TPF: TCP Plugged File System for Efficient Data Delivery over TCP

Sang Seok Lim, *Member*, *IEEE*, and Kyu Ho Park, *Member*, *IEEE*

Abstract—Most Internet services, including Web, FTP, and streaming, have been realized on top of TCP, which is the de facto protocol for data delivery over the Internet. Therefore, in order to achieve high-performance data delivery over TCP, we thoroughly analyze TCP-based data delivery and identify three critical mismatches in a general file system design while supplying data to TCP. The first is the frequent sleeping/waking up of a server process that accompanies excessive context switching overheads due to processing TCP data and ACK segments in different contexts. The second is the inefficient uniform data prefetching for TCP connections, irrespective of their characteristics such as bandwidth, latency, and the status of a send buffer. The third is inefficient disk access due to the ignorance of abrupt changes in TCP connections. As a remedy to these mismatches, we newly design a TCP-plugged file system (TPF) which is comprised of three novel mechanisms, each of which relieves the identified mismatches, integrated data sending routine, TCP aware data prefetching, and eager disk request cancellation. With these mechanisms, TPF becomes capable of supplying data managed by a file system to TCP connections timely and seamlessly and becomes reactive to abrupt changes in TCP connections. As a consequence, TPF provides minimal context switching overhead, high buffer utilization, and highly effective disk access utilization. We have implemented and tested the mechanisms in Linux 2.4. The experimental results show that the number of context switching is reduced by up to 40 percent and the overall system performance is improved by 3-34 percent.

Index Terms—Operating systems, file systems, TCP, data prefetching, disk scheduling.

1 Introduction

Broadly deployed Internet server systems, such as the Web, FTP, and streaming servers, are generally equipped with two indispensable I/O subsystems: network subsystems, such as TCP and UDP protocol stacks, and disk subsystems, such as ordinary file systems and databases. Although each server system provides different types of Internet services, there is one common and fundamental operation of the systems: to deliver data managed by disk subsystems to clients via network subsystems. During data transfer in the server system, network and disk subsystems must cooperate with each other.

For performance improvement of Internet server systems, most studies have focused on either optimizing individual I/O subsystems [1], [2], [3], [4], [5], [6] or eliminating data copying among the subsystems [7]. There have been few studies about interoperation between two subsystems, considering each subsystem's specific mechanisms. For instance, previous studies have not addressed the following issues: 1) how TCP mechanisms such as congestion control and flow control will affect the internal operations of a file system, such as file prefetching, disk buffer management, and disk scheduling, or 2) what the most efficient reaction of a file system would be when a TCP protocol stack receives ACK (acknowledge), FIN, RST (reset), and ECN (explicit congestion notification) [8] segments. In an effort to find the best answers to these questions, we decided to review a current file system thoroughly in the context of TCP-based data transfer. Thus, we have closely investigated the dynamics of running Apache (HTTP) and Wu-FTPd (FTP) servers by observing the following details:

1. TCP buffer profiles according to the bandwidth and latency of a TCP connection,
2. TCP ACK and data segment processing mechanisms,
3. a data prefetching mechanism in a file system and prefetched data access timing, and
4. the above three behaviors under severe network congestion and a high connection abortion rate.

For this investigation, we narrowed down the network subsystems to a TCP protocol stack since it is the de facto transport protocol over the Internet and uses the Ext2 file system of Linux, which has a page cache for data prefetching and a disk scheduler to maximize disk throughput. The overall I/O subsystem architecture of the Linux is illustrated in Fig. 1. In the architecture, for sending data via TCP connections, first, a server application loads data from disk to a page cache and then copies the data to a send buffer, initiating TCP/IP protocol processing. Last, the processed data is delivered to NIC by DMA.

Our investigation disclosed the three rudimentary operations of the file system, which are summarized as follows:

- We observed frequent sleeping/waking up of the server processes while sending TCP segments due to the limited size of the send buffer for a TCP connection. This problem becomes worse when the connection has high latency because it leads to a large number of unacknowledged TCP segments in the

• *The authors are with the Department of Electrical Engineering, Korea Advanced Institute of Science and Technology, Guseongdong Yuseonggu Daejoen, Korea, 305-701.*
E-mail: sslim@core.kaist.ac.kr, kpark@ee.kaist.ac.kr.

Manuscript received 19 Jan. 2006; revised 27 May 2006; accepted 18 Aug. 2006; published online 1 Feb. 2007.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0020-0106.
Digital Object Identifier no. 10.1109/tc.2007.1009.

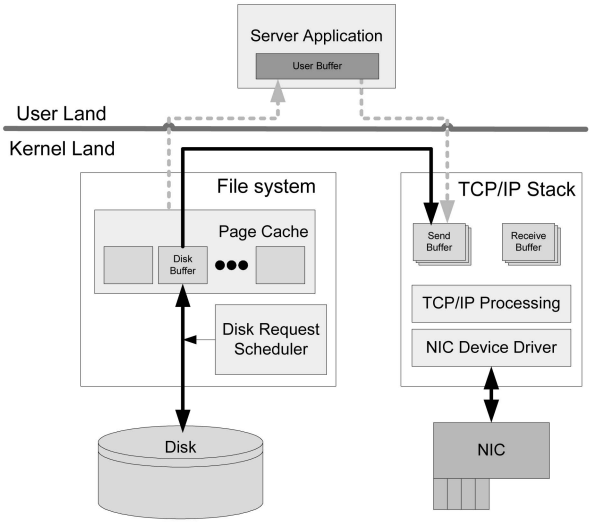

Fig. 1. The architecture of an I/O subsystem in Linux 2.4.

buffer and, finally, increases the number of pending TCP segments in the buffer. Although the size limitation of the buffer can be partly alleviated by the adaptive resizing of the send buffer [9], a server still suffers from the severe context switching overheads which accompany frequent sleeping/waking up of server processes. This is because TCP data segment sending routines and TCP ACK segment receiving routines are executed in different contexts.

- TCP has a flow control and congestion control mechanism to adapt itself to a dynamically changing status of network paths between two end points of a TCP connection. Therefore, at a given time, each TCP connection has a different bandwidth and latency so that it needs a different amount of data from a storage in place to transmit. However, conventional data prefetching mechanisms load the fixed amount of data without considering the bandwidth and latency of each connection. If too much data is loaded for a low-bandwidth TCP connection, it will waste disk bandwidth and memory. We need a new prefetching mechanism to make a file system capable of supplying data in a timely and continuous manner to TCP connections.
- To deal with frequent network congestion in the Internet, a variety of congestion control mechanisms have been proposed. When a TCP connection detects congestion symptoms, such as reception of duplicated ACK and ECN segments, it reduces the size of a congestion window by half. From the view point of data delivery in a server, the bandwidth of the connection will be cut in half immediately so that it will not need data from storage as much as it did a second before. However, current data prefetching algorithms will continue to pump data out of storage at the same rate. In addition, a recent study [10] pointed out that abnormal TCP connection termination frequently occurs due to human behavior and browser implementation. It reported that more than 15 percent of TCP connections were terminated by RST segments. On the reception of an RST segment, a server will not deliver data over the connection immediately. However, a current file system keeps pumping data out of a storage which will not be delivered to clients until the corresponding file is explicitly closed by the server application. Thus, we need a file system which can deal with abrupt changes in a TCP connection by either reducing the data prefetching rate or by stopping prefetching for an aborted connection.

In an effort to solve the identified design imperfection in the disk subsystem, we have designed and implemented TPF in Linux. Deliberating on our investigation into the TCP protocol stack and the disk subsystem, we devised three new mechanisms for the performance improvement of data delivery over TCP: integrated data sending routine (IDRC), TCP aware data prefetching (TDAP), and eager disk request cancellation (EDRC).

First, IDSR is targeted for minimizing context switching while sending TCP data segments and receiving TCP ACK segments. The ACK reception routines and TCP data sending routines of a TCP protocol stack are executed in different contexts, leading a sender process to frequent process blocking/waking up. It is inevitable in modern operating systems since, in order to make them resilient and responsive to multiple prioritized events, they have adopted multiple and prioritized contexts such as a hardware interrupt context, software interrupt context, and process context. However, from the viewpoint of a TCP data flow, it is more efficient to integrate the TCP data sending and ACK reception routines that are originally executed in different contexts so as to minimize the frequent sleeping/waking up of the server process.

Second, we design an adaptive prefetching mechanism, TDAP, which dynamically adjusts the amount of data to be prefetched according to the dynamic states of a given TCP connection characterized by a TCP flow control, a congestion control, and the bandwidth and latency of two end points of the TCP connection. By observing the movement of a TCP sliding window along a TCP sequence space and the number of pending segments in the send buffer, our new mechanism calculates how much data needs to be loaded into a memory buffer within a certain time constraint to sustain the outgoing data rate of the connection.

Third, an eager disk request cancellation (EDRC) is to maximize an effective disk throughput by making a disk request scheduler responsive to the abrupt changes in TCP connections, such as packet loss (duplicated ACKs), congestion notification (ECN), FIN, and RST. On the reception of those segments, EDRC scans the disk requests in the disk scheduler. If there is any request which is related to the TCP connection where abrupt changes occur, it either moves the queued requests into a delay queue or cancels the requests and frees the related memory pages, depending on the disk load. As a consequence, it enhances the efficiency of the disk access and the memory utilization.

TPF is a part of our cluster-based Internet server development project [11], [12], [13], [14]. In our server, TPF has been integrated into Apache (HTTP), FTP, and TCP-based streaming servers as the key primitive for a TCP-based data delivery over the Internet. It can be also extended to other application areas, including a content distribution network [15] and peer-to-peer file sharing, as long as they use TCP as a transfer protocol. TPF is not targeted to small file transfers that do not appear to be

affected by TCP network dynamics, but, instead, to large files and streaming data transfers.

The remainder of this paper is organized as follows: Previous studies about data delivery over the Internet in operating systems will be explained in Section 2. In Section 3, TPF with elaboration on three novel mechanisms will be given in detail. Section 4 presents our experimental results and we conclude our paper in Section 5.

