Performance of optimized software implementation of the iSCSI protocol

Fujita Tomonori and Ogawara Masanori
NTT Network Innovation Laboratories
1-1 Hikarinooka Yokosuka-Shi Kanagawa, Japan
Email: tomof@acm.org, ogawara.masanori@lab.ntt.co.jp

Abstract— **The advent of IP-based storage networking has brought specialized network adapters that directly support TCP/IP and storage protocols on the market to get comparable performance that specialized high-performance storage networking architectures provide.**

This paper describes an efficient software implementation of the iSCSI protocol with a commodity networking infrastructure. Though several studies have compared the performances of specialized network adapters and commodity network adapters, our iSCSI implementation eliminates data copying overhead unlike straightforward iSCSI implementations used in previous studies. To achieve it, we modified a general-purpose operating system by using techniques studied for improving TCP performance in the literature and features that commodity Gigabit Ethernet adapters support. We also quantified their effects.

Our microbenchmarks show, compared with a straightforward iSCSI driver that does not use these techniques, the iSCSI driver with these optimizations reduces CPU utilization from 39.4% to 30.8% when writing with an I/O size of 64 KB. However, when reading, any performance gain is negated due to the high cost of operations on the virtual memory system.

I. Introduction

To cope with the rapidly growing volume of data, many companies have started to adopt new storage architectures, in which servers and storage devices are interconnected by specialized high-speed links such as Fibre Channel. The new storage architectures, called storage networking architectures, have superseded traditional storage architectures, in which servers and storage devices are directly connected by system buses. Storage networking architectures make it easier to expand, distribute and administer storage than with the traditional storage architectures.

With advances in Ethernet technology, more advanced storage networking technology, i.e., the iSCSI protocol [1] is on the horizon. It encapsulates a block-level storage protocol, that is, the SCSI protocol into the TCP/IP protocol, and it carries packets over IP networks. The iSCSI protocol has some advantages over other storage networking architectures: it is based on two well-known technologies, that is, the SCSI protocol and the TCP/IP protocol, which have been used for many years. Integrating storage networking with mainstream data communications is possible by using Ethernet networks. Furthermore, many engineers who are familiar with these technologies may provide economic and management advantages.

The authors' current affiliation is NTT Cyber Solutions Laboratories.

Though the iSCSI protocol is fully compatible with commodity networking infrastructures, specialized network adapters, such as TCP Offload Engine (TOE) and Host Bus Adapter (HBA), exist on the market. Vendors claim that they can obtain comparable performance with existing specialized storage networking architectures by offloading the overheads of IP-based storage networking. A TOE adapter offloads the TCP/IP protocol overhead on the host CPU by directly supporting the TCP/IP protocol functions. Furthermore, an HBA adapter offloads the TCP/IP protocol and data copying overheads by directly supporting the TCP/IP protocol and storage protocol functions. Several studies have compared the performance of specialized network adapters and commodity Gigabit Ethernet adapters.

This paper provides an analysis on how high a level of performance the iSCSI protocol can provide without having specialized hardware. We made changes to the Linux kernel to get the best performance with commodity networking infrastructures by using techniques studied for improving TCP performance in the literature. We also used three features that commodity Gigabit Ethernet adapters support, i.e., checksum offloading, jumbo frames, and TCP segmentation offloading (TSO). Unlike the straightforward iSCSI implementations used in previous studies, ours avoids copying between buffers in the virtual memory (VM) system and those in the network subsystem.

This paper focuses on the implementation of an initiator, i.e., a client of a SCSI interface that issues SCSI "commands" to request services from a target, typically a disk array, that executes SCSI commands. However, most of the discussion is applicable to a target in the case it is implemented by software like some iSCSI appliances on the market.

The outline of the rest of this paper is as follows. Section II summarizes related work. Section III covers the issues related to optimizations for to achieve the high performance, and discusses some of the detailed implementations. Section IV presents our performance results. And then Section V summarizes the main points to conclude the paper.

II. Related work

Some performance studies of the iSCSI protocol have been conducted in the past. Sarkar et al. [2] evaluated the performance of a software implementation and those of implementations with TOE and HBA adapters. Stephen Aiken

et al. [3] contrasted the performance of a software implementation with that of Fibre Channel. Wee Teck Ng et al. [4] investigated the performance of the iSCSI protocol over wide area networks with high latency and congestion. Unlike our iSCSI implementation, their software implementations are straightforward, that is, they are based on unmodified TCP/IP stacks, copy data between buffers in a VM system and those in a network subsystem in an operating system, and do not exploit some of the features that the network adapter supports.

We cannot directly compare the performance that we obtained with those obtained from their studies on TOE adapters, HBA adapters, and Fibre Channel, due to differences in the experimental infrastructure. However, we believe that the indirect comparison between their studies and ours may provide some useful information.

There are a number of papers that point out that TCP performance on high-speed links is limited by the host system and study several optimization approaches. Chase et al. [5] outline a variety of approaches. In this paper, we adopt a technique called *page flipping* or *page remapping* to avoid copying, and evaluate how much it affects the performance of optimized software implementation of the iSCSI protocol.

Unlike many studies [6]–[8] on TCP performance, this paper does not address the avoidance of copying data between the kernel and user address spaces. Instead, it focuses on the avoidance of redundant copying between buffers in several subsystems in an operating system. That is, our goal is "single-copy" from the application point of view. Even if using techniques studied in the literature to avoid copying data between the kernel and user address spaces may possibly result in better performance, making the techniques work with the iSCSI protocol requires extensive modifications to subsystems in an operating system, such as file systems, a memory management architecture and a network subsystem.

TCP overhead on the host system is mostly inherited from its design. It leads to many newly low overhead networking protocols, called user level protocol or light weight protocol [9]–[11]. These works result in specialized file systems such as DAFS [12]. These protocols that are free from negative inheritances of the TCP protocol can achieve better performance. Furthermore, to take advantage of the merits of both new protocols and TCP protocol, integrating them with TCP protocol has been addressed.

