Flexible Event Delivery for Kernel Extensions in ELinux

Christian Poellabauer, Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Richard West
Computer Science Department
Boston University
Boston, MA 02215

Abstract

Many applications rely on the kernel's ability to specialize and extend its functionality in order to achieve high performance and meet application-specific QoS requirements. However, general-purpose operating systems lack sufficient support for flexible and scalable event-based communication between user-level applications and kernel-level extensions. This presents problems especially for real-time applications, which typically require notification of events and response to events in a timely manner. This paper introduces ECalls, a lightweight, bidirectional user/kernel event delivery facility. ECalls not only supports the timely delivery of events, but it also reduces the cost and frequency of user/kernel boundary crossings. ECalls is a communication tool that allows (a) kernel extensions to register their offered services and (b) applications to register their interest in these services. Using ECalls, applications use lightweight system calls to generate events, while kernel extensions raise real-time signals or invoke handler functions (residing in either user or kernel space), or they may use kernel threads to handle events on behalf of applications. ECalls can also influence the CPU scheduler such that a process with pending events is given preference over other processes. To demonstrate their utility, this paper implements an I/O event delivery mechanism using ECalls. This mechanism is shown to improve the performance of two applications: a web server and a distributed video player.

1 Introduction

Server systems heavily utilize operating system facilities, as exemplified by web servers handling thousands of requests per second, and mail servers maintaining millions of mailboxes in file structures. To address the resulting high rates of system calls on server machines, researchers have sought ways to reduce system call overheads [1, 2], and they have created facilities by which operating system kernels are extended with application-specific functionality [3, 4, 5]. One goal of such work is to reduce the number of user/kernel interactions required for certain service requests. In addition, solutions have been created for better integration of certain kernel- and user-level actions taken on behalf of requests, thereby reducing request latencies. Examples include scheduler activations [6] to exploit kernel-level knowledge of request state to guide user-level scheduling decisions, upcalls [7, 8] to integrate kernel- with user-level actions for protocol processing, and real-time variants of the latter [9, 10, 11] to address the specific needs of multi-media and real-time applications.

Common elements of these solutions are (1) the need to share information about performance-critical events between kernel- and user-level facilities, and (2) to be able to act on such information in a timely fashion. For instance, communication rates can be adjusted based on information about buffer fill-levels [12], if such information is made available and acted upon with little delay. The same requirements exist when exploiting knowledge about the ACK/NAK behavior of communication protocols to alter the behavior of media streaming [13] or even scientific applications [14]. In fact, past work has shown that system quality may be reduced rather than improved by runtime adaptation if such actions are not performed within certain tolerances [15].

This paper presents a general mechanism for coordinating kernel/user actions and for sharing information between both, termed *E(event)Calls*. ECalls may be used to link existing kernel facilities with user-level functionality, and also, to link application-specific kernel extensions with appropriate user-level functions. That is, via ECalls, unprivileged instructions can access the services implemented by their (or other) ECall-equipped kernel extensions. The implementation of ECalls provides separate facilities for information sharing and coordination across the user/kernel boundary, thereby providing flexibility in both data and control passing not offered by system call facilities existing in general purpose operating systems [16].

The interaction model supported by ECalls is that of events and event channels, where parties interested in certain events subscribe to shared channels to which events are produced and from which they are received. Event publication and receipt also imply an action triggered by the event, e.g., the execution of a handler function. Such an action (e.g., the execution of a user-defined kernel extension that filters kernel events on behalf of a specific user) is defined at the time the event channel is created. The publish/subscribe paradigm implemented utilizes events that are described by event-specific formats known to both producers and consumers. Currently, only one consumer and one producer can subscribe to an event channel. The ability of kernel-level facilities to subscribe to multiple event channels is subject of our future work, where such multiple subscriptions will result in event descriptions being copied to multiple user-level address spaces.

The exact representation of ECall events and the manner in which their contents are read/written by kernel- and user-level programs depend on the formats and use of these events. For instance, ECalls does not proscribe some specific synchronization strategy for access to event data. Therefore, an application can create any number of different events and use them as it sees fit, subject only to restrictions in the total memory available for storing the events' representations[1].

Event formats and the control semantics of event channels are defined at the time of channel creation. Specifically, the implementation of ECalls supports a variety of well-defined, per-channel ways in which control is passed between user and kernel upon event production and receipt. By separating data and control passing in this fashion, different specializations of each may be used to implement efficient user/kernel event sharing for different applications and usage scenarios. For instance, a real-time ECall channel implies that upon event generation by the kernel, a real-time signal will be generated to the address space subscribed to this channel. Other control passing methods offered by ECalls include high performance, lightweight non-blocking downcall methods (i.e., system calls), and downcalls that reduce the total number of system calls performed, thereby improving performance by aggregation of events. In the reverse direction, from kernel to user, ECalls can directly invoke handler functions residing in either kernel or user space, or they can execute such functions in kernel threads, or even raise real-time signals to applications. In addition, we have modified both the standard Linux scheduler and a hard real-time scheduler developed by our group to efficiently combine the scheduling of user-level processes and events. The effect is that kernel-level events can result in the re-scheduling of the user-level processes interested in these events. Past work has shown that such coordinated kernel/user behavior is particularly important for real-time and multi-media applications [17, 10].

The results shown in this paper heavily exploit the ability of ECalls to vary control semantics. For instance, we attain substantial performance improvements in a web server's ability to handle high rates of incoming requests by using ECalls to coordinate the necessary kernel- and user-level actions taken in response to web service requests, in place of the standard system call facilities offered by Linux. More precisely, using ECalls we observe improved

[1] The memory areas shared between kernel and user must be pinned in memory, thereby defining certain limits on the amounts of data possibly shared between them.

throughput in overload scenarios by replacing expensive select() system calls through a kernel extension, which monitors activity on sockets and uses ECalls to inform interested applications of these activities. Throughput and response times are further improved by using ECalls to collect load information from the user-level web server as well as certain kernel structures, to reject new requests already arrived in the kernel in the case of high system loads. One result that will be discussed later in this paper is an improvement in throughput of 50% and more for a web server application, and an improvement in response times of 200% and more, both in overload situations.

The importance of offering not just 'fast' but also richer control passing semantics for event sharing is demonstrated by a distributed video player application, where a single host runs several instances of multimedia applications receiving video streams from different servers. In this example, ECalls allows applications to share information with kernel extensions enabling the applications to achieve a desired quality of service.

