SAN Optimization for High Performance Storage with RDMA Data Transfer

Jae Woo Choi, Young Jin Yu, Hyeonsang Eom, Heon Young Yeom
Department of Computer Science and Engineering
Seoul National University
Seoul, South Korea
jwchoi@dcslab.snu.ac.kr

Dong In Shin
Taejin Infotech
Seoul, South Korea
tuataras@taejin.co.kr

***Abstract*— Today's server environments consist of many machines constructing clusters for distributed computing system or storage area networks (SAN) for effectively processing or saving enormous data. In these kinds of server environments, backend-storages are usually the bottleneck of the overall system. But it is not enough to simply replace the devices with better ones to exploit their performance benefits. In other words, proper optimizations are needed to fully utilize their performance gains. In this work, we first applied a high performance device as a backend-storage to the existing SAN solution, and found that it could not utilize the low latency and high bandwidth of the device, especially in case of small sized random I/O pattern even though a high speed network is used. So, we propose a new design that contains three optimizations: 1) removing disk legacies to lower I/O latency; 2) parallelism to utilize the high bandwidth of the device; 3) *temporal merge* mechanism to reduce network overhead. We implemented them as a prototype and found that our solution makes substantial performance improvements in terms of both the latency and bandwidth.**

Keywords—SAN; RDMA; High Performance Storage;

I. INTRODUCTION

Whereas the computing capability and networks are more powerful today due to the increasing number of cores on systems and high throughput of data transfer, the performance development of storage systems is not up to speed with these evolutions, and therefore become the bottleneck of the overall server environment. To overcome this problem, demands for high performance storages are ever more increasing and the storages based on HDD can no longer satisfy this need. So, other types of storages and techniques such as flash-SSDs, DRAM-based SSDs or using main memory for data storage are rising as alternatives, and research for next generation storages such as PCM, FeRAM and MRAM are actively in progress [3,5].

SAN(Storage Area Network) is the system that replaces the data bus between host computer and storage with high speed networks and provides access to block level data storage via standard network protocol. In SAN environment, the host computer and the storage server are typically called *Initiator* and *Target* respectively. This kind of system is widely used in enterprise and small to medium sized business environment and can be effectively implemented with high-performing interconnect solutions supporting Gigabit Ethernet, Fibre Channel, Infiniband and so on [6,8,9].

In this work, we employed DRAM-based SSD using PCI-E interface and Infiniband network supporting RDMA data transfer [10,13]. We first applied the storage device to existing Infiniband SAN solution, SCST project using SCSI RDMA Protocol (SRP) [12], and discovered huge performance degradation compared with the device capability, especially in case of small sized random I/O pattern.

The first problem we figured out is Disk legacies in SAN I/O path. Current block device I/O path has been developed with diverse optimizations based on HDD under the assumption that disk is very slow. This kind of assumption causes large overheads to the SAN I/O path when high performance storages are applied. Existing SAN solutions usually conduct storage virtualization on the SCSI layer [1]. But the SCSI layer itself can make the I/O path ineffectively longer on the low latency device and there are some researches that support layer overheads [4]. Also, I/O scheduler usually adopts a policy that utilize delays which is very effective for HDD based device but not for others.

The second is the lack of parallelism on the procedure handling I/O requests in *Target* side. In case of the high performance storages, great parallelism is implemented on device level so that simultaneously incoming I/O requests can be handled effectively by exploiting their high bandwidth. Actually, most procedures on *Target* side are independent of each other which mean that they can be processed in parallel. However serialized executions still exist on the procedures, especially relating to device I/O, which cause enormous performance penalty for small sized random I/O pattern.

The last one is the processing overhead in networks. Actually, Infiniband with RDMA data transfer shows great performance. But when small sized random requests are transmitted, the overhead that was unnoticed due to huge latency from disk I/O is exposed in high performance storage environments. Therefore, we designed some optimizations for SAN I/O path to overcome the problems mentioned earlier.

II. TRADITIONAL SAN I/O PATH

Every I/O request holds essential information such as read/write type, start address and data size to execute I/O operations. In Linux I/O subsystem, the unit of the I/O request can be changed by going through the system layers, but the information is constantly maintained in units. Figure1 shows a typical SAN I/O path abstractly. As you can see, the I/O path in *Initiator* side is quite similar to the path for local SCSI devices. An I/O request initiated from the application layer goes through the file system and is transferred to the block layer. The request can be merged with an existing request if their data locations are adjacent. This is called "*spatial merge*". The requests are then inserted into the request queue and must wait until the I/O scheduler dispatches them to the SCSI layer. Finally, the SCSI command is constructed from the request at the SCSI Mid-level and forwarded to Front-End *Initiator* driver to make a command including information for I/O processes in *Target* and transmitted via high speed network.