III. Optimization techniques

Figure 1 shows the difference between the architectures of general SCSI drivers and iSCSI drivers, and data flow inside an operating system[1]. In general, an iSCSI initiator is implemented as a SCSI host bus adapter driver. However, unlike general SCSI drivers interacting with their own host bus adapter hardware, an iSCSI driver interacts with a network subsystem.

[1]We assume that cached file system data are managed by a virtual memory system, as in the Linux kernel.

Commonly cited disadvantages of IP-based storage networking over existing specialized storage networking architectures are the TCP/IP protocol and data copying overheads. They can decrease the performance of applications by consuming CPU and memory system bandwidth.

Though general SCSI host bus adapter drivers directly move data between buffers in a VM system and the target, straightforward iSCSI drivers copy data between buffers in a VM system and those in a network subsystem. We were able to find four implementations of an iSCSI initiator [13]–[16], which are licensed under open source licenses [17], for the Linux operating system. All implementations copy data between the VM cache and the network subsystem. However, the copying can be avoided in some cases.

We discuss techniques for copy avoidance and the issues of implementing them in the Linux kernel in the following sections.

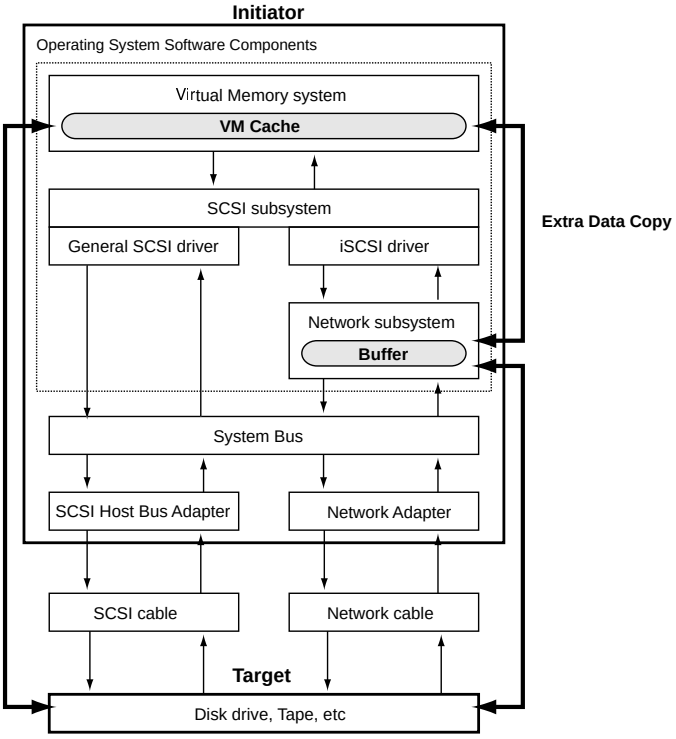


Fig. 1. SCSI driver architecture.

A. Network adapter support

Though commodity Gigabit Ethernet adapters cannot directly process the TCP/IP protocol and the iSCSI protocol unlike specialized network adapters, they provide some features that can be useful for offloading the TCP/IP protocol and data copying overheads, which IP-based storage networking has. We use the following three features.

- **Checksum offloading.** Calculating checksums to detect the corruption of a packet is an expensive operation. Checksum calculation on a network adapter instead of the host CPU can improve performance, and it is common

today. Six out of nine kinds of chipsets for a Gigabit Ethernet adapter that the Linux kernel version 2.6.0-test1 supports provide this feature.
Checksum offloading works on both the transmitting and receiving sides[2].

- **Jumbo frames.** Large packets can improve performance by reducing the number of packets that the host processes. However, standard IEEE 802.3 Ethernet frame sizes allow the MTU size to be 1500 bytes at the most. Nowadays most Gigabit Ethernet adapters support jumbo frames, that allow the host to use the larger MTU size (typically up to 9000 bytes). Seven out of nine kinds of chipsets for a Gigabit Ethernet adapter that the Linux kernel version 2.6.0-test1 supports provide this feature.
  Jumbo frames are mandatory to achieve copy avoidance when reading data.
- **TCP Segmentation offloading.** A network subsystem must break a packet whose size is larger than the TCP segment size into smaller pieces at the TCP layer before it is passed to the IP layer. TCP Segmentation offloading (TSO) can improve performance by executing some parts of this work on a network adaptor, and is relatively new. Two out of nine kinds of chipsets for a Gigabit Ethernet adapter that the Linux kernel version 2.6.0-test1 supports provide this feature.
  Though copy avoidance is possible without TCP Segmentation offloading, it can improve the performance when writing large data.

B. Write

The avoidance of copying when data travel from an initiator to a target (i.e., write) is easier than when data travel from a target to an initiator (i.e., read). Instead of copying data from the VM cache into buffers in the network subsystem, our iSCSI driver hands over references to buffers in the VM system to the network subsystem when writing.

In the Linux kernel, an *sk_buff* structure is used for memory management in the network subsystem. It is called *skbuff*, and is equivalent to an *mbuf* structure in the BSD lineage. An skbuff holds not only its own buffer space, but also references to a *page* structure describing a page in order to support avoiding redundant copying between the VM cache and skbuffs. Moreover, the Linux kernel also provides interfaces called *sendpage* interfaces to use these data structures.

These interfaces pass references to *page* structures describing pages in the VM system to the network subsystem. And, a network adapter driver receives skbuffs holding the references from the network subsystem, and then the contents of the referred pages are transferred to the wire.

By using these interfaces, our iSCSI driver can avoid copying data between the VM cache and skbuffs without modifications to the Linux kernel. Our iSCSI driver performs the following operations when writing.