In summary, the main contribution of ECalls is the ability to specialize user/kernel event delivery. This includes the efficient execution of kernel-level handlers provided by applications. It also includes the prioritization and batching of events, the latter typically resulting in reductions in the number of context switches incurred during user/kernel and kernel/user interactions.

ECalls is part of the ELinux [18] project, our group's effort to build a version of the Linux operating system able to deal with dynamic resource changes and able to exploit and adjust itself to runtime changes in user needs. The following two chapters introduce both ELinux and ECalls. The different methods of event delivery and event handling that are offered in ECalls are described in Chapter 4 and our experimental results are presented in Chapter 5.

2 ELinux Overview

ELinux consists of a number of modifications and enhancements to the Linux kernel, which enable the operating system to dynamically adapt itself to changes in resource requirements of its applications. ELinux not only offers basic mechanisms for runtime system extension, but it also supports several extensions specifically targeting the real-time domain. These extensions are implemented as kernel-loadable modules.

Real-Time CPU Scheduling. One of the extensions offered by ELinux is a hard real-time scheduler that replaces the standard Linux scheduler. DWCS (Dynamic Window-Constrained Scheduler) [19, 21] schedules processes according to their service constraints: a *request period* T and a *window-constraint* x/y. The end of a request period determines a *deadline* by which a process must be serviced. The numerator x of the window-constraint is the number of times a request for a process can be serviced later than the deadline or not serviced at all for every y requests. Given a set of processes, DWCS is able to guarantee a maximum delay of service to a process, even in overload scenarios (when processor utilization is larger than 100%). Furthermore, feasible schedules exist as long as the utilization does not exceed 100%. Details about the real-time capabilities of DWCS can be found in [20].

Real-Time Network Packet Scheduling. DWCS is also capable of scheduling network packets, where the end of a request period T determines a *deadline* by which a packet of a certain stream must be serviced. The numerator x of the window-constraint x/y is the number of packets that can be transmitted late or even lost for every window of y packets in a stream. The same real-time guarantees as described above apply. Consequently, the packet scheduling extension of ELinux permits applications to control their outgoing packet streams, which is particularly useful for server-based applications, such as media and web servers.

QoS-aware Implementation of Sockets. The DWCS-based packet scheduling supported by ELinux does not offer the 'convenient' interfaces end users may require for controlling their servers' output streams. The 'QSocket' interface implemented in ELinux uses the Linux kernel's recently developed framework for the implementation of QoS technologies, the latter supporting a set of queuing disciplines that include Class Based Queuing (CBQ), Stochastic Fair Queuing (SFQ), and Random Early Detection (RED). We have added DWCS to this list of queuing disciplines, but are also improving the interfaces provided to end users. Specifically, access to the QoS architecture of Linux is currently quite cumbersome; it is typically through the netlink protocol, specifically designed to allow users to call kernel-internal functions. In response, we are implementing a socket library that adds QoS parameters to what appear to be regular socket calls. This library, called *QSockets*, extends the original attribute list of socket calls by a text string containing QoS parameters for the corresponding socket stream. Qsockets allow users to assign and modify per-stream QoS-attributes dynamically for DWCS and other queuing disciplines supported by Linux.

Resource Monitoring – ECalls and Monitoring Handlers. Quality management is not possible without understanding dynamic resource availability. Toward this end, we have implemented a general mechanism for extracting such resource information from the kernel, termed ECalls, and we are demonstrating its utility by construction of specific resource monitoring kernel extensions. The ECall interface and functionality is described in detail in this paper and is overviewed below. Next, we describe briefly one of the monitoring extensions we are currently constructing. Specifically, we are implementing a network monitor in the ELinux kernel that will allow applications to gather information and statistics about network utilization, loss-rate, etc., in an application-specific manner, thereby permitting them to enact application-level adaptations to try to adjust to dynamic changes in network behavior. The monitoring tools reside within the kernel such that only network information desired by the application (using filters) and network statistics are passed to the user-space. We are basing our implementation on Linux Socket Filters (LSFs).

ECalls – Asynchronous and Synchronous User/Kernel Communication. To efficiently access user-provided kernel extensions, such as the CPU and network schedulers and the monitoring handlers described above, the E(vent)Calls mechanism implements event-based communications from user to kernel space and vice versa. While ECalls is the topic of the remainder of this paper, we first outline the nature of the distributed quality management systems we are constructing with ECalls, with monitoring extensions, and with the user-specific and real-time kernel extensions described above.

Adaptive Resource Management with ELinux. Our general approach to quality of service management is the runtime adaptation of resources used by applications. This is achieved by enabling applications to determine or at least influence the adaptation decisions being made on their behalf. This approach is spelled out in more detail in a paper describing the Dionisys quality of service infrastructure [17], which permits applications to determine how, when, and where adaptations take place. This is done by executing application-specific handler functions with service providers, including those in the kernel. Such handlers have two tasks: (1) they monitor current resource usage and (2) they adapt resource allocation in conjunction with application needs. Our previous implementation of Dionisys was as a user-level resource manager on Solaris. Our ongoing work includes the implementation of suitable handlers as well as the resulting handler interactions within the kernel and across the user/kernel boundary in ELinux. In this case, monitor and handler functions can reside either within the application in user space or in loadable kernel-modules in kernel space. Thus, the ECalls mechanism described in this paper is a critical element of the manner in which quality management is performed in ELinux.

Event-based Communication. ECalls permits quality management to take place on a single machine, but it does not support inter-machine quality management actions, as indicated by previous and ongoing research on distributed quality management infrastructures. To address multi-machine quality management, we are adding another critical mechanism to the ELinux system, termed *real-time events*. The resulting *Artemis* real-time event system is offering an event communication library resident in the ELinux kernel, based on which inter-service and inter-machine events may be sent and received. In this fashion, we will construct a distributed implementation of the Dionisys approach to quality management. With Artemis (a) applications can communicate in an asynchronous and anonymous manner with other processes at the same host or at remote hosts, and (b) resource managers, residing either in kernel space or in user space, can exchange adaptation events within one host and across multiple hosts.

An effort somewhat orthogonal to the work described above, but also ongoing, is to more effectively use the fast networks and large main memories now existing for many Linux machines. Towards this end, our group is extending the Linux file system. One outcome of interest to the real-time community will be the file system's ability to control data caching and also the scheduling of caching actions, thereby offering quality of service support for distributed file management. The remainder of this paper focuses on ECalls.