Depending on the command from *Initiator*, a sequence of procedures such as RDMA data transfer or device I/O is executed on the *Target* side. After the operations are complete, *Target* sends a completion response to *Initiator* to finish the I/O request entirely.

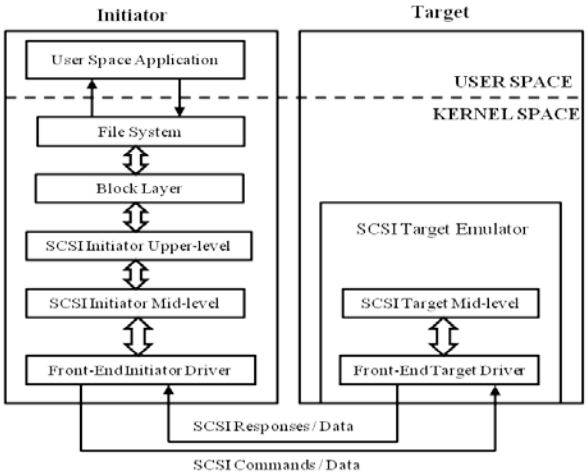


Figure 1: Traditional SAN I/O path in Linux subsystem[1]

III. DESIGN EXPLORATION

In this section, we propose the following three optimizations needed for the new SAN solution based on high performance storages with RDMA data transfer. 1) Removing Disk legacies in I/O path, 2) Increasing Parallelism for device I/O and 3) *temporal merge* in RDMA data transfer

A. Removing Disk Legacies.

As mentioned earlier, SCSI layer itself can be a penalty for low latency device because it makes the I/O path longer. So, many drivers for these kind of up-to-date devices usually do not adopt the SCSI layer. But existing SAN solutions conduct storage virtualization on the SCSI level and transfer SCSI command to *Target*, including SRP and SRPT in SCST project for the Infiniband [12].

Therefore, we removed the SCSI layer in the SAN I/O path and virtualized the storages on generic block layer. Figure 2 shows the abbreviated path. The block layer directly interacts with Front-End Initiator Driver that we implemented. Thus the key I/O unit in the driver is *bio structure*, not the SCSI command. In addition, the information from *bio structure* is referenced to compose a command what will be sent to *Target.*

Current I/O scheduler in Linux subsystem commonly adopts a policy with disk assumption. CFQ is the typical optimization for HDD-based device I/O which admits relatively long delays. Therefore, if the existing scheduler is used for the low latency device, the policy should be changed to NOOP.

However we did not change the scheduler because it still conducts complex operations not suitable for high-end devices. These operations include the usage of delays for expecting data merge and the usage of *request structure* that is employed by the existing scheduler not *bio structure* that we use to maintain waiting queues. So, we implemented a very simple I/O scheduler that dispatches the I/O requests from waiting queues and sends them to *Target* as soon as possible. Since the transfer protocol we designed between the machines is also based on *bio structure*, it becomes easier and more efficient to construct *bio structure* again with the information from *Initiator* in *Target* side. From these modifications, we could simplify the SAN I/O path and lower the I/O latency substantially.

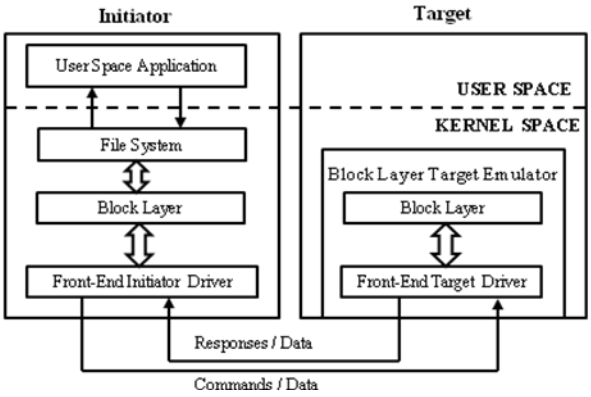


Figure 2: new SAN I/O path which removed the SCSI Layer

B. Increasing Parallelism

The second optimization is to increase parallelism in a sequence of procedures handling I/O requests from *Initiator* in *Target side*. In high performance storage system, parallelism is implemented on device level so that simultaneously incoming I/O requests can be handled effectively by exploiting its high bandwidth.

