On Performance of Parallel iSCSI Protocol for Networked Storage Systems

Qing (Ken) Yang
Dept. of Electrical and Computer Engineering
University of Rhode Island
Kingston, RI 02881
qyang@ele.uri.edu

Abstract:

A newly emerging protocol for storage networking, iSCSI [1,2], was recently ratified by the Internet Engineering Task Force [3]. The iSCSI protocol is perceived as a low cost alternative to the FC protocol for networked storages [5,6,7,8]. It allows block level storage data to be transported over the popular TCP/IP network that is widely understood. This paper presents a new storage architecture allowing parallel processing of iSCSI packets. By leveraging inexpensive Ethernet ports, we are able to greatly improve the throughput of iSCSI storages through parallel processing. We have carried out a performance analysis to evaluate the performance of the new architecture as compared to the existing iSCSI storages. Our numerical results show that the new architecture has good performance potential.

1. Introduction

Increasing performance of a shared storage in a SAN has been the design goal of storage designers for a long time. Because of the inherent limitations of storage devices such as disks or tapes that involve mechanical operations for each data access, storage operations have been the major bottleneck in an IT system compared to CPU, RAM, and network that have all improved by several orders of magnitudes over the past decade. Technologies to improve storage performance can generally be classified into two categories: caching and parallel processing. Our previous research [4] is an example of reliable and effective caching technology. Examples of parallel processing include RAID (redundant array of independent disks) and various parallel interconnect technologies such as InfiniBand, Ultra Wide SCSI, Gigabit FC, etc. These technologies aim at increasing the storage throughput by means of parallel data accesses (RAID), parallel connectivity (InfiniBand and Ultra Wide SCSI), and high data rate (FC).

With rapid advances in network technologies, speed of networks such as Ethernet and FC increased dramatically at the same time the cost of network components such as NIC (network interface card) and switches decreases dramatically. The cost of per network port is very inexpensive, which makes it economically feasible to have multiple ports at server and multiple ports at a storage system. Such multiple ports provide parallel connections between servers and storages allowing parallel data/control transfers between servers and storages. Parallel communications between servers and storages can happen in two dimensions: multiple logical connections over one physical link and parallel connections across multiple physical links.

Existing storage technologies make use of these parallel connections to improve storage throughput in a straightforward manner. For example, the iSCSI protocol allows parallel connections but will collect and order packets in a buffer queue in a session before execution. The protocol mandates that all packets have to be handled in the order in which the packets were originally generated from the initiator side. Such requirement limits the potential parallelism and performance gains since it essentially serializes the processing of packets coming from parallel ports. Analyzing such networked storage systems, we found a surprising similarity between parallel network connections in a storage system and the parallel processing technology in processor designs. The logical parallel connections over a physical channel is analogous to pipelining machine that issues one instruction per cycle with parallelism occurring at different processing stages, and multiple physically connected channels are analogous to super-scalar machine that issues multiple instructions per cycle. The fundamental technology that makes the greatest leap forward from the traditional computers to today's high performance processors is the trick of allowing out of order execution and in order commit inside processor designs to maximize parallelism. We believe the same is true for parallel storage systems. However, to the best of our knowledge, there has been no attempt in doing this in the research community neither in the storage industry.

Our objective here is to propose a new algorithm in a storage system to process packets as soon as they arrive at the storage through the parallel connections irrespective of their order of initiation. Data dependencies among the operations are resolved without delays. The results of these parallel executions will commit in the order of their initiations. Our prior research has shown that 30% of packets arrive at an iSCSI storage target out of order over 3 parallel network connections. We believe such a high percentage of out-of-order arrivals present us with an opportunity to improve networked storage performance using our parallel packet processing algorithm.

2. System Architecture and the Parallel Algorithm

We consider an IT infrastructure where a number of application servers share a target storage. Connections between an application server and the target storage are plural both logically and physically. The storage should therefore be able to handle multiple connections from a server and multiple connections from multiple servers.