2 Related Works

2.1 Previous Research on Internet Server

There have been many studies on the performance improvement of operating systems in the context of Internet servers [1], [2], [3], [4], [5], [6], [12]. Most of the studies optimized network or disk subsystems in the scope of individual subsystems. In the intersubsystem scope, a few studies have explored intersubsystem data copy elimination [7] and inbound/outbound TCP connections splicing [16], [17]. In *IO-Lite* [7], with a unified buffer which can be accessed and managed by both the disk and the network subsystem, all data copying and multiple buffering across subsystems can be eliminated. Our study has focused on optimizing data delivery timing and context between subsystems so that it is complementary to *IO-Lite*. *TCP splice* maximizes the performance of an application-layer proxy by achieving the tightest possible coupling between the two TCP connections coterminating at the proxy. With *TCP splice*, incoming data packets from one TCP connection are forwarded to the other TCP connection without intervention of the proxy so as to reduce the scheduling and context switching overhead. Conceptually similarly, we have spliced a disk subsystem to a TCP protocol stack by providing a new prefetching mechanism, a disk scheduler, integrated TCP data, and ACK segment processing routines especially optimized for the dynamic characteristics of TCP connections.

UCFS, a user-space file system for Web proxy servers [18], was devised to drastically improve their I/O performance. It manages its own data and metadata in a user space by bypassing an in-kernel file system. Its cluster-based data block management scheme, or an extension of the log-structured file system (LFS) [19], significantly improves disk I/O performance by clustering Web files that are likely to be requested together. In their approach, they only focused on the reduction in disk access without considering data delivery over a network connection, which is the integral part of proxy servers. We assert that TPF must work together with UCFS for further performance improvement.

ECEM [12] manages correlated I/O events from the network and disk subsystem in such a way that it minimizes the chance of process scheduling during data copying between an application buffer and kernel buffer. By doing so, it also improves data cache efficiency since two data copying operations between an application buffer and a kernel buffer are executed continuously without yielding the CPU to other processes. This approach, however, is only applicable to the case where data is copied via an application buffer and is not applicable to the case where data is copied from a disk buffer to a network buffer directly, which is quite a popular practice in modern networked servers. Besides, this approach requires program modification to use an ECEM library and recompilation, which makes it less viable in practical servers. Our TPF is designed to support direct copying from a disk subsystem to a TCP protocol stack and to require no program modification.

The dynamic buffer allocation method [9] addresses how to allocate an optimally sized send buffer for an individual TCP connection that enables the TCP connection to use the maximum available bandwidth. The size is determined by considering the bandwidth-delay product, congestion window, RTT, and *ReadTime* (data fetching time from a disk) of a given connection. It minimizes the frequency of data misses in a TCP connection by dynamically increasing or decreasing the size of the send buffer. In this approach, *ReadTime* plays a key role in deciding how much data needs to be loaded into the buffer from a disk. Basically, this approach tries to fetch enough data to fill the send buffer at every RTT, which will severely damage overall disk throughput due to frequent and fragmented disk access, which was not identified in their simulation-based study. Most modern operating systems are equipped with a data prefetching mechanism for maximizing disk throughput, minimizing disk access latency, and overlapping computation and disk access (asynchronous disk access). Our TPF is based on a prefetching mechanism and optimizes the amount of data to be prefetched by considering the characteristics of a given TCP connection as well as a disk subsystem. As a result, we believe that TPF and this method (dynamic buffer allocation excluding the *ReadTime* part) complement each other, producing further performance improvement.

2.2 Previous Research on Multimedia System

This research group proposed many techniques for the improvement of streaming data transfer from servers to clients and guaranteeing stream rates [20], [21], [22], [23]. In the proposed techniques, buffer management and prefetching mechanisms for streaming data have been the essential components of the servers. A prefetching method is used to allocate data ahead of the actual data transfer, using read-ahead techniques which are usually customized for the streaming characteristics. There are two kinds of read-ahead techniques: fixed size (FS) [20] and fixed duration (FD) [21]. In FS, a disk subsystem prefetches the fixed amount of data. It is simple, but inefficient if there are multiple, different rate streams in the servers. In FD, the amount of prefteched data is proportional to the stream rate. Our work, TDAP, is based on the FD.

In [22], the size of the read-ahead buffers changes dynamically according to the requirements of each stream media. At initialization, a fixed amount of memory is allocated to all connections and then the amount is adjusted with respect to the corresponding stream rate over time. Using CTL (Constant Time Length)-based models, the optimal number of buffers can be allocated to each stream. When it comes to implementation, CTL requires very restricted meta information that is hardly provided by modern operating systems.

Recently, [23] was proposed to guarantee the bandwidth of stream data and the fast response of text data in an integrated multimedia server. It is based on FD and tackles the limitation of CTL by dynamically sensing characteristics of each stream. It senses the data consumption rate of a stream server and monitors the disk I/O time of its request

dynamically. With the collected information, it adjusts the read-ahead size and the size of data buffer for each stream.

Our newly proposed prefetching mechanism has a few strengths over the two previously mentioned approaches. When it comes to data transferring over TCP, the send buffers of TCP connections play a crucial role in reflecting the dynamic characteristics of TCP connections. Previous studies have not taken the buffering effect in the send buffers caused by high-latency and the TCP flow control mechanism into account, but TPF takes full advantage of the effect. Also, our direct mapping TCP sequence space into a file prefetching space provides more fine control than previous approaches do. Furthermore, our approach is enhanced with a new disk-scheduling mechanism which copes with abrupt changes in TCP connections by either delaying or cancelling queued disk requests if necessary.

3 Design of the TCP-Plugged File System

One of the most common, fundamental operations in Internet server systems is to deliver data managed by a file system to clients via TCP connections. In order to improve the operation, we have investigated the server systems in the context of TCP-based data transfer, yielding two critical questions: 1) How will TCP mechanisms such as congestion control and flow control affect the internal operations of a file system such as file prefetching, disk buffer management, and disk scheduling and 2) what will the most efficient reaction of a file system be when a TCP protocol stack receives ACK (acknowledge), RST (reset), and ECN (explicit congestion notification) [8] segments. By deliberating on the questions, we found three operations to be improved in a file system when it was used as a data provider to a TCP protocol stack, which are summarized as follows:

- The routines for sending TCP segments and receiving TCP ACK segments are executed in different contexts. Considering the fact that TCP ACK segment receiving routines are invoked approximately half as frequently as TCP segment sending routines due to the nature of TCP, the context switching problem should be taken care of. Moreover, the limited size of the send buffer of a TCP connection worsens frequent context switching among processes.
- The fixed amount of data is prefetched, irrespective of the dynamically changing status of each TCP connection.
- Once issued, disk requests cannot be withdrawn even if the will-be-loaded data is highly unlikely to be used due to the abrupt change in the corresponding TCP connections.

In an attempt to improve the above three operations, we have designed a new file system, or TPF, which is especially optimized for the components and mechanisms of a TCP protocol stack. TPF has three novel mechanisms: IDSR, TDAP, and EDRC. In the following sections, each mechanism will be elaborated.

3.1 Integrated Data Sending Routine (IDSR)

IDSR is targeted to minimizing the frequent sleeping/waking up of a server process while processing TCP data and ACK segments by newly providing integrated routines for sending data and receiving ACK in the same context. In order to integrate the routines without breaking any semantic of a TCP protocol stack and the integrity of operating systems, we need to understand the TCP protocol stack and the prioritized multiple-context concept of an operating system in depth. In the following paragraphs, both issues will be covered in detail and then IDSR will be explained.

The key component of a network subsystem is a TCP/IP protocol stack which is comprised of

1. TCP-packet processing routines including checksum calculation, TCP packet header processing, TCP segmentation, etc.;
2. TCP packet flow control routines, including a TCP sliding window control, congestion control, flow control, and packet retransmission;
3. memory buffer management, including a send and receive buffer; and
4. four timers for maintaining established TCP connections.

In this paper, we only focus on data delivery over TCP that is steered by a TCP/IP protocol stack, as stated in the previous section. By the TCP's nature, all TCP packets sent to clients must be ACKed by the clients and packet transferring is accordingly controlled by the three TCP control mechanisms mentioned above [24]. The packet transfer rate and latency of each TCP connection vary according to the packet consumption rate of a client-side application and the dynamic status of the network path established between the server and the client. From the perspective of a TCP/IP protocol stack, a disk subsystem is a data provider whose role is to load an adequate amount of data from a disk to memory buffers by considering a data sending rate of a corresponding TCP connection.

The two subsystems, that is, a network subsystem and a disk subsystem, are mostly realized within an operating system which has adopted an interrupt-based I/O request processing scheme and multiple control contexts, such as a process context, soft interrupt context, and H/W interrupt context [25] with different priorities. The priorities of the contexts are ordered like "H/W interrupt context $>$ soft interrupt context $>$ process context." Therefore, on the occurrence of an H/W interrupt, a user process or a system daemon, running in either a process context or a soft interrupt context, should relinquish CPU for an H/W interrupt handler running in an H/W interrupt context. In this type of operating system, an Internet server application running in the lowest priority, a process context, needs to collaborate with H/W interrupt handlers and soft interrupt handlers running in an H/W and soft interrupt context, respectively, in order to transfer data from a disk into a network interface card (NIC). Since processing I/O requests in the different contexts incurs context switching and scheduling, data transferring between the two subsystems is not contiguous intrinsically in terms of context switching. In the following paragraph, we explain how two subsystems are implemented in multiple-context operating systems such as Linux, Unix, BSD, etc.

The execution of routines for sending TCP data and receiving TCP ACK in a multiple-context operating system occurs in the order shown in Fig. 2a.

