RC24543 (W0804-113) April 30, 2008
Computer Science

IBM Research Report

A Scalable, High Performance InfiniBand-Attached SAN Volume Controller

D. Scott Guthridge
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

A Scalable, High Performance InfiniBand-Attached SAN Volume Controller

Scott Guthridge
Thomas J. Watson Research Center
IBM Research
Yorktown Heights, NY 10598
guthridg@us.ibm.com

Abstract— **We have developed a highly reliable InfiniBand host attached block storage management and virtualization system that allows use of off-the-shelf fibrechannel RAID controllers on the back end. The system is based on the existing IBM TotalStorage SAN Volume Controller (SVC) product, and therefore offers performance, a wide array of storage virtualization features, and support for many existing storage controllers. We provide an overview of the driver design as well as performance results. Read performance from SVC cache reaches 2 GB/s in a minimal two-node cluster configuration for large I/O requests.** [1]

I. Introduction

InfiniBand is a high-bandwidth, low-latency open systems interconnect. It provides fast message passing between hosts as well as remote direct memory access (RDMA).

InfiniBand is pervasive in high-performance computing (HPC) environments such as the Top 500 supercomputers list [1] where higher latencies of other network interconnects and the CPU overhead of data copying in protocols such as TCP/IP are prohibitive. Roughly a quarter of the Top 500 supercomputers rely on InfiniBand as the communication fabric. Increasingly, InfiniBand is being used in Grid Computing [2], clustered database systems [3] and in the financial sector for applications such as algorithmic stock trading [4], [5], [6].

In addition to host-to-host communication, InfiniBand can also be used to connect hosts to storage devices using the ANCI/INCITS standardized SCSI RDMA Protocol (SRP) [7]. Using the same InfiniBand fabric for both communication and storage simplifies server configuration by reducing the required number of PCI card slots and the associated cabling from host systems, especially in highly dense blade server environments.

InfiniBand storage is not as well developed as Fibrechannel storage, leaving few commercial IB storage options. We have tried to address the shortage of options by extending one of IBM's storage products, the TotalStorage San Volume Controller (SVC) to support InfiniBand in addition to Fibrechannel. The result is a highly reliable, InfiniBand-capable storage management and virtualization system with a rich set of features, that allows use of off-the-shelf fibrechannel RAID controllers as the underlying storage. Main benefit of our approach is that users of InfiniBand clusters can now attach, through SVC, to an existing FC SAN fabric and as well as having a choice of plethora of FC storage controllers that SVC product already supports today.

Section II provides an overview of the San Volume Controller product. Section III gives a high-level description of the IB driver implementation. Section IV shows the life of an I/O request through the driver. Section V describes driver synchronization and resource management. Finally, section VI gives performance results.

II. San Volume Controller

IBM TotalStorage San Volume Controller (SVC) is a storage management and virtualization system built on clustered Pentium-based servers [8]. SVC provides a centralized storage pool, block I/O access through virtual disks (*vdisks*), fast write-cache, copy services including point in time copy, remote copy and transparent migration, quality of service metering and reporting, and use of off-the-shelf RAID controllers. Deployment of SVC in an enterprise environment enhances manageability and improves storage utilization by providing centralized management and pooling of storage resources.

SVC provides high reliability by using UPS-backed server nodes running in I/O pairs. The paired nodes provide fast write caching by communicating their modified data blocks to each other and returning status back to the host as soon as both nodes have saved a copy of the modified data in their respective caches. If either node should fail before the modified blocks have been committed to disk, the other node will ensure the data is written. In the event of a power failure where the external storage devices may go offline before SVC can commit the modified blocks, each SVC node will save a copy of the modified cache blocks to a small local disk, running under UPS power. These blocks will then be committed to the external storage when the system is brought back on-line.

SVC system is scalable. Up to 8 SVC nodes form an SVC cluster. Each host connects to a redundant pair of SVC nodes in the cluster. All the physical storage attached to the back-end of the SVC cluster is pooled for efficient sharing of the storage bytes and performance. For example, the virtual disks that SVC exports to the hosts may be striped across multiple storage controllers at the back-end,

[1]Note: this paper describes a research prototype. It is not an IBM product announcement.

therefore providing performance not possible with a single storage controller. Thus, SVC performance far exceeds the performance of other storage devices in existence. In SPC-2 industry benchmarks SVC throughput is twice the throughput of the next best performing storage controller, reported as of Apr. 2008 [10].