3 Architecture of ECalls (Event Calls)

The ECalls mechanism (also simply referred to as *ECalls* in the remainder of this paper) allows user-level applications and kernel-level extensions to exchange events. It consists of five main components, as shown in Figure 1. The shaded area in this figure depicts the kernel-loadable module comprising the units *Service Registration*, *Application Registration*, *Event Dispatcher*, and *Event Queue*. The fifth and last component, a set of two pinned memory areas shared by an application and a kernel extension, is accessible from both user- and kernel-level.

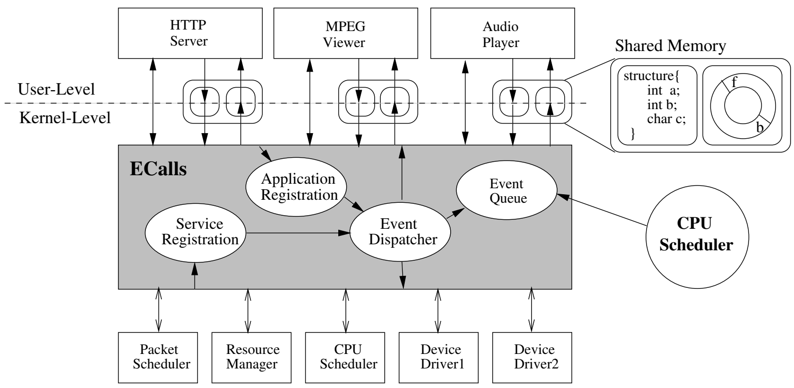

Figure 1: Architecture of ECalls.

The remainder of this chapter briefly introduces the different components of ECalls depicted in Figure 1:
Service Registration. A kernel extension registers with ECalls (using the function register_service()) to make its services publicly available. Along with the registration of its services, a kernel extension also informs ECalls of *how* it wishes to be notified of events generated in user-level applications.
Application Registration. Once a kernel extension has registered with ECalls, applications are free to register

for these kernel services (using the function ECalls_register()). In order to exchange events with kernel extensions, applications must know certain service-specific information, which is described in a C header file supplied by the kernel extension. Besides a unique character string identifying the service, the header file also includes information about the structure of the areas of memory shared between the application and the kernel. Along with the registration, an application also determines *how* it wishes to be notified of events generated by the kernel extension.

Event Dispatcher. Once the registration process is complete, both applications and kernel extensions are able to generate and receive events. The event dispatcher manages a heap structure containing the information necessary to direct events from the event producer to the event consumer.

Event Queue. Our work on ECalls also includes modifications to the standard Linux CPU scheduler and a hard real-time CPU scheduler (DWCS) in order to support the cooperative scheduling of tasks and events. That is, tasks with events pending can be favored by the scheduler. In order to inform the CPU scheduler about pending events, the Event Dispatcher can put events in an Event Queue, which is ordered according to the event priority (e.g., a timestamp or an integer identifying the importance of the event). As a consequence, the CPU scheduler needs only to look at the first entry of the Event Queue when selecting the next runnable process.

Shared Memory. ECalls uses two separate data segments per application and extension, which are pinned in memory (to prevent paging) and are accessible from both user space and kernel space. The application uses the first data segment to transfer data to the kernel extension, and the second data segment to receive data coming from the kernel extension. Memory is pinned by the kernel during application registration, which means that applications using ECalls do not require superuser privileges.

4 Event Delivery and Event Handling

We now describe the actual implementation and usage of the ECalls module. In the following we introduce the three stages that constitute event communication in ECalls: *data transfer*, *event generation*, and *event delivery*.

4.1 Data Transfer

As stated above, during registration of an application with a kernel extension, two memory areas are created and locked into memory; they are used for the exchange of data between the application and the extension. This memory's structure is described in a C header file. It defines two data structures, both having an integer value (termed *flag*) as the first entry. This integer is incremented each time an event is generated and decremented or reset to zero each time an event (or all events) are received, therefore serving as an event counter. Beyond such functionality, the precise usage of this memory segment is determined by each kernel extension. The extensions used in this paper organize the memory as a list of data entries (of any desired data types) or as a ring buffer. In the first case, the memory segment is organized as follows:

```
struct ecall_data {
   int flag;
   unsigned long bit_pattern[MAX];
   [data part]
 }
```

After the *flag* entry, an array termed *bit_pattern* holds a bit per data entry in the following data part. When an event is generated, *flag* is incremented, the event data is written into the corresponding position in the data part, and the corresponding bit in *bit_pattern* is set. The bit pattern facilitates the search for new events in the memory segment. There are three possibilities to use the memory: (1) each entry in the data part is of the same data type and new event data is put into the next available slot; (2) each entry is of the same data type as in (1), but the position of the new event data denotes its priority and therefore the sequence in which new data is read; and (3) entries in the data part have different data types and new event data is put into the corresponding entry according to its type. As an alternative, the memory segment can be structured as a ring buffer, in that case the *flag* entry is followed by a *front* and a *back* entry, pointing to the beginning and the end of the momentarily written part of the memory segment, respectively:

```
struct ecall_data {
   int flag;
   int front;
   int back;
   [data part]
 }
```

In the case of the ring buffer, new event data is put at the the end of the written part of the memory segment (indicated by *back*) as long as there is enough space.

4.2 Event Generation

The following sections describe the different methods of event generation that are available to applications and kernel extensions. Applications use *Fast User-ECalls*, *Deferred User-ECalls*, and *System Calls* to generate events. Kernel extensions generate events by calling a function supplied by ECalls, do_upcall().

4.2.1 Application-generated Events

Fast User-ECalls. Fast User-ECalls are a lightweight version of system calls, with the restriction that the invoked handler function is not allowed to block. Figure 2 shows the simplified pseudo code for both system calls and Fast User-ECalls. On return of a regular system call function, the kernel first checks for pending *bottom halves* (the *slow* part of interrupts). Next, the kernel checks if it is necessary to invoke the scheduler, and finally, the kernel looks for pending signals and invokes signal handlers if necessary. In the case of Fast User-ECalls, the default situation is to avoid these steps altogether and directly return to the user application. The return value of the short non-blocking function executed by a Fast User-ECall decides if any or all of the steps described above are required to be executed. In addition, the return value can indicate that it is necessary to turn the non-blocking Fast User-ECall into a regular (and possibly blocking) system call. In that case, the function executed upon a Fast User-ECall acts as an *optimistic* handler function, which returns immediately if the optimistic assumption that the handler function is not required to block, holds true. If the assumption fails, a regular system call is invoked. Fast User-ECalls are useful for short and simple actions such as toggling flags in the kernel, updating QoS attributes for resource managers, etc., i.e., where ordinary system calls would be too expensive.