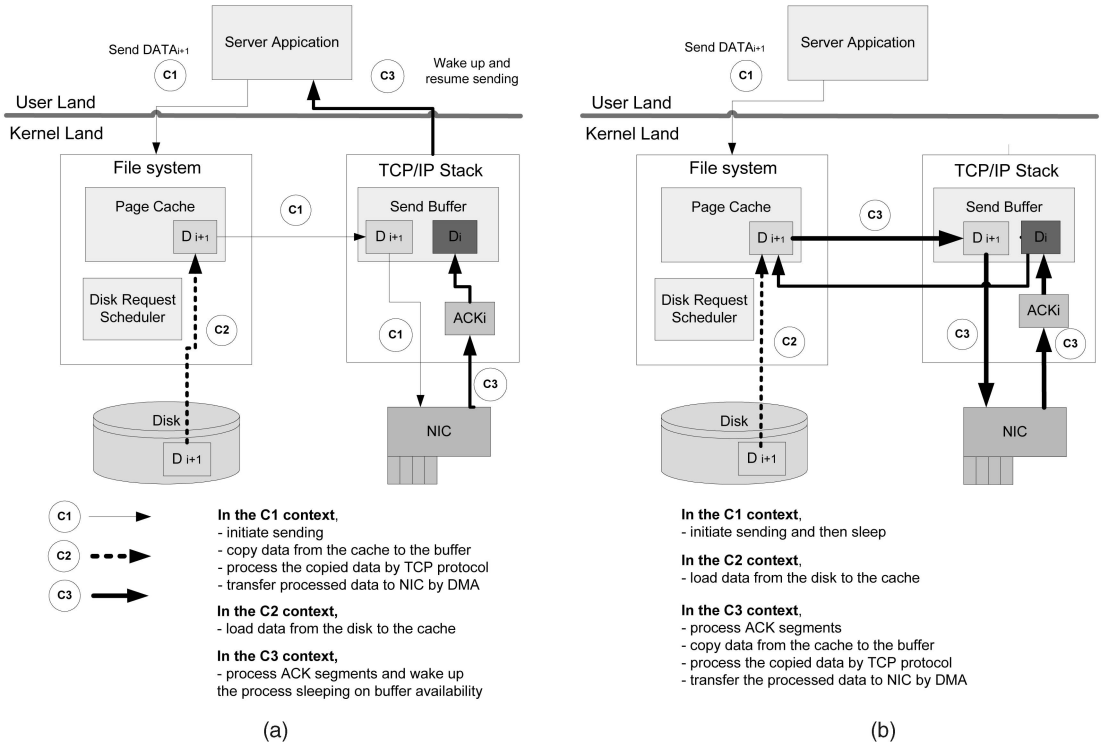

Fig. 2. Data transferring through multiple contexts. (a) Conventional data transfer. (b) IDSR data transfer.

1. A server process reads data from a disk into a data buffer (a page cache in Linux) in an H/W interrupt context, C2. Depending on the server implementation, the process may copy the data into buffers in a user space.
2. In the C1 context, the loaded data is copied into the send buffer until the buffer has no more available space in it. Then, the current process is scheduled out, relinquishing the CPU to one of the other running processes and waiting for available space in the buffer. It occurs very frequently due to the limited size of the send buffer.
3. The data in the send buffer is processed by a TCP protocol stack in the C1 context and then it is copied to NIC by DMA in the H/W context.
4. As a response, corresponding ACK segments are transmitted and they are processed through a TCP protocol stack in the C3 context.
5. Finally, in the same context, it relieves the ACKed data in the send buffer, checking to see if there is a waiting process for free space in the buffer. If there is one, it wakes up the process.

As explained so far, whenever data is sent to a client, the data should be ACKed [24], introducing additional context switching for processing incoming ACK segments in a server-side operating system. Worse, an ACK arrival rate is approximately half that of a TCP data sending rate [24]. Thus, it is obvious that a current operating system suffers from severe context switching overheads, being exacerbated by the context switching overheads between Internet server processes due to the limited size of a send buffer, as described in the previous section. Therefore, in an effort to reduce the overall amount of context switching, we developed IDSR, which integrates the data sending routines with the ACK segment receiving routines so as to be executed in the same context.

A set of routines executed in the different contexts, C1 and C3 of Fig. 2a, are integrated into the one context C3 of Fig. 2b, which is represented with the thick black line. In IDSR, the data sending routines are executed in the same context just after the execution of the ACK receiving routines is completed without context switching. In other words, the ACK processing routines executed in a soft interrupt context continuously send the next data whose amount is identical to the amount of data just acknowledged by the incoming ACK segments in the current context. By virtue of the *self-clocking*[1] behavior of TCP [24], IDSR becomes able to continue to pump data to clients without any delay. As a result, the sending rate of TCP data segments can be synchronized with the incoming rate of ACK segments, which reflects the status of an established network path to clients.

3.1.1 Implementation Detail in Linux

This section illustrates how to implement IDSR in Linux 2.4. The detail of data delivery over TCP at a function level is pictured in Fig. 3.

Initially, a server process examines whether a target data is in a page cache. If not, it will queue a disk read request and then sleep. When the data from the disk is ready to transfer, *device_intr_handler()*, or an H/W interrupt handler, puts the data into the page cache by DMA. After that, *end_buffer_io_async()* is executed in a soft interrupt context. It manipulates a few flags for buffer management and synchronization and then wakes up the sleeping process.

1. In TCP networks, when transferring data between a sender and a receiver, the spacing of incoming ACK segments from the receiver is almost identical to the spacing of the outgoing data segments in the sender.

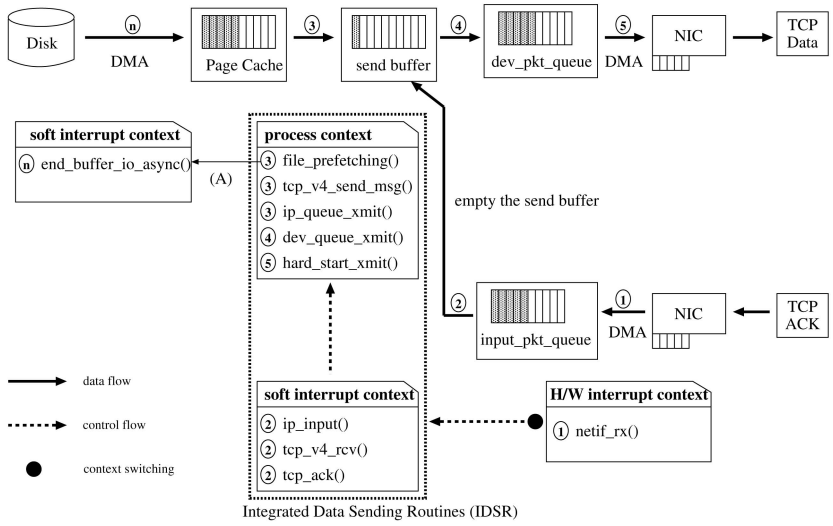

Fig. 3. IDSR implementation example in Linux 2.4.

The waken process copies the data into the send buffer of a corresponding TCP connection in a process context. After processing the data through a TCP/IP protocol stack in the same context, the data is put into *dev_pkt_queue*. Finally, *hard_start_xmit()* initiates DMA to copy the data into NIC in a soft interrupt context.

In response to the data, a client generates an ACK segment every two full TCP data segments. On the reception of the ACK segments, *netif_rx()* is executed in an H/W interrupt context, putting the segments into the *input_pkt_queue*. In the soft interrupt context, the segments are processed through the TCP/IP protocol stack. Last, *tcp_ack()* is called, emptying the ACKed data in the send buffer and waking up the sleeping process for a free buffer space, if any.

In IDSR implementation in Linux, a set of routines executed in a process and soft interrupt context in Fig. 3 is integrated into one soft interrupt context, represented with the dashed box. In the dashed box, the data sending routines are executed in the same soft interrupt context just after the execution of the ACK receiving routines without context switching.

3.1.2 Scheduling Fairness Issue in Linux

As explained so far, IDSR integrates a set of routines for sending data in a process context into the routines (soft interrupt handler in Linux) for processing ACK segments in a soft interrupt context. The soft interrupt handler always preempts the execution of user processes as long as the handlers are not turned off. From the standpoint of scheduling and priority, it needs to be addressed that TCP data sending routines are quite heavy due to checksumming, data copying, and TCP segmentation and IDSR makes those heavy routines run in a higher priority context than a process context. Obviously, it will lengthen the overall execution of a soft interrupt handler, leading other concurrently running processes in a process context to starvation. To mitigate this potential starvation problem, we introduce a low-priority kernel daemon which deals with TCP ACK and data segments in the same priority level as the process context's. In our implementation, we extended the *softirqd* [25] which is already included in Linux in order to prevent the interrupt live-lock problem of network packet processing from degrading overall system throughput.

3.2 TCP Dynamics Aware Prefetching (TDAP)

To transfer data from disks to a client, first it needs to be loaded into a memory buffer (a page cache in Linux). On loading data, in a modern operating system, a prefetching mechanism plays a key role in deciding how much data needs to be loaded in advance for the performance improvement of disk access. Conventional prefetching mechanisms are rudimentary because they prefetch a statically predetermined amount of data into a memory buffer only if the data is accessed sequentially without considering the dynamically changing bandwidth and latency of corresponding TCP connections. For example, in Fig. 4, the TCP connection for 1 Mbps video streaming is handled in the same manner as the TCP connection for 64 kbps audio streaming; 31 pages are prefetched for each connection in Linux. To send 31 pages, the TCP connection for the video streaming takes 1 second and the other one for the audio streaming takes 15.5 seconds. The last page of the prefetched data for the audio streaming needs to be retained in the page cache for 15.5 seconds. Obviously, too much data is loaded for the audio streaming, causing inefficient disk access and memory buffer utilization. What is worse, in data-intensive servers, prefetched data are prematurely evicted from the cache even before being accessed, causing additional and unnecessary disk I/O [26]. To alleviate this problem, we devised a novel prefetching mechanism, TDAP, which is adaptive to dynamic changes in the bandwidth and latency of TCP connections. Prior to illustrating how TDAP works, we need to explain how the states of TCP connections are maintained in a TCP protocol stack and how the bandwidth and latency of TCP connections affects the dynamic profile of a send buffer. Both issues will be explained in the following paragraphs.

Dynamic TCP states can be observed by reading the variables of a TCP sequence space at a given moment that is maintained by a TCP protocol stack [24].

- *snd_nxt*: The sequence number of the next data byte to be sent.