Our iSCSI target storage consists of a packet processing logic that manages the parallel network ports, an execution engine that executes our parallel algorithm, a command queue that buffers received storage commands, a reservation station that records commands being executed; a commit cache that stores committed data, and a storage system. The basic idea of our algorithm is to start execution of a storage command as soon as it arrives at the storage. Such an execution is recorded in the reservation station associated with dependencies of the command on other commands in the reservation station as well as commands that have not arrived at the storage yet. The actual commits of such executions will be done in the order of commands sequence. To describe how the algorithm works, let us first consider the following example command sequence.

EXAMPLE: *Suppose that the correct command sequence as generated by an initiator is*

1. *write block A:* $\boldsymbol{W_A^1}$
2. *read block B:* $\boldsymbol{R_B^2}$
3. *read block C:* $\boldsymbol{R_C^3}$
4. *write block D:* $\boldsymbol{W_D^4}$
5. *read block A* **:** $\boldsymbol{R_A^5}$
6. *write block B* $\boldsymbol{W_B^6}$
7. *read block A* $\boldsymbol{R_A^7}$

The sequence of these commands is shown blow. Note that network storage protocols such as iSCSI define command sequence numbers for all storage commands issued from an initiator to target storage. The superscript of each command indicates this command sequence number.

$\boldsymbol{R_A^7}$	$\boldsymbol{W_B^6}$	$\boldsymbol{R_A^5}$	$\boldsymbol{W_D^4}$	$\boldsymbol{R_C^3}$	$\boldsymbol{R_B^2}$	$\boldsymbol{W_A^1}$

During the network transmission of these commands, the arrival sequence to the storage has been changed as follows:

1. *read block B:* $\boldsymbol{R_B^2}$
2. *read block C:* $\boldsymbol{R_C^3}$
3. *write block D:* $\boldsymbol{W_D^4}$
4. *read block A* **:** $\boldsymbol{R_A^5}$
5. *write block B* $\boldsymbol{W_B^6}$
6. *write block A:* $\boldsymbol{W_A^1}$
7. *read block A* $\boldsymbol{R_A^7}$

The arrived storage commands will be buffered in a command queue as shown below after all commands have arrived at the storage.

$\boldsymbol{R_A^7}$	$\boldsymbol{W_A^1}$	$\boldsymbol{W_B^6}$	$\boldsymbol{R_A^5}$	$\boldsymbol{W_D^4}$	$\boldsymbol{R_C^3}$	$\boldsymbol{R_B^2}$

We would like to start execution as soon as $\mathbf{R_B^2}$ arrives even though its sequence number is not what is expected by the storage that is expecting for $\mathbf{W_A^1}$. By doing this, we are effectively allowing parallel and out of order execution of these commands in the command queue. However, there are data dependencies among these commands. For example, there is a read after write (RAW) dependency between $\mathbf{R_A^5}$ and $\mathbf{W_A^1}$. There is a write after read (WAR) data dependency between $\mathbf{W_B^6}$ and $\mathbf{R_B^2}$. And there are also possibly write after write data dependencies not shown here. To guarantee data correctness and allow parallel out of order execution, we start executing these arrived commands by putting them in a data structure called Reservation Station as shown in Table 1. As shown in this reservation station, command #5 has a RAW data dependency on command #1 and command #6 has a WAR dependency with command #2 buffered in reservation station entry 0, RS[0], and so forth. Note that, some of the data dependencies shown in the table may not be available while commands are arriving at the storage. However, the command dependencies are readily available whenever a command arrives after comparing with the expected command sequence number. In this case, all the 5 commands in the top 5 rows of the table arrived earlier than command #1 and therefore such dependencies are marked in the CMD dependencies column of the reservation station.