Deferred User-ECalls. Deferred User-ECalls are similar to Fast User-ECalls in that they invoke a non-blocking handler function. However, in contrast to Fast User-ECalls, they never handle bottom halves or signals, and the invocation of the function is deferred to a later point in time. More specifically, the application can decide *when* the handler function is invoked, on a per-call basis. Possible invocation times are: (a) at return from the next system call, (b) after a certain time delay (in multiples of jiffies, i.e., 10ms on Intel computers), or (c) before the next invocation of the CPU scheduler. The advantage of Deferred User-ECalls is the reduced number of crossings of the user/kernel boundary, particularly when several events cause only one invocation of the handler function. This is useful when kernel extensions want to handle several events at once (*batched events*), or when only the most recent event is of interest to the kernel service, while in both cases a delay in handler execution does not result in a significant performance loss. As an example, an application might want to update QoS attributes in the kernel, but the updated values will not be required until the next invocation of the resource manager.

System Calls. ECalls also offers a generic system call, which takes the unique character string identifying the service as parameter. ECalls then redirects the system call to the corresponding kernel extension, which executes a system call function. This is useful when it is not desired to implement new system calls for each new kernel extension. Instead, this generic system call can be used, with any desired number of parameters, which are written into the corresponding shared memory.

System Calls	Fast User-ECalls

trap_to_kernel;
save_all_register;
call_syscall_function;
if (bottom_halves_pending)
   call_bottom_halves;
if (need_resched)
   call_scheduler;
if (signals_pending)
   call_signal_handler;
return;
	trap_to_kernel;
	save_all_registers;
	return_value = call_fast_syscall_function;
	if (return_value & run_syscall)
	   call_syscall_function;
	if (return_value & run_bottom_halves)
	   call_bottom_halves;
	if (return_value & need_resched)
	   call_scheduler;
	if (return_value & signals_pending)
	   call_signal_handler;
	return;

Figure 2: Simplified pseudo code for (a) regular system calls and (b) Fast User-ECalls. Fast User-ECalls differ from regular system calls in three ways: (1) they always invoke non-blocking handler functions, (2) they perform less tasks upon return from the handler function (per default), and (3) they can act as optimistic system calls through their ability to directly invoke a regular system call without the need to return to the user-level application.

4.2.2 Kernel-generated Events

ECalls offers kernel extensions a function, do_upcall(), which can be invoked to generate an event. do_upcall() takes the process ID of the receiving process as a parameter. This function can be called from within the context of a user-level process or by a kernel-level thread, but also within the context of an interrupt. In both cases, ECalls uses the process ID to look up the event recipient and the method of notification desired by this recipient.

4.3 Event Delivery

Upon registration, applications and kernel extensions determine how they are to be notified when events are generated. This section describes the different methods of event delivery supported in ECalls.

4.3.1 Application-generated Events

A kernel extension has to supply two functions: (1) a short non-blocking handler function, which is invoked by Fast and Deferred User-ECalls, and (2) a system call function, which is allowed to block. The system call function is invoked when (a) the application makes a call to the generic system call and (b) the return value of the non-blocking handler function (invoked by a Fast User-ECall) determines that the invocation of the system call function is required. As an alternative, the kernel service may decide to poll for User-ECalls periodically if it runs frequently enough (e.g., a kernel thread running in a loop). It does so by calling the *poll function* provided by the ECalls module, which then checks the *flag* entries in all data segments of all applications registered with the kernel service for pending events. A second alternative is to let ECalls poll the data segments periodically with a period T (provided by the kernel extension). This allows the kernel extension to be passive, in which case ECalls invokes the non-blocking handler function as soon as an event has to be delivered.

4.3.2 Kernel-generated Events

When a kernel extension generates an event, one or more of the following actions may be performed:
Polling. An application can poll for events anytime by simply checking the *flag* entry in the memory segment for updates.
Kernel Handler Function. Here, ECalls acts on behalf of the process receiving the event, by invocation of an application-specific kernel handler function. This function may (a) be supplied by the application in a separate kernel-loadable module or (b) be selected from a set of functions provided by the kernel extension itself.
User Handler Function. In this case, the process provides the handler function, which is pinned in memory to prevent paging. In Linux, to be able to pin a function in memory, superuser permissions are required.
Kernel Handler Thread. As before, a handler function has to be run, but this time ECalls takes a kernel thread from a thread pool and executes the function in the context of the kernel thread. In contrast to the two methods described above, the thread can block anytime. After the execution of the handler function has terminated, the thread is returned to the pool.
Real-Time Signals. The last method raises a real-time signal to the application. When an application registers, it also specifies which real-time signal(s) will be used. The details of the implementation of real-time signals are described in the POSIX.4 standard [9]. Currently, there are up to 64 signals supported in Linux, 32 non-real-time signals, and 32 real-time signals. Real-time signals differ from ordinary signals in that they are queued to processes, they are handled according to their priorities, and they can carry some small amount of data. In Linux, signals are handled in the following order: (1) All non-real-time signals (signals 0 to 31) and (2) all real-time signals from signal 32 (SIGRTMIN) to 63 (SIGRTMAX), giving signal 32 the highest priority among all real-time signals. Since it is possible to catch non-real-time signals (except SIGKILL and SIGSTOP) and execute user-defined handlers instead of the default action, it is possible to delay real-time signals unpredictably. Therefore, we changed the order of checking for pending signals to the following order: (1) we first check the two signals which cannot be caught (and which will either exit or stop the application): SIGKILL and SIGSTOP; (2) next, we check all real-time signals beginning with SIGRTMIN to SIGRTMAX; and (3) finally we check all remaining non-real-time signals. Using this scheme we are able to prevent a real-time signal from being delayed unpredictably by a non-real-time signal.

An application may decide to register for one or more of the methods described above, e.g., it may want to let ECalls invoke a handler function residing within the kernel and perform some tasks whose results determine if further actions (e.g. raising a real-time signal) have to be taken. This allows for *preprocessing* or *filtering* of events.