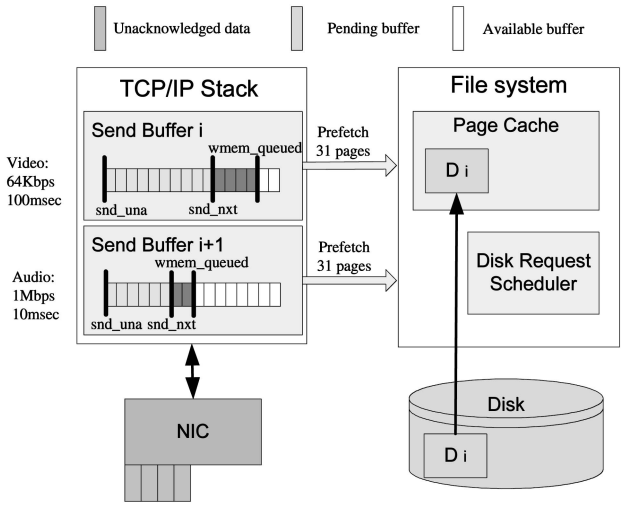

Fig. 4. Conventional prefetching mechanism (Linux).

- *snd_una*: The sequence number of the first unacknowledged data byte. It is equivalent to the sequence number of the greatest ACK received.

The left and right edge of the sliding window in Fig. 5 are *snd_una* and

$$snd\_una + min(advertised\ window, usable\ send\ buffer),$$

respectively. When a receiver acknowledges data, the sliding window moves to the right along the TCP sequence space. The relative motion of the two edges of the window either increases or decreases its available buffer space for outgoing data. The speed of the window movement is mainly determined by the bandwidth of the TCP connection. For our new prefetching mechanism, we defined three new variables, UAB, PB, and AB, in terms of the absolute size of a buffer space in a send buffer.

- UAB (Un-Acknowledged Buffer): A portion of a send buffer that is already sent but yet-to-be-acknowledged so that the portion cannot be released yet for retransmission. It is identical to $snu\_nxt - snd\_una$.
- PB (Pending Buffer): A portion of a send buffer that is already written into the send buffer by a user process (server process) but not sent to a client due to the TCP flow control mechanism.
- AB (Available Buffer): A portion of a send buffer that is immediately available for new data.

To see how the bandwidth and latency of a TCP connection affect the size of PB, we set an experimental environment that was comprised of a client, a server, and a WAN router [27]. The router emulates a WAN environment by providing a set of configurable parameters for each connection such as latency, bandwidth, packet drop ratio, etc. We opened a TCP connection between the client and the server via the router and varied the bandwidth and latency of the connection. When the connection became stable, the size of PB was probed. Fig. 6 illustrates that, as bandwidth and latency increase, the sizes of PBs of 200 kbps, 1 Mbps, 5 Mbps, and 10 Mbps connections also increase up to 1, 38, 48, and 48 segments, respectively, whose upper bound is limited by the size of the send buffer (128 Kbytes, 90 segments) and maximally allowable number of packets-in-flight. When the number of packets-in-flight reaches a certain value (in Linux, approximately half of the send buffer size), the actual packet transmission is suspended until some of the sent packets are acknowledged. Therefore, from the view point of a disk subsystem, PB is an important factor for prefetching data from disks because PB can be considered as a margin prior to the shortage of data in a page cache. Thus, we decide to take full advantage of PB in the design of our new prefetching mechanism, while conventional prefetching mechanisms have not considered the amount of PB in calculating the amount of data to be prefetched.

In order to devise a TCP-friendly prefetching algorithm from scratch, TDAP is based on two of the previously mentioned factors: bandwidth and PB (by latency). New state variables required for preserving the status of file prefetching are defined in TDAP for an individual TCP connection: LP, PP, and NPP in addition to UAB, PB, and AB. These variables have the following meanings:

- LP: The greatest index of pages that are loaded into a page cache.

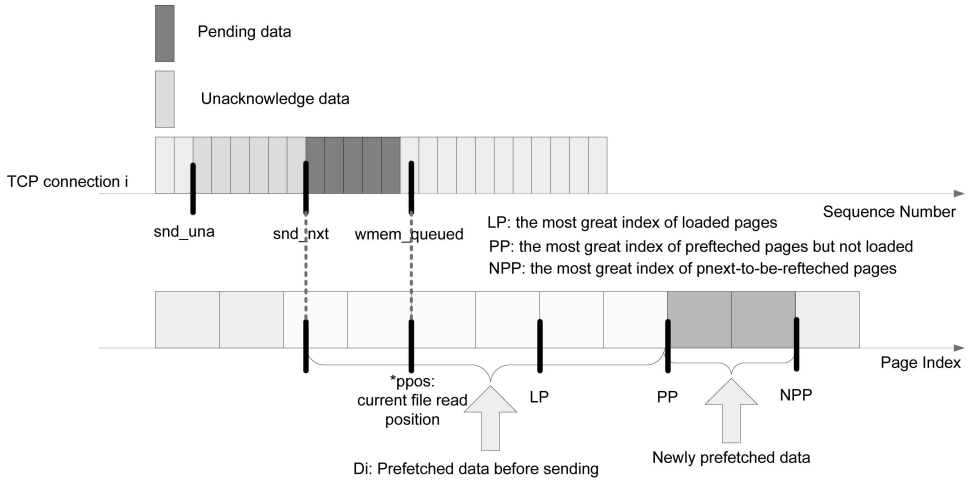

Fig. 5. A data prefetching based on network bandwidth and latency.

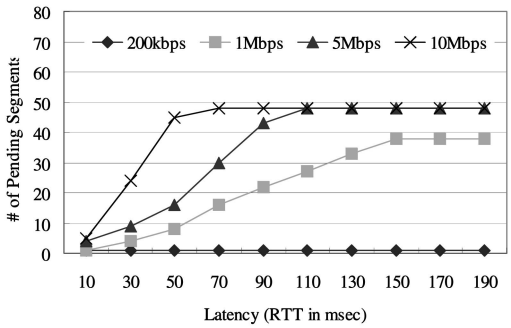

Fig. 6. The size of the Pending Buffer (PB) with varying bandwidth and latency.

- PP: The greatest index of pages that a prefetch has requested, but that the file system has not yet completed processing.
- NPP: The greatest index of the next-to-be-prefetched pages

Whenever TDAP is activated, the following variables are calculated to determine the optimal amount of data to be prefetched:

- $D1$: The set of data that has already been put into the send buffer, but has not been sent to a client due to the TCP flow control mechanism.

$$D1 = PB = *ppos - snd\_nxt. \tag{1}$$

- $D2$: The set of data that resides in a page cache, waiting to be moved to a send buffer.

$$D2 = LP - *ppos. \tag{2}$$

- $D3$: The set of data that may not reside in the page cache, but for which the corresponding disk requests are already generated and enqueued for prefetching.

$$D3 = PP - LP. \tag{3}$$

- $D4$: The set of data which has not been prefetched yet, but will be prefetched in the next prefetching trial.

$$D4 = NPP - PP. \tag{4}$$

$D1$ is identical to PB, as shown in Fig. 5.

To estimate the amount of data to be prefetched, or $D4$, we consider the current network characteristics that are reflected in $D1$ and the moving speed of *snd_una*, or the TCP ACK arrival rate with which we can measure the approximate bandwidth of a given TCP connection. The amount of data which resides in a physical memory, or not in the memory but prefetched, is $D1 + D2 + D3$. It takes

$$T_n = \frac{D1_n + D2_n + D3_n}{B_n} \tag{5}$$

for the data to be sent to clients where n denotes the nth connection and B_n denotes bandwidth of the nth connection. In other words, T_n is the time margin before data miss occurs in the page cache in data transmission of the nth connection. TDAP needs to keep T_n close to T_D to sustain the outgoing bandwidth of the nth connection without data miss, where T_D is the ideal data buffering time and is calculated as follows:

$$T_D = 2 \cdot \frac{B_D}{B_{net}}, \tag{6}$$

where B_D is a measured disk bandwidth against random disk I/O access and B_{net} is $\sum_{i=1}^{N} B_i$ or the overall network bandwidth at the moment. The 2 coefficient is necessary since, at the moment when $ppos$ is moving into a current prefetching window, queuing disk requests for the next prefetching window is heuristically optimal.

Whenever the sliding window moves to the right and $T_n \leq \frac{T_D}{2}$, TDAP will prefetch D4 KBytes to keep T_n close to T_D, as follows:

$$D4 = \begin{cases} min(\frac{T_D - T_n}{B_n}, \frac{M_a}{B_n/B_{net}}, H_{max}), & \text{if } \frac{T_D}{2} \geq T_n \\ 0, & \text{if } \frac{T_D}{2} < T_n, \end{cases} \tag{7}$$

where M_a is the amount of memory that is available for prefetching. M_a is obtained by subtracting the amount of memory holding data that is prefetched but not yet accessed from overall memory in the page cache. The second term of (7) is to prevent page thrashing, where prefetched but yet-to-be-accessed pages are prematurely evicted from the cache, causing additional disk I/O. In other words, $\frac{T_D - T_n}{B_{net}}$ is for proportionally sharing disk bandwidth and $\frac{M_a}{B_n/B_{net}}$ is for proportionally sharing available memory among TCP connections. The last term of (7), H_{max}, is the maximum number of pages bounded by underlying devices. For example, in an SCSI disk controller, the maximum number of elements in a scatter-gather list is bounded above by 128. Thus, at most 128 pages in Linux can be included in a single scatter-gather list, which is mapped to a single disk I/O request. For a 516 Kbyte prefetching operation, however, two disk I/O requests, one for 512 Kbytes and the other for the remaining 4 Kbytes, are generated due to the constraint of a scatter-gather list, degrading the performance of disk access. In order to elude this problem, H_{max} limits the maximum size of a single prefetching operation.

3.2.1 TDAP Comparison

For prefetching, Linux defines the following variables [25] for each open file:

- f_raend: Position of the first byte after the read-ahead group and the read-ahead window.
- f_ralen: Length in bytes of the current read-ahead group. The group of data requested in the last read-ahead operation.
- f_rawin: Length in bytes of the current read-ahead window. The group of data requested in the last two read-ahead operations.
- f_ramax: The maximum number of characters to get in the next read-ahead operation.
- $^*ppos$: Byte position of read data at a given moment.