When a command is in the reservation station, it is eligible for execution. The results of such execution will be buffered in a data cache called ***commit cache***. The commit cache has a separate read cache and a write cache. Each entry in the cache has an additional field indicating the condition for commit. For example, the image of the commit cache after commands #2 through #6 have been executed is shown in Table 2. When command #1 finally arrives at the storage and is placed in RS[5], we will be able to determine all data dependency conditions and fill out the last column of the reservation table. After the completion of successful execution of command #1, the command dependency column of the reservation station will be cleared and the algorithm will commit all the commands in the reservation station by transmitting data in the commit cache to the initiator for read operations and committing writes to the storage. Whether the committed data come from the read cache or write cache will depend on data dependency conditions listed in the last column of the reservation station. Table 3 shows the cache image after all commands in the sequence have arrived at the storage. Data are committed as follows:

1. block B in CRi will be sent for command: $\mathbf{R_B}^{\mathbf{2}}$
2. block C in CRj will be sent for command: $\mathbf{R_C}^{\mathbf{3}}$
3. block D in CWj will be written to storage for command: $\mathbf{W_D}^{\mathbf{4}}$
4. block A in CWk will be sent for command**:** $\mathbf{R_A}^{\mathbf{5}}$
5. block B in CWi will be written to storage for command $\mathbf{W_B}^{\mathbf{6}}$
6. block A in CWk will be written to storage for command: $\mathbf{W_A}^{\mathbf{1}}$
7. block A in CWk will be sent for command: $\mathbf{R_A}^{\mathbf{7}}$

As shown in this example, data dependencies such as WAR and WAW are eliminated by renaming of the destination addresses (separate cache entries for them). True data dependencies such as RAW are short circuited because they are resolved and data are forwarded between the two caches. This example shows clearly the benefit of parallel out of order execution as compared to sequential in-order execution. By the time when command #1 arrives at the storage, the data for read and write operations of the 5 commands that have arrived ahead of command #1 have been in the commit cache or on the way to the commit cache. Substantial time savings can be obtained as result of this because accessing storage such as disk and disk arrays is usually time consuming. Another way to look at this is that such out-of-order execution and in order commit provides the best and optimal pre-fetching for storage caches. The general algorithm is shown in Figure 1.

Table 1. Reservation Station

RS Entry #	CMD Seq #	Storage CMD	LBA	Data Size	CMD dependencies	Data Pointer To Cache	Data Dependencies
0	2	R	B	4K	1	CRi	
1	3	R	C	4K	1	CRj	
2	4	W	D	4k	1	CWj	
3	5	R	A	4k	1	CRk	RAW, RS[5]
4	6	W	B	4K	1	CWi	WAR, RS[0]
5	1	W	A	4K		CWk	RAW, RS[3]
6	7	R	A	4K	1		RAW, RS[5]

Table 2. Commit Cache image 1

Read Cache

indx	Data	Dependency
CRi	B	1
CRj	C	1
CRk	A	1

indx	Data	Dependency
CWi	B	1
CWj	D	1

Write Cache

Table 3. Commit Cache image 2.