4.4 ECall-Scheduling

In most systems, CPU scheduling ignores the communication between processes on the same or between remote hosts. ECalls is able to influence scheduling decisions, i.e., processes with events pending can be favored over other processes. In addition to delivering events, ECalls is therefore able to cooperate with the CPU scheduler to minimize the delay of event delivery. For this purpose, we modified the standard Linux scheduler and also a novel real-time CPU scheduler (DWCS [19, 21, 20]). The intent is to support the integrated scheduling of tasks and events.

Event Scheduling with the Linux Scheduler. The Linux scheduler has been modified as follows. If a real-time process (a process in either the SCHED_FIFO or the SCHED_RR queue) has any Kernel-ECalls pending, it will be given preference over processes with the same or smaller priority. If only non-real-time processes are runnable (SCHED_OTHER queue), a process with Kernel-ECalls pending will always be given preference.

Event Scheduling with DWCS. To be able to efficiently support real-time processes, we use the hard real-time CPU scheduler DWCS (Dynamic Window-Constrained Scheduler), which assigns each process a period T, a runtime C, and a window-constraint x/y, meaning that a process will be scheduled y-x times for C time slots each, in a window of T*y time units. Each process can be scheduled once in a period of T, unless it is marked as *work-conserving*, in that case it is possible to schedule this process several times within a period. Each time a process gets scheduled within its period T, both the nominator and the denominator of the current window constraint are decremented and therefore the window-constraint is relaxed. On the other hand, if a process misses to be scheduled within its period T, only the nominator is decremented, resulting in a tighter window-constraint. Details about DWCS can be found in [19, 22].

The original DWCS algorithm works according to the following (simplified) rules:

- The process with the closest deadline (i.e., the time until its current period T expires) will be selected. If several processes have the same deadline, the process with the tightest current window-constraint x/y is chosen.
- If all processes have been scheduled at least once in their respective current periods, a work-conserving process will be selected according to the rules described above.
- If no real-time process is runnable, the next available best-effort task will be selected.

Elsewhere [20], we present boundaries for the worst-case delays of processes and show that we are able to guarantee schedulability of a set of processes as long as the *processor utilization U* does not exceed 100%. The following modifications minimize average event delivery delay, without violating the real-time guarantees mentioned above:

- If two or more processes have the same deadline, the process with an ECall pending is selected as long as this does not result in an immediate violation of the other process' real-time guarantees.
- If a work-conserving process has been selected that already ran once in its current period, the next process with an ECall pending is scheduled instead.
- If a best-effort task has been selected by the scheduler, the next process with an ECall pending is scheduled.

In addition, we introduce the notion of an *ECall server*, which is a pseudo task with the attributes x/y, T, and C determined in the following way: x/y = 0/YMAX with YMAX being the highest possible value for the denominator. This assigns the ECall server the tightest window constraint possible. The service time C is the same as the service time of the process with the highest priority Kernel-ECall pending or C = 1 time slice if no Kernel-ECalls are pending. The *rest utilization Ur* of the system, which is the *maximum utilization* minus the *current utilization,* is used to determine the value of the period T (T=C/Ur). Each time the ECall server becomes the highest priority task, the process with the highest priority Kernel-ECall pending is selected instead. If there are no Kernel-ECalls pending, the scheduler selects a process according to the rules of the original algorithm as described above.

5 Experimental Results and Discussion

ECalls has been implemented based on the Linux 2.2.13 kernel. The ECalls module consists of less than 1400 lines of code, where the modifications to the Linux kernel to support ECalls consist of approximately 200 additional lines of code. In the first part of this chapter, we investigate the overhead of the different methods implemented in ECalls. Then, we introduce an I/O event delivery module as an example for a kernel extension and demonstrate the utility of ECalls with two applications: a web server and a distributed video player.

5.1 Microbenchmarks

The following measurements have been performed on an AMD Athlon computer with 550Mhz and 64MB of RAM running Linux 2.2.13. The first graph in Figure 3 compares the costs of the getpid() system call ($4.9\mu s$) with the

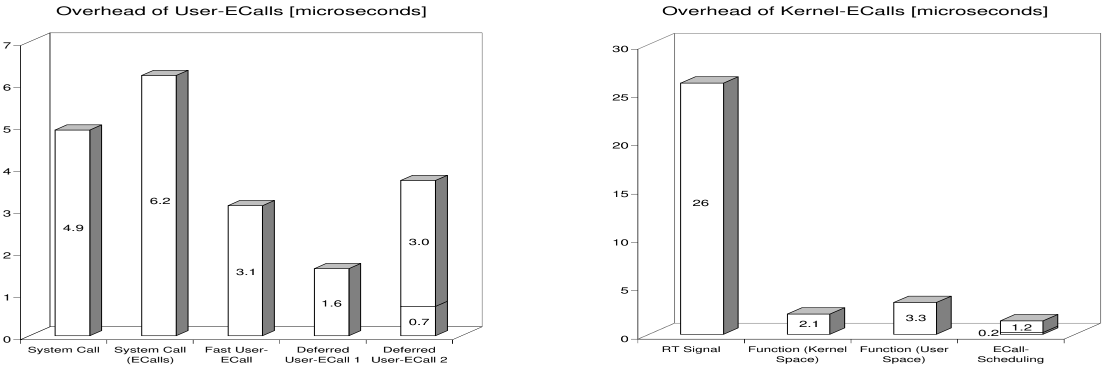

Figure 3: Overhead comparisons of (a) regular system calls with the generic system call implemented in ECalls, Fast User-ECalls, and Deferred User-ECalls and (b) real-time signals with invocation of kernel-level and user-level handler functions and ECall-Scheduling

costs of the methods offered by ECalls. The generic system call requires $6.2\mu s$, where the additional $1.3\mu s$ are mainly due to the overhead of the lookup of the handler function. Overhead increases minimally with the number of modules registered with ECalls. In the case of 200 modules, the system call requires less than 100ns more. On the other hand, Fast User-ECalls require only $3.1\mu s$ ($3.2\mu s$ with 200 registered modules), which is a gain of $1.8\mu s$ compared to standard system calls. In the case of a Deferred User-ECall, the library function to generate the event consumes approximately 700ns per event. The last two bars in the first graph in Figure 3 show the overhead for Deferred User-ECalls: In the case of event deferral until the next invocation of the scheduler, the additional kernel overhead when no new events have been generated is $1.6\mu s$. If there are one or more events pending, the additional overhead is $3\mu s$. In other words, in the case of pending events, the cost for Deferred User-ECalls is $3\mu s$ plus 700ns for each new event. For instance, the overhead is $3.7\mu s$ for one new event, $4.4\mu s$ for two and so forth. This is cheaper than the getpid() system call with $4.9\mu s$ as described above and is more than the $3.1\mu s$ for a Fast User-ECall. But since we invoke the handler function only once even when several events have been generated, this alternative is cheaper than Fast User-ECalls in situations where there are at least two events generated per handler invocation.