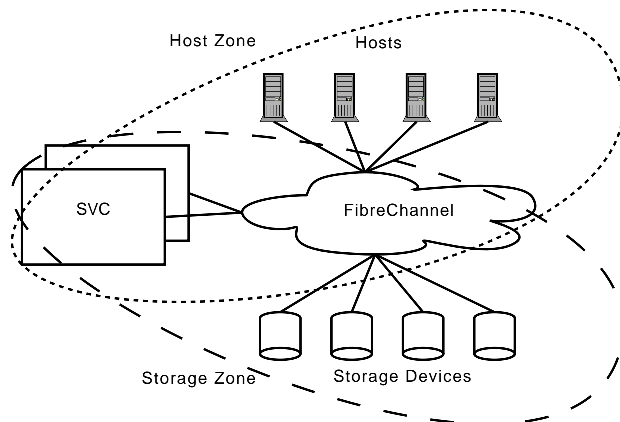

Fig. 1. Standard SVC Fibrechannel Topology

Logically, SVC stands between the hosts and the storage devices. Figure 1 illustrates the standard fibrechannel configuration using a fibrechannel SAN. Here, hosts and storage devices connect to SVC through a common SAN; however, the storage devices are isolated from the hosts using fibrechannel zoning. This protects the SVC-managed storage from direct access by hosts.

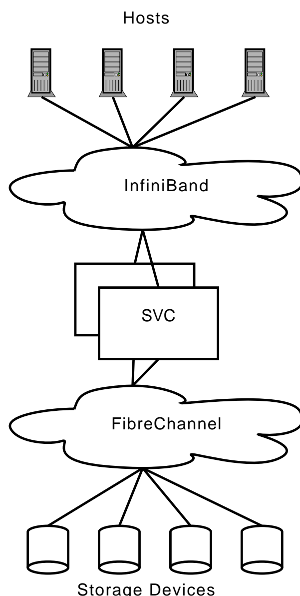

Fig. 2. SVC with IB Topology

In our modified SVC, the SVC nodes have a mixture of fibrechannel and InfiniBand ports. We added a PCIe Mellanox 4X InfiniBand host channel adapter to each SVC node in addition to the existing FC adapter. Figure 2 shows a configuration where all hosts connect to SVC via the InfiniBand fabric. The back-end is still fibrechannel, allowing the ordinary RAID controllers as the back-end storage. Fibrechannel target functionality in the modified SVC is still intact; so a mixture of InfiniBand and Fibrechannel hosts is permitted. It's possible for hosts to share a virtual disk independent of the interconnect used between host and SVC.

III. Implementation Overview

The SVC code, including its device drivers, run primarily at user-level. Main reasons for this design choice was performance and simplified development. In the case of the fibrechannel driver, a small kernel module maps the fibrechannel adapter's PCI registers into user-space, where the driver can control the FC adapter without incurring system call overhead. I/O completions as well as the other functions of SVC are handled by a small number of polling threads.

We developed an InfiniBand driver which also controls its adapter's from user-space, but instead of accessing PCI registers, it uses the standardized InfiniBand user-level *verbs* API. This API communicates with the IB adapter via shared memory queues and thus also avoids system call overhead, but has the benefit of not being specific to a particular IB adapter model.

Much work was done in the IB driver for performance and efficient use of resources. The driver has a completely asynchronous and re-entrant design, permitting a high-level of I/O parallelism while fitting well into SVC's threading model.

The driver is highly modular, consisting of five cooperating state machines, each implementing a different portion of the SCSI RDMA protocol. Breaking the logic out in this way, we were able to implement the target mode protocol fully with robust error handling, without putting unmanageable complexity into any one component.

Resource management and synchronization between the state machines is handled by an asynchronous semaphore mechanism, described later. In the next two sections we describe our IB driver design in detail.

IV. Driver State Machines

Several instances of five state machines implement the logic of the SVC IB driver. The five state machines control, respectively, SRP channel establishment and disestablishment, management of the SCSI I-T nexus, SCSI command and task management requests, buffer descriptor table fetches and DMA transfers. These state machines use the asynchronous semaphore mechanism described below to synchronize with each other and to control access to common resources. For brevity, we'll only describe the request and DMA state machines here.