Fig. 7 illustrates the interaction between a TCP sequence space and a Linux prefetching space. Whenever the right edge of a sliding window increases, data is put into the

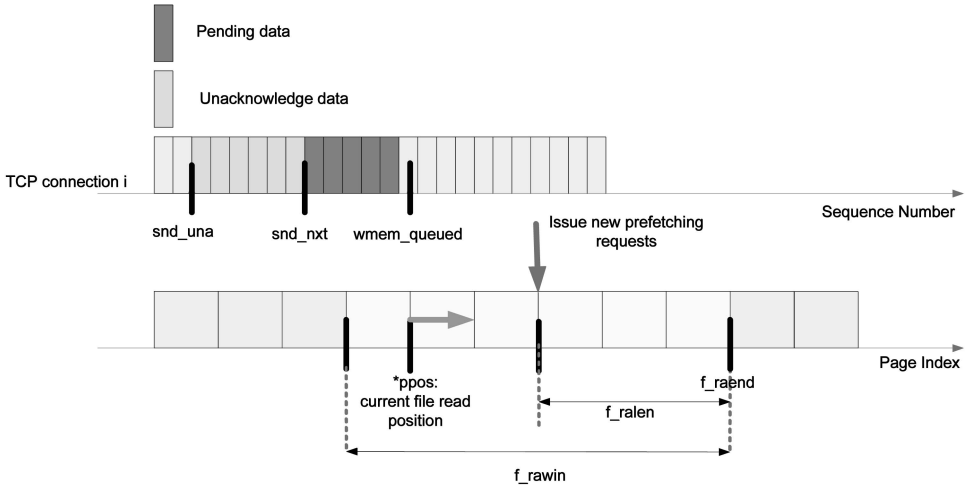

Fig. 7. TCP sequence space and variables for Linux prefetching.

usable window, moving $^*ppos$ to the right. When $^*ppos$ reaches the point, $f\_raend - f\_fralen$, f_rawin, and f_ralen move to the right by f_ramax, 31 pages (4,096 bytes/page) in a steady state. f_ramax is fixed and it is not sensitive to a sliding speed of the window (connection's bandwidth).

Prefetching a fixed amount of data leads to low efficiency of memory-buffer utilization and disk access. For example, TCP connections for both 64 Kbps music streaming and 1 Mbps high-quality video streaming prefetch the same f_ramax amount of data, or 31 pages (124 kbytes) into a page cache. It will take 17.32 seconds and 1 second to send all data in the cache, respectively. In the presence of memory pressure, prefetched data is likely to be evicted from the cache by a Linux memory management daemon, *bdflush*, even before they are sent due to the shortage of memory buffer. It is unlikely, in TDAP, if T_D is 2 seconds, that 1 page and 248 pages will be prefetched, respectively, for both cases and the occupied memory space will be released in 2 seconds, improving the efficiency of memory-buffer utilization and disk access.

3.3 Eager Disk Request Cancellation (EDRC)

In this section, we describe the solution for the most effective reaction of a disk subsystem when a TCP protocol stack receives a duplicated ACK, FIN, RST, and ECN segment.

On the reception of an RST and FIN segment, a disk subsystem releases kernel-data structures, such as TCP send/receive buffers and related metadata structures owned by the corresponding TCP connections, terminating data transmission over the TCP connections immediately. On the other hand, the reception of duplicated ACK and ECN segments activates a TCP congestion control mechanism that cuts down the congestion window by half, also reducing the transmission rate of the corresponding TCP connections by half. In both cases, a conventional disk subsystem will not take any immediate action against the changes. Therefore, it continues to load data whose disk requests are already created and queued to the disk queue, wasting disk bandwidth and memory. The prefetching will stop only after the corresponding file control (generally called as a file descriptor) is explicitly closed by a server application. Prefetched data will remain in the memory until an operating system runs out of memory so as to invoke a buffer replacement daemon [25]. In this design, there is an obvious downside; data which is highly unlikely to be accessed in the near future will be loaded at the expense of disk bandwidth and memory buffer. For example, let us assume that an ECN segment arrives at a NIC, as pictured in Fig. 8a. A TCP protocol will reduce the congestion window size by half, decreasing the transmission rate of the corresponding TCP connection by half. At that moment, disk requests for prefetching in a disk request scheduler will be issued to a disk without reflecting the abrupt changes in the connection. The prefetching will load twice the amount of data to sustain the half-reduced outgoing transmission rate of the connection. If there is insufficient memory and a disk is busy, the prefetching will sacrifice the overall performance of a server system.

In order to get deep insight into a disk subsystem, we profiled a running Apache HTTP server, mainly focusing on a disk request queue and page allocation/reclamation for prefetching. For profiling, we interconnected a server with 1.8 GHz Pentium 4 and 1 G memory, and clients by a Gigabit ethernet. The clients established HTTP connections to the server and then downloaded movie files from the server as fast as possible. Fig. 9a shows the average number of queued disk requests (AQDR) in the disk request queue and the number of the queued disk requests per TCP connection. As the number of connections increased, AQDR also increased up to the point of 150 connections. After that, the system is saturated due to heavy disk load. When there are 200 connections, AQDR appears at around 250, which means 250 issued requests are waiting for disk service at a given moment. As the number of client connections increased AQDR/connection decreased from 2.3 to 0.8. Actually, the number of disk requests generated during prefetching outnumbered 250, but the elevator scheduling algorithm [25] sorts and merges incoming requests to boost disk throughput, yielding relatively small AQDR/connection. Fig. 9b shows the number of pages (4 Kbytes) allocated for prefetching and the page reclamation time that shows how long the prefetched pages stay in the page cache prior to reclamation. Each disk request reserves 300—70 pages for will-be-loaded data with respect to the number of connections. After prefetched pages stay in the page cache for

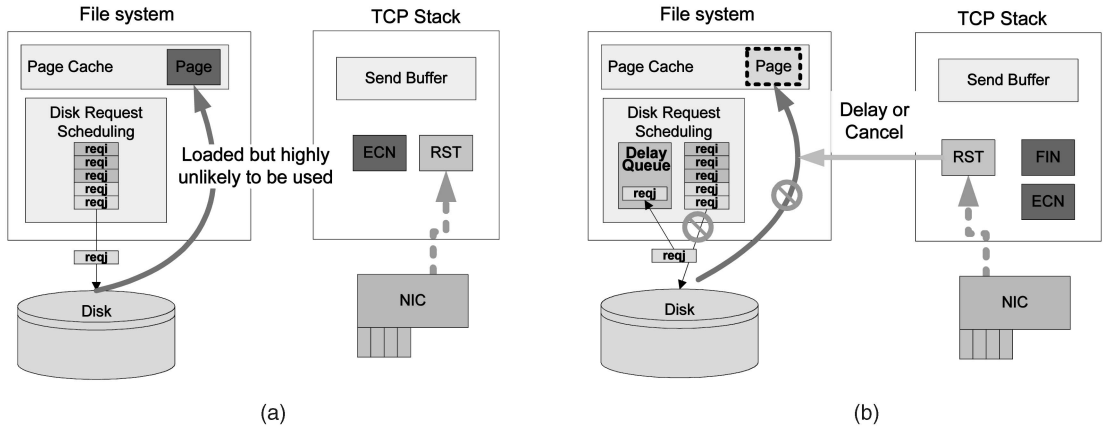

Fig. 8. Eager disk request cancellation. (a) Conventional disk scheduling. (b) EDRC-based disk scheduling.

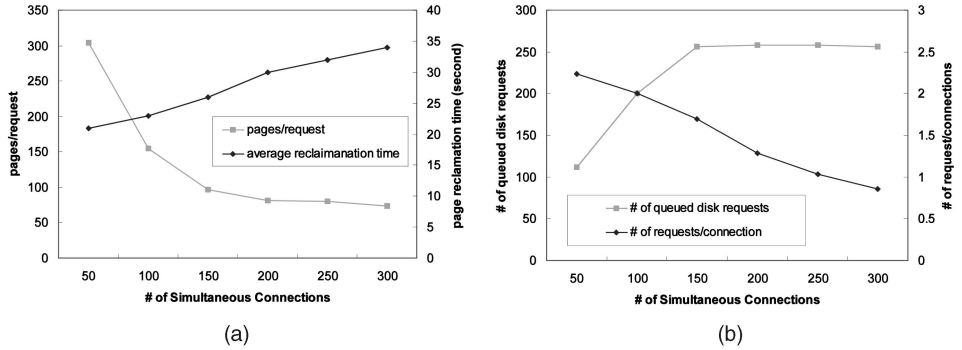

Fig. 9. Disk subsystem profiling result. (a) Disk queue profiling. (b) Page allocation/reclamation profiling.

21-34 seconds, they are freed for other data. Based on the profiling result, we can derive one fact that, even when a TCP connection is aborted by RST or FIN, the current Linux will continue to prefetch 1.2 Mbytes to 0.29 Mbytes and the loaded pages will stay for 21 to 34 seconds in the cache, wasting the memory pages and disk bandwidth.

We devised an eager disk request cancellation mechanism (EDRC) which scans queued disk requests in a disk queue, either moving selected requests into a delay queue or cancelling them immediately according to the type of received segments in the corresponding connections. Fig. 8b illustrates EDRC with an additional delay queue. On the reception of an RST segment, EDRC scans the disk requests queued in a disk request queue. If there is any request that is germane to the TCP connection, it eagerly removes the requests out of the queue, freeing the memory allocated to the requests for DMA. On the reception of both ECN and duplicated ACKs, related disk requests are moved to the delay queue temporarily or the deadline of the disk requests is increased by $\frac{T_D}{2}$, depending on disk load. If a disk is underutilized, EDRC will not take any action against the reception of duplicated ACKs and ECN. In brief, EDRC enhances the efficiency of resources, such as disk-bandwidth and memory utilization, by reclaiming the resources allocated to TCP connections when the connections require less resources.

