A Caching Strategy to Improve iSCSI Performance

Xubin He,
Electrical and Computer Engineering
Tennessee Technological University
Cookeville, TN 38505, USA
hexb@tntech.edu

Qing Yang, and Ming Zhang
Electrical and Computer Engineering
University of Rhode Island
Kingston, RI 02881 USA
{qyang, mingz}@ele.uri.edu

Abstract

iSCSI is one of the most recent standards that allows SCSI protocols to be carried out over IP networks. However, to encapsulate SCSI protocol over IP requires significant amount of overhead traffic for SCSI commands transfers and handshaking over the Internet. In this paper, we propose a caching scheme, called iCache, to improve the iSCSI performance. iCache uses a log disk along with a piece of non-volatile RAM to cache the iSCSI traffic. Through efficient caching algorithm, iCache can significantly improve performance over current iSCSI systems. Numerical results using popular benchmark program and real world trace have shown dramatic performance gain.

1 Introduction

Data storage plays an essential role in today's fast-growing data-intensive network services. New standards and products emerge very rapidly for networked data storages. Given the mature Internet infrastructure, overwhelming preference among IT community recently is using IP for storage networking [3, 4, 6, 10, 12] because of economy and convenience. iSCSI is the most recent emerging technology with the goal of implementing the SAN technology over the better-understood and mature network infrastructure: the Internet (TCP/IP).

Implementing storage over IP brings economy and convenience whereas it also raises issues such as performance and reliability. Currently, there are basically two existing approaches: one encapsulates SCSI protocol in TCP/IP at host bus adapter (HBA) level [13] and the other carries out SCSI and IP protocol conversion at a specialized switch [2]. Both approach have severe performance limitations. To encapsulate SCSI protocol over IP requires significant amount of overhead traffic for SCSI commands transfers and handshaking over the Internet. Converting protocols at a switch places special burden to an already-overloaded switch and creates another specialized networking equipment in a SAN. On a typical iSCSI implementation, we have measured around 58% of TCP/IP packets being less than 127 bytes long, implying an overwhelming quantity of small size packets to transfer SCSI commands and status (most of them are only one byte). Majority of such small packet traffic over the net lowers the network bandwidth utilization.

This paper introduces a cache scheme called *iCache* (iSCSI cache) aimed to speedup current iSCSI performance. Based on our previous DCD [7] technology, we use a small amount of non-volatile RAM (NVRAM) and a log disk to form a two-level hierarchical cache for iSCSI requests as shown in Figure 1. *iCache* converts small requests into large ones (logs) before writing data into remote storage though the network, and utilizes the Log-structured file system to quickly write data into log disks for caching data. *iCache* can also improve the reliability of the system because both meta data and user data are cached in a log disk that is much more reliable than RAM. Moreover, *iCache* is completely transparent to the OS. It does not require any changes to the OS nor does it need accesses to the kernel source code. Furthermore, *iCache* also localizes SCSI commands and handshaking operations to reduce unnecessary traffic over the Internet. In this way, it acts as a storage filter to discards a fraction of the data that would otherwise move across the Internet, reducing the bottleneck imposed by limited Internet bandwidth and increasing storage data rate.

To quantitatively evaluate the performance potential of *iCache* in real world network environment, we have implemented the *iCache* prototype under the Linux OS as a loadable kernel driver. We have used PostMark [9] benchmark and EMC's trace to measure system performance. PostMark results show that *iCache* provides 53% to 78% performance improvement over iSCSI implementation in terms of average system throughput. An order of magnitude performance gain is observed for 90% of I/O requests under the EMC's trace in terms of response time.

The paper is organized as follows. Next section presents the design and implementation of *iCache*. Section 3 presents our initial experiments and performance evaluations. We discuss the related research work in Section 4 and conclude our paper in Section 5.

2 Design and Implementation

This section describes the data structures and algorithms used to implement our *iCache* driver. The cache organization in *iCache* consists of two level hierarchies: a RAM cache and a log disk. Frequently accessed data reside in the RAM that is organized as LRU cache as shown in Figure 2. Whenever the newly written data in the RAM are sufficiently large or whenever the log disk is free, data are written into the log disk. There are also less frequently accessed data kept in the log disk. Data in the log disk are organized in the format of *segments* similar to that in a Log-structured File System [14]. A segment contains a number of slots each of which can hold one data block. Data blocks in segments are addressed by their *Segment IDs* and *Slot IDs*.

