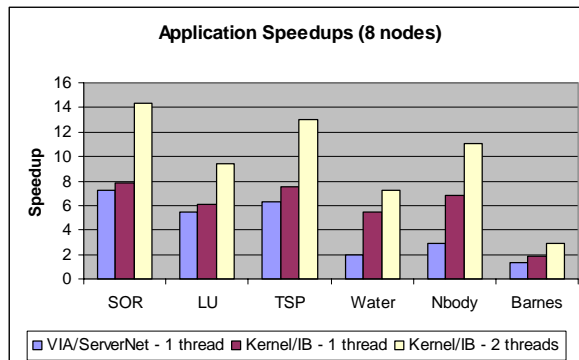


Overall Performance

- We compared our implementation with a VIA simulation implemented as follows:
 - Asynchronous-event handling executed in process context
 - Page protections are changed using the mprotect system call
 - We inserted a 1 μ s delay before any action that would require a system call
- Our system improved page fault latency by up to 50%
- Execution time was improved by up to 23%
- When we introduced additional load, the relative gap increased

Application Scalability

- Our implementation achieves better speedups than a HW VIA implementation (ServerNet-II) for all tested applications:



- In demanding (communication intensive) applications our platform doubled the VIA speedups

Conclusions

- High performance communication does not suffice for low latency message handling
- Efficient integration of Infiniband and OS primitives in the kernel enables attaining the full benefits of high performance SANs
 - Fewer user-kernel crossings
 - Less overhead in accessing OS functions
 - Better control over the scheduling of network related events
- Our buffer manager offers message-passing primitives that are both efficient and easy to use
- Applications do not need to be implemented in the kernel: integrated OS and network services can provide high performance to applications through an appropriate user-level API.