The procedures can be classified into four different schemes. 1) Analyzing commands from *Initiator*, 2) Executing the RDMA data transfer, 3) Device I/O, and 4) Sending response to *Initiator*. These operations from different I/O requests are almost independent of each other so that they can be processed in parallel. Especially, the parallelism of the device I/O is very critical for overall performance because it determines whether to exploit the higher bandwidth of the device or not. Actually, the existing SAN solution also

conducts parallel processing for procedures with multi-threads. However, we have discovered that the device I/O is executed in serial fashion and it has resulted in severe performance degradation for small sized I/O patterns. Therefore, we optimized the procedures to enable all operations to be handled by multi-threads in parallel, including device I/O, which lead to great throughput increment for the overall system.

C. Temporal Merge

Every data transfer via network includes pre-processing and post-processing operations. They include the essential processes that should be executed before and after the data transfer respectively.

Actually, Infiniband with RDMA data transfer shows great performance. But there are some inefficient aspects in using the existing transfer protocol as it is. The first picture in figure 3 shows how the existing RDMA data transfer works with write requests. A command including I/O information from *Initiator* and a response including completion information are needed for every RDMA data transfer. And network transfers such as command, data and response all have pre-processing and post-processing operations as well.

The processing overheads have been relatively very small in HDD-based SAN environments because the device I/O latencies are so long causing network overheads to be unnoticeable. But in case of the low latency devices, overhead can affect the system performance. Actually, when large sized data is transmitted, the data transfer time dominates the whole latency in the network so the processing overhead hardly affects the system. However once small data is transferred, the processing overhead in the network is finally exposed because data transfer time per request is relatively small.

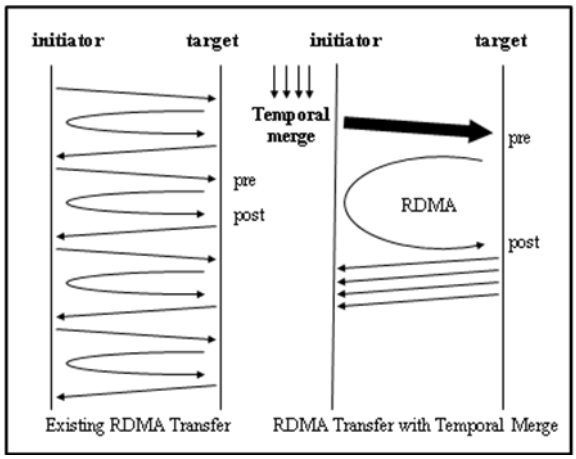


Figure 3: Comparison between Existing RDMA transfer and Temporal Merge applied RDMA transfer

Thus, we propose *temporal merge* to mitigate the processing overhead for small sized random I/O pattern. *Temporal merge* is a technique that merges a number of requests regardless of their spatial adjacency and has been already introduced to exploit peek throughput in low latency device [15]. We applied this idea to RDMA data transfer.

The second picture in Figure 3 describes the RDMA transfer method with *temporal merge*. As you can see, the small requests on *Initiator* are merged and constructed to a big command, called *jumbo command* and sent to *Target* in one. This can lead to latency benefits. According to table1, it is more efficient to send one 256bytes data than four 64bytes data in a row.

In addition, RDMA supports scatter /gather DMA, which can transfer a number of spatially discontinuous data in one RDMA execution. We believe that this also helps reduce the processing overhead. After the RDMA data transfer, device I/O will be conducted in parallel with multi-threads. Finally, the responses for I/O completion are transferred to *Initiator* one by one in order of termination, which increase the response speed for each request.

However, *temporal merge* is not always beneficial because *temporal merge* can cause some serial executions among the merged requests. This obstructs the active parallelism in *Target* side and degrades overall performance in the system. From the experiments, we found that *temporal merge* is effective in intensive I/O situation, so we needed to set a criterion for determining if the I/O condition is intensive or not. Therefore, we implemented the *temporal merge* that is activated depending on the type of I/O condition.