A. Request State Machine

The request state machine, shown in Fig. 3, manages the life of each SRP SCSI command or task management request. The driver stores the state associated with a request

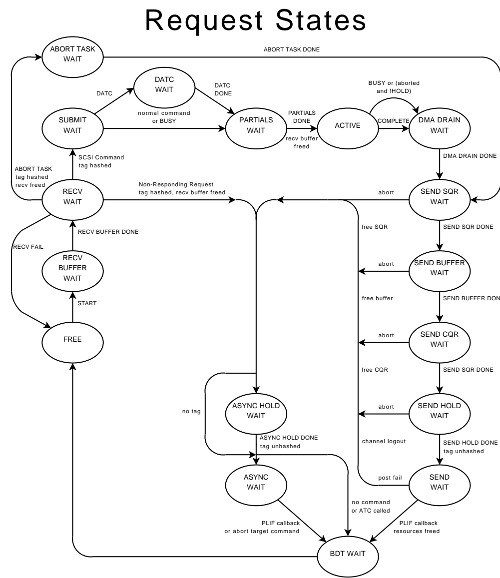

Fig. 3. Request State Machine

in a request structure, which is allocated from a fixed-size pool of 2000 structures per IB port.

All requests begin in the FREE state. When an IB port is brought on-line, each of the port's requests receives a START event, taking it to the RECV BUFFER WAIT state. Here, the request state machine tries to allocate an IB receive buffer.

There are fewer receive buffers than requests available per port, therefore some of the requests remain in RECV BUFFER WAIT waiting for a buffer, while others post their newly-allocated receive buffers to the InfiniBand shared receive queue and continue onto the RECV WAIT state. Requests remain in RECV WAIT until a message from a remote host is received into the corresponding buffer.

When a message arrives, the receive logic looks up the IB queue pair number from the message header to identify the the SRP channel, checks that the message is well-formed and classifies it based on the message type. SCSI commands and most task management requests bring the request state to SUBMIT WAIT where the request is inserted into a tag hash table (used for abort processing) and submitted to the SVC code. Abort task requests and miscellaneous SRP requests such as initiator logout are handled by the driver directly and take different paths in the state machine.

For SCSI commands that transfer data, the request state machine initiates a fetch of the buffer descriptor table (BDT) in parallel with submitting the command to SVC. The buffer descriptor table fetch is controlled by another state machine (not shown). Usually, buffer descriptors are included in-line with commands, and in this case, all the BDT state machine has to do is allocate a buffer to hold the descriptor entries and copy them from the receive buffer. But if necessary, the BDT state machine will initiate RDMA's to the host to fetch any part of the descriptor table not included with the request.

Once a command has been submitted to SVC, the request enters the PARTIALS WAIT state, where it remains until the BDT state machine has copied any partial (in-line) buffer descriptors from the receive buffer. After partial descriptors have been copied out, the receive buffer is freed where it can be used by another request. After the receive buffer is freed, the request enters the ACTIVE state where it waits for the SVC common code to direct DMA transfers and the final status transfer. DMA operations are controlled by the DMA state machine, described below.

Once the common SVC code has returned status for the execution of the SCSI command, the the request enters DMA DRAIN WAIT where it waits for RDMA's associated with the request to complete. The next series of states are used to allocate resources for the SRP response message. This involves reserving a slot in the IB send queue, allocating a send buffer, reserving a slot in the completion queue and posting the response message. Once the response is posted, the request enters BDT WAIT where it releases the buffer descriptor table and returns to FREE where the process starts again.

The remaining request states: deferred abort target command (DATC), SEND HOLD WAIT, ASYNC HOLD WAIT and ASYNC WAIT are used to handle errors and to synchronize with SVC when processing abort task requests.

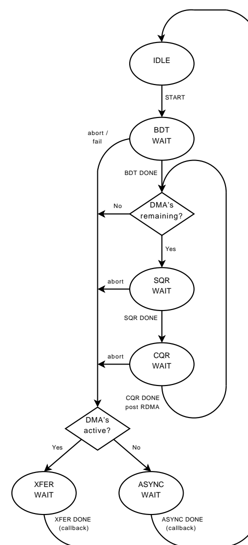

Fig. 4. DMA State Machine

B. DMA State Machine

Figure 4 shows the DMA state machine, which controls data transfers between hosts and SVC. When a DMA operation is requested, the IB driver allocates a DMA control structure. From the initial IDLE state, a DMA enters BDT WAIT where it synchronizes with the buffer descriptor table state machine. It then enters its main loop where it initiates RDMA read or write operations to the host. This is slightly tricky in that both SVC and the host provide scatter-gather lists for the transfer, making it sometimes necessary to submit several RDMA operations to complete a single DMA operation. As resources permit, the driver executes these RDMA operations in parallel.

