Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software 74 (2005) 229–242
The Journal of Systems and Software
www.elsevier.com/locate/jss

# ECEM: an event correlation based event manager for an I/O-intensive application

Sang Seok Lim, Kyu Ho Park *

*Department of Electrical Engineering, Korea Advanced Institute of Science and Technology, Gusong-dong, Yusong-Gu, Taejun 305701, South Korea*

Received 29 April 2003; received in revised form 15 November 2003; accepted 18 November 2003
Available online 21 January 2004

**Abstract**

In Internet servers that run on general purpose operating systems, network subsystems and disk subsystems cooperate with each other for user requests. Many studies have focused on optimizing the data movement across the subsystems to reduce data copying overhead among kernel buffers, a network send buffer and a disk buffer. When data are moved across the subsystems, events such as read requests and write requests for data movement are also delivered across the subsystems by the servers and the operating system. However, there have been fewer studies on the optimization of event delivery across the subsystems. In conventional operating systems, an event from a disk subsystem is delivered to a network subsystem regardless of the status of the network subsystem. If the network subsystem is not ready for data sending, the execution of the server will be blocked, which causes scheduling and context switching overheads. This non-contiguous execution will incur deficiencies such as avoidable process blocking, context switching, cache pollution and long response time. To alleviate the deficiencies, we have developed inter-subsystem event delivery mechanisms that define event dependencies among the subsystems involved. We define an event correlation based on the happened-before relation. We propose deferred event delivery (DED) and disk-to-network splicing (DNS) to suppress scheduling and context switching during I/O request processing. We performed experiments on Linux 2.4 and the experimental results show that the number of context switching is reduced by up to 20% and server data transmit rate is improved by 4.0–8.1%.
© 2003 Elsevier Inc. All rights reserved.

*Keywords:* Operating system; Internet server; I/O event; Event manager

## 1. Introduction

With the exponential growth of the Internet, services are being provided by different types of Internet servers including WWW, Proxy, FTP and streaming servers. Many of these servers run on the one of general purpose operating systems such as UNIX, Linux, Windows NT, etc. Although the goals of the servers are different from each other, they are intrinsically I/O intensive. In other words, the server should intensively access I/O subsystems such as network subsystems and disk subsystems with the aid of the operating systems. The requests from Internet users arrive via the network subsystems and the responses generated by the servers are delivered to the users through the subsystem again. To process the requests, the server, with the aid of a disk subsystem, needs to access some data that may reside in the disks. All Internet servers are attached to the Internet through a network subsystem and data are provided by a disk subsystem so that the two subsystems need to collaborate to process the requests.

With the series of accesses to a disk and network subsystem, the servers achieve their original goals: web services, file downloads/uploads, and streaming service. Internet users differ in their network performance and computing power; Network bandwidth (Modem, ADSL, LAN, etc.) and computing power depending on the type of CPU and clock speeds, etc. These diversities affect the server system behavior directly. For example, a network subsystem is directly connected to the users and a disk subsystem is connected to the network subsystem indirectly via an Internet server. In this system, although the disk subsystem is ready to transfer data, the network subsystem may not be ready to transfer data due to the

* Corresponding author. Tel.: +82-42-869-5425; fax: +82-42-869-8025/5028.
*E-mail address:* kpark@ee.kaist.ac.kr (K.H. Park).

0164-1212/$ - see front matter © 2003 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2003.11.014

low bandwidth of user's network connection. In this situation, the server that delivers the data to a user must be blocked at the network subsystem (in a blocking mode) or return control immediately to the process (in a non-blocking mode). Since Internet servers deliver data to a client through the network subsystems, if the subsystem is not ready to deliver data, even though other operations that generate the data are already over, the data cannot be delivered to the user. Moreover, if the executions of the servers are blocked, it will cause scheduling and context switching. This non-contiguous execution will incur deficiencies such as avoidable process blocking, context switching, cache pollution and long response time.

With a conventional operating system, the deficiencies cannot be solved because the network subsystem and disk subsystem operate independently of each other with little sharing of information. The bridge between the two subsystems is processes (the active objects of the Internet server). Therefore, processes could handle the deficiencies by investigating the subsystems. Unfortunately, they do not have the necessary tools to handle this problem because accessing an operating system is usually restricted for security and safety. Therefore, we came to the conclusion that we should develop a mechanism to detect and manage inter-subsystem dependency so as to deal with the deficiencies. We define an *I/O event correlation* concept and develop an event manager based on the concept. With our event correlation based event manager (ECEM), we can detect and manage the inter-subsystem dependency and suppress the deficiencies. With ECEM, we propose two I/O request processing mechanisms; *deferred event delivery* (DED) and *disk-to-network splicing* (DNS).

There have been many studies on methods of optimizing I/O subsystems of the operating systems on which Internet servers run (Druschel and Banga, 1996; Brustoloni et al., 1999; Pai et al., 1999; Banga et al., 1998, 1999; Aron and Druschel, 1999; Banga and Mogul, 1998). Most of the studies only have focused on optimizing either the network or the disk subsystem. As explained above, however, there are inter-subsystem dependencies. To optimize inter-subsystem behavior, a *unified buffer* (Pai et al., 1999) issued the inter-subsystem dependency from the point of view of data copying. With the unified buffer which can be accessed and managed by both the disk and network subsystem, all data copying and multiple buffering across inter-subsystem (disk and network subsystems) can be eliminated. Besides data copying, there is event delivery between the subsystems. We have focused on optimizing event delivery between the subsystems so as to minimize the scheduling and context switching overhead. Our mechanisms can collaborate with the unified buffer.

This paper is organized as follows. Section 2 discusses the interaction between an operating system and Internet servers, Section 3 describes the concepts of an event and event correlation and explains a deferred event delivery mechanism. Section 4 describes a disk-to-network splicing mechanism. Section 5 explains implementation issues in Linux. Section 6 presents our experimental results of ECEM, and Section 7 concludes our paper.

## 2. Interaction between an operating system and applications

To process a user request, the servers need to access a network and disk subsystem, which generate a number of I/O requests. Therefore, we need to explain the general organizations of Internet servers and request/response processing mechanisms in general operating systems. To process an Internet user request, an Internet server generally accesses disks to read data and sends them to a client via a network interface card (NIC). Fig. 1 illustrates the architecture of I/O subsystems in Linux 2.4 and the general data flow on top of the subsystems. In the disk subsystem, so called file system, the page cache (Bovet and Cestati, 2001) is placed between the disk and a user process to accelerate disk access. When the process reads the data in the disk, it retrieves the page cache first. If the data already exists in the cache (cache hit), it will be copied to the process buffer immediately. If not, the disk subsystem loads the data from the disk into the cache and then it will be copied to the buffer. In the network subsystem, there are send and receive buffers that are managed by a TCP/IP protocol stack. The user process copies the data into the send buffer and then the stack puts the data into the NIC buffer by DMA. The incoming packet will be put into the receive buffer by the TCP/IP stack and then the data will be read by the user process. If the user process does not need to manipulate the data in its buffer, [1] it will copy the data from a disk buffer into the send buffer directly without copying the data into the process buffer which is depicted in Fig. 1 with the black arrow. The representative primitives are *sendfile( )* and *transmitFile( )* (Joubert et al., 2001). We developed two mechanisms for data copying with/without a process buffer respectively; deferred event delivery and disk-to-network splicing.

The general flow of I/O processing involved with the network and disk subsystem for file transferring via a process buffer is illustrated in Fig. 2. Such file transferring is the key operation of an Internet server. The flow is illustrated along the time line with snapshots of three buffers: a *process buffer* in a user process, a *disk buffer* in

[1] Internet server usually manipulates data in its buffer for data caching in its memory space, data encoding and bandwidth management.