1) The SCSI subsystem receives references to *page* structures describing dirty pages[3] from the VM system, and hands over them to the iSCSI driver.
2) The iSCSI driver associates these references with skbuffs for iSCSI packets by using sendpage interfaces.
3) References are passed on to a network adapter driver and the contents of referred pages are transmitted by using direct memory access (DMA).

Unlike the avoidance of copying data between the kernel and user address spaces, the iSCSI driver does not require pinning and unpinning of transferred pages during DMA transfers, because the VM system increases a usage count of these pages. It guarantees that the VM system does not reclaim these pages until the iSCSI command related them finishes.

To get the best performance improvement from copy avoidance, a network adapter has to provide checksum offloading to calculate data checksums during DMA transfers. Without this feature, the performance gain from copy avoidance becomes negligible [5], because in the Linux kernel the TCP stack integrates calculating the checksum on data with copying the data.

1) iSCSI digest: Without modifications to the Linux kernel, it is possible to avoid copying when writing data in most cases. However, this scheme for copy avoidance does not work in the case the *data digest* feature that the iSCSI protocol supports is enabled.

The iSCSI protocol defines a 32-bit CRC digest on an iSCSI packet to detect the corruption of iSCSI PDU (protocol data unit) headers and/or data because a 16 bit checksum used by TCP to detect the corruption of a TCP packet was considered to be too weak for the requirements of storage on long distance data transfer [18]. These digests are called *header digest* and *data digest* respectively. When these digests are used, the iSCSI driver performs in a slightly different way.

1) The SCSI subsystem receives references to dirty pages from the VM system, and hands over them to the iSCSI driver.
2) The iSCSI driver associates these references with skbuffs for iSCSI packets by using sendpage interfaces.
3) If the header digest feature is enabled, the iSCSI driver computes a digest on an iSCSI PDU header.
4) If the data digest feature is enabled, the iSCSI driver computes a digest on an iSCSI PDU data, that is, the contents of dirty pages.
5) References are passed on to a network adapter driver and the contents of referred pages are transmitted by using DMA.

Note that the Linux kernel permits modifying the contents of a page that are being written to persistent storage. Therefore, a process or a subsystem can modify the contents of a page at any time during all of the above operations.

If the contents of a transferred page are modified between the fourth and fifth operations, a target sees the combination

[2]Some network adapters can compute checksums on only transmitting packets.

[3]A dirty page means a page whose contents are modified in the VM system that must be written to persistent storage.

of the data digest on the old contents and the new contents. Thus, the verifying of the data digest fails on the target. In this case, the iSCSI driver must compute again a data digest on the new contents of the page before a network adapter transmits them. However, the iSCSI driver has no chance to do it. Therefore, the iSCSI driver must keep a transferred page untouched after computing a data digest on the page until the iSCSI command related to the page finishes. A synchronous scheme to avoid such a situation is necessary to enable our copy avoidance scheme to work with the data digest feature. The current version of our iSCSI driver simply copies data between the VM cache and skbuffs instead of using sendpage interfaces in the case the data digest feature is used.

On the other hand, the header digest feature always works with our scheme to avoid copying, because a header of an iSCSI PDU can be modified only by the iSCSI driver.

C. Read

With regard to reading data, the avoidance of copying can be more problematical. This is because an iSCSI driver has to place incoming data from a network subsystem at arbitrary virtual addresses specified by a VM system without copying the data.

An HBA adapter processing iSCSI and network protocols can determine the virtual addresses chosen by a VM system corresponding to incoming data, and directly place the data in the appropriate places. Thus the driver does not have to do anything with it. On the other hand, a commodity network adapter cannot process these protocols. Therefore, it cannot directly place incoming data at the virtual addresses specified by a VM system.

One technique to avoid copying data with a commodity network adapter is page remapping. It changes address bindings between buffers in a VM system and a network subsystem. This scheme is widely used in research systems; however, it has some restrictions.

- The Maximum Transfer Unit (MTU) size must be larger than the system virtual memory page size.
- The network adapter must place incoming data in a way that the data that a VM system requires are aligned on page boundaries after removing the Ethernet, IP, TCP, and iSCSI headers.
- Page remapping requires the high-cost operations of manipulating data structures related to a memory management unit (MMU) and flushing an address-translation cache, called a translation lookaside buffer (TLB). These operations are particularly slow in symmetric multiprocessor (SMP) environments.

The first issue means that the technique does not work with standard IEEE 802.3 Ethernet frame sizes, and thus jumbo frames are mandatory.

The requirement in the second issue can be achieved with some modifications to a general-purpose operating system. We explain these modifications in subsequent paragraphs.

The operations in the last issue can be avoidable with data movement between buffers in a VM system and those in a network subsystem[4], though such operations are mandatory with copy avoidance between the kernel and user address spaces. The reason for this is that any kernel virtual address into which a page in a VM system will do in UNIX operating systems as long as the page is mapped.

In general, pages in a VM system are temporarily mapped into the kernel virtual address space. When a process or a subsystem reads or writes the contents of a page, the page is mapped into a kernel virtual address. When it finishes the operation, the page is unmapped. The page can be mapped into another kernel virtual address next time. That is, a process does not directly see the kernel virtual address into which a page is mapped.

This scheme does not work if a page in the VM system is mapped into user virtual addresses, e.g, if files or devices are mapped into user virtual addresses by using the *mmap* system call, high-cost MMU operations are inevitable. In this case, a process directly sees user virtual addresses, and thus a new page must be mapped into the same user virtual addresses.

Our iSCSI driver requires some modifications to two parts in the Linux kernel: the memory management in the network subsystem and the VM cache management.

1) Memory management in the network subsystem: The *alloc_skb* function does not allow allocating space for the skbuff with arbitrary alignment. Thus, we slightly modified the function and an *sk_buff* structure.