3.3.1 Interaction with a Current Linux Disk Scheduler

The ultimate goal of a Linux disk scheduler is to maximize throughput by minimizing a disk-head seek time. In addition, starvation and fairness also need to be considered. The organization of the scheduler is illustrated in Fig. 10. The block layer generates disk I/O requests into the I/O scheduler. Then, the scheduler sorts and merges the requests according to the scheduling policy (Linux 2.4 adopts an elevator algorithm). Finally, the requests are queued to the device queue and a disk interrupt handler services the requests one by one.

In our Linux implementation, EDRC scans the device queue and moves selected disk requests to a delay queue. After a specified delay time passes, it puts the requests back to the device queue. Since EDRC does not modify any current I/O scheduling policy and its implementation, it is complementary to the existing I/O scheduler. By adding minimal extra codes for a scanning operation and a delay queue, EDRC improves an effective disk throughput and memory utilization, making the disk scheduler responsive to abrupt network changes such as packet loss (duplicated ACKs), congestion notification (ECN), and RST.

4 Performance Evaluation

In this section, we present performance results obtained with our prototype implementation of TPF in Linux 2.4. It was developed as a kernel module with an internal code patch to *sendfile()* without any modification to the current syntax of *sendfile()*. Hence, neither application modification nor recompilation is required to utilize TPF as long as applications use *sendfile()* as a data transfer primitive. An

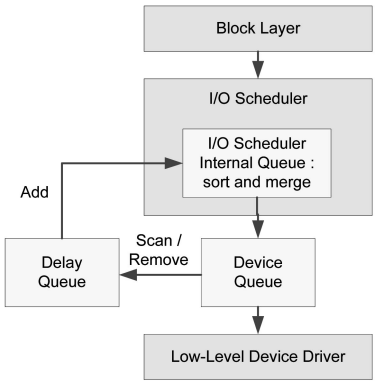

Fig. 10. Linux disk I/O scheduler.

Apache HTTP server [28] was configured to use *sendfile()* in our experiments.

A server running Apache has a 1.8 GHz Pentium 4 processor, 1 GB main memory, and Seagate SCSI ST-373307LC disk. Four client machines with the same H/W specification to the server were used to drive the server with a variety of input patterns. Especially, for WAN environment emulation, we used a NIST WAN router [27]. All machines were interconnected by Gigabit Ethernet. TPF is targeted for performance improvement of servers which handle a number of concurrent TCP connections. Therefore, the performance metric to be measured in the following experiments is overall server transmission rate under various workload patterns.

4.1 Various Disk Load and Memory Cache Hit Rate of IDSR

The first experiment measures the performance of IDSR by varying the locality of downloaded data. Increases in the locality decrease the disk load in the server so that we can observe how IDSR operates with respect to changing disk load and memory requirements. At the time when an experiment is initiated, all clients establish TCP connections, send HTTP requests to Apache, and then download the target movie files (384 MBytes) as fast as possible. At 20 locality, 20 percent of clients download the same movie file.

The overall server bandwidth with increasing the number of simultaneous client connections and the locality of the downloaded data is illustrated in Fig. 11. When the locality is at 0 percent, the bandwidth of the conventional Linux and IDSR decreases from 312 Mbps to 152 Mbps and from 324 Mbps to 157 Mbps, respectively, as the number of simultaneous connections increases. Our observation into the degradation demonstrates that more than 70 percent of CPU time is idle when there are 10 connections and the CPU idle time is increased as the number of simultaneous connections is increased by up to 84 percent, as shown in Fig. 11c. Therefore, it is ensured that the performance inhibitor of the server is the disk bottleneck. Even though both cases are hit by significant performance degradation, IDSR shows 5-12 Mbps (4-5 percent) bandwidth improvement as compared to the conventional Linux. Fig. 11b confirms that around a 10 percent decrease in the number of context switching is achieved by IDSR.

As the locality is increased, the bandwidths of both cases are improved. This is because the disk load is decreased due to the proportional increase in the memory cache hit rate. When the locality is 100 percent, 612 Mbps, the peak performance of IDSR is obtained. The performance improvement ranges from 49 Mbps to 68 Mbps (11-13 percent) as compared to Linux. The CPU utilization reaches to 0.7-0.91. One interesting observation at 100 percent locality is that the peak performance degradation with increasing connections is approximately 12 percent, a significantly reduced result, compared to 50 percent of 0 percent locality. Differently, the number of context switching soars over 50,000, yielding the 12 percent performance degradation. IDSR always achieves the better performance in overall ranges, showing around 5 percent to 13 percent according to the locality.

In order to get deep insights into the running Linux kernel with IDSR, we performed a system-level profiling by using oprofile [29], which is capable of collecting execution profiles of all execution entities of the Linux OS including user processes (thread-level), libraries, and the kernel itself. Fig. 12 shows the profiling results with 200 TCP connections and 100 percent memory cache hit rate. The symbols in boldface represent different entities: Apache (httpd), network device driver (ns83820), and the Linux kernel (vmlinux). The most interesting routines are listed selectively in Fig. 12. Around

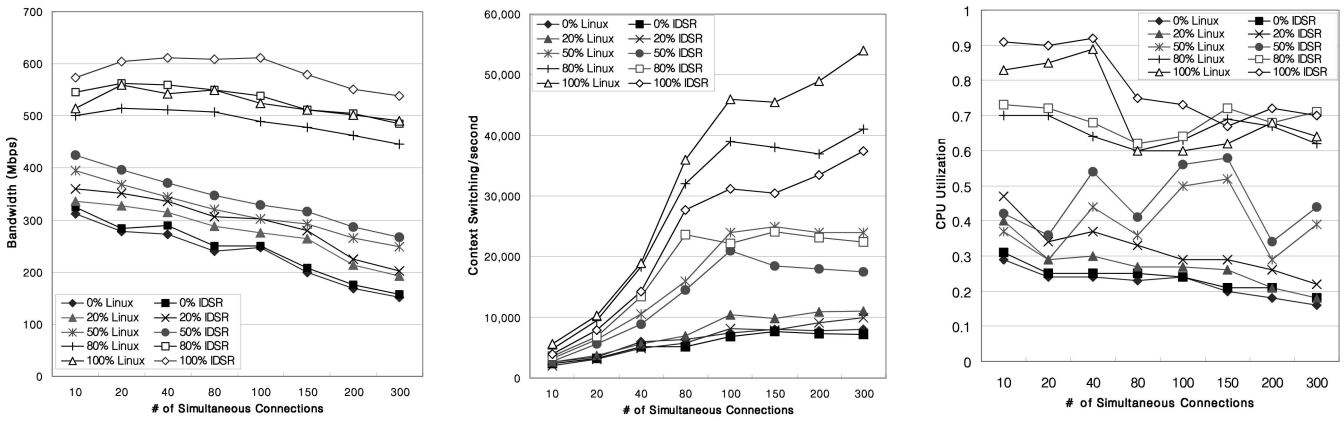

Fig. 11. Network transmission rate and context switching overhead. (a) Transmission rate. (b) Context switching. (c) CPU utilization.

		Linux (%)	IDSR (%)
vmlinux		**86.10**	**90.20**
	memory/scheduling	**33.54**	**35.24**
	__copy_user_xxx	**14.654**	**15.244**
	__kmalloc	**3.861**	**4.001**
	kmem_cache_alloc	**3.304**	**3.420**
	wait_for_tcp_memory	**0.045**	**0.001**
	__wake_up	**0.770**	**0.191**
	TCP	**20.68**	**21.68**
	tcp_sendmsg	**3.555**	**3.601**
	tcp_write_xmit	**3.121**	**3.229**
	tcp_ack	**2.352**	**2.380**
	tcp_v4_rcv	**2.154**	**2.299**
	IP	**14.67**	**15.02**
	ip_queue_xmit	**4.081**	**4.341**
	dev_queue_xmit	**2.477**	**2.526**
	ip_rcv	**1.540**	**1.647**
	misc	**17.21**	**18.11**
httpd		**0.27**	**0.29**
oprofiled		**3.42**	**3.62**
ns83820		**8.71**	**8.90**

Fig. 12. Execution profile (100 percent memory cache hit rate).

33 percent of CPU cycles are consumed for data copying, memory allocation, and scheduling, while TCP and IP take 20 percent and 15 percent, respectively, for protocol processing. On the other hand, Apache itself takes only 0.27 percent of the CPU cycles. This is because HTTP protocol processing in a user mode is minimal in our experiments since a single TCP connection generates only one HTTP request to download a movie file, so, overall, we have a total of 200 HTTP requests during profiling. With IDSR, we found two outstanding routines: *wait_for_tcp_memory* and *__wake_up*. The names of functions are self-explanatory. *wait_for_tcp_memory* at 0.045 percent, 58 C-code lines and 0.77 percent of *__wake_up*, eight C-code lines are reduced to 0.001 percent and 0.191 percent, respectively. They are significantly reduced by IDSR, considering the small number of C-code lines. Therefore, our profiling result confirms that IDSR minimizes the frequency of scheduling.

4.2 Fairness for Low Priority Process of IDSR

In order to investigate the process scheduling fairness problem caused by IDSR, we run an HTTP server together with a user process that does matrix multiplications (MM) repeatedly. The MM process runs entirely in a process context, the lowest priority in Linux, making it vulnerable to higher priority routines. Fig. 13 illustrates the throughput of MM under three cases: intact Linux, IDSR running in a soft interrupt context, and IDSR with *sofirqd*, a low-priority daemon. In the experiment, to keep the CPU busy, the locality was set at 0 percent and, to impose the same amount of load to the server system, clients were set to download files at a fixed speed ranging from 100 Mbps to 400 Mbps. As the downloading speed is increased, the MM throughput is decreased in all cases. As expected, IDSR shows the worst throughput. The throughput degradation of MM by IDSR is approximately 21-40 percent compared to Linux. The degradation was compensated for by introducing *softirqd*, showing almost the same throughput to Linux. This result convinced us that the fairness issue in IDSR can be minimized by using a low-priority kernel daemon.

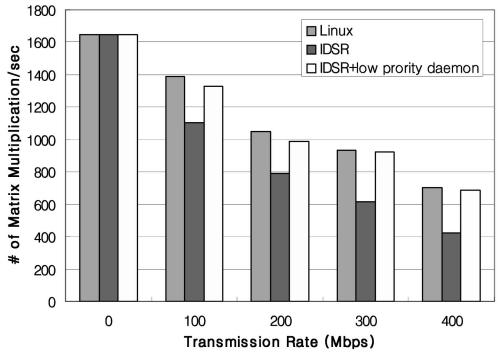

Fig. 13. IDSR fairness measurement.