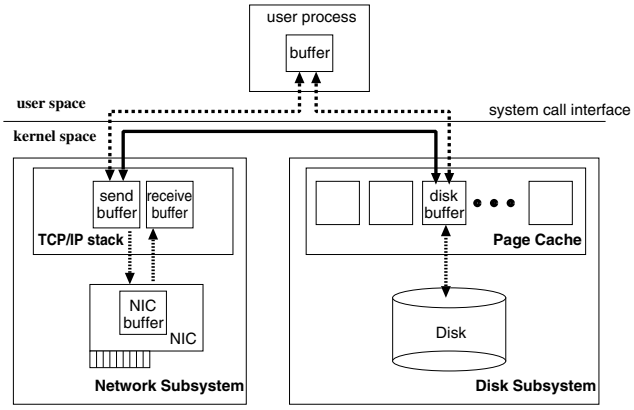


Fig. 1. The Architecture of I/O subsystem in Linux 2.4.

the page cache and a *send buffer* in the TCP/IP stack, all of which are depicted in Fig. 1. For simplicity, we assumed that multiple processes are running concurrently and the scheduling policy is non-preemptive. The process A has two phases: one is to read data from a disk into the process buffer and the other is to write the data into the send buffer. In the first phase, to read the data from the disk, the process allocates the process buffer and gives the pointer (memory address) of the buffer to the operating system. It checks if the data is in the page cache. If the data is already in the page cache, it copies the data into the process buffer and returns. In Fig. 2, we assume that the data are not in the page cache. If not, to load the data from the disk, the operating system allocates the disk buffer in the page cache and it enqueues a disk read request. After that, process A will sleep until the read operation completes. This execution is marked by the black line from $t0$ to $t1$. As soon as a disk interrupt handler copies the data to the disk buffer, it also wakes the sleeping process. When the process gets next CPU cycles by the CPU scheduler, it copies the data to the process buffer. This execution is represented by the black line from $t2$ to $t3$.

Now, the second phase begins. The process copies the data in the process buffer to the send buffer. If the send buffer has sufficient space for the data, it is immediately copied into the send buffer and the process returns. This flow is shown in the lower time-line of Fig. 2. In case that the send buffer is not available due to the lack of the buffer space (previously written data are not completely transmitted yet), the process must sleep again until the send buffer becomes available. This flow is illustrated in the upper time-line of Fig. 2. When the send buffer becomes available, the process will continue to copy.

### 2.1. The operating system deficiencies during interaction with applications

The request processing flow explained in the previous section is generally used in contemporary operating systems. A process generates a series of I/O requests to complete its task. At the time a process generates an I/O request through a system call, the process does not have enough current status information of an operating system. Therefore, it always generates the request as fast as possible. Similarly, when the operating system is ready to respond to the request, it will deliver the response immediately without considering the current status of the process. Since the process and the operating system

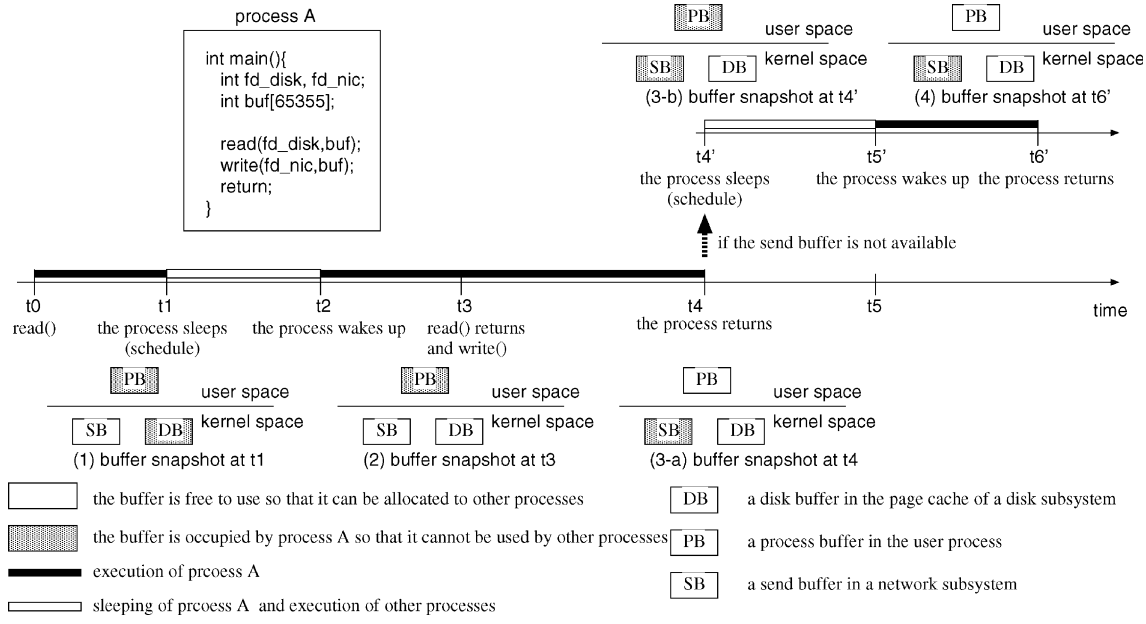


Fig. 2. Request processing flow in an general operating system.

share little information about each other, the I/O request processing mechanism incurs three deficiencies that degrade the overall system performance. The deficiencies are summarized as follows:

- Avoidable process blocking and Context switching.
- Long response time.
- Cache pollution.

Now, we will explain the three deficiencies in detail. Avoidable process blocking and context switching are directly related to each other. In the request processing flow of Fig. 2, there are two scheduling points at $t1$ and $t4'$. Without a page cache hit, the process is always blocked at $t1$. It is inevitable. But at $t4$, if the send buffer is available, the process will not be blocked again. If the operating system predicts the near future of the process, it can defer waking up the process until the send buffer becomes available at $t2$. By waking up the process only after the send buffer becomes available, the process will progress without sleeping, or blocking. Therefore, due to the lack of status information the process may experience scheduling and context switching overhead by process blocking. Fig. 3 illustrates the long response time deficiency. For easy explanation, we assume that the four processes, A–D have the same task size, which is 8, and the context switching overhead is $\alpha$. Fig. 3(1) shows what happens when the send buffer of processes A–D are available at the time when the processes write the data to the buffers. Fig. 3(2) shows the case that the send buffers of process A and B are not available on writing. On *read( )*, all processes are blocked and pass the CPU to the other processes. In Fig. 3(1), when *read( )* returns, *write( )* is called and it returns immediately because the send buffers are available. The average response time of all processes are $20 + 4\alpha$ and the number of context switching is 7. In Fig. 3(2), since the send buffers of process A and B are not available, they sleep again. The response times of process A and B increase to $30 + 8\alpha$. The average response times of all processes are $23 + 6\alpha$ and the number of context switching times is 9. The average response time and the number of context switching are increased by $3 + 2\alpha$ and by 2 respectively. Fig. 4 illustrates the cache pollution deficiency with the same process execution in Fig. 2. In Fig 4(1), during the time from $t1$ to $t2$, the data is copied from the disk buffer to the process buffer, being cached into the CPU data cache which usually adopts the *write allocation* cache mechanism (Hennessy and Patterson, 1995). Therefore, while the data is copied from the process buffer into the send buffer from $t3$ to $t4$, most of data will be hit at the CPU data cache. In Fig. 4(2), however, the process is blocked at $t4'$ due to the lack of send buffer space. Then, the other processes will be executed until the blocked process is scheduled again. Thus the data in the cache may be evicted due to the capacity limit of the cache. Finally, with the high chance of a cache miss, the data will be copied from the process buffer into the send buffer (from $t5'$ to $t6'$). The three deficiencies explained above will degrade the performance of the servers.

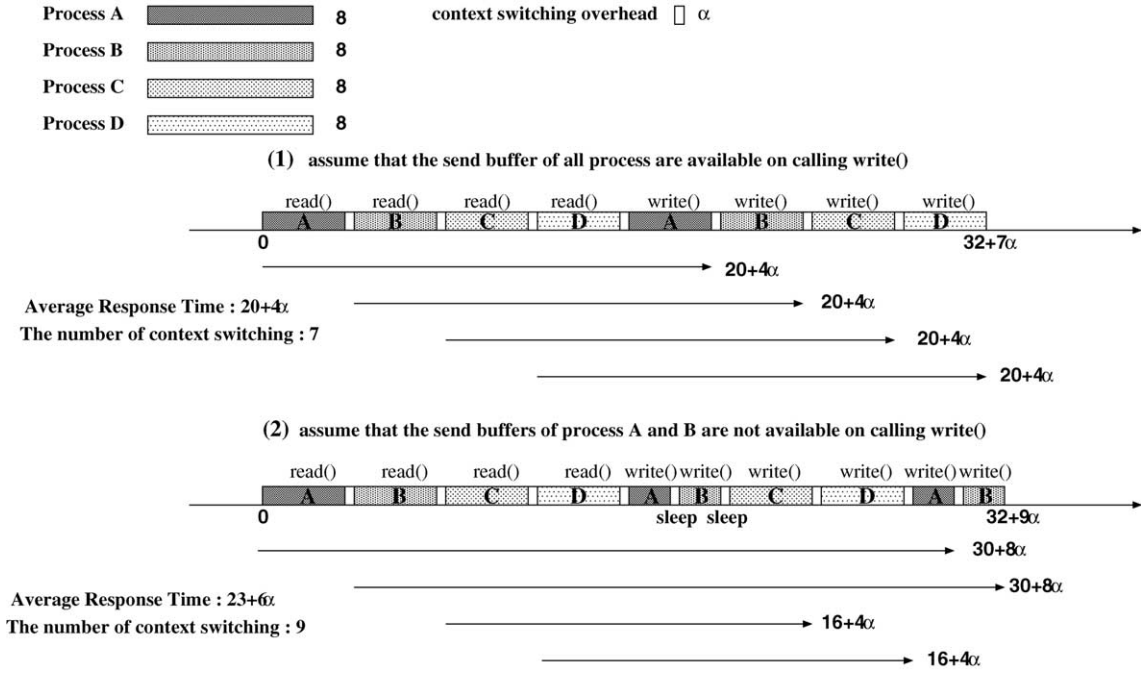


Fig. 3. Long response time during processing a request.