Within the main DMA loop, there are two states: SQR wait and CQR wait. In SQR WAIT, the driver reserves a space in the IB send queue, taking care not to exceed the negotiated number of concurrent RDMA reads that the host's channel adapter can accept. In the CQR WAIT state, the driver reserves a slot in the IB completion queue. Once both table slots have been reserved, the DMA state machine submits an RDMA operation and loops until all RDMA's have been started.

Once all RDMA operations have been submitted, the DMA state machine enters XFER WAIT where it waits for the RDMA's to complete. It finally executes a callback to notify the requester of the completion and returns to the IDLE state. In the case of a zero-length DMA request or the receipt of an abort before RDMA's could be started, the DMA state machine enters ASYNC WAIT instead of XFER WAIT where it uses the sched_async mechanism (described below) to generate a context for the completion callback.

V. Driver Synchronization

At the core of the InfiniBand driver is a counting and queuing semaphore mechanism that, instead of blocking, schedules an asynchronous callback when the semaphore count is too low to satisfy a **P** (down) operation [9]. This mechanism is used to control access to limited driver resources, to synchronize between the state machines and to schedule asynchronous callbacks outside of the caller's context for deadlock avoidance. Moreover, the mechanism has a cancel operation, making it possible to interrupt the wait in case an error or arrival of new information obviates the need for the resource. Using a single, generalized mechanism for driver synchronization greatly simplifies exception handling.

The semaphore count and queue are implemented by a *wait channel*. When the wait channel is initialized, it's given the initial semaphore count and an associated run queue.

```
void sched_wait_channel_init(wait_channel *wcp,
    run_queue *rqp, uint32_t count)
```

The **P** operation is provided by the sched_wait and sched_wait_n functions. If the count is positive, sched_wait decrements the count by one and returns true; otherwise, it saves the given callback function and argument into the caller-supplied wait entry, enqueues the wait entry onto its internal queue and returns false.

The sched_wait_n function is a more general interface. If the semaphore count is at least n, it decrements the count by n and returns true; otherwise, it leaves the semaphore count unchanged, saves the requested n value, enqueues the wait entry and returns false.

```
int sched_wait(wait_channel *wcp, wait_entry *wep,
    callback *fn, void *arg)
int sched_wait_n(wait_channel *wcp, uint32_t n,
    wait_entry *wep, callback *fn, void *arg)
```

Alternatively, the semaphore count can be decremented by conditional interfaces, sched_try and sched_try_n. These work exactly as sched_wait and sched_wait_n, except that if the semaphore count is insufficient, they simply fail without scheduling a callback.

```
int sched_try(wait_channel *wcp)
int sched_try_n(wait_channel *wcp, uint32_t n)
```

There are also two regular and two conditional forms of the **V** (up) operation. The regular forms, sched_signal_wait and sched_signal_wait_n, increase the semaphore count by one or n, respectively, and move any waiters satisfied by the new count from the wait channel queue to the run queue. As waiters are removed, the semaphore count is decremented by the amount that was specified in the corresponding wait call.

```
void sched_signal(wait_channel *wcp)
void sched_signal_n(wait_channel *wcp, uint32_t n)
```

The conditional forms, sched_signal_first and sched_signal_all, only have an effect if there are queued waiters. If there are queued waiters, sched_signal_first moves the first waiter to the run queue, leaving the semaphore count unchanged. Similarly, sched_signal_all function moves all queued waiters to the run queue, leaving the count unchanged.

```
int sched_signal_first(wait_channel *wcp)
void sched_signal_all(wait_channel *wcp)
```

A special function, sched_async, enqueues a request directly to the run queue. This can be used to schedule an immediate callback outside of the calling context.

```
int sched_async(run_queue *rqp, wait_entry *wep,
    callback *fn, void *arg)
```

A pending wait entry can be cancelled by a call to wait_cancel. This function restores any changes the request has made to the semaphore count (possibly awakening other waiters) and returns true on success. It returns false if the wait entry is neither on the wait channel nor the run queue, i.e. is already being executed.

```
int wait_cancel(wait_entry *wep)
```

Finally, a call to sched_run removes all wait entries from the run queue and executes their callback functions. If these callbacks schedule more work onto the run queue, sched_run loops internally (up to a limit of a few iterations) and processes the new entries, thereby avoiding the introduction of unnecessary context switches when the completion of one event triggers the start of another.