As described earlier, we applied *temporal merge* technique and we figured out that it works properly and gain performance benefits when small sized random I/O workloads occurs intensively.

bytes	min[usec]	max[usec]	typical[usec]
64	4.69	12.64	4.73
128	4.87	12.72	4.91
256	5.24	11.03	5.28
512	6.51	12.42	6.56
1024	8.27	15.84	8.32

Table 1: Infiniband RDMA send latency

IV. EVALUATION

We experimented with two machines, one is *Initiator* and the other is *Target*. They both have two Intel Xeon E5630 2.53GHz quad core CPUs (total 8 cores), and have 8GB, 16GB main memory respectively. Experiment is executed in Linux 2.6.32 kernel. DRAM-based SSDs are equipped in *Target* side (total 512GB = 64GB*8) and we use one of them in this work as a backend-storage. We also have two ConnectX-2 MHQH18B-XTR Infiniband cards from Mellanox for *Initiator* and *Target*.

We first evaluated the performances of the existing SAN solution, and then compared the result with the new solution we implemented. We specify our solution, Block RDMA Protocol (BRP), with numbering of BRP-1, BRP-2 and BRP-3: BRP-1 includes only the optimization removing disk legacies; BRP-2 adds the parallelism optimization to BRP-1; BRP-3 adds *temporal merge* to BRP-2. BRP alone means BRP-3, the final version.

A. *Latency & Bandwidth*

As described before, we removed SCSI layer and implemented simple and fast I/O scheduler to lower the I/O latency in SAN environment. Table2 shows the read/write latency comparison between the existing solution and the new one. The results are measured by *dd* test with the direct I/O option. We conducted 10,000 direct I/Os repeatedly for 4KB size and calculated the average of latencies. The numbers in parentheses are the values for the software only overhead which does not include the device latency, which is between 12usec and 13usec. According to the results, read latency is lowered about 31.7(39.2)% and write latency is also reduced about 28(-33.8)%. These results show that our optimization mechanism of disk legacy removal is effective for low latency devices.

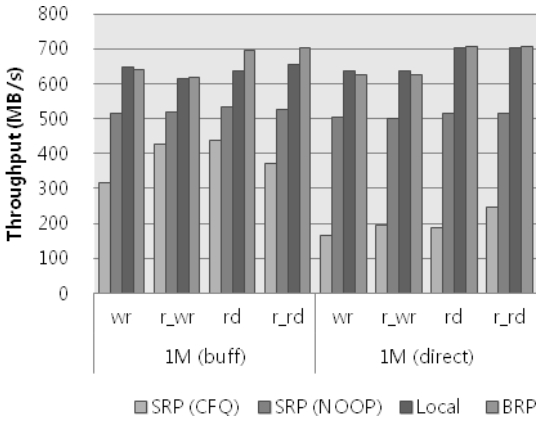
I/O Type	SRP-SRPT (usec)	BRP-BRPT (usec)	Latency Reduction
Read	63 (51)	43 (31)	-31.7 (-39.2)%
Write	75 (62)	54 (41)	-28 (-33.8)%

Table 2: Latency Comparison between SRP-SRPT I/O path and BRP-BRPT I/O path

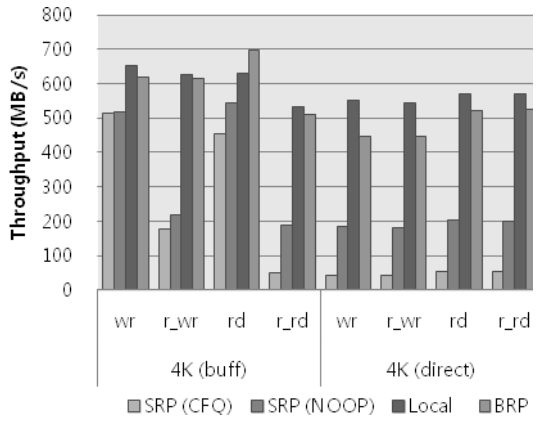
Next, we conducted FIO micro-benchmark to evaluate throughput for diverse I/O patterns. Figure 4 depicts the throughput comparison between SRP-based SAN and BRP-based SAN. We executed 16 I/O threads and each thread has 3GB workloads, which in total is 48GB. I/O types are sequential read/write, random read/write and buffered/direct I/O. Two request sizes are set for the small (4KB) and the large (1MB). The bars on the charts indicate SRP with CFQ schedule policy, SRP with NOOP schedule policy, local I/O test on the DRAM-SSD and the BRP we implemented.