Read Cache

CRi	B	1
CRj	C	1
CRk	A	1

CWi	B	1
CWj	D	1
CWk	A	

Write Cache

Figure 1. The general algorithm for parallel packet processing.

```
A storage command arrives at the storage called Current_CMD;
/**Reserved Execution stage**/
Compare Sequence_No(Current_CMD) with Sequence_No_Expected
If Not Match Then
{
Find an entry in RS, called RS[free];
RS[free].S# ← Sequence_No[Current_CMD];
RS[free].CMD ← CMD[Current_CMD);
RS[free].LBA ← LBA[Current_CMD);
RS[free].size ← Size[Current_CMD);
RS[free].CMD_dependency ← Sequence_No_Expected, Sequence_No_Expected+1, …
                                                  Sequence_Number[Current_CMD)-1
Examine all entries in RS to find any data dependency;
If data dependency is found with RS[i] then
          {RS[free].Data_Dep ← ( the data dependency, RS[i]);
          RS[i].Data_Dep ← (reverse of the data dependency, RS[free])};
If CMD[Current_CM]==Read
                  then {Allocate an entry in read cache called CR[i];
                           RS[free].DataPtr ← CR[i];
                           CR[i].Dependency ← RS[free].CMD_dependency;
                           CR[i].Data ← read data at LBA[Current_CMD] from storage}
                  Else {Allocate an entry in write cache called CW[i];
                           RS[free].DataPtr ← CW[i];
                           CW[i].Dependency ← RS[free].CMD_dependency
                           CW[i] ← Data[Current_CMD]};
}
Else /** The command sequence number matches the expected sequence number**/
{For all i, Look up RS[i].CMD_dependency to find a match for Sequence_No[Current_CMD]
 If there is no match then issue Current_CMD for normal execution
  Else
   {Find an entry in RS, called RS[free];
    RS[free].S# ← Sequence_No[Current_CMD];
    RS[free].CMD ← CMD[Current_CMD);
    RS[free].LBA ← LBA[Current_CMD);
      RS[free].size ← Size[Current_CMD);


    Examine all entries in RS to find any data dependency;
    If data dependency is found with RS[i] then
          {RS[free].Data_Dep ← ( the data dependency, RS[i]);
          RS[i].Data_Dep ← (reverse of the data dependency, RS[free])};
    If CMD[Current_CM]==Read
                  then {Lookup RS and Commit cache if there is a hit;
                           If hit in read cache, called CR[i]
                           Else { Allocate an entry in read cache called CR[i];
                             CR[i].Data ← read data at LBA[Current_CMD] from storage or CW}
                           RS[free].DataPtr ← CR[i];
                           Clear Sequence_No[Current_CMD] in all RS[i], CR[i], CW[i]}
                  Else {Allocate an entry in write cache called CW[i];
                           RS[free].DataPtr ← CW[i];
                           CW[i] ← Data[Current_CMD];
                           Clear Sequence_No[Current_CMD] in all RS[i], CR[i] CW[i]}
  };
/**Commit Stage**/
Loop
For i do
  {
         If RS[i].CMD_dependency = null then
         {
                  If  RS[i].Data_Dep = null then
                    Commit data pointed by RS[i].Data_Ptr
                  Else
                    Commit data based on dependency and CMD sequence;
         }
}
```

3. Performance Analysis

In order to understand the performance potential of the parallel algorithm, we present here a simple and approximate analysis of the possible performance gains resulting from this new algorithm. We intend to compare the possible performance gains of our algorithm with a baseline storage system that orders arrived commands before execution. To make our analysis tractable, we make the following assumptions:

- We call a packet an *out-of-order packet* if it is expected by the storage at a point of time and has not arrived at the storage until this time point, while other packets with greater (latter) command sequence numbers have arrived. Let Ω be the number of *unexpected* packets with greater command sequence numbers arrived before the out-of-order packet arrives.
- Let δ be the time delay from the time the first unexpected packet arrives until the out-of-order packet arrives at the storage. Clearly, with the baseline storage system, this unexpected packet that has arrived will be delayed for δ seconds before it can be processed.
- In a baseline storage system, each of the Ω *unexpected* packets will be delayed for certain amount of time. We assume that the delays experienced by these Ω packets are uniformly distributed between δ and δ/Ω, implying that arrival times of these packets are evenly scattered across δ. That is, the first of the Ω packets will be delayed for δ seconds, the second for $\delta(\Omega-1)/\Omega$, the third one for $\delta(\Omega-2)/\Omega$ …. and the last one of the Ω packets will be delayed for δ/Ω. The purpose of this simplifying assumption is to evenly distribute the delays among the unexpected packets to adjust the average service time of each packet. Clearly, this assumption is not realistic because the Ω packets may experience delays that are impossible to estimate depending on the exact arrival times. Such an unrealistic assumption can greatly simplify our analysis.
- It is assumed that the aggregated packet arrival process to the storage follows the Poisson distribution with arrival rate λ. The storage service time is assumed to be a random variable with mean $\boldsymbol{S}$. The adjusted service times taking into account of possible delays caused by waiting for command ordering are assumed to be exponentially distributed with mean $\boldsymbol{S_B}$, $\boldsymbol{S_P}$, for baseline storage and our parallel storage, respectively. This assumption is also an approximation.