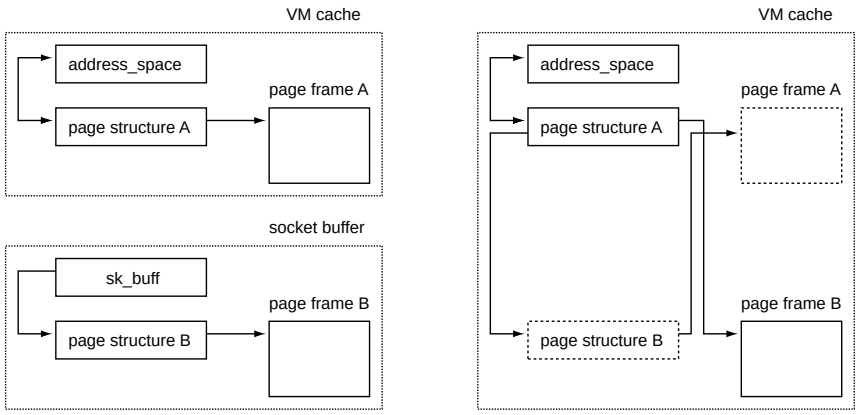
A network adapter allocates all skbuffs in expectation of receiving an iSCSI packet including a response to a READ command. Therefore, after the network subsystem and the iSCSI driver finish processing the protocol headers, iSCSI PDU data, that is, request data stored on the disk of the target, show up on page boundaries. This scheme can only be applied to the first TCP packet in the case that one iSCSI PDU consists of several TCP packets. This is because the length of the header of the first packet is different from that of the rest. Therefore, after using page remapping for the first TCP packet, our iSCSI driver copies data in the rest of the packets.

2) VM cache management: Figure 2 shows how the copy avoidance scheme works after the iSCSI driver finishes processing an iSCSI PDU header.

Figure 2(a) represents data structures just before page remapping. One *page* structure 'A' describes a page 'A' in the VM system and one *page* structure 'B' describes a page 'B' used for the space for an skbuff. Figure 2(b) shows the structures after page remapping. The *page* structure 'A' describes the page 'B' in the VM system. The *page* structure 'B' and the page 'A' are not used, and the *sk_buff* structure is freed. If necessary, the iSCSI driver performs MMU operations to change the address bindings between these pages.

This scheme allows the contents of the page that *page* structure 'A' describes to be updated without copying. Before page remapping, when a process or a subsystem tries to read

[4]The word 'page remapping' means changing address bindings by performing MMU operations. Therefore, page remapping may not be the appropriate name for the scheme to avoid copying between buffers in a VM system and those in a network subsystem.



(a) Before page remapping

(b) After page remapping

Fig. 2. VM cache and skbuff management.

the contents of the page that *page* structure 'A' describes, the page 'A' is mapped into the kernel virtual address space, and then they see the contents of a page 'A'. After page remapping, the page 'B' is mapped into the kernel virtual address space, and then they see the contents of a page 'B'.

To implement our scheme, we made two modifications to the Linux VM system.

First, in the modified VM system, the relationship between a *page* structure and the page that it describes is dynamic. In the Linux VM system, a *page* structure is allocated for every page frame when the system boots, and the relationship between them is static. We add the index of a page frame to a *page* structure to enable it to describe any page. This modification increases the overheads of some operations related to memory management. For example, it becomes complicated to know the kernel virtual address into which a page is mapped by using a *page* structure describing the page. It also becomes complicated to know a *page* structure describing a page by using the physical address of the page.

To keep modifications to a minimum, we implement page remapping by using a special kernel virtual address space, called *high memory*, which is used for a host system with a large amount of physical memory. Though the original Linux VM system maps most pages into the kernel virtual address space all the time, some pages are temporarily mapped into the high memory address space as the need arises. In the Linux VM system, pages are mapped into the kernel virtual address space starting from 3 GB with Intel 32 bit architecture. Without high memory, therefore, about only 1GB of the main memory at most can be used. The iSCSI driver expands this address space and uses some amount of the physical memory exclusively for page remapping.

Secondly, in the modified VM system, a *page* structure holds the reference to another *page* structure as shown in Figure 2(b). This extension allows these *page* structures to be restored to their original state when the page is reclaimed. The old page is not freed until the new page is freed, or replaced with another page.

D. Another potential solution

Another potential solution to avoid copying when reading would be to directly replace a *page* structure with another.

As shown in Figure 2(a), we can directly replace a *page* structure 'A' with a *page* structure 'B'. This solution requires extensive modifications to the Linux kernel, because a *page* structure contains various data structures that subsystems use. Take for example the case where a process is waiting for the contents of a page to be updated by using data structures in the *page* structure that describes the page. If the *page* structure is replaced, the operating system must guarantee that the replaced *page* structure remains untouched until all processes that use it to wait for the contents resume.

IV. Performance

We performed microbenchmarks to get the basic performance characteristics of our iSCSI driver.

A. Experimental infrastructure

- **Initiator** The Dell Precision Workstation 530 uses two 2 GHZ Xeon processors with 1 GB of PC800 RDRAM main memory, an Intel 860 chipset and an Intel Pro/1000 MT Server Adapter connected to a 66-MHz 64-bit PCI slot. The network adapter supports checksum offloading on both the transmitting and receiving sides, jumbo frames, and TSO.
  The initiator runs a modified version of the Linux kernel version 2.5.67 and our iSCSI initiator implementation that is based on the source code of the IBM iSCSI initiator [16], [19] for the Linux kernel version 2.4 series. The operating system has an 18GB partition on the local disk, and the iSCSI driver uses a 73.4 GB partition on the target server.

- **Target** The IBM TotalStorage IP Storage 200i is an iSCSI appliance on two 1.13 GHz Pentium III processors with 1 GB of PC133 SDRAM main memory and an Intel Pro/1000 F network adapter connected to a 66-MHz 64-bit PCI slot. Our system houses Ultra 160 73.4 GB 10000 RPM SCSI disks.
  While the details of the software have not been disclosed, it is believed that the appliance runs the Linux kernel version 2.4.2 and the iSCSI target implementation that runs in kernel mode.