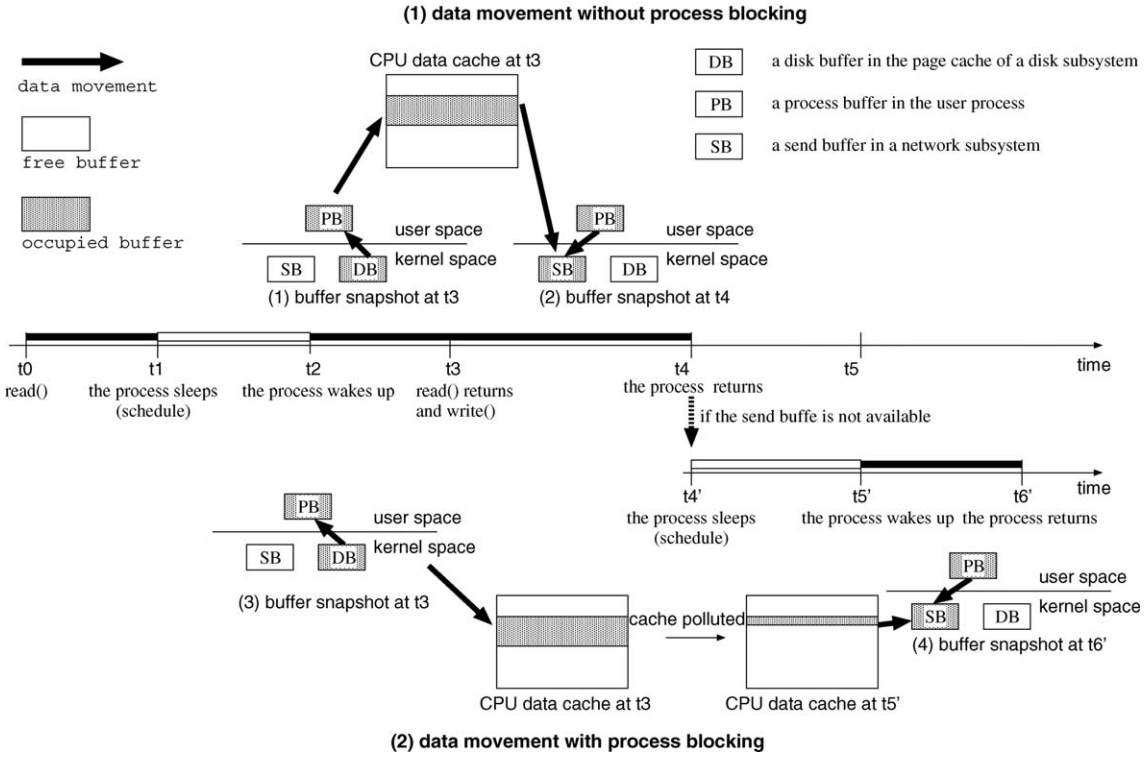


Fig. 4. Cache pollution deficiency during processing a request.

## 3. I/O events in an operating system

In this paper, we define an *event* as *an action generated by a signal, interrupt and system call which are invoked by an operating system or process*. A set of interesting events are grouped according to the related H/W devices for I/O and are listed in Table 1.

### 3.1. Event correlation

The series of events are invoked by a process and an operating system. In Fig. 5, the example of a file transmission process and its invoked events are illustrated. In the example, the four events, E1, E2, E3 and E4 are invoked sequentially. When *read( )* is called, then *disk read request*, E1 is invoked and then the process is blocked until *disk read response*, E2 is invoked. Both E1 and E2 are related to the disk subsystem. As soon as E2 is invoked by the disk subsystem, *network write request*, E3 will be invoked by the process which calls *write( )*. E4 is invoked when the data is copied into the send buffer. Both E3 and E4 are related to the network subsystem. In the view of time, E3 must be invoked only after E2 is invoked. Therefore, the two events which are related to the different subsystems are correlated by the process. In this manner, a series of events will be correlated according to the operating system access-pattern of the process. We define an *event pair* as *two events from different subsystems whose actions are correlated to each other directly, or which have inter-subsystem dependency*. For example, in Fig. 5, E2 of a disk subsystem and E3 of a network subsystem are directly correlated to each

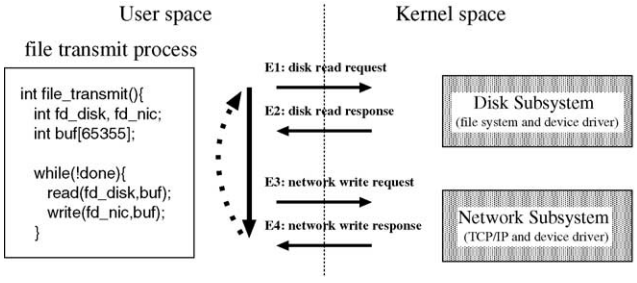


Fig. 5. Events in the example program.

Table 1
Events in an operating system

| H/W | Event |
|---|---|
| Network subsystem | Network read request, network read response<br>Network write request, network write response,<br>Arrival of SYN segment, arrival of Data segment<br>Arrival of ACK segment, arrival of RST segment |
| Disk subsystem | Disk read request, disk read response,<br>Disk write request, disk write response |

other's behavior because only after E2 is invoked, the data is copied to the process buffer, E3 will be invoked, the process initiates copying of the data from the process buffer to the send buffer. These two events, E2 and E3 are grouped together as an *event pair*.

The attribute of an event is *state*. The state is set to either *ready* or *wait*. A *ready* state indicates that the device is free to use or its resource is available. A *wait* state indicates the opposite. The attributes of an event pair consist of a *state* and *happened-before relation* (Lamport, 1978). The happened-before relation indicates that one event in an event pair should be invoked only after the other event is invoked in the order specified by a process. A happened-before relation is symbolized by "$E1 => E2$". We call ***E1*** pre-event and ***E2*** post-event. For example, in Fig. 5, $E2 => E3$. The states of an event pair are described in Table 2. For an event pair, there are four states; RR, RW, WR and WW. An RR state indicates that both of the events in the pair are ready, which means that the two events can be executed without blocking. An RW state indicates that a pre-event is ready but a post-event is wait. Although the pre-event can be executed but there is the high probability of its execution being blocked due to the post-event. A WR state indicates that the pre-event is wait and a post-event is ready. Therefore, as soon as the pre-event becomes ready, a post-event can be executed without blocking. Finally, a WW state indicates that both states are wait so that it has to wait that the pre-event is set to be ready.

Table 2
States of an event pair (A => B)

| Happened-before relation | State |
|---|---|
| A(Ready) => B(Ready) | RR |
| A(Ready) => B(Wait) | RW |
| A(Wait) => B(Ready) | WR |
| A(Wait) => B(Wait) | WW |

### 3.2. Deferred event delivery (DED)

An event pair in the either an RR or an RW state will be executed in conventional operating systems. However, the event pair in an RW state will most likely be blocked due to the post-event that is set to wait unless the post-event becomes ready during the processing of the pre-event. To avoid blocking, we suggest that the execution of the event pair in the RW state be deferred until the state of post-event becomes ready. For example, in Fig. 5, the delivery of a *block read response* to the process will be deferred until *network write request* becomes ready. By deferring, we can avoid process blocking which reduce the possibility of the occurrence of the three deficiencies mentioned in Section 2.1. Our approach cannot always avoid process blocking since the execution of an event pair whose state is ready will be scheduled to other process by expiring its time quantum or high priority interrupt handlers. However, we can reduce the chance of process blocking through our approach. In brief, conventional operating systems will execute an event pair in the states of both RR and RW. But an operating system enhanced with our event manager can detect and defer the execution of a process in an RW state. We call this event processing mechanism *deferred event delivery*.