With these assumptions, the average additional delay that each of the Ω packets experiences in a baseline storage system is given by

$$\Delta\delta = \delta(\Omega+1) / (2\Omega).$$

The adjusted storage service time for the baseline storage system is given by

$$\boldsymbol{S_B} = \boldsymbol{S} + \delta(\Omega+1) / (2\Omega).$$

With the new parallel algorithm, a storage request can start as soon as it arrives. The request would not experience any additional delay if the out-of-order packet arrives any time before the storage operation complete, i.e. $\boldsymbol{S} >= \delta$. The adjusted storage service time in this case is given by

$$\boldsymbol{S_P} = \boldsymbol{S}, \qquad \textbf{if } \boldsymbol{S} >= \delta.$$

If the out-of-order packet has not arrived when the storage operation of the request is done, additional delay is needed to wait for the commit time. That is, when $\boldsymbol{S} < \delta$, the command in the reservation station may finish its execution before the out-of-order packet finally arrives. In this situation, the commands would need to wait for the out-of-order packet to arrive before commit. This post-execution waiting time is on average $(\delta - \boldsymbol{S})/2$, and the probability that a packet in the reservation station needs to wait for it is $(\delta - \boldsymbol{S})/\delta$. Therefore, the adjusted service time in this case is

$$\boldsymbol{S_P} = \boldsymbol{S} + (\delta - \boldsymbol{S})^2/(2\delta), \qquad \textbf{if } \boldsymbol{S} < \delta.$$

Using M/M/1 queue system to approximate the average I/O response time, we have

$$\boldsymbol{R_B} = \boldsymbol{S_B} / (1 - \lambda\, \boldsymbol{S_B}),$$
$$\boldsymbol{R_P} = \boldsymbol{S_P} / (1 - \lambda\, \boldsymbol{S_P})$$

for the I/O response times of baseline storage ($\boldsymbol{R_B}$) and the parallel algorithm ($\boldsymbol{R_P}$), respectively.

Figure 2 shows the numerical results calculated using the above approximate formulae. In this figure, we vary the delay time of the out-of-order packet from 10 microseconds to 1 millisecond and fix the storage service time to 10 milliseconds. We choose these numbers based on our prior experiments with LAN switches and disk storage access times. For example, our measured packet delay over a 1 Gbps switch is about 100 microseconds and disk operations took about 10 milliseconds (6 milliseconds seek time and 4 millisecond rotation latency). Modern hardware may have much faster speed than what we measured before. But what is more important here is the relative value rather than absolute numbers. We plotted the I/O response times as a function of the packet delay time assuming that the packet arrival rate is 45 per second and there are 10 packets arrived before the out-of-order packet. As shown in the figure, performance improvement of our algorithm (RespT_P3 for parallel packet processing) is noticeable for this workload. The performance difference between the two storage systems increases rapidly as the packet delay increases.

Figure 3 shows the performance results by varying the packet arrival rate while fixing the packet delay at 5 milliseconds. Again, there is a significant performance improvement of our algorithm over the baseline storage systems. While the I/O response times of both storage systems increases as the arrival rate increases, the P3 algorithm shows consistently lower I/O response times as compared to the baseline storage systems.

4. Conclusions

We have described a new parallel packet-processing algorithm for networked data storage systems. The main objective of this algorithm is to maximize parallelism among multiple and inexpensive network ports in a storage target to improve storage performance. Multiple network port is clearly a cost-effective approach to achieving high storage throughputs because of the wide availability of high speed and low cost network ports. Our algorithm is shown to be advantageous over existing storage systems. An approximate queuing model is used to estimate the potential gains of the algorithm with promising results.