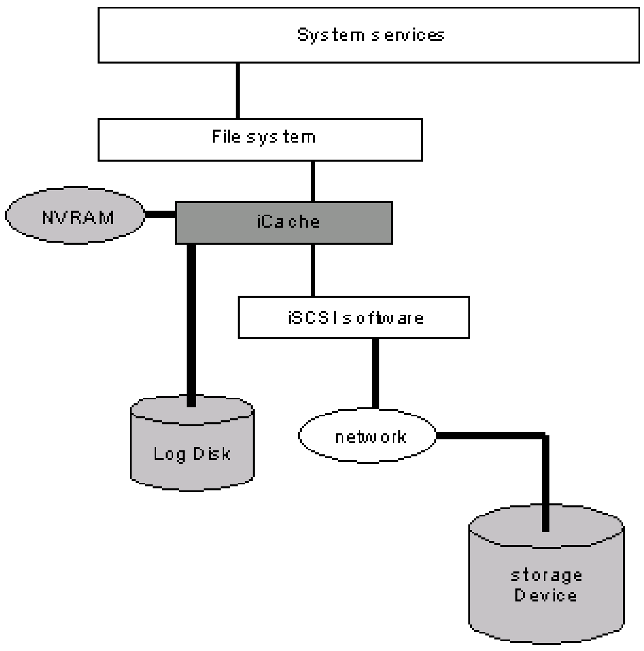


Figure 1. iCache Architecture

2.1 RAM Cache Structure

The RAM used in *iCache* is a set of continuous memory locations reserved and allocated from the system non-paged pool upon loading of the *iCache* driver. It consists of

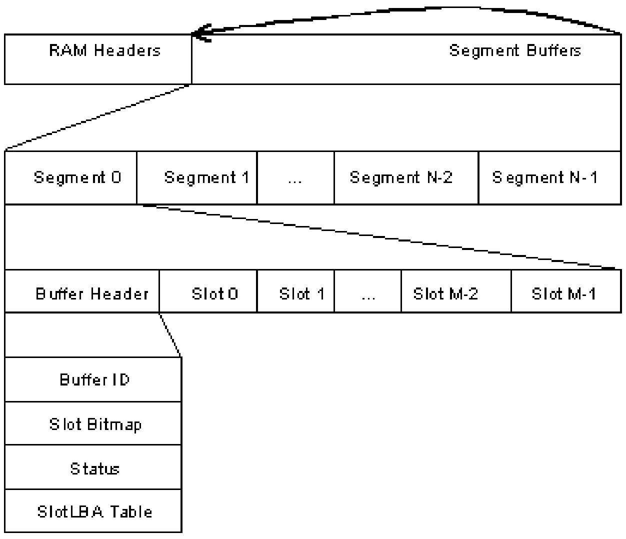


Figure 2. RAM buffer organization

two main parts, RAM Headers and Segment buffers. RAM headers are in-memory copies of the segment header of the log disk which will be discussed shortly. Each segment buffer consists of a buffer header and several slots. Each segment has its own unique Segment Buffer ID. Our driver makes the segment buffers appear infinite by reusing it circularly while guaranteeing that it does not overwrite useful information. A slot is the basic caching unit (Figure 2). In our implementation, a segment buffer is 64KB and a slot can store 1KB data. Therefore, our 1 MB RAM buffer contains 16 segment buffers and each segment buffer consists of 64 slots.

Data blocks stored in the RAM cache are addressed by their *Logical Block Addresses (LBAs)* and organized by a hash table. The Hash Table contains location information for each of the valid data blocks in the cache and uses LBAs of incoming requests as search keys. The buffer header describes the contents and the status of the corresponding segment buffer. It contains:

1. A *Buffer ID* which is used to identify a segment buffer.
2. A *Slot bitmap* which is used to describe the status (free or used) of each slot within the segment buffer.
3. A *Status* field to describe the status of the segment buffer. A segment can be in one of 3 states: *free* which means no data on this segment; *dirty* which means that the segment contains changed data that has not been written to the log disk yet; *valid* which means that the segment contains data which has already been written to the log disk.