### 3.3. Reducing operating system deficiencies by DED

In this section, we describe how to deal with three deficiencies in an operating system enhanced with ECEM. Fig. 6 illustrates how to deal with *avoidable process blocking* and *context switching*. In Fig. 6, the process sleeps at $t1$ and the process is supposed to wake up at $t2$. At $t2$, ECEM investigates the status of the send buffer. If the buffer is available, ECEM will wake up the process immediately but if not, it will defer waking up until the buffer becomes available. Therefore, with

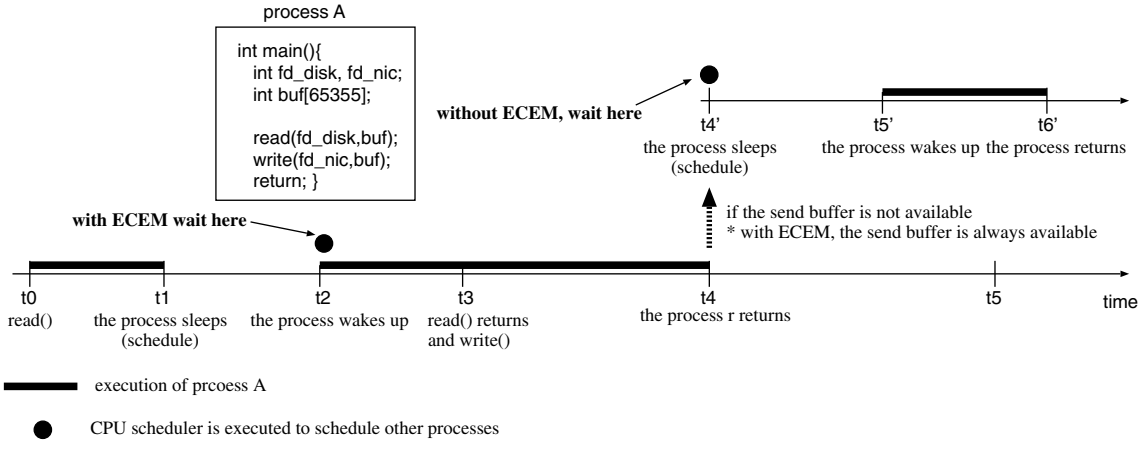


Fig. 6. Request processing flow in an operating system with ECEM.

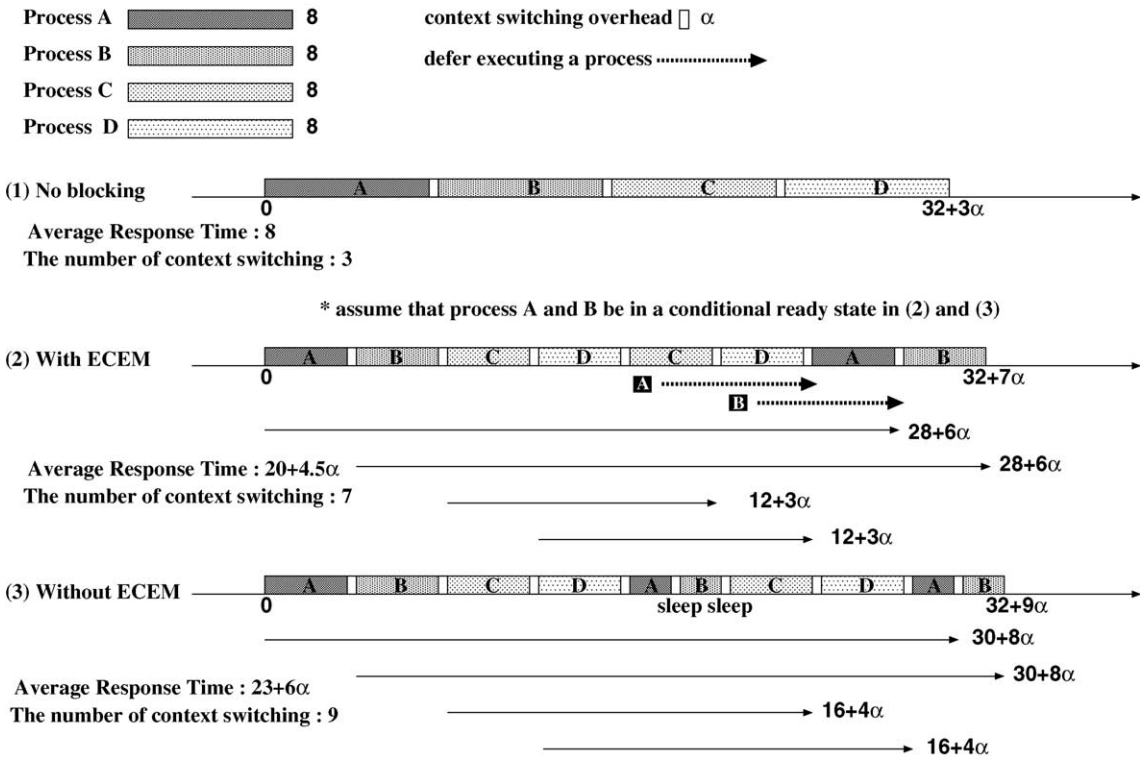


Fig. 7. Time diagram during request processing with ECEM.

ECEM, the process will wait at $t2$. However, without ECEM, the operating system wakes up the process immediately even though it may be blocked again at $t4'$. The response time with ECEM might be smaller than that without ECEM. which is explained in Fig. 7. For easy explanation, we assume that the four processes, A–D have the same task size, which is 8, and the context switching overhead is $\alpha$. We also assume that process A and B are in an RW state. Fig. 7(1) illustrates that the four processes are executed without any blocking. This is possible only when not only all the data are in a page cache but also the send buffers are available. In this case, the average response time is $8 + 0.75\alpha$. Fig. 7(2) shows the process execution timing diagram with ECEM. At the fifth time slot, process A is supposed to be scheduled. However, with ECEM, the execution is deferred to the seventh time slot when process A switches to an RR state. Fig. 7(3) shows the process execution timing diagram without ECEM. At the fifth time slot, even though process A is in an RW state, process A is executed without deferring. The process is blocked again, passing the CPU to the next process B. It is then scheduled at the ninth time slot again. The average response times of the three cases are 8, $20 + 4.5\alpha$ and $23 + 6\alpha$. The average response time with ECEM is about 14% lower than the response time without ECEM. The improvement ratio, 14% shows only the possibility not the absolute improvement ratio in a real system. With a DED mechanism, as long as the data movement is not interrupted by time quantum expiring and interrupt handling, the data is moved from the process buffer to the send buffer with high probability of cache hit as illustrated in Fig. 4(1).

## 4. Disk-to-network splicing (DNS) using ECEM

We have described an optimization mechanism based on a deferred event delivery mechanism. The mechanism is used for data copying via a process buffer. Internet servers need to copy data from the disk subsystem into its process buffer for data manipulation such as data caching in their memory space, data encoding/encryption and bandwidth management. To eliminate the data copying between a process and an operating system, data can be copied directly from a disk subsystem to a network subsystem. The representative primitives are a *sendfile( )* and *transmitFile( )* (Joubert et al., 2001). Even though the primitives are used, there is still data copying between two subsystems. This data copying can be eliminated with a unified buffer (Pai et al., 1999). The *sendfile( )*, *transmitFile( )* and unified buffer focused on eliminating data copying overhead between subsystems. In our method, however, we optimized the subsystems from the view of event delivery to reduce the number of context switching and scheduling overhead.