We call sched_run from the driver's polling thread after it has processed completions from the IB completion queue. Therefore, in a single poll, the driver handles all pending IB completions and starts all new work that can be started without having to wait for the polling thread to be rescheduled.

```
int sched_run(run_queue *rqp)
```

How the semaphore count is initialized and which **P** and **V** functions are used depends on the application of the semaphore. Following are some examples:

To implement a simple mutex, the semaphore count is initialized to 1. The semaphore is acquired using sched_wait and released using sched_signal. If several requesters try to to acquire the semaphore at the same time, one will be successful while the others will be queued. When the first requester releases the semaphore, the count increases by one which releases the next in line.

To limit access to a pool of n resources, the count is initialized to n. Resources are reserved by a call to sched_wait_n and released by a call to sched_signal_n. This usage provides first come first served fairness by releasing waiters in order and, only when sufficient resources have been freed to satisfy their requested counts.

In order to hold a single requester until an event occurs, the count is initialized to 0. The requester waits via sched_wait and the event is signalled by sched_signal. If the requester waits before the event occurs, it will be queued and then released by the event. If the event occurs first, the semaphore count will become 1 allowing the requester to decrement the count without waiting.

In order to hold zero or more requesters at a barrier until an event occurs, the count is initialized to 0. The requesters wait via sched_wait and the event is signalled by sched_signal_all. In this case requesters must check if the event has already occurred and call sched_wait under a lock in order to guarantee that they do not miss the event. Because sched_wait never actually blocks the calling thread, it's safe to hold the lock through the sched_wait call.

VI. Performance Results

We evaluate the performance using three different measures: I/O throughput, average number of SCSI completions per second and I/O latency as seen by a single requester.

Our benchmarking environment consists of the following: the client machines generating the I/O workload consist of up to 8 IBM model LS21 blade servers each having two sockets of 2.4 GHz AMD Opteron 2216-HE and a Mellanox MT25208 InfiniBand host channel adapter. The client blade servers are running SuSE SLES 10 2.6.18.8-xen Linux. The client IB adapters connect to the SVC cluster through a number of highly-interconnected InfiniBand switches. Our SVC cluster prototype consists of two IBM System-X Model 3650 servers each with 16 GB of RAM and a Mellanox 4X InfiniBand host channel adapter.

Note that we measured only the front-end performance of our SVC prototype, namely the 100 percent SVC cache

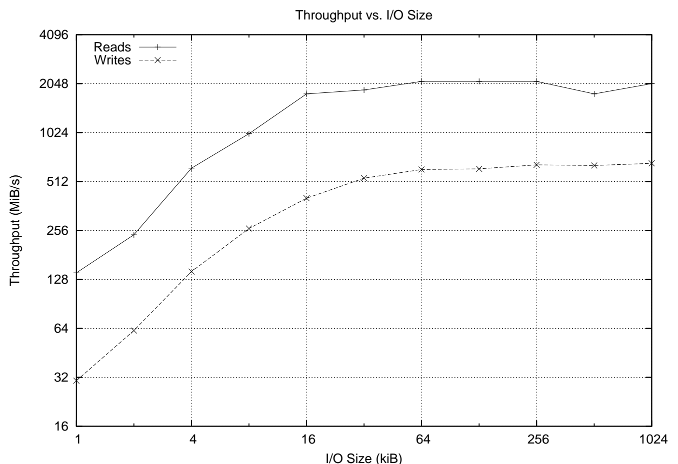

Fig. 5. I/O Throughput by Request Size

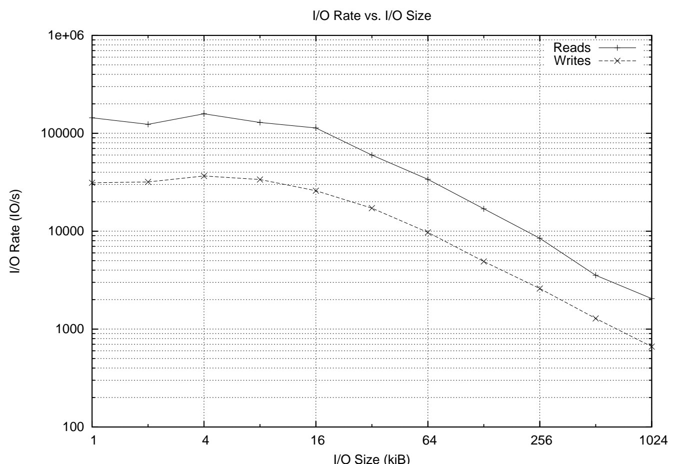

Fig. 6. I/O Rate by Request Size

hit scenario. In our prototype, the InfiniBand interface exists only in the SVC front-end. Therefore, the back-end performance of our prototype would have been the same as the SVC product. Note also that since SVC is a storage virtualization engine which sits inline between hosts and the multiple storage controllers possibly from multiple vendors in the back-end. As such the back-end performance of SVC is very much dependent on the number of storage controllers and disk drives in the back-end and therefore not reported here; see the published benchmark results for those [10].