As you can see, CFQ has a huge disadvantage for high performance storage and NOOP policy can be a solution to overcome the drawbacks of CFQ and it actually works for most I/O patterns. But it still shows relatively low performance compared with the Local I/O. Buffered I/O using page cache in the File system can mitigate the penalties for some cases, but in case of direct I/O or small sized random I/O, large performance gap still exist between them.

We implemented the new SAN solution with three optimizations mentioned before, and accomplished great throughput close to the Local I/O performance. Figure 4(a) describes the throughputs of large sized I/O patterns. They generally utilize the high bandwidth well in both SRP and BRP. We also figured out that BRP can be an effective solution under large size workloads as well. Since *temporal merge* is not conducted with large size request, the performance increments are only result of removing disk legacies and increasing parallelism. In some cases, the result of Local I/O is inferior to the ones of BRP, and this is because I/O operations in the DRAM SSD driver are based on polling, not interrupt. In low latency devices, polling might be better than interrupt [7], but cause CPU burst. In SAN environment, the consumption of computing resources can be distributed. Figure 4(b) shows the throughput of small sized I/O patterns. As described in prior sections, the biggest goal of our optimization was to increase the performance of this pattern and we achieved great throughput increments comparable with Local I/O result in almost all cases. *Temporal merge* technique affects intensive I/O patterns, and the case of random buffered write can be the best beneficiary of the technique. This experiment was conducted under 16 threads so that means other cases such as direct I/O is not faced with I/O intensive situation. Thus, *temporal merge* are not executed in those cases at all.



(a) Large size I/O



(b) Small size I/O

Figure 4: Throughput comparison between SRP-SRPT and BRP-BRPT for a variety of I/O patterns

B. *BRP Performance Analysis*

In this section, we analyze the performance of our three optimizations. Figure 5 indicates the throughput variation as the number of threads gradually increases. It is small sized random write pattern in direct I/O. BRP-3T on the chart is distinguished with BRP-3 in that BRP-3T always executes *temporal merge* technique if it is possible. BRP-3 has a threshold that activates *temporal merge* at proper time. According to the chart, the throughputs in SRP and BRP-1 which are not optimized for parallelism seem to be saturated at 16 threads while the performance of others continuously increase as the number of threads increase.

We can see the effect of *temporal merge* from the results of BRP-2 and BRP-3T. The results indicate that in the intensive

I/O situation, *temporal merge* can be effective and the throughput is close to the device performance itself. But in sporadic I/O situation, *temporal merge* obstructs the parallelism. In other words, the penalties from interrupting the parallelism are bigger than the benefit from *temporal merge*. Therefore, we needed to set the threshold to determine the intensive I/O situation, and the number of in-flight requests became the condition value. The proper threshold was decided from the results of repeated experiments.

Figure 6 depicts the performance analysis of the three optimizations with 16 threads. The results indicate normalized throughputs for each optimization based on the Local I/O. The throughput of SRP usually reaches about 35% of Local I/O throughput. In case of buffered random write, each optimization affects the performance and the effect of BRP-1 is bigger than other I/O types because the lower latency increases the efficiency of page cache. Direct I/O does not have any benefit from *temporal merge* under 16 threads as we explained before. So, we can see the performance increases only at the BRP-1 and BRP-2. Generally, the performances of BRP seem reasonable enough. In case of small I/O pattern, every case is over 80% of Local I/O performance and buffered random write reaches 98% of that.

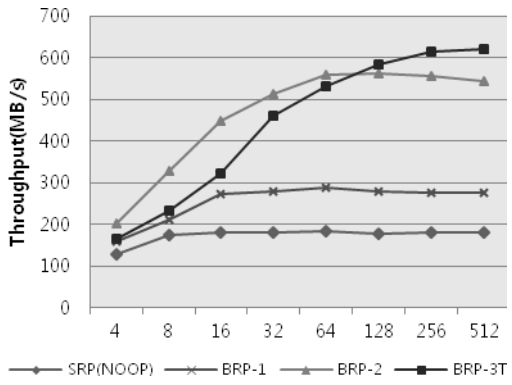


Figure 5: performance analysis with the number of threads on random write pattern