4. A *SlotLBA table*. Each slot in the segment has a corresponding entry in the table. Each entry is an integer representing the target Logical Block Address (LBA) of the data in the slot.

Initially all Segment Buffers are free. When a write request arrives, the driver picks a free Segment Buffer to become the Current Segment Buffer. Meanwhile the driver also obtains a free disk segment from the Free-Segment-List to become the current Disk Segment. The driver then p̈airsẗhe RAM segment buffer and the disk segment together by writing the Segment Buffer ID to the corresponding field in the RAM header. From this point until the current Segment Buffer is written into the current Disk Segment on the cache-disk, incoming write data are written into the slots of this Segment Buffer.

2.2 Log-disk organization

The entire log disk is divided into a number of fixed-size segments (clusters). The *iCache* driver always writes an entire segment to the log disk. The organization of the log disk is similar to that of the RAM image. The log disk consists of segments, each of which also has a header (*segment header*) and several slots. The segment and slot sizes are 64KB and 1KB, respectively, so the log disk contains 320 segments and each segment consists of 64 slots. The segment header contains the following information:

1. A *Segment ID* which identifies the segment.

2. A *Buffer ID*, which associates the segment with the RAM segment buffer.

3. A *Slot bitmap* which is used to describe the status (free or used) of each slot within the segment.

4. A *Time Stamp* which records the time when the segment is written into the log disk. During a crash recovery period, the time stamps help the *iCache* driver to search for segments.

5. A *SlotLBA* table same as the SlotLBA table in the Buffer header of RAM cache.

2.3 Basic operations

2.3.1 Write

After receiving a write request, the *iCache* first searches the Hash Table by the LBA address. If an entry is found, the entry is overwritten by the incoming write. Otherwise, a free slot entry is allocated from the Free List, the data are copied into the corresponding slot, and its address is recorded in the Hash table. The relevant data structures are then updated. When enough data slots (64 in our preliminary implementation) are accumulated or when the log disk is idle, the data slots are written into log disk sequentially in one large write. After the log write completes successfully, *iCache* signals the host that the request is complete.

2.3.2 Read

After receiving a read request, the *iCache* searches the Hash Table by the LBA to determine the location of the data. Data requested may be in one of three different places: the RAM buffer, the log disk(s), and iSCSI storage device. If the data is found in the RAM buffer, the data are copied from the RAM buffer to the requesting buffer. The *iCache* then signals the host that the request is complete. If the data is found in the log disk, the data are read from the log disk. Otherwise, the *iCache* encapsulates the request including LBA, and destination IP into an IP packet and forwards it to the remote iSCSI storage device.

2.3.3 Destages

The operation of moving data from a higher-level storage device to a lower level storage device is defined as *destage* operation. There are two levels of destage operations in *iCache*: destaging data from the RAM buffer to the log disk (*Level 1 destage*) and destaging data from log disk to a iSCSI storage device (*Level 2 destage*). We implement a separate kernel thread, *LogDestage*, to perform the destaging tasks. The *LogDestage* thread is registered during system initialization and monitors the *iCache* states. The thread keeps sleep at most of the time, and is activated when one of the following events occurs: 1) the number of slots in the RAM buffer exceeds a threshold value, 2) the log disk is idle, 3) the *iCache* detects an idle period, 4) the *iCache* RAM buffer and/or the log disk becomes full. *Level 1 destage* has higher priority than *Level 2 destage*. Once the *Level 1 destage* starts, it continues until a log of data in the RAM buffer is written to the log disk. *Level 2 destage* may be interrupted if a new request comes in or until the log disk becomes empty. If the destage process is interrupted, the destage thread would be suspended until the *iCache* detects another idle period.

As for *Level 1 destage*, the data in the RAM buffer are written to the log disk sequentially in large size (64KB). The log disk header and the corresponding in-memory slot entries are updated. All data are written to the log disk in ”append” mode, which ensures that every time the data are written to consecutive log disk blocks. For *Level 2 destage*, we use a l̈ast-write-first-destageälgorithm according to the LRU list. Each time 64KB data are read from the consecutive blocks of the log disk and written to the remote iSCSI storage device.