One important component of a network subsystem is the TCP/IP protocol stack. All data transferring is

managed by the TCP flow control mechanism (Stevens, 1994). That is, all data that are sent to a client must be ACKed and the data transfer rate is increased or decreased by the mechanism. In a disk subsystem, however, there is no flow control mechanism. The two subsystems have different read/write bandwidths and control mechanisms. Therefore, to transfer data from a disk subsystem to a network subsystem efficiently, we integrate a TCP/IP protocol stack with disk read routines so that the two subsystems are spliced to each other seamlessly. We named it the *disk-to-network splicing* (DNS) mechanism. With the mechanism, data is moved across the subsystem at the optimum timing so as to minimize context switching and scheduling overhead. In our paper, *sendfile( )* is enhanced by DNS because Linux provides *sendfile( )* only. However, the *transmitFile( )* and the unified buffer can also be enhanced by our mechanism because our mechanism have focused on data copying timing instead of eliminating data copying overhead. To transfer data from the disk into an NIC, a process needs to collaborate with H/W interrupt handlers and soft interrupt handlers (the bottom half handler in Linux). The transfer is not contiguous because I/O related operations of an operating system are originally interrupt-driven and priority-based. Basically, an H/W interrupt handler has the highest priority. A soft interrupt handler has lower priority than the H/W interrupt handler but higher priority than a user process. These priority mechanisms are generally adopted in modern operating systems. In this operating system, several different interrupt handlers are involved to transfer data as illustrated in Fig. 8. First, the process which calls *sendfile( )* checks whether the data is in the page cache or not. If not, it enqueues a disk read request and sleeps. When the data is ready in the disk, *device_intr_handler( )*, or the H/W interrupt handler puts the data into the disk buffer in the page cache by DMA and sets a soft interrupt mask to be scheduled by a CPU scheduler. After the H/W interrupt hander returns, *end_buffer_io_async( )* is executed by the scheduler. It manipulates buffer management flags and synchronization, and then wakes up the sleeping process. All the execution steps are depicted in the right of Fig. 8. The waken process copies the data in the page cache into the send buffer. Then, the data is processed through the TCP/IP protocol stack (*tcp_v4_sendmsg( )*,

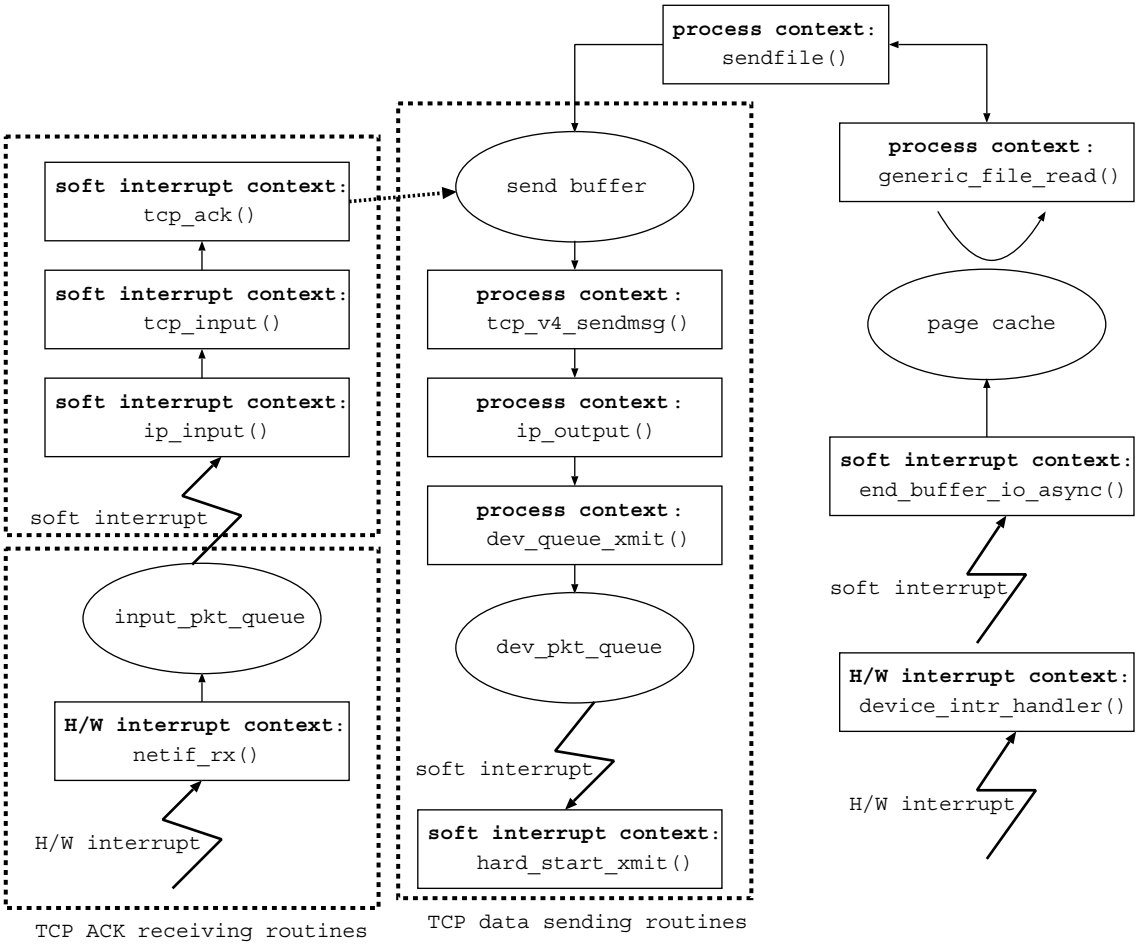


Fig. 8. Data transferring in the context of a hardware interrupt, software interrupt and process.

*ip_out_put( )* and *dev_queue_xmit( )* in Fig. 8) in the context of the process. After protocol processing, the data are put into *dev_pkt_queue* and a soft interrupt mask is set to be scheduled soon. Finally, the soft interrupt handler, or *hard_start_xmit( )* initiates DMA to copy the data into NIC. Whenever data are sent to a client, the data should be ACKed (Stevens, 1994). Therefore, we will explain incoming ACK segment processing. When data arrives at the NIC, *netif_rx( )* is executed in the H/W interrupt context. It puts the data into the *input_pkt_queue* and sets a soft interrupt mask to be scheduled. In the soft interrupt context, the data are processed through the TCP/IP protocol stack(*ip_input( )* and *tcp_input( )*). If the data is an ACK segment, *tcp_ack( )* will be called. It investigates the send buffer and empties the ACKed data in the buffer. After that, it checks whether there is a process that is waiting for free buffer space. If there is one, it will wake up the process. As explained, to transfer the data from a disk to NIC, soft interrupt and H/W interrupt handler need to be involved in the process execution, which causes scheduling and context switching as depicted in Fig. 8. There are several explicit scheduling points in the operating system illustrated in Fig. 8. First, the data from the disk must be loaded into the page cache. Second, the data must be put into *dev_pkt_queue*, to initiate DMA in a soft interrupt context. Third, DMA must be initiated to copy incoming data into the *input_pkt_queue* in an H/W interrupt context and TCP/IP protocol processing in a soft interrupt context. Finally, the process is scheduled if the send buffer is full.

In TCP connections, while transferring data between the sender and receiver, the spacing of the ACKs returned to the sender are identical to the spacing of the outgoing data segments in the sender. This is called the *self-clocking* behavior of TCP (Stevens, 1994). Since the receiver can only generate ACKs when the data arrives, the spacing of the ACKs at the sender identifies the arrival rate of the data at the receiver. Therefore, an incoming ACK rate reflects the transmission speed of network paths deployed between a server and a client. Whenever an ACK segment arrives at the server, it empties a certain amount of the send buffer. In conventional I/O processing steps, if the buffer has available space for new data, the space is filled with the data in a process context only after the process is scheduled by a CPU scheduler. To be scheduled, there is a time gap between the time when the send buffer becomes available and the time when the send buffer is filled. The context switching between different interrupt handlers and the time gap are inevitable in conventional operating systems. Therefore, to get rid of the scheduling and time gap, we developed a DNS mechanism. The mechanism integrates the ACK segment receiving routine and the data sending routine. The overall architecture is illustrated in Fig. 9. In Fig. 8, the two groups of routines in the three dashed boxes on the left are executed in different contexts. We integrate *TCP data sending* routines in the process context into *TCP ACK receiving* routines in the soft interrupt context. The soft interrupt handler which receives a TCP ACK segment continuously sends the new data whose size is identical to the size of data ACKed in the same soft interrupt context as depicted in the dashed box of Fig. 9. Therefore, there is no additional schedulings for sending the data and no time gap between TCP ACK receiving and TCP data sending. To read the data in a page cache into the send buffer, the handler needs the help of ECEM explained in the previous section. The handler executes *disk_read_request* event correlated to *Arrival of ACK segment* event, being supported by ECEM. That is, "*Arrival of ACK segment => disk read request*" is managed by ECEM.