The initiator and the target are connected by an Extreme Summit 7i Gigabit Ethernet switch.

The iSCSI initiator and target implementations are compatible with the iSCSI draft version 8. Though it is not the latest, we believe this issue does not significantly affect the performance under our experimental conditions, i.e., high-speed and low-latency networks.

B. Measurement tool

We implemented microbenchmarks that run in kernel mode and directly interact with the VM system in order to bypass the VM cache. All iSCSI commands request the operation on the same disk block address. This leads to the best cache usage on the target. Therefore, the effect of the target server factors, such as disk performance, on the results is kept to a minimum.

To measure the CPU utilization and where the CPU time is spent while running the microbenchmarks, we use OProfile suite [20], which is a system-wide profiler for the Linux kernel. The OProfile suite uses Intel Hardware Performance Counters [21] to report detailed execution profiles.

C. Configuration

We conducted the microbenchmarks with an MTU of 1500 bytes (standard Ethernet) or 9000 bytes (Ethernet with jumbo frames).

The microbenchmarks were run with the following four configurations: *zero-copy, checksum offloading and TSO*, for which the iSCSI driver avoids copying data between the VM cache and skbuffs, and uses checksum offloading and TSO that the network adapter supports; *zero-copy and checksum offloading*, for which the iSCSI driver avoids copying data, and uses checksum offloading; *checksum offloading*, for which the iSCSI driver copies data in the same way that a straightforward iSCSI driver does and uses checksum offloading; and *no optimizations*, for which the iSCSI driver copies data and does not use checksum offloading or TSO.

The microbenchmarks read or write data 100,000 times with the I/O sizes ranging from 512 bytes to 64 KB. Except in the 64 KB case, one I/O request is converted to one iSCSI command. For the 64 KB case, one I/O request is converted to two iSCSI commands due to a restriction on the maximum I/O size on the target.

Here, we report the averages of five runs for all experiments.

D. Results

1) CPU utilization and bandwidth: Figure 3(a) shows the results of CPU utilization during the write microbenchmarks with an MTU of 1500 bytes. Except for the sizes ranging from 512 bytes to 1024 bytes, the two configurations with which the iSCSI driver avoids copying, the *zero-copy, checksum offloading and TSO* and *zero-copy and checksum offloading* configurations, yield a lower CPU utilization than the others. It means that copy avoidance improves performance over the large I/O sizes, and that over the small I/O sizes, the cost of sendpage interfaces is slightly higher than the cost of coping data.

Though TSO is used for all I/O sizes, it effectively works for I/O sizes greater than 2048 bytes. This is because in the block sizes of 512 bytes to 1024 bytes TCP segmentation does not happen. That is, one TCP/IP packet can carry all data that constitute one iSCSI PDU. Therefore, when the I/O sizes are over 2048 bytes, the *zero-copy, checksum offloading and TSO* configuration yields a lower CPU utilization than the *zero-copy and checksum offloading* configuration does. But the gain from TSO is not clear. For example, CPU utilization is 29.0% in the *zero-copy, checksum offloading and TSO* configuration and 30.2% in the *zero-copy and checksum offloading* configuration at 32 KB.

As explained in Section III, the gap between the *checksum offload* and *no optimizations* configurations is small.

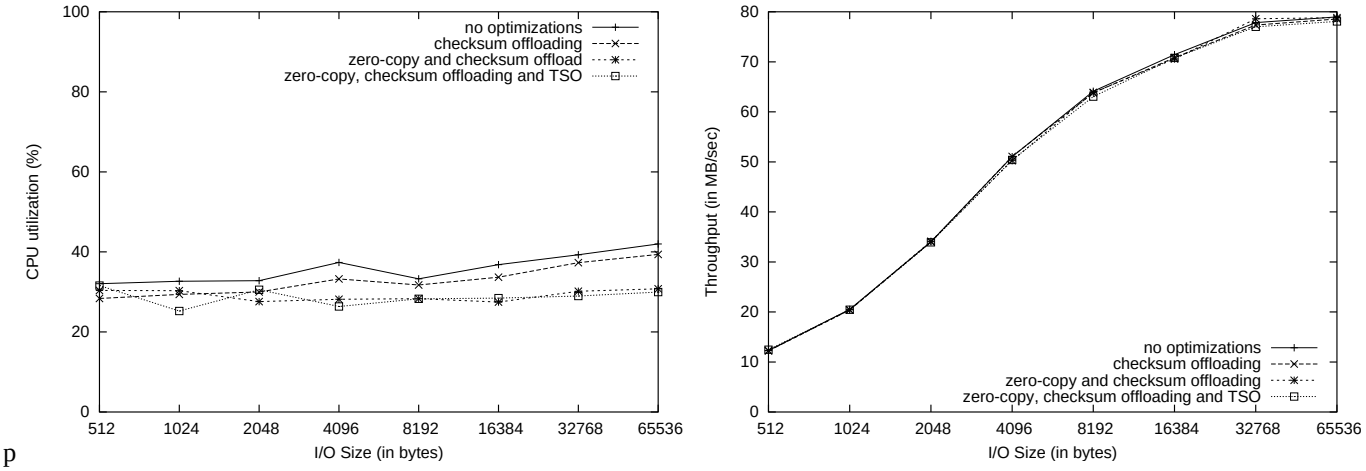
We see the largest gain from copy avoidance at 64 KB. CPU utilization is 30.0% in the *zero-copy, checksum offloading and TSO*, 30.8% in the *zero-copy and checksum offloading*, 39.4% in the *checksum offloading*, 42.0% in the *no optimizations* configuration with the 1500-byte MTU.

We could not determine the reason why the *zero-copy and checksum offloading* configuration yields a lower CPU than the *zero-copy, checksum offloading and TSO* configuration in the block size of 2048 bytes.

Figure 3(b) shows the results of throughput during the write microbenchmarks with the 1500-byte MTU. All configurations provide comparable throughputs.