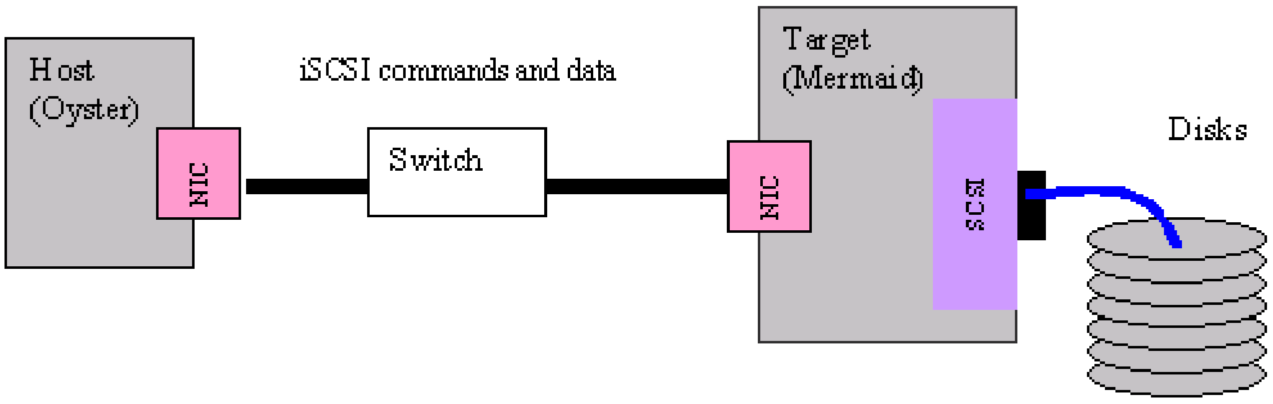


Figure 3. iSCSI configuration. The host *Oyster* **establishes connection to target, and the target** *Mermaid* **responds and connects. Then the** *Mermaid* **exports hard drive and** *Oyster* **sees the disks as local.**

Table 1. Machines configurations

	Processor	RAM	IDE disk	SCSI disk
Oyster	PII-450	128MB	Maxtor 91366u4 x2	N/A
Mermaid	K6-500	128MB	Maxtor 91020D6	IBM DNES-318350

3 Performance Evaluations

3.1 Experimental Setup

For the purpose of performance evaluation, we have implemented *iCache* prototype and deployed a software iSCSI. For a fair performance comparison, both iSCSI and *iCache* have exactly the same CPU and RAM size. This RAM includes RAM buffer used in *iCache*. All I/O operations in both iSCSI and *iCache* are forced to be remote operations to target disks through a switch.

Our experimental settings for the purpose of evaluating the performance of iSCSI and *iCache* are shown in Figure 3. Two PCs are involved in our experiments, namely *Oyster* and *Mermaid*. The *Oyster* serves as the host and the *Mermaid* as the iSCSI target. We load our *iCache* driver on the machine Oyster for *iCache* testing. These two machines are interconnected through a 100Mbps switch to form an isolated LAN. Each machine is running Linux kernel 2.4.2 with a 3c905 TX 100Mbps network interface card (NIC) and an Adaptec 39160 high performance SCSI adaptor. The configurations of these machines are described in Table 1 and the characteristics of individual disks are summarized in Table 2.

For iSCSI implementation, we compiled and run the Linux iSCSI developed by Intel Corporation [1]. The iSCSI is compiled under Linux kernel 2.4.2 and configured as shown in Figure 3. For *iCache* testing, we use 4 MB of the system RAM to simulate *iCache* NVRAM buffer, and the log disk is a standalone hard drive. A hash table and a LRU list are maintained in the NVRAM.

3.2 Benchmark program and workload

It is important to use realistic workloads to drive our *iCache* for a fair performance evaluation and comparison. For this reason, we chose to use real world trace and benchmark program.

The benchmark we used to measure system throughput is PostMark [9] which is a popular file system benchmark developed by Network Appliance. It measures performance in terms of transaction rates in an ephemeral small-file environment by creating a large pool of continually changing files. P̈ostMark was created to simulate heavy small-file system loads with a minimal amount of software and configuration effort and to provide complete reproducibility [9].P̈ostMark generates an initial pool of random text files ranging in size from a configurable low bound to a configurable high bound. This file pool is of configurable size and can be located on any accessible file system. Once the pool has been created, a specified number of transactions occur. Each transaction consists of a pair of smaller transactions, i.e. *Create file* or *Delete file* and *Read file* or *Append file*. Each transaction type and its affected files are chosen randomly. The read and write block size can be tuned. On completion of each run, a report is generated showing some