## 5. Implementation issues in Linux

### 5.1. Overall system architecture with ECEM

The overall architecture of a server system with ECEM is illustrated in Fig. 10. A server program developer must know the event correlations, but there is currently no methods to inform the operating system of the correlations. Therefore, we create API to register event correlations required for ECEM. With the API, the process manages the event correlation table in the operating system. By retrieving the table, an event manager in the operating system detects a process in an RW state and deals with the process. In this approach, a developer must write a server program using the ECEM API. The event correlation library provides a set of data structures and interface functions. Two key interface functions to a user program are as follows:

- *int ecem_add(int fd_disk, int fd_net, int flag)*
- *int ecem_del(int fd_disk, int fd_net, int flag)*

*ecem_add( )* and *ecem_del( )* add and delete a new *event pair* described in Section 3 to the event correlation table whose entry is added, deleted and retrieved by a hash function. An example program with the functions is shown in Fig. 11. With *ecem_add( )*, the process associates the *fd_nic* of network I/O and *fd_disk* of disk I/O. In addition to the interface functions, we have more functions to find correlated events and check the state of event pairs, which are as follows:

- *int ecem_retrieve_correlated_disk(int fd_net)*
- *int ecem_retrieve_correlated_network(int fd_disk)*
- *int ecem_check_if_RW_state(int fd_net,int fd_disk)*

With these functions, the event manager retrieves the entries of the table and check if the event pair is in a RW

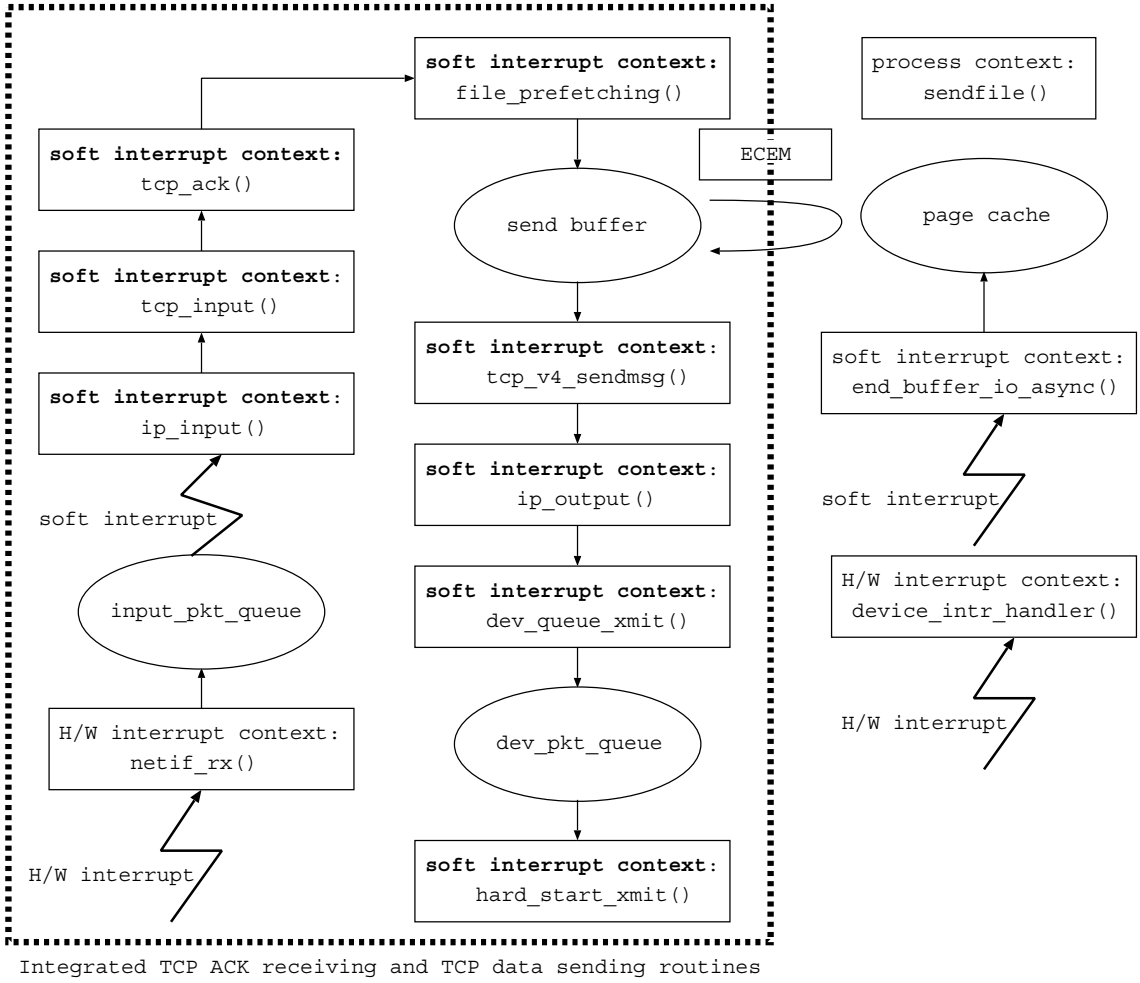


Fig. 9. Data transferring in the context of a hardware interrupt, software interrupt and process.

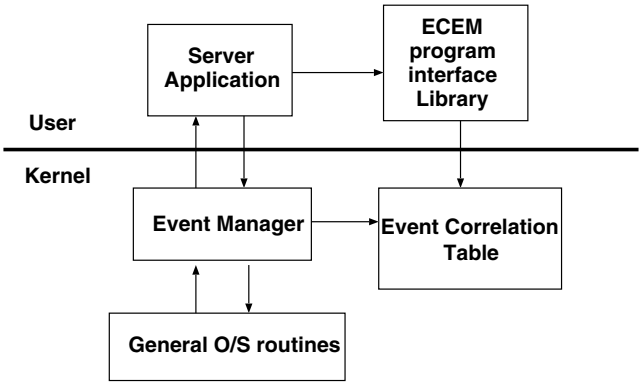


Fig. 10. Overall server system architecture with ECEM.

state. In the Internet server, there might be a significant number of the table entries so that we use a hash function to reduce overheads of managing and accessing the entries instead of linear search. Event manager can be a single point of failure if the manager is implemented as kernel daemon process. To avoid this problem, we integrated all the ECEM codes into Linux kernel codes as a module and patch so that we eliminate the single point of failure as well as reducing overhead of an event manager.

### 5.2. Implementation issues of DNS

To integrate the TCP data sending routines into the TCP ACK receiving routines, we address a few design issues; non-atomic operation, synchronization and data prefetching. The data sending routine is originally executed in the process context as shown in Fig. 8 and can be blocked due to non-atomic operations such as memory allocation, synchronizations (a spin lock or semaphore) and a disk read operation. A soft interrupt handler that processes incoming TCP ACK segments has to be modified to send data in the same soft interrupt context. A data sending routine has to be executed atomically so as not to be blocked. If the routine in the soft interrupt is blocked, the overall system will be blocked. Thus the memory allocation operation is executed with a non-blocking flag set. If a spin lock or semaphore is already held, the sending routine in a soft

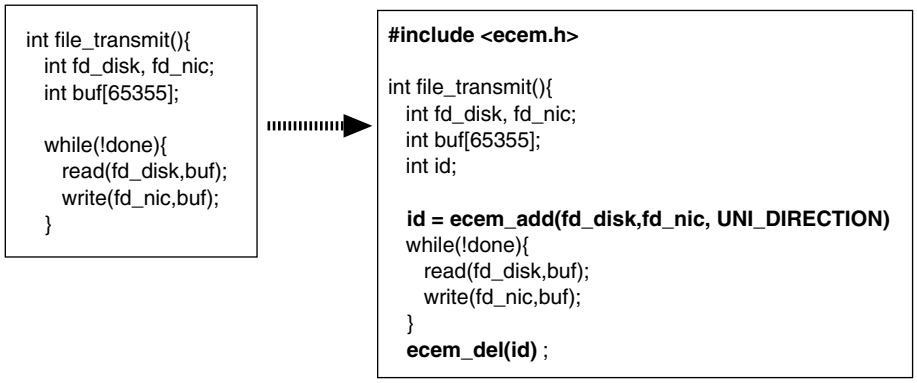


Fig. 11. Example program with ECEM.

interrupt context will not be executed any further, but just returned for synchronization. The disk read operation is intrinsically a blocking operation. Fortunately, a disk subsystem has two important components; a page cache and page prefetching mechanism. When a page is loaded into a page cache, multiple pages which are located ahead from the page within a file are prefetched into the cache (in Linux, maximum of 31 pages (4 Kbytes) are prefetched). Therefore, using the prefetching mechanism, the data sending routine can read data from the cache without blocking. When the data is not in the page cache, the routine will return immediately and try again in the next TCP ACK processing.