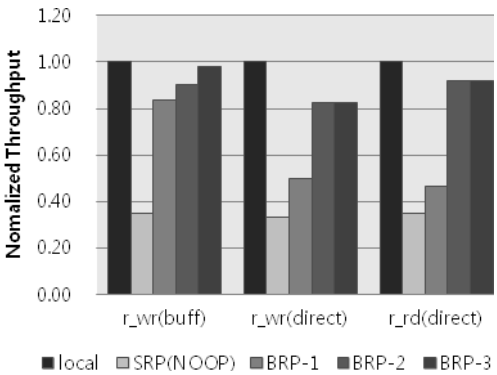


Figure 6: Performance analysis for three optimizations on BRP

C. *Real Workload Benchmark*

Finally we tested our solutions on DBMS. We used BenchmarSQL tool with TPC-C benchmark, and postgreSQL for database program [2,11,14]. The number of warehouse is 320 and we evaluated the average of the transactions per minute as the number of terminal increases from 8 to 128. Figure 7 shows that BRP has better performance than SRP in DBMS environment as well. But the degree of performance increment is relatively smaller than the result of micro benchmark. After the trace analysis, we discovered that I/O rate from DBMS is hardly intensive and the thing is the optimizations in DBMS. Actually, it conducts many optimizations internally, but they are unnecessary for high performance storages. Therefore, optimizations on diverse layers, from application to device, are needed in high performance storage environments

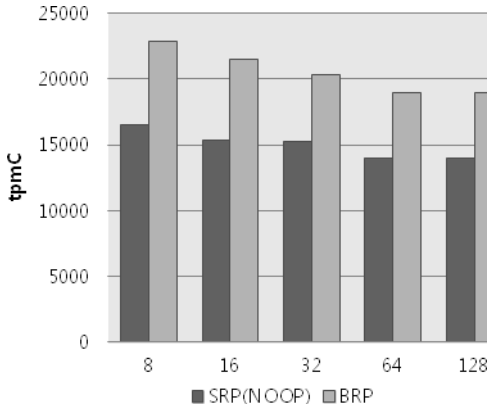


Figure 7: TPC-C benchmark on PostgreSQL

V. CONCLUTION

We have designed and implemented a new SAN solution for high performance storages with RDMA data transfer. In this work we proposed three optimizations that are really effective. The first one was removing disk legacy in SAN I/O path, so we eliminated the SCSI layer that makes I/O path longer and replaced the I/O scheduler with simple and fast one. From this work, we could lower the I/O latency a lot. The second optimization that we proposed was increasing parallelism. We analyzed the existing SAN solutions and found several places where I/O request is serially handled one by one, which we changed that every work involved I/O requests is handled in parallel with multi-threads. From the result, the throughputs in every type of I/O patterns are increased. And finally we proposed *temporal merge* technique to reduce network processing overhead for small sized I/O pattern. The results showed us that this technique is effective in intensive I/O situations.

REFERENCES

[1] ASHISH PALEKAR, DESIGN AND IMPLEMENTATION OF ALINUX SCSI TARGET FOR STORAGE AREA NETWORKS, In ALS' 01

[2] BenchmarkSQL, http://sourceforge.net/projects/benchmarksql/

[3] Burr, G. W., Overview of candidate device technologies for storage-class memory, IBM Journal of Research and Development, Vol. 52 Issue 4.5, July 2008, p449-464

[4] CAULFIELD, A. M., ET AL. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In MICRO'10 (2010), pp. 385–395.

[5] Freitas, R. F., Storage-class memory: The next storage system technology, IBM Journal of Research and Development, Vol. 52 Issue 4.5, July 2008, p439-447

[6] IBM, Introduction to Storage Area Networks, http://www.redbooks.ibm.com/redbooks/pdfs/sg245470.pdf

[7] Jisoo Yang, When Poll is Better than Interrupt, Fast'11

[8] Mellanox, Building a Scalable Storage with InfiniBand, http://www.mellanox.com/relateddocs/whitepapers/WP_Scalable_Storage_InfiniBand_Final.pdf

[9] Mellanox, InfiniBand Storage, http://www.mellanox.com/pdf/whitepapers/InfiniBand_Storage_WP_050.pdf

[10] Mellanox, Introduction to InfiniBand, http://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf

[11] PostgreSQL, http://www.postgresql.org/

[12] SCST, Generic SCSI Target Subsystem for Linux, http://scst.sourceforge.net/

[13] TAEJININFOTECH, HHA 3804, http://www.taejin.co.kr

[14] TPC-C benchmark, http://www.tpc.org/tpcc/

[15] Young Jin Yu, Exploiting Peak Device Throughput from Random Access Workload, In Hotstorage'12