Table 2. Disk parameters

Disk Model	Interface	Capacity	Data buffer	RPM	Latency (ms)	Transfer rate (MB/s)	Seek time (ms)	Manufacturer
DNES-318350	Ultra SCSI	18.2G	2MB	7200	4.17	12.7-20.2	7.0	IBM
91366U4	ATA-5	13.6G	2MB	7200	4.18	Up to 33.7	9.0	Maxtor
91020D6	ATA-4	10.2G	256KB	5400	5.56	18.6	9.0	Maxtor

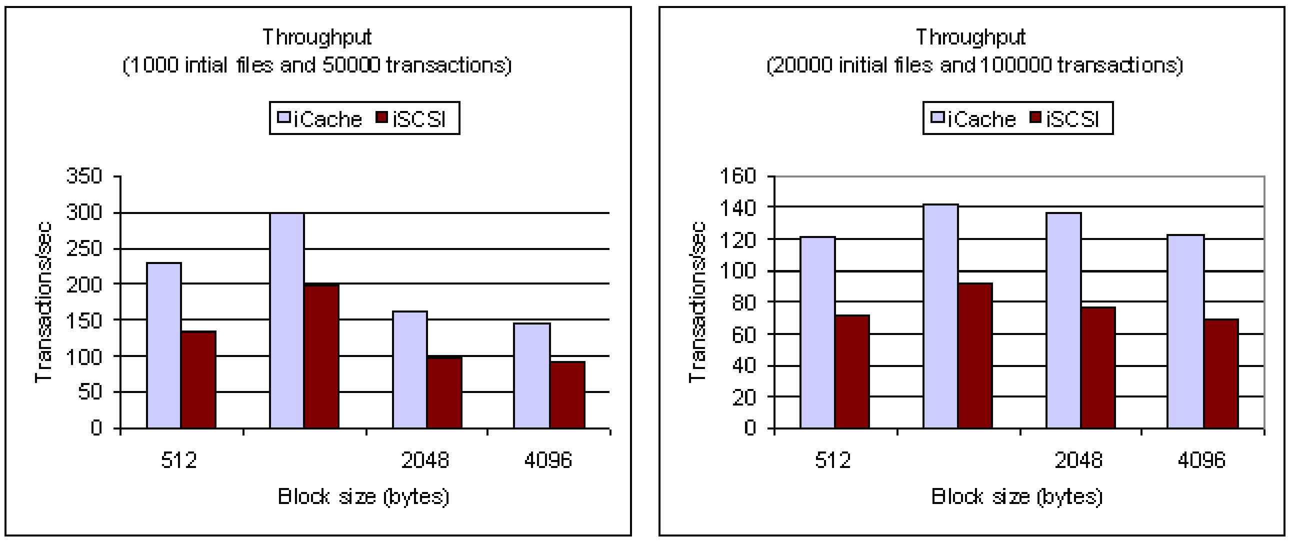


Figure 4. Postmark measurements

metrics such as elapsed time, transaction rate, total number of files created and so on.

In addition to PostMark, we also used a real-world trace obtained from EMC Corporation. The trace, referred to as *EMC-tel* trace hereafter, was collected by an EMC Symmetrix disk array system installed at a telecommunication consumer site. The trace file contains 230370 requests, with a fixed request size of 4 blocks. The trace is write-dominated with a write ratio of 89%.