4.3 Bandwidth Effect of TDAP

In the previous experiment, all clients download movie files from the server as fast as possible, evenly sharing available server bandwidth so that all connections have a similar bandwidth. As a result, the same amount of data is prefetched for all connections, making TDAP have no impact on server performance. So, in order to highlight how TDAP works, we divided connections into two groups; one is 1 Mbps connections and the other is 200 Kbps. One Mbps represents high-quality video stream and 200 Kbps represents audio streaming on the Internet. The 1 Mbps clients consume received data at 1 Mbps. In this experiment, the number of 1 Mbps connections is fixed at 180 and the number of 200 Kbps connections is increased from 100 to 300. The average bandwidth of each group is illustrated in Fig. 14. Three bars represent the average transmission rate of the conventional Linux, IDSR, and IDSR + TDAP from left to right.

Fig. 14a shows the measured average bandwidth of the 1 Mbps group. In Linux, as the number of 200 Kbps connections is increased, the bandwidth drops steeply from 675 Kbps to 440 Kbps. Comparatively, IDSR shows much improved bandwidth ranging from 718 Kbps to 580 Kbps. Last, IDSR enhanced with TDAP outperforms the other two methods. Its bandwidth ranges from 816 Kbps to 680 Kbps, achieving 11-15 percent and 16-36 percent improved bandwidth compared to that of IDSR only and Linux, respectively. On the other hand, the bandwidth degradation ratio of 200 Kbps connections ranges from 15 percent to 30 percent, much smaller than that of 1 Mbps, 32-56 percent.

In the experiment, T_D ranges from 2.76 to 3.08 seconds with respect to the number of 200 Kbps connections. Therefore, around 184 Kbytes and 24 Kbytes are loaded on prefetching for 1 Mbps and 200 Kbps connections, respectively, while 128 Kbytes are loaded for 1 Mbps and 200 Kbps connections in the other two cases. For a 1 Mbps connection, TDAP generates a disk request to load 184 Kbytes, which is 56 Kbytes larger than for other cases. Also, disk request generation rates are 0.69 requests/sec for TDAP and 1.02 requests/sec for the other cases. When it comes to disk access, a rule of thumb is to load as much data as possible in a single request in order to achieve high disk throughput. One hundred eighty-four Kbytes is determined by TDAP so as to increase overall disk throughput without sacrificing the transmission rates of 200 Kbps connections. We also measured page thrashing, wherein prefetched pages are prematurely evicted from a page cache without being accessed. In the IDSR case,

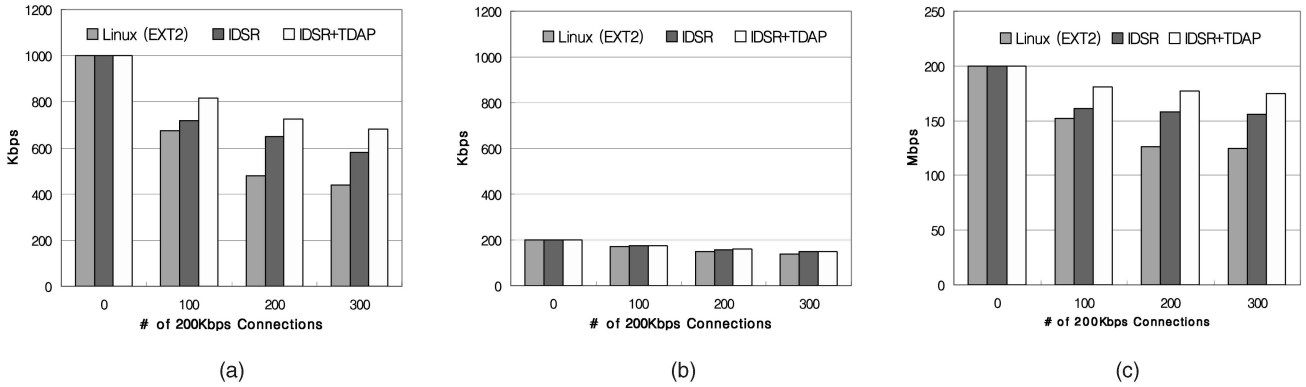

Fig. 14. Transmission rates in the combination of low and high bandwidth connections. (a) Transmission rate of 1 Mbps connections. (b) Transmission rate of 200 kbps connections. (c) Overall server transmission rate.

approximately 160 pages/sec was observed, while TDAP had no page thrashing because of the second term of (7). Based on these observations, it becomes clear that, without sacrificing the performance of connections with a low transmission rate, we achieve more bandwidth for high transmission rate by preventing page thrashing and prioritizing disk accesses according to the transmission rate of a TCP connection.

Overall server transmission rates of the three methods are illustrated in Fig. 14c. The rate of Linux is degraded by 23 percent to 37 percent as the number of 200 Kbps connections is increased. The degradation is partly relieved by IDSR, which shows a 20 percent rate drop. It is further improved by TDAP, topping an approximately 12 percent rate drop.

4.4 High Latency Effect (Large PB) of TDAP

In an attempt to observe the latency effect on the prefetching mechanism, we put 200 msec and 10 msec RTT into connections between a server and the clients. The longer RTT leads to the larger PB, as pointed out in Section 3.2. In our network configuration of Linux, the maximum size of a send buffer is 128 Kbytes so that, with 200 msec RTT, a maximum of 40 segments will remain as a pending segment in the buffer. We generate two input patterns; Pattern 1 generates 100 1 Mbps connections and terminates the connections all together after one second; Pattern 2 is the same as Pattern 1 except that the connection lifetime is increased to 3 seconds. Before starting the input patterns, we establish 100 1 Mbps connection with 10 msec RTT and 100 1 Mbps connection with 200 msec RTT, and then run the server around one minute to bring the TCP connections to steady states.

Fig. 15a illustrates the average bandwidth with Pattern 1. Without TDAP, when 100 new connections are established, the average bandwidth of 10 msec RTT connections abruptly drops from 100 Mbps to 54 Mbps. The bandwidth drop of 200 msec RTT connections is smaller than that of 10 msec RTT connections because more pending segments in the send buffer of 200 msec connections are able to tolerate the short-term bursting disk load. TDAP takes the number of pending segments into its prefetching calculation in such a way that 10 msec RTT connections take precedence of disk access over the 200 msec RTT connections. As illustrated in Fig. 15a, TDAP improves the bandwidth of 10 msec RTT connections so that the bandwidth reaches 200 msec RTT connections. In Fig. 15b, the lifetime of bursting connections increases to 3 seconds. After 2 seconds, all pending segments of 200 msec connections are consumed. Then, all connections in the groups start competing against each other to win disk access so that all four cases show similar bandwidth at 63 seconds. This observation confirms that our TDAP becomes capable of tolerating a short-term load variation by utilizing pending segments in a send buffer.

4.5 TCP Connection Termination of EDRC

EDRC enables a disk subsystem to cancel disk requests queued in a device queue on the reception of FIN or RST. We observed the performance of EDRC when 15 percent of connections were terminated by RST, as shown in Fig. 16a. At the beginning of this experiment, clients established 300 connections and let them run for 60 seconds. At 60 seconds, clients instantly terminated 45 connections by sending RST, reducing the overall number of connections from 300 to 255. On the steady states of 255 connections, the expected bandwidth is 168 Mbps. Thus, we expect that the termination of 45 connections will increase the overall bandwidth from 157 Mbps to 168 Mbps. Without EDRC, it takes approximately 3 seconds to reach the target bandwidth of 168 Mbps. EDRC reduces the time delay by a third, or 2 seconds. On the reception of RST, it cancels all corresponding disk requests and frees the memory pages allocated for the disk requests immediately. As a result, available disk bandwidth for other connections increases. Our profiling shows that, during the first second, 70 queued disk requests to load 5 Mbytes data into the page cache are removed from the disk queue immediately and it saves 5,000 memory pages in the cache.

4.6 Network Congestion of EDRC

Fig. 16b shows the same experimental result on the reception of duplicated ACK or ECN segments. In order to emulate the reception of ACK and ECN segments, we modify a server-side kernel to have a new system call that cuts the size of the congestion window in half. In this case, not all disk requests are delayed. Depending on the bandwidth of a given connection, a portion of disk requests are delayed by putting them into the delay queue of EDRC. After 60 seconds, 20 percent of an established connection's congestion windows shrink to half the size. Without EDRC, during that

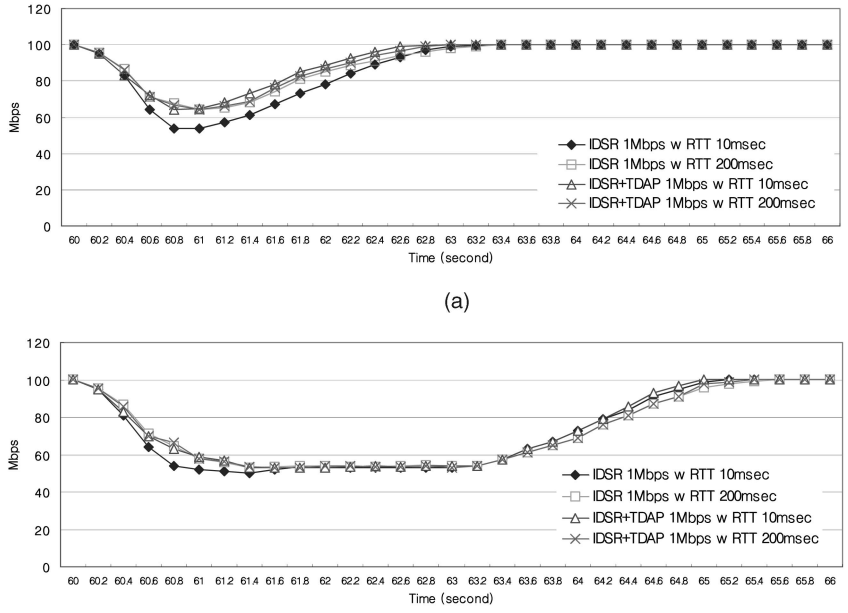

Fig. 15. WAN delay effect: RTT 200 msec. (a) The bandwidth against Pattern 1: 1 second burstiness (60-61). (b) The bandwidth against Pattern 2: 3 second burstiness (60-63).