Acknowledgements:

This research is supported by Gemini Storage Corporation and partly by National Science Foundation under grants CCR-0073377 and CCR0312613. Any opinion, findings and conclusions are those of authors and do not necessarily reflect the views of NSF.

[1] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. iSCSI draft standard. http://www.ietf.org/ internetdrafts/draft-ietf-ips-iscsi-20.txt.

[2] UNH. iSCSI reference implementation. http://www.iol.unh.edu/consortiums/iscsi/.

[3] C. Boulton. iSCSI becomes official storage standard. http://www.internetnews.com/storage/article.php/1583331.

[4] X. He, Q. Yang, and M. Zhang, "Introducing SCSI-To-IP Cache for Storage Area Networks," in *Proceedings of the 2002 International Conference on Parallel Processing*, Vancouver, Canada, Aug. 2002, pp. 203-210.

[5] W. T. Ng, B. Hillyer, E. Shriver, E. Gabber, and B. Ozden, "Obtaining high performance for storage outsourcing," in *Proceedings of the Conference on File and Storage Technologies (FAST)*, Monterey, CA, Jan. 2002, pp. 145-158.

[6] S. Aiken, D. Grunwald, A. R. Pleszkun, and J. Willeke, "A performance analysis of the iSCSI protocol," in *IEEE Symposium on Mass Storage Systems*, San Diego, CA, Apr. 2003, pp. 123-134.

[7] Y. Lu and D. H. C. Du, "Performance study of iSCSI-based storage subsystems," *IEEE Communication Magazine*, vol. 41, no. 8, Aug. 2003.

[8] P. Radkov et al. "A performance comparison of NFS and iSCSI for IP-Networked storage," *Proc. of the 3rd USENIX Conf. On File and Storage Technologies*, CA, 2004.

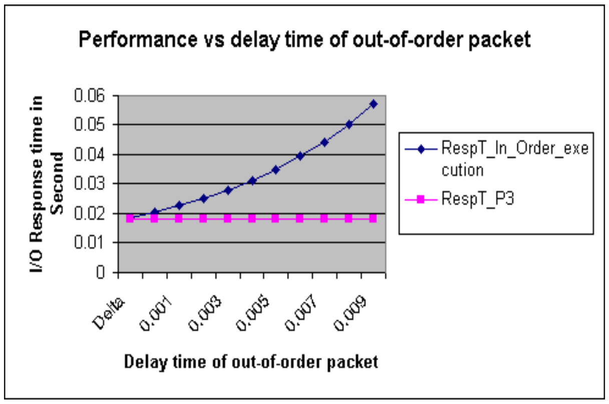

Figure 2. Performance comparison between baseline storage and P3 storage in terms of I/O response time as a function of packet delay (δ). (packet arrival rate, $\lambda = 45$ pkts/s, storage service time, $\boldsymbol{S} = 0.01$s, No. of packets prior to the our-of-order packet, $\Omega = 10$).

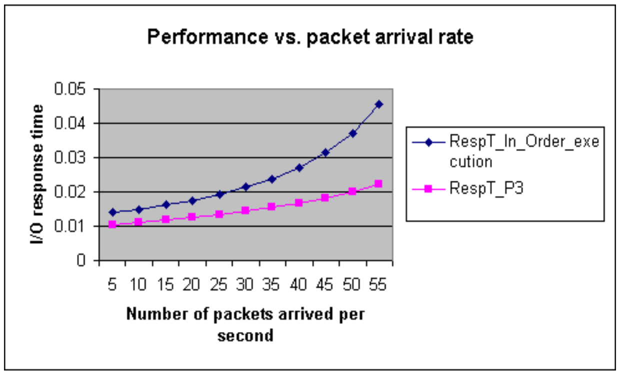

Figure 3. Performance comparison between baseline storage and P3 storage in terms of I/O response time as a function of packet arrival rate (λ). (packet delay, δ: 0.005s, storage service time, $\boldsymbol{S}$: 0.01s, No. of packets prior to the out-of-order packet, $\Omega = 10$).