3.3 Numerical Results

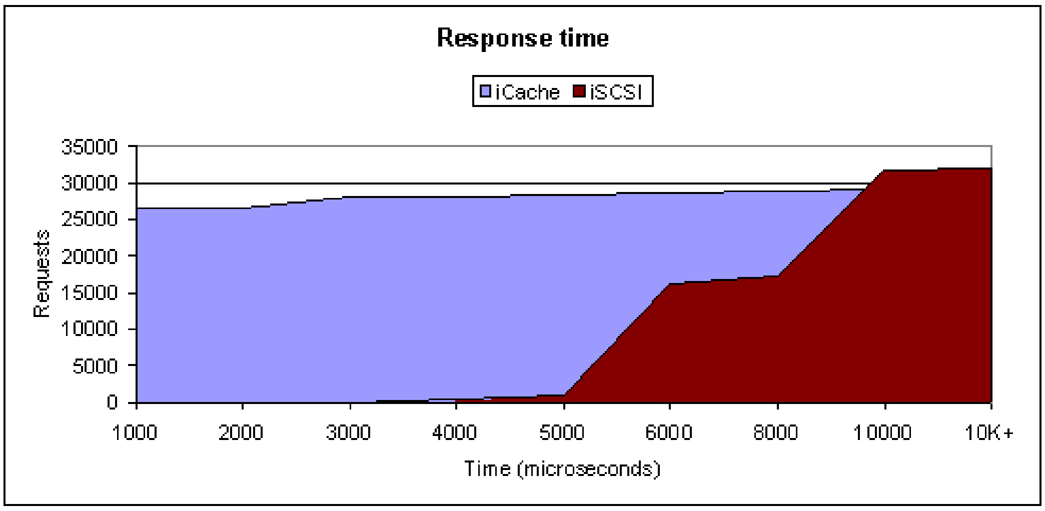
3.3.1 Throughput

Our first experiment is to use PostMark to measure the I/O throughput in terms of transactions per second. In our tests, PostMark was configured in two different ways as in [9]. First, a small pool of 1,000 initial files and 50,000 transactions; and second a large pool of 20,000 initial files and 100,000 transactions. The total sizes of accessed data are 330MB (161.35MB read and 168.38MB write) and 740MB (303.46 MB read and 436.18MB write) respectively. They are much larger than the system RAM (128MB). The block sizes change from 512 bytes to 4KB. The IO operations are set to synchronous mode. We left all other PostMark parameters at their default settings.

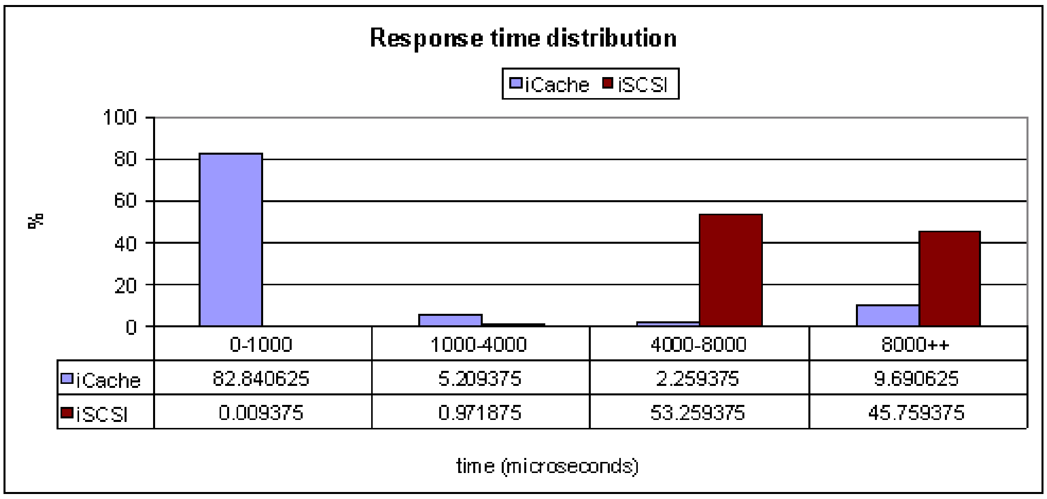
In Figure 4, we plotted two separate bar graphs corresponding to the small file pool case and the large one, respectively. Each pair of bars represents the system throughputs of *iCache* (light bars) and iSCSI (dark bars) for a specific data block size. It is clear from this figure that *iCache* shows obvious better system throughput than the iSCSI. The performance improvement of *iCache* over iSCSI is consistent across different block sizes and for both small pool and large pool cases. The performance gain of *iCache* over iSCSI ranges from 53% to 78%.