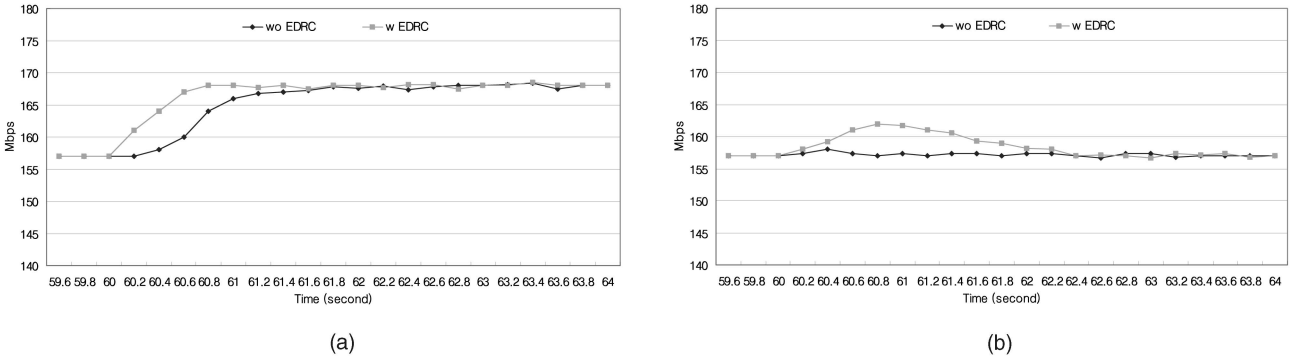

Fig. 16. The performance impact of eager disk request cancellation (in both cases, IDSR and TDAP are set on). (a) RST effect (b) Duplicated ACK or ECN effect.

period, no performance difference is observed, as shown in Fig. 16b. It is unlikely that EDRC will present a bandwidth increase, topping 162 Mbps. This improvement is less than that of the reception of RST segments. Our profiling result explains that, at 60 seconds, EDRC defers the execution of 23 disk requests to load 1.7 Mbytes data. This performance result confirms that EDRC makes a file system more resilient to abrupt changes in a TCP connection.

5 Conclusion

We have thoroughly analyzed the interaction between a file system and a TCP protocol stack in general operating systems on which widely deployed Internet servers run. We found that there were three mismatches during the interaction between a disk subsystem and a TCP/IP protocol stack that limit the data delivery performance of the servers. As a remedy to the mismatches, we have designed a new file system amortized with three mechanisms, each of which proposes an efficient TCP segment sending method, a TCP dynamics-aware data prefetching algorithm, and an eager disk request cancellation or a delay mechanism by sensing abrupt changes in TCP connections. Regarding a programming API, we patched the internal codes of *sendfile()* so as to require neither modification nor recompilation of server program codes for utilizing TPF. Thus, our implementation in Linux 2.4 presents its feasibility and practicality and the experimental results confirm 3-34 percent throughput improvement by reducing the number of context switches and improving the effective disk access ratio.

All three mechanisms are applicable to large and continuous data transfer rather than small file transfer. Only IDSR is beneficial for small file transfer due to the TCP's slow start mechanism. Three mechanisms can be selectively adopted to improve a file system by considering the characteristics of target applications. For example, for a caching server, where most contents are retained in memory without requiring disk accesses, only IDSR will help to improve the performance of file transfer since the other two mechanisms, TDAP and IDSR, are designed to streamline the disk access of a file system. Differently, a

streaming server that supports a multiple of different streaming rates concurrently can utilize TDAP for the performance improvement. We believe that TPF can facilitate the development of not only conventional applications, such as FTP, Web, and streaming over TCP, but also emerging applications, such as NAS, Web-Disk, and Peer-to-Peer, as long as they utilize TCP as a transport protocol and file systems as a disk access method.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their useful feedback on the paper. This work was supported by the NRL (National Research Laboratory) of the Ministry of Science and Technology of Korea.

REFERENCES

[1] P. Druschel and G. Banga, "Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems," *Proc. Second USENIX Symp. Operating Systems Design and Implementation,* 1996.
[2] A.S.J. Brustoloni, E. Gabber, and A. Singh, "Signaled Receiver Processing," *Proc. Gigabit Networking Workshop (GBN '99),* 1999.
[3] P.D.G. Banga and J.C. Mogul, "Operating System Features for Faster Network Servers," *Proc. Workshop Internet Server Performance,* 1998.
[4] M. Aron and P. Druschel, "TCP Implementation Enhancements for Improving Web Server Performance," Technical Report TR99-335, Rice Univ., 1999.
[5] G. Banga and J.C. Mogul, "Scalable Kernel Performance for Internet Servers Under Realistic Loads," *Proc. USENIX Ann. Technical Conf.,* 1998.
[6] P.D.G. Banga and J.C. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," *Proc. Third USENIX Symp. Operating System Design and Implementation,* 1999.
[7] P.D.V.S. Pai and W. Zwaenepoel, "Io-Lite: A Unified I/O Buffering and Caching System," *Proc. Third USENIX Symp. Operating Systems Design and Implementation,* 1999.
[8] S.F.K. Ramakrishnan and D. Black, "The Addition of Explicit Congestion Notification (ECN) rfc 3168," 2001.
[9] A. Cohen and R. Cohen, "A Dynamic Approach for Efficient TCP Buffer Allocation," *IEEE Trans. Computers,* vol. 51, no. 3, pp. 303-311, Mar. 2002.
[10] M. Arlitt and C. Wiliamson, "An Analysis of TCP Reset Behavior on the Internet," *ACM Comm. Rev.,* Mar. 2005.
[11] S.S. Lim and K.H. Park, "A Disk-to-Network Splicing (DNS) for Network Driven Data Transferring," *Proc. IEEE/IPSJ Symp. Applications and the Internet,* Jan. 2004.
[12] S.S. Lim and K.H. Park, "ECEM: Event-Correlation Based Event Manager for an I/O-Intensive Application," *J. Systems and Software,* vol. 74, no. 3, pp. 229-242, Mar. 2005.
[13] J.Y. Hwang, C.W. Ahn, and K.H. Park, "A Scalable Multi-Host Raid-5 with Parity Consitency," *IEICE Trans. Information Systems,* vol. E58-D, pp. 1086-1092, Dec. 2001.
[14] C. Lee, S.-H. Lim, S.S. Lim, and K.H. Park, "Autonomous Management of Clustered Server System Using Jini," *Proc. 15th IFIP/IEEE Distributed Systems: Operations and Management (DSOM),* 2004.
[15] K. Park and V.S. Pai, "Deploying Large File Transfer on an HTTP Content Distribution Network," *Proc. USENIX Workshop Real, Large Distributed Systems (WORLDS),* Dec. 2004.
[16] A. David, "TCP Splicing for an Application Layer Proxy Performance," IBM Technical Report RC 21139, 1998.
[17] R.M. Catalin, "An Evaluation of TCP Splice Benefits in Web Proxy Servers," *Proc. World Wide Web Conf.,* 2002.
[18] J. Wang, Y. Zhu, and Y. Hu, "UCFS: A Novel User-Space, High Performance, Customized File System for Web Proxy Servers," *IEEE Trans. Computers,* vol. 51, no. 9, Sept. 2002.
[19] M. Rosenblum and J. Outerhout, "The Design and Implementation of a Log-Structured File System," *ACM Trans. Computer Systems,* vol. 10, pp. 26-52, Feb. 1992.
[20] S. Ghandeharizadeh and L. Romas, "Continuous Retrieval of Multimedia Data Using Parallelism," *IEEE Trans. Computers,* vol. 44, no. 5, pp. 40-49, May 1995.
[21] K.W. Bg, K. Hau, and A. Yeung, "Analysis on Disk Scheduling for Special User Function," *IEEE Trans. Circuits and Systems for Video Technology,* vol. 9, no. 5, pp. 752-766, Aug. 1999.
[22] J. Nang and S. Heo, "An Efficient Buffer Management Scheme for Multimedia File System," *IEICE Trans. Informations and Systems,* vol. E83-D, pp. 1225-1236, June 2000.
[23] S.-H.L.K.-H. Kim and K.-H. Park, "Adaptive Read-Ahead Ahead and Buffer Management for Multimedia Systems," *Proc. Eighth Int'l Conf. Internet and Multimedia Systems and Applications,* 2004.
[24] W.R. Stevens, *TCP/IP Illustrated, Volume I.* Addison Wesley, 1994.
[25] D.P. Bovet and M. Cesati, *Understanding the Linux Kernel.* O'Reilly & Assoc., 2005.
[26] C. Li and K. Shen, "Managing Prefetch Memory for Data-Intensive Online Servers," *Proc. Fourth USENIX Conf. File and Storage Technologies,* 2005.
[27] M. Arlitt and C. Wiliamson, "Nist Net—A Linux-Based Network Emulation Tool," *ACM SIGCOMM Computer Comm. Rev.,* vol. 33, July 2003.
[28] A.H.S. Project, http://httpd.apache.org, 2006.
[29] oprofile, http://oprofile.sourceforge.net, 2006.

Sang Seok Lim received the BS degree in computer engineering from Chonnam National University in 1998 and the MS and PhD degrees in electrical engineering from the Korea Advanced Institute of Science and Technology in 2000 and 2006, respectively. His research interests include high-performance Internet server systems, operating systems, and embedded systems. He is a member of the IEEE.

Kyu Ho Park received the BS degree in electronics engineering from Seoul National University, Korea, in 1973, the MS degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST) in 1975, and the DrIng degree in electrical engineering from the University de Paris, France, in 1983. He was awarded a France Government Scholarship during 1979-1983. He is now a professor in the Department of Electrical Engineering at KAIST. His major interests include computer architecture and parallel processing. Dr Park is a member of KISS, KITE, the IEICE, and the IEEE.