In the other direction (second graph in Figure 3), real-time signals require $26\mu s$, which is relatively costly. On the other hand, the invocation of a handler function residing in kernel space requires $2.1\mu s$ and in user-space $3.3\mu s$. The overhead in these cases is mostly incurred by the lookup process ECalls has to perform in order to identify the application consuming the event. To add an event to the event queue for the CPU scheduler requires 200ns, while the overhead for the scheduler to determine the next process eligible for running, increases by $1.2\mu s$ if the event queue contains one or more entries.

5.2 Implementation of an I/O Event Delivery Module

Unix systems provide select() and poll() system calls, which query a set of file descriptors passed in an array for activity. The system call returns when there is activity in at least one of these descriptors or when the system call times out. The application then has to scan a returned array to find the descriptors that are actually active. Web servers such as Zeus, Flash [23], or thttpd use the 'select' approach, which for thousands of file descriptors does not scale very well. The problem here is that the kernel has to scan the entire array each time a system call is executed.

We used ECalls to implement a scalable I/O event delivery module (termed *SockMon*) using the Linux 2.2.13 kernel. Applications register their interest in sockets via the ECall mechanism. If data arrives at one of these sockets, the registered application will be notified using one or more of the methods described in the sections above. A similar example has been presented in Banga, et al. [24], which introduces a scalable event notification mechanism to replace the select() system call, which has been shown to have poor scalability. To be able to support this notification mechanism, we added approximately 20 lines of code to the networking code inside the kernel and one additional entry into the *sock* structure, namely a flag that indicates interest for the owner of the socket.

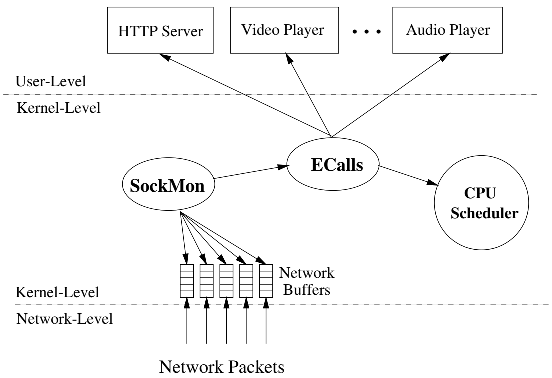

Figure 4: SockMon: An I/O Event Delivery Mechanism using ECalls

When SockMon detects activity on one of the monitored sockets, it generates an event for the application owning this socket. Each application has two data structures, which are both locked into memory and shared between the application and the event delivery mechanism. The first data structure has the following entries:

```
struct socket_interest {
   int flag;
   unsigned long fd_list[MAX];
   unsigned long updated_fd_list[MAX];
 }
```

The first entry, *flag* is incremented each time the application submits a change in interest. The next entry, *fd_list* is an array used to indicate which file descriptors the application has registered for, each bit in *fd_list* corresponds to a file descriptor. The second array, *updated_fd_list*, is used to indicate the changes in *fd_list* since the last time the registration module read from this data structure. Its purpose is to accelerate the registration process.

The second data structure looks as follows:

```
struct socket_ready {
   int flag;
   unsigned long fd_active[MAX];
 }
```

The value of *flag* indicates how many sockets are active, i.e., how many sockets have data in the receive buffer. The actual file descriptors for the active sockets can be found in the next entry, *fd_active*.

5.3 Application 1: Web Server

Our first experiment investigates the performance changes in a web server running on top of ECalls. We modified thttpd, a small and fast single-process event-driven webserver, such that it uses the I/O event delivery module described above. thttpd is a single-process web server that uses the select system call for all HTTP requests. We modified the API such that thttpd subscribes every new incoming request with the I/O event delivery module and instead of a select call, thttpd continues to service requests and selects the next connection to service via the *fd_active* array.

The client requests are generated using the httperf Version 0.8 [25] performance tool. The HTTP server is a Pentium II with 450 MHz and 512MB RAM, running our modified HTTP server. The client machines are five Sun Ultra 30 with a 248MHz processor and 128MB RAM each. The machines are connected via a switched 100Mbps Ethernet.

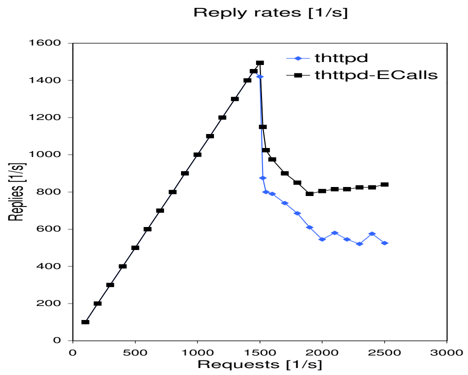

Figure 5: Reply rates and response times for both the thttpd web server and the modified thttpd web server using ECalls and the I/O event delivery module.

In this experiment, the clients request a small static web page for a duration of 180s, each request with a timeout value of 1s. The first graph in Figure 5 shows the achieved reply rates with both the original thttpd server and our modified server using ECalls (thttpd-ECalls). The web server thttpd is highly optimized, so that the difference in performance for small loads is irrelevant, as evident from the graph. On the other hand, in the case of overload, thttpd displays poor behavior: its reply rate drops to 25% at request rates of 2000 per second. The reply rate for thttpd with ECalls also decreases in the case of overload, but less dramatically (to 40% at a request rate of 2000 per second).

The second graph in Figure 5 shows the average response times for thttpd and thttpd-ECalls. Here, the response times of thttpd settle at approximately 150ms when the server is overloaded, whereas in the case of thttpd-ECalls the response time increases until it settles at approximately 300ms. The reason for this behavior is the higher reply rate of thttpd-ECalls, i.e., thttpd-ECalls is able to respond to more requests than thttpd, resulting in higher response times.