3.3.2 Response times

Our next experiment is to measure and compare the response times of *iCache* and iSCSI under EMC trace. In Figure 5a, we plotted histogram of request numbers against response times, i.e. X-axis represents response time and Y-axis represents the number of storage requests finished within a particular response time. For example, a point (X, Y)=(1000, 25000) means that there are 25,000 requests fin-



(a)



	0-1000	1000-4000	4000-8000	8000++
iCache	82.840625	5.209375	2.259375	9.690625
iSCSI	0.009375	0.971875	53.259375	45.759375

(b)

Figure 5. Response time distributions

ished within 1000 microseconds. The lighter (blue) part of the figure is for *iCache* whereas the darker (red) part for iSCSI. To make it clearer, we also draw a bar graph representing percentage of requests finished within a given time as shown in Figure 5b. It is interesting to note in this figure that *iCache* does an excellent job in smoothing out the speed disparity between SCSI and IP. With *iCache*, over 80% of requests are finished within 1000 microseconds and most of them are finished within 500 microseconds. For iSCSI with no *iCache*, more than 99% of requests take over 4000 microseconds, where about 46% take over 8000 microseconds, and almost no request finishes within 1000 microseconds. These measured data are very significant and represent dramatic performance advantages of *iCache*.

While *iCache* improves the iSCSI performance by an order of magnitude for 90% of storage requests, the average speedup of *iCache* is only about 85%. The average response time of *iCache* is 3652 whereas the average response time of iSCSI is about 6757. Figure 6 shows average response times of groups of 1000 requests each, i.e. we average the response times of every 1000 requests as a group and show the average response times for all groups. In our experiments, we noticed that around 10% of requests take over 8000 microseconds for *iCache*. Some requests even take up to 120,000 microseconds. These few peak points drag down the average performance of *iCache*. These excessive large response times can be attributed to the *destaging* process. In our current experiment, we allow the *level 1 destaging* pro-

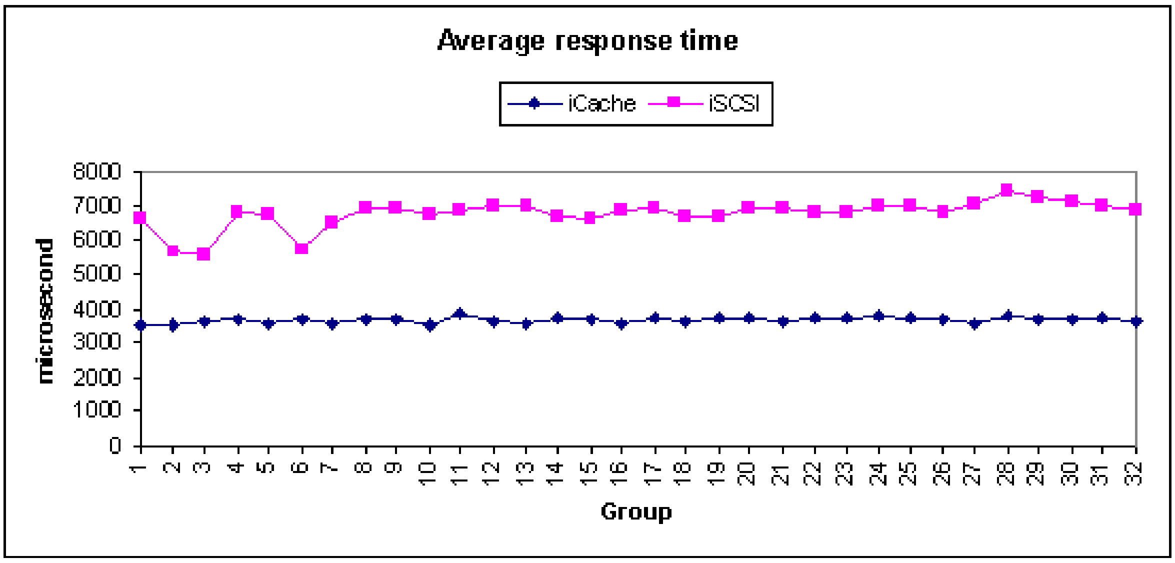


Figure 6. Average response time of iCache and iSCSI

cess to continue until the entire RAM buffer is empty before serving a new storage request. It takes a long time to move data in a full RAM to the log disk. We are still working on the optimization of the *destage* algorithm. We believe there is sufficient room to improve the *destaging* process to avoid the few peak response times of *iCache*.