## 6. Experimental results

This section presents an experimental evaluation of ECEM with DED and DNS. We have implemented an ECEM kernel module for Linux 2.4. We developed an Apache-like multi-process server whose architecture is similar to an Apache web server (Apache 1.3, 2003), wuFTP WU-FTPD (2003) and HTTP based stream server (Scuturici et al., 2001). The server has two versions; one transfers data via a process buffer enhanced by DED and the other transfers data with *sendfile( )* enhanced by DNS. The server ran on a PC that was equipped with a single 1.5 GHz Pentium and 1 GBytes memory. We used Apachebench (ApacheBench, 2003) as a load generator which generated the HTTP requests as fast as possible.

In the first experiment, we have run the server enhanced with the DED mechanism and the non-enhanced server. The performance with a different number of concurrent connections is illustrated in Fig. 12. Each connection downloads a 10 MBytes file from the server. The number of concurrent connections are 10, 30, 60 and 120. As the number of concurrent connections increases, the system loads (scheduling overhead, memory utilization and disk load) also increase. Fig. 12 illustrates the performance of a DED mechanism which defers the scheduling of the process in an RW state. The average response time and total network transmission rate are improved for all ranges of concurrent connections except for 10. As shown in Table 3, the average response times are improved by about 4.0%. As the number of concurrent connections increases, the number of concurrent processes handling clients requests also increases. The more processes run concurrently, the more scheduling overheads are incurred. In Fig. 12(b), as the number of concurrent connections increases, the total network transmission rate decreases. The transmission rate of the enhanced server is decreased less steeply, compared to that of the non-enhanced server. This implies that the enhanced server suffers less from the deficiencies explained in Section 5 than the non-enhanced server. The performance improvement can be proved by Fig. 13(a) which shows the number of context switchings during the experiment. As the number of concurrent connections increases, the number of context switching increases due to the increasing number of server processes. With DED, the number of context switchings is decreased for all ranges by about 20% except for with 10 concurrent connections.

In the second experiment, we ran the server enhanced with a disk-to-network splicing mechanism and a non-enhanced server. We used the *sendfile( )* which transfered data in the disk subsystem into the network subsystem directly without a process buffer. A DNS mechanism was integrated into the *sendfile( )*. The measured performances are illustrated in Fig. 14. According to the number of concurrent connections, the average response times are improved by 10.1–4.9%. As the number of concurrent connections increases, the improvement ratio decreases. This is because with an increase in the number of concurrent connections the disk load, memory utilization and synchronization overhead is increased proportionally so that the hit ratio in a page cache decreases.

The system network transmission rate is increased by 4.15–9.19 Mbps. The number of context switching during experiments is illustrated in Fig. 13(b).

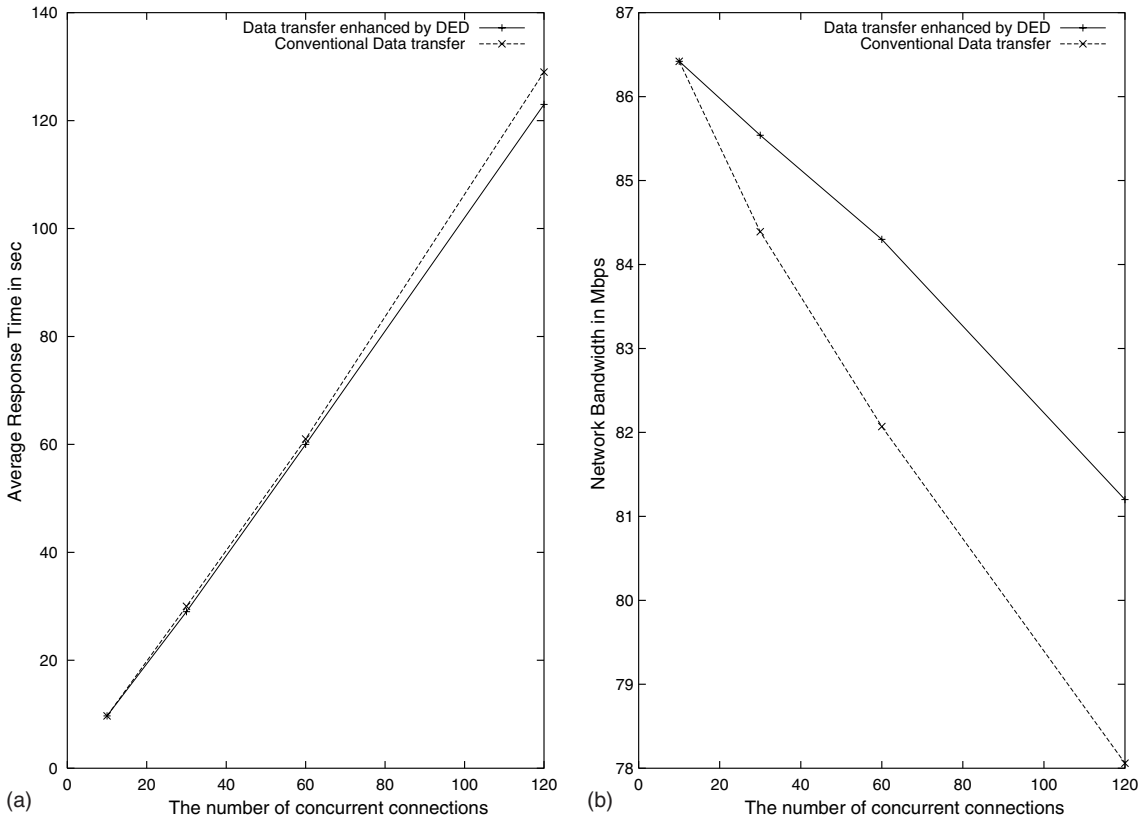


Fig. 12. Performance of DED: (a) Average response time, (b) total network transmission rate.

Table 3
Performance improvement ratio(%)

| The number of concurrent connections | 10 | 30 | 60 | 120 |
|---|---|---|---|---|
| Enhanced by DED | 0.0 | 2.4 | 2.2 | 4.0 |
| Enhanced by DNS | 8.1 | 7.2 | 7.0 | 6.9 |

Compared with that of a DED mechanism, the number of context switching of the *sendfile( )* enhanced with a DNS mechanism is decreased by 1/2–1/3. Comparing with the non-enhanced *sendfile( )*, the number of context switching is decreased by up to 17%. This shows that a DNS mechanism reduces scheduling overheads significantly. As explained in Section 4, data sending routines are integrated into a soft interrupt handler which processes incoming TCP ACK segments so that the context switching between different contexts is eliminated. Table 3 summarizes overall performance improvement ratio.

## 7. Conclusion

We have analyzed the behavior of an I/O intensive operating system on which Internet servers were running and have found that there are three deficiencies: *avoidable process blocking and context switching, long response time* and *cache pollution*. These deficiencies are caused by non-optimal inter-subsystem event delivery. To alleviate the deficiencies, we have developed inter-subsystem event delivery mechanisms. To describe the inter-subsystem dependencies of events, we defined an event correlation concept in which events from different subsystems were correlated by a happened-before relation. Based on the concept, we have developed two event handling mechanisms; deferred event delivery and disk-to-network splicing. With a deferred event delivery mechanism, the event manager detects events in an *RW* state and defers the notification of the events until their states become *RR*. The mechanism improves the data transferring via a process buffer. The disk-to-network splicing mechanism improves direct data transferring without a process buffer. The splicing mechanism reduces the scheduling and context switching overhead by integrating the TCP data sending routines and TCP ACK receiving routines in the same soft interrupt context. We have implemented an ECEM module on Linux 2.4 and the experimental results showed that the number of context switching is reduced by about 20% and the performances of the servers were improved between 4.0% and 8.1%.