The next experiment investigates whether the use of ECalls can improve the overload behavior shown above. Provos et al. [26, 27] analyzed the overload behavior for the thttpd and phhttpd web servers. In [26], real-time signals are used to notify the web server of new requests, and the number of signals and therefore, the number of pending requests is used as indicator for server overload. The overload behavior shown in Figure 5 is referred to as *receive livelock*; Mogul, et al., suggest to drop requests as early as possible to achieve more request completions [28].

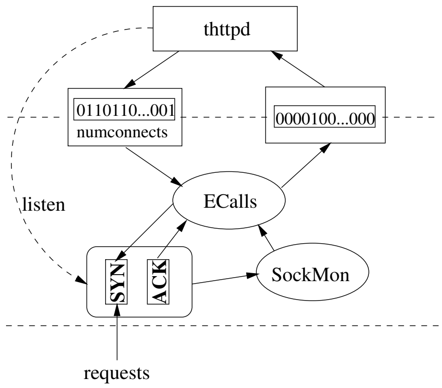

Figure 6: Modified thttpd web server using ECalls and SockMon: ECalls monitors both the number of open connections (*numconnects*) and the buffer fill level of the listen queue with completed requests (ACK-queue) to determine the size of the listen queue with incomplete requests (SYN-queue).

While in [26] requests are dropped by the server if overload is detected, we use ECalls to drop requests in the kernel. That is, we flood the server with requests, this time with a more complex web page, and monitor overload behavior, but then we improve overload behavior by using the ability of ECalls to cheaply exchange information between user- and kernel-level. Specifically, the web server continuously updates a new variable, *numconnects*, in the memory segment, telling ECalls the current number of open connections being serviced by the server. In addition, ECalls monitors the buffer fill level of the *completed connection queue* of the listening socket (ACK-queue in Figure 6). If both values (number of connections and buffer fill level) are above a certain threshold, ECalls reduces the buffer length of the *incomplete connection queue* (SYN-queue in Figure 6), until either the number of connections or the number of accepted requests drops under their respective thresholds. The reason why we use two criteria is to prevent ECalls from decreasing the queue size in case of transient overloads. As an example, if the ACK-queue is above the threshold but the number of open connections is under its threshold, we assume that the server will soon be able to service this burst of requests. On the other hand, if the number of connections is over the threshold, but the buffer fill level is under its threshold, we assume that the small number of pending requests will reduce the server load soon such that it is not necessary to drop requests.

Figure 7 shows the results of this experiment. Using ECalls, we are able to improve the overload case such that more replies are generated and at the same time the average response times are reduced. As an example, the first graph in Figure 7 shows that thttpd is able to service 80 requests per second at a request rate of 300 per second, while thttpd using ECalls is able to service 140 requests per second. The second graph indicates a 250% better response times than the original thttpd web server. The values for the thresholds have been determined via experimentation, and could also be made adaptive instead of fixed as done in this experiment. Also, more knowledge from both kernel-level and user-level can help to improve the overload case even more (such as the average response time measured in the web server or the type of a request).

5.4 Application 2: Distributed Video Player

Our second application is a distributed MPEG video player, modified such that it uses ECalls to communicate with the *SockMon* module. In contrast to the web server application, here we run a number of players on one computer acting as clients, requesting video streams from several video servers. We use the same setup as in the previous experiments. This time, the Ultra 30 computers act as video servers, and the Linux computer runs the video players, each reading and displaying a single video stream.

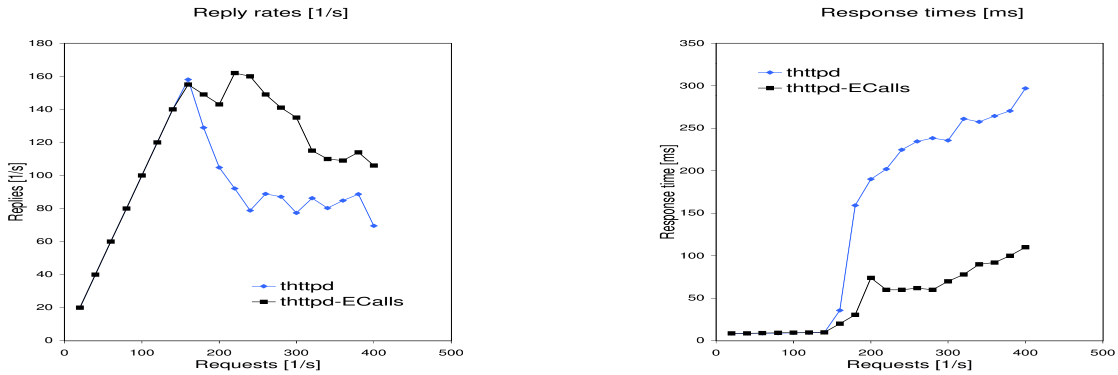

Figure 7: Graph (a) shows the improved overload behavior of the web server when ECalls modifies the size of the *incomplete connection queue* of the listening socket according to the load. Graph (b) indicates how this adaptive approach also improves the response time of requests.

The purpose of this experiment is to show that the interaction between user-level applications and kernel-level services enabled by ECalls allows these applications to achieve their desired QoS. Every video player writes the *desired frame rate* into the shared memory supplied by ECalls. This frame rate can be changed dynamically if desired (e.g., as image resolution or compression changes). Additional information placed into shared memory includes a *frame counter* and a *timestamp* of the last frame displayed. ECalls uses the timestamp and the frame rate to compute the display time of the next frame. Incoming frames are monitored by SockMon, and ECalls places the notification events into the Event Queue ordered by the display time of the next frame. The CPU scheduler uses this information to modify the scheduling priority of the video players. In this experiment all players are placed into the round-robin scheduling queue (SCHED_RR) of the Linux scheduler with time slices of 150μs, each process with the same priority, i.e., all players operate at the same priority level. Using ECall-Scheduler, the players with the shortest deadlines are given preference by the scheduler.

Graph (a) in Figure 8 shows the behavior of the video players, where each video player displays a new image and then sleeps for a period of time, determined by the desired frame rate. The graph shows that the achieved frame rates are close to the desired rates until we increase the number of video players to 4. After this point, the CPU is overloaded, and all players receive the same frame rates. Graph (b) in Figure 8 shows the behavior of the video players when we use ECalls to delay event notification for a period of time determined by the frame rate and to inform the scheduler to favor players with events pending, resulting in better QoS achieved, even in overload scenarios. Using ECalls we are able to achieve better quality of service, because video players that become runnable again, are not necessarily being placed at the end of the run queue. Instead, ECall-Scheduling favors an application if this application has one or more events pending.