4 Related Work

The idea of using a disk-based log to improve system performance or to improve the reliability of RAM has been used in both file system and database systems for a long time. For example, the Log-structured File System (LFS [14, 15]), Disk Caching Disk (DCD [7]), and other similar systems all use disk-based data/metadata logging to improve file system performance and speed-up crash recovery. Several RAID systems have implemented the LFS algorithm at the RAID controller level [8, 11, 16]. LFS collects writes in a RAM buffer to form large logs and writes large logs to data disks. Our previous research, STICS [5], introduces a SCSI-to-IP cache for storage area networks. iCache differs from STICS in two ways. Firstly, iCache is used to cache data on the iSCSI host end, while STICS aims at building storage area networks by coupling reliable and high-speed data caching with low-overhead conversion between SCSI and IP protocols. Secondly, iCache is used together with iSCSI, while STICS can be used alone. While many implementation techniques are borrowed from existing work, our contributions are as follows. *iCache* speedups the performance of iSCSI, which is a new concept to smooth the disparities between SCSI and IP protocols. *iCache* is implemented as a device driver layer, where we need neither modify the operating system nor the existing file systems.

5 Conclusions

In this paper, we have introduced a cache scheme called *iCache* to improve the iSCSI performance. Using a two-level hierarchical cache consisting of a small amount of NVRAM and a log disk, *iCache* smoothes out the storage data traffic between SCSI and IP. We have carried out a prototype implementation of *iCache* under the Linux operating system. We measured the performance of *iCache* as compared to a typical iSCSI implementation using a popular benchmark (PostMark) and a real world I/O workload (EMC's trace). PostMark results have shown that *iCache* outperforms iSCSI by 53%-78% in terms of average system throughput. Numerical results under EMC's trace show an order of magnitude performance gain for 90% of storage requests in terms of response time.

Acknowledgements

This research is sponsored in part by National Science Foundation under grants MIP-9714370 and CCR-0073377. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. The authors would like to thank the anonymous reviewers for their many helpful comments and suggestions.

References

[1] Intel Corp.,*"Intel iSCSI project"*, http://sourceforge.net/projects/intel-iscsi,2001.

[2] Nishan System white paper,*"Storage over IP (SoIP) Framework - The Next Generation SAN,"* http://www.nishansystems.com/techlib/techlib_papers .html.

[3] D. Fetters,*"RFP: Storage Area Networks,"* in Network Computing, 2000.

[4] G. Gibson and R. Meter,*"Network Attached Storage Architecture,"* Communications of the ACM, vol. 43, pp. 2000.

[5] X. He, Q. Yang, and M. Zhang, *"Introducing SCSI-To-IP Cache for Storage Area Networks"*, International conference on Parallel Processing (ICPP'2002), August 2002.

[6] R. Hernandez, C. Kion, and G. Cole,*"IP Storage Networking: IBM NAS and iSCSI Solutions,"'* Redbooks Publications (IBM), 2001.

[7] Y. Hu and Q. Yang, *"DCD-disk caching disk: A New Approach for Boosting I/O Performance,"* Proc of 23rd Annual Intl. Symposium on Computer Architecture (ISCA'96), 1996.

[8] Y. Hu, Q. Yang, and T. Nightingale, *"RAPID-Cache - A Reliable and Inexpensive Write Cache for Disk I/O Systems,"* Proc. of the 5th International Symposium on High Performance Computer Architecture (HPCA-5), Orlando, Florida, 1999.

[9] J. Katcher, *"PostMark: A New File System Benchmark,"* Network Appliance Technical Report TR3022, 1999.

[10] R. Khattar, et al., Introduction to Storage Area Network: Redbooks Publications (IBM), 1999.

[11] J. Menon, *"A Performance Comparison of RAID-5 and Log-Structured Arrays,"* Proc. Of 4th IEEE Int'l Symp. High Performance Distributed Computing, 1995.

[12] D. Nagle, et al., *"Network Support for Network-Attached Storage,"* Proc of Hot Interconnects'1999, 1999.

[13] J. Satran, et al., *"iSCSI draft standard,"* http://www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-09.txt,2001.

[14] M. Seltzer, et al., *"An Implementation of a Log-Structured File System for UNIX,"* Proc of Winter USENIX Proceedings, 1993.

[15] R. Wang, T. Anderson, and D. Patterson, *"Virtual Log Based File Systems for a Programmable Disk,"* Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), 1999.

[16] J. Wilkes, et al., *"The HP AutoRAID Hierarchical Storage System,"* Proc. Of the Fifteenth ACM Symposium on Operating System Principles, 1995.