Figs. 5 and 6 show read and write throughput and number of I/O operations per second, respectively, as a function of I/O size. Here, 8 hosts are each queuing 16 simultaneous requests to the two SVC nodes, so I/O latency is masked by pipelining.

Toward the left-hand side of these figures, the system is limited by I/O rate, peaking at about 144,000 read operations per second [2] Toward the right-hand side, reads are

[2] Given zero-byte read and write operations (i.e. no RDMA phase), the SVC nodes process more than 200,000 requests per second.

limited by InfiniBand bandwidth. In both cases, writes are slower as the SVC nodes commit the data to both nodes caches before returning status back to the driver.

Figure 7 shows the I/O latency for a single node issuing requests serially, i.e. no command queuing. In this test, small reads complete in about 30 microseconds. This figure is much different from the reciprocal of the average I/O rate above where pipelining effects greatly improve the overall I/O rate.

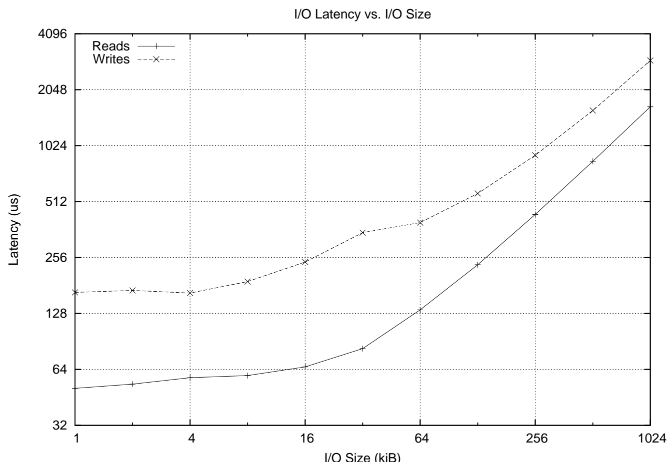

Fig. 7. Single-Client I/O Latency by Request Size

Acknowledgements

We're grateful to the technical support and advice of Bill J. Scales in the SVC development organization in Hursley, UK.

References

[1] Mellanox Technologies. *InfiniBand Strengthens Leadership as The High-Speed Interconnect Of Choice*, http:www.mellanox.com/pdf/applications/Top500Nov07.pdf.

[2] Olaf Schneider, Frank Schmitz Ivan Kondov, and Thomas Brandel *Opus Grid Enabled Operton Cluster with InfiniBand Interconnect*, Lecture Notes in Computer Science, ISBN 978-3-540-75754-2, pp. 840-849.

[3] Gordon Haff *High Performance Computing Meets Databases*, Computational Methods in Science and Technology, Special Issue 2006, 71-74.

[4] Ivy Schmerken *Test Finds Infrastructure Eliminates Latency Spikes*, http:www.advancedtrading.com/blog/archives/2007/03/

[5] Security Technology Analysis Center *Exegy Ticker Plant with InfiniBand*, http://www.stacresearch.com.

[6] *Designing for Performance on Wall Street - The Need For Speed* The Techdoer Times, Jan 29, 2008, http://www.techdoer.com.

[7] ANSI/INCITS Technical Committee T10 *SCSI RDMA Protocol (SRP)*, INCITS 365-2002 [R2007]

[8] J. S. Glider, C. F. Fuente, and W. J. Scales. *The software architecture of a SAN storage control system.*, IBM Systems Journal, 42(2):232–249, 2003.

[9] E. W. Dijkstra. *Solution of a problem in concurrent programming control.*, Communications of the ACM, v.8 n.9, p.569, Sep. 1965.

[10] Storage Performance Council. *SPC-2 Benchmark Results: IBM San Volume Controller 4.2*, Sep. 2007, http://www.storageperformance.org