Figures 4(a) and 4(b) show the results of CPU utilization and throughput during the write microbenchmarks with the 9000-byte MTU, respectively. These results are similar to those with the 1500-byte MTU. But, as expected, CPU utilization with the 9000-byte MTU is lower than one with the 1500-byte MTU. The throughput with the 9000-byte MTU is better than the one with the 1500-byte MTU.

Figures 5(a) and 5(b) show the results of CPU utilization and throughput during the read microbenchmarks with the 1500-byte MTU. As explained in Section III, our scheme to avoid copying is impossible with the 1500-byte MTU. In addition, TSO has no effect with regard to the read microbenchmarks. Therefore, only the *checksum offloading* and *no optimizations* configurations are presented. Like the write microbenchmarks, the *checksum offloading* configuration yields a slightly lower CPU utilization than the *no optimizations* configuration. Furthermore, both configurations perform comparably with regard to throughput.

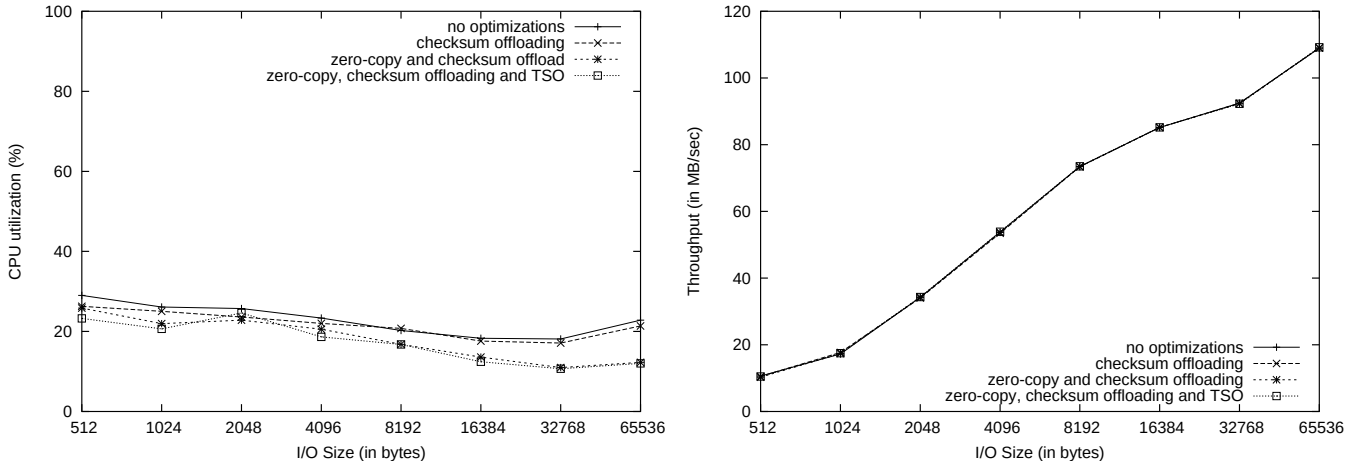


p

(a) Write CPU utilization (%).

(b) Write Throughput (MB/s).

Fig. 3. Results of write microbenchmarks with a 1500-byte MTU.



(a) Write CPU utilization (%).

(b) Write Throughput (MB/s).

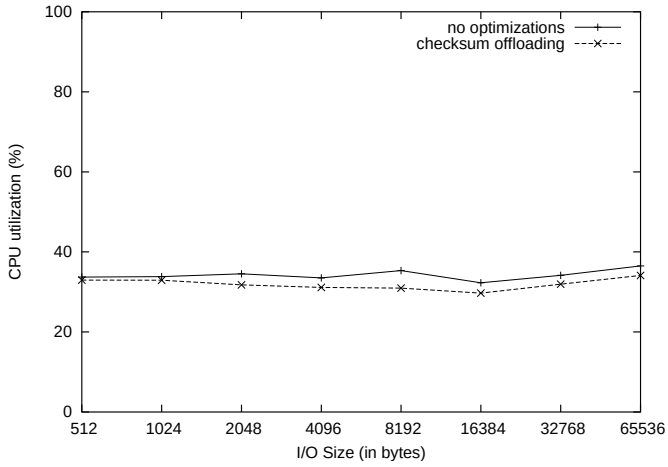
Fig. 4. Results of write microbenchmarks with a 9000-byte MTU.

Figures 6(a) and 6(b) show the results of CPU utilization and throughput during the read microbenchmarks with the 9000-byte MTU. Though the *zero-copy and checksum offload* configuration is feasible under 4096 bytes (the system virtual memory page size) through the combination of page remapping and copying data, its gain is smaller than the one for the 4096-byte I/O size and over. Therefore we concentrate on the results over 4096 bytes with regard to the *zero-copy and checksum offload* configuration.

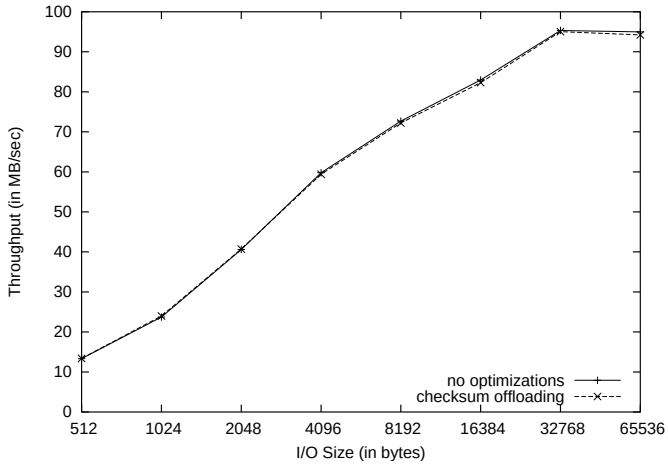
While at over 16 KB, both page remapping and copying data are required, at 8192 bytes, the iSCSI driver does not need to copy; that is, it only needs to manipulate some data structures for memory management. Therefore, it gets the largest gain from the optimizations at 8192 bytes. However, we cannot see a clear gain. CPU utilization is 21.1% in the *zero-copy and checksum offloading* configuration and 21.3% in the *checksum offloading* configuration at 8192 bytes. Moreover, at 4096 bytes and over 16 KB, the *checksum offloading* configuration yields the lower CPU utilization. We investigate the reasons for this behavior in subsequent paragraphs.