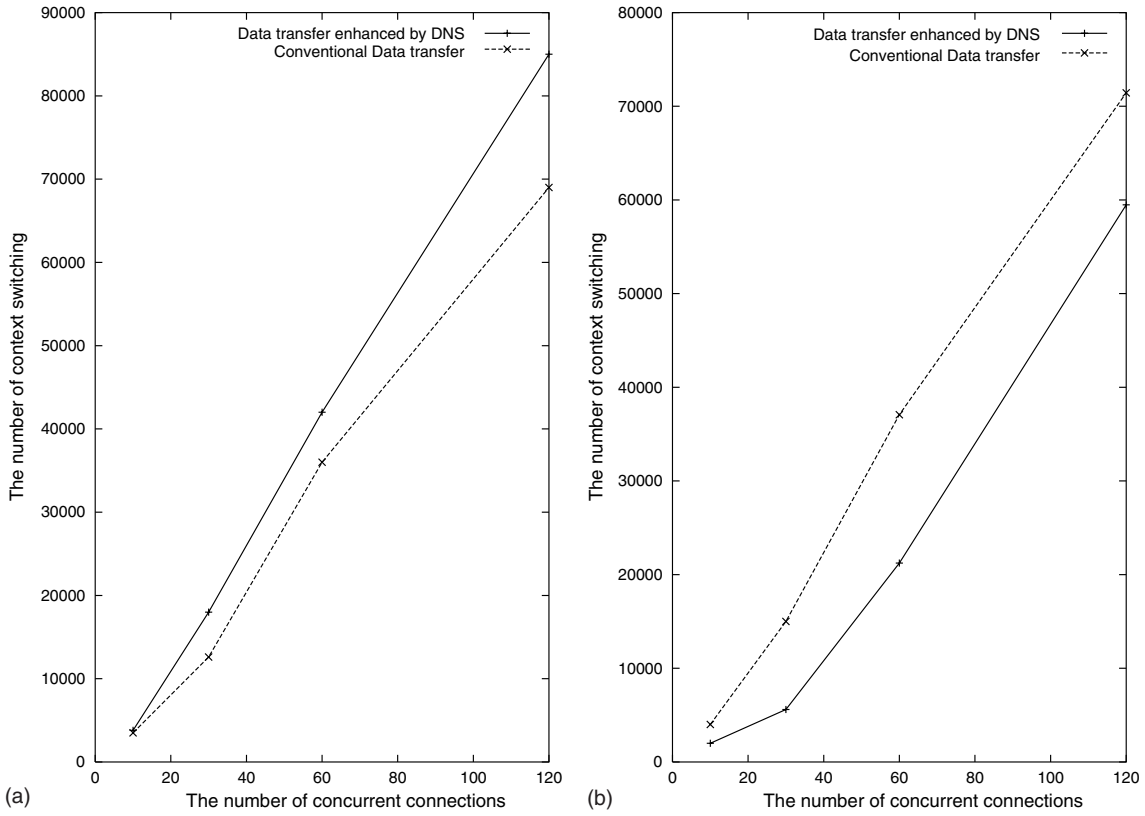


Fig. 13. The number of context switching: (a) DED, (b) DNS.

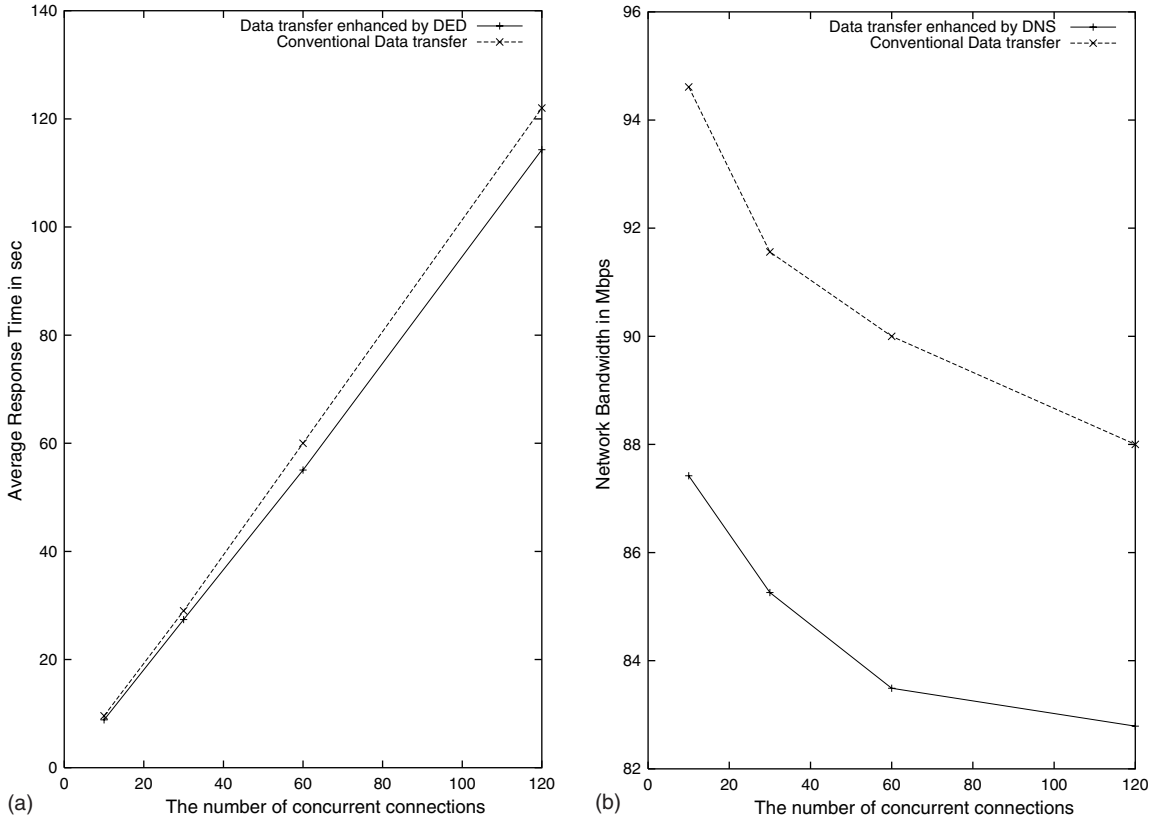


Fig. 14. Performance of disk-to-network splicing: (a) average response time, (b) total network transmission rate.

## References

ApacheBench. Available from <http://www.remotecommunications/com/apache/ab/>.

Aron, M. Druschel, P. 1999. TCP Implementation Enhancements for Improving Web server Performance, Rice University Technical Report, TR99-335.

Banga, G., Druschel, P., Mogul, J.C., 1998. Better operating system features for faster network servers. In: Proceedings of the Workshop on Internet Server Performance, Wisconsin, USA.

Banga, G., Druschel, P., Mogul, J.C., 1999. Resource containers: a new facility for resource management in server systems. In: Proceedings of the 3rd USENIX Symposium on Operating System Design and Implementation, New Prleans, USA.

Banga, G., Mogul, J.C., 1998. Scalable kernel performance for Internet servers under realistic loads. In: Proceedings of the USENIX Annual Technical Conference, New Orleans, USA.

Bovet, D.P., Cestati, M., 2001. Understanding the Linux Kernel. O'Reilly & Associate.

Brustoloni, J., Gabber, E., Silberschatz, A., Singh, A., 1999. Signaled receiver processing. In: Proceedings of Gigabit Networking Workshop GBN99, New York, USA.

Druschel, P., Banga, G., 1996. Lazy receiver processing (LRP): a network subsystem architecture for server systems. In: Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation, Seattle, USA.

Hennessy, J., Patterson, D., 1995. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers.

Joubert, P., King, R.B., Neves, R., Russinovich, M., Tracey, J.M., 2001. High-performance memory-based servers: kernel and user-space performance. In: Proceedings of the USENIX Annual Technical Conference, Boston, USA.

Lamport, L., 1978. Time, clocks, and the ordering of events in a distributed system. ACM J. Commun. 21 (7), 558–565.

Pai, V.S., Druschel, P., Zwaenepoel, W., 1999. IO-Lite: A unified I/O buffering and caching system. In: Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation, New Orleans, USA.

Stevens, W.R., 1994. TCP/IP Illustrated, Volume I. Addison Wesley.

Scuturici, V.-M., Scuturici, M., Miguet, S., Pinon, J.-M., 2001. Measuring Web servers performance in VoD. In: Proceedings of international Conference on Telecommunications, Bucharest, Romania.

WU-FTPD Development Group. Available from<http://www.wu-ftpd.org/>.

**Sang Seok Lim** received B.S in Computer Engineering from Chonnam National University and M.S in Electrical Engineering from Korea Advanced Institute of Science and Technology, 1998 and 2000 respectively. He is now a Ph.D. student in KAIST. His research interests are high-performance Internet server systems and operating systems.

**Kyu Ho Park** received the B.S. degree in Electronics Engineering from Seoul National University, Korea in 1973; the M.S degree in Electrical Engineering from the Korea Advanced Institute of Science and Technology (KAIST) in 1975; and Dr.Ing. 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 at the Department of Electrical Engineering, KAIST. His major interests include computer architecture, parallel processing. Dr. Park is a member of KISS, KITE, IEICE and IEEE.