In the read microbenchmarks, the iSCSI driver provides comparable throughput in all configurations with the 9000-byte MTU, as it does with the 1500-byte MTU.

2) Distribution of CPU time: Next, we investigate where the CPU time is spent while running the microbenchmarks. Table I and II show the distribution of CPU time during the write and read microbenchmarks for the 8192-byte I/O size, respectively. The reason why we choose the I/O size of 8192 bytes is that this size most clearly shows the characteristics of the *zero-copy and checksum offloading* configuration in the

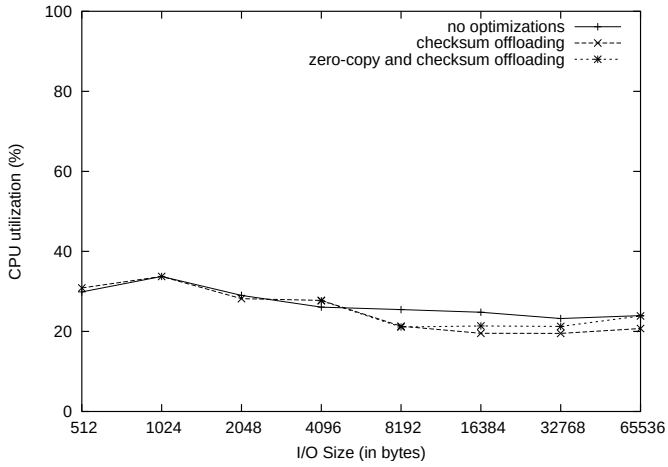


(a) Read CPU utilization (%).

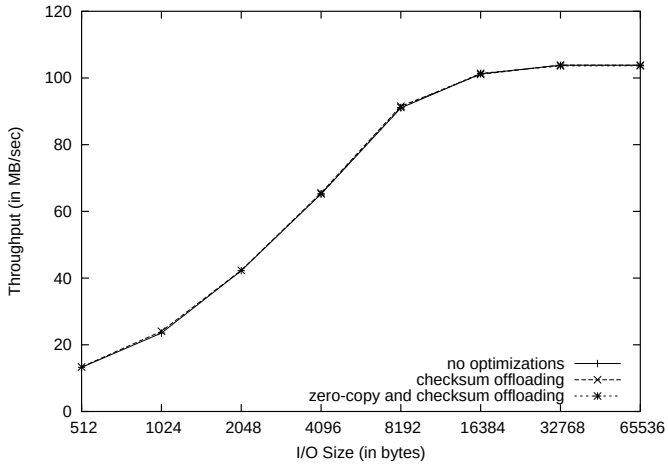


(b) Read Throughput (MB/s).

Fig. 5. Results of read microbenchmarks with a 1500-byte MTU.



(a) Read CPU utilization (%).



(b) Read Throughput (MB/s).

Fig. 6. Results of read microbenchmarks with a 9000-byte MTU.

read microbenchmarks.

As mentioned in the previous paragraph, we cannot see a clear gain from TSO. Therefore, we omit the result of the *zero-copy, checksum offloading and TOE* configuration.

There are six types of large overhead: the iSCSI driver, copying data and checksums, the SCSI subsystem, the network adapter driver, the TCP/IP stack, and the VM system.

The results shows that in the write microbenchmarks, the combination of copy avoidance and checksum offloading decreases the CPU time spent on copying data and calculating checksums. And we also clearly see the effect of a large MTU. It greatly decreases the CPU time spent on the TCP/IP stack and also decreases the time spent on the network adapter driver.

The read microbenchmarks perform like the write microbenchmarks. The combination of copy avoidance and checksum offloading decreases the CPU time spent on copying data and calculating checksums. Furthermore, a large MTU decreases the CPU time spent on the TCP/IP stack and on the network adapter driver. The major difference is that the optimization for copy avoidance increases the CPU time spent on the VM system and the TCP stack.

The increased overhead of the VM is mainly due to the complexity of the modified VM system. As explained in Section III, in the modified VM system, the relationship between a *page* structure and the page that it describes is dynamic. This increases the overhead of memory-management operations. Furthermore, the increased overhead of memory-management operations increases the CPU time spent on the TCP/IP stack, which often uses these operations. As a result,

TABLE I

DISTRIBUTION OF CPU TIME DURING THE WRITE MICROBENCHMARKS FOR THE 8192-BYTE I/O SIZE.

	Percent Time					
	no optimizations		checksum offload		zero-copy and checksum offload	
MTU (byte)	1500	9000	1500	9000	1500	9000
Idle	66.81	79.85	68.35	79.23	71.86	83.32
iSCSI driver	1.52	1.34	1.53	1.41	1.32	1.30
SCSI subsystem	1.90	1.25	1.86	1.42	1.17	1.23
Copy and Checksum	3.76	3.40	2.93	2.41	0.27	0.27
Network adapter driver	2.12	0.96	2.27	1.07	2.60	1.06
TCP/IP	12.16	3.72	11.75	4.39	11.48	3.82
VM system	4.21	2.98	4.11	3.16	3.67	2.71

TABLE II

DISTRIBUTION OF CPU TIME DURING THE READ MICROBENCHMARKS FOR THE 8192-BYTE I/O SIZE.

	Percent Time					
	no optimizations		checksum offload		zero-copy and checksum offload	
MTU (byte)	1500	9000	1500	9000	1500	9000
Idle	64.84	74.77	69.08	78.74	n/a	79.89
iSCSI driver	2.01	1.99	1.93	1.79	n/a	2.12
SCSI subsystem	1.44	1.49	1.25	1.19	n/a	1.29
Copy and Checksum	6.78	6.65	3.81	3.77	n/a	0.16
Network adapter driver	2.28	1.92	2.14	1.93	n/a	2.18
TCP/IP	9.84	3.36	9.75	2.92	n/a	4.79
VM system	4.54	3.22	4.28	3.08	n/a	4.89

the copy avoidance's performance gain is negated.

These investigations also give some ideas on how much performance the iSCSI protocol provides with specialized adapters. The important observation is that the overhead of processing the iSCSI protocol and the SCSI subsystem consumes a relatively small amount of CPU time. For example, during the write microbenchmarks in the *zero-copy and checksum offloading* with the 1500-byte MTU, it accounts for only 8.9% of the processing time in which the CPU is not idle. The largest source of overhead is the TCP/IP protocol. It accounts for 40.8% of the total processing time.

V. CONCLUSION

This paper presents an efficient implementation of an iSCSI initiator with a commodity Gigabit Ethernet adapter, and experimentally analyzed its performance. Our implementation of an iSCSI initiator avoids copying between the VM cache and buffers in the network subsystem by using page remapping, and exploits features that commodity Gigabit Ethernet adapters support, i.e., checksum offloading, jumbo frames, and TCP segmentation offloading.

Our analysis of writing showed that the combination of copy avoidance and checksum offloading reduces CPU utilization from 39.4% to 30.8% with an I/O size of 64 KB in our microbenchmarks as compared with a straightforward iSCSI driver copying data. In addition, the copy avoidance technique that we used can be easily integrated into the Linux kernel by using existing interfaces. However, we cannot see a clear gain from TCP segmentation offloading.

With regards to reading, the copy avoidance technique requires modifications to the Linux kernel. Furthermore, the overhead due to our modifications to the VM system negates any gain in performance that could be had from copy avoidance.

Our experiments show that the CPU is not the performance bottleneck in all configurations and the iSCSI driver provides comparable throughput with each MTU.

The quantitative analysis shows that the overhead relative to the iSCSI protocol and the SCSI subsystem consumes a relatively small amount of CPU time (8.9% of the total processing time during the write microbenchmarks with an MTU of 1500 bytes) and the largest overhead comes from processing the TCP/IP protocol with standard IEEE 802.3 Ethernet frame sizes.

REFERENCES

[1] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner, "IPS Internet Draft : iSCSI," January 2003.

[2] P. Sarkar, S. Uttamchandani, and K. Voruganti, "Storage over IP: When Does Hardware Support Help?" in *the Conference on File and Storage Technologies (FAST 03)*. San Francisco, CA: USENIX, January 2003, pp. 231–244.

[3] S. Aiken, D. Grunwald, A. R. Pleszkun, and J. Willek, “A Performance Analysis of the iSCSI Protocol,” in *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*. San Diego, California: IEEE, April 2003, pp. 123–134.

[4] W. T. Ng, B. Hillyer, E. Shriver, E. Gabber, and B. Özden, “Obtaining High Performance for Storage Outsourcing,” in *Conference on File and Storage Technologies*. Monterey, CA: Usenix, January 2002, pp. 145–158.

[5] J. S. Chase, A. J. Gallatin, and K. G. Yocum, “End-System Optimizations for High-Speed TCP,” *IEEE Communications*, vol. 39, no. 4, pp. 68–74, 2001.

[6] Hsiao-keng Jerry Chu, “Zero-Copy TCP in Solaris,” in *the USENIX 1996 Annual Technical Conference*, San Diego, CA, January 1996, pp. 253–264.

[7] V. S. Pai, P. Druschel, and W. Zwaenepoel, “IO-Lite: A Unified I/O Buffering and Caching System,” *ACM Transactions on Computer Systems*, vol. 18, no. 1, pp. 37–66, 2000.

[8] J. C. Brustoloni and P. Steenkiste, “Interoperation of Copy Avoidance in Network and File I/O,” in *the INFOCOM’99 Conference*. New York: IEEE, March 1999, pp. 534–542.

[9] S. Pakin, M. Lauria, and A. Chien, “High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet,” in *Supercomputing ’95*, San Diego, CA, December 1995.

[10] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd, “The Virtual Interface Architecture,” *IEEE micro*, vol. 18, no. 2, pp. 66–76, March/April 1998.

[11] T. von Eicken, A. Basu, V. Buch, and W. Vogels, “U-Net: A User-Level Network Interface for Parallel and Distributed Computing,” in *the Fifteenth ACM Symposium on Operating System Principles*, December 1995, pp. 40–53.

[12] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle, “The Direct Access File System,” in *2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, March 2003, pp. 175–188.

[13] Intel iSCSI Reference Implementation, http://sourceforge.net/projects/intel-iscsi/.

[14] InterOperability Laboratory, http://www.iol.unh.edu/consortiums/iscsi/.

[15] Cisco iSCSI initiator Implementation, http://sourceforge.net/projects/linux-iscsi/.

[16] IBM iSCSI Initiator Implementation, http://www-124.ibm.com/developerworks/projects/naslib/.

[17] Open Source Initiative (OSI), http://www.opensource.org/.

[18] K. Z. Meth and J. Satran, “Design of the iSCSI Protocol,” in *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*. San Diego, California: IEEE, April 2003, pp. 116–122.

[19] K. Z. Meth, “iSCSI Initiator Design and Implementation Experience,” in *Tenth NASA Goddard Conference on Mass Storage Systems and Technologies Nineteenth IEEE Symposium on Mass Storage Systems*. Maryland, USA: IEEE, April 2002.

[20] OProfile - A System Profiler for Linux, http://oprofile.sourceforge.net/.

[21] Intel, *The IA-32 Intel Architecture Software Developer’s Manual, Volume 3: System Programming Guide*, 2003.