6 Summary and Future Work

Extensibility in operating systems is key to flexibility and configurability. We propose ECalls to support the implementation of kernel services in extension modules, by supplying a flexible interface for both real-time and best-effort applications. ECalls transports events both from user to kernel space and vice versa. For events originating in user space, two lightweight system calls may be used, termed *Fast User-ECalls* and *Deferred User-*

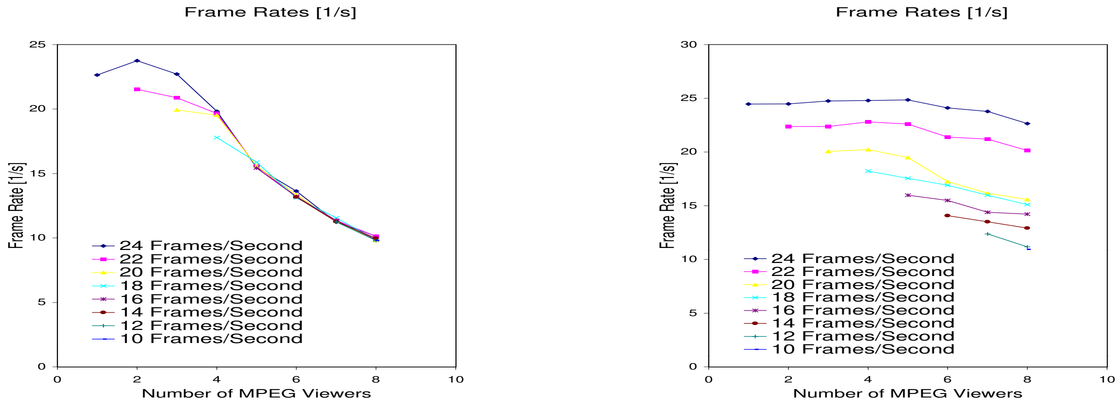

Figure 8: Graph (a) shows the achieved frame rates of the video applications for the original implementation, (b) shows the frame rates achieved with the support of ECalls.

ECalls. Their use aims to reduce the frequency and cost of event communication and context switches. For events originating in the kernel (in addition to providing low latency upcall implementations) ECalls can also provide real-time guarantees for events. This functionality utilizes a modification of the standard Linux scheduler or it utilizes a novel hard real-time scheduler, DWCS. In either case, we can give preference to processes with pending events, therefore improving the average case delay. Using DWCS, we are able to provide the same hard real-time guarantees under worst-case scenarios as those described in [20].

In general, ECall-Scheduling can enhance the CPU scheduler with knowledge about event reception, e.g., which processes have events pending, and it influences scheduling decisions correspondingly. This functionality builds on and extends earlier work. For example, in [29], Manimaran et al. propose an integrated framework for interacting process and message schedulers for distributed real-time systems. Similarly, Lee et al. [30] introduce an architecture that is aware of the real-time characteristics of tasks sending and receiving network packets. The goal is to overcome the traditional deficiencies like FIFO ordering of incoming packets and processing in the kernel of all packets regardless of their priority to the receiving application.

Andersen et al. [31] describe an kernel module, the Congestion Manager (CM), an architecture that allows for sharing of congestion information between multiple concurrent flows. In CM, a select call on a special socket (*control socket*) is used to transfer information between kernel and application.

Several proposals try to minimize data copies, including the work in [2] and [32], where the latter aims at the efficient transfer of data and control between user and kernel space and also provides rate-based flow control. This is achieved by using I/O efficient buffers and independently scheduled kernel threads. Banga, Mogul, and Druschel [24] introduce an event delivery system allowing applications to register interest in event sources like sockets. However, the application still has to poll for events, whereas ECalls is able to notify a process of pending events by executing a handler function and raising its scheduler priority. Finally, Pai et al. [23] introduce a new flexible and general I/O approach that avoids data copying with minimal overhead.

Future Work. Kernel extensions allow applications to specialize kernel functionality and achieve better performance, but they also introduce new threats to system security and reliability. Small and Seltzer [33] consider three methods of ensuring safety of the kernel: hardware protection, software protection, or the use of a run-time code generator (interpretation). In this paper, we concentrated on the flexible event communication supported by ECalls, protection is the topic of future work. The implementation of protection techniques will include run-time checks for the execution of kernel-level and user-level functions such that long running functions are either interrupted and continued in the context of a kernel thread or aborted. Also, memory access checks will ensure that functions (especially user space functions) do not access or modify memory segments of other processes. In addition, we are working on a mechanism to ensure that recursive calls between applications and kernel extensions are prevented.

ECalls is able to collect information from both user and kernel space to improve kernel performance. We intend to use ECalls to monitor resource allocations like network (bandwidth, loss rates, round trip times), CPU, or memory, which can be combined to generate useful information which will be pushed into user space. Another role of ECalls to be explored in future papers is its use for adaptive resource management, offering applications a flexible mechanism to pass resource management events to the kernel and receive events from the kernel, with overheads and associated functionalities acceptable to the different resource management actions being taken.

References

[1] J. Liedtke, "Improving IPC by Kernel Design," in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1993.

[2] P. Druschel and L. Peterson, "Fbufs: A High-bandwidth Cross-domain Transfer Facility," in *Proc. of the Fourteenth ACM Symposium of Operating Systems Principles*, 1993.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility, Safety and Performance in the SPIN Operating System," in *Proceedings of the* 15^{th} *ACM Symposium on Operating System Principles*, December 1995.

[4] D. R. Engler, F. M. Kaashoek, and J. O'Toole Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management, booktitle = Proceedings of the 15^{th} ACM Symposium on Operating System Principles, year = 1995, month = December,,"

[5] D. Ghormley, S. Rodrigues, D. Petrou, and T. Anderson, "SLIC: An Extensibility System for Commodity Operating Systems," in *Proceedings of the 1998 USENIX Annual Technical Conference*, June 1998.

[6] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," in *Proceedings of the* 13^{th} *ACM Symposium on Operating System Principles*, pp. 95–109, December 1991.

[7] D. D. Clark, "The Structuring of Systems Using Upcalls," in *Proceedings of the* 10^{th} *ACM Symposium on Operating Systems Principles*, pp. 171–180, December 1985.

[8] N. Hutchinson and L. Peterson, "The x-Kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, vol. 17, pp. 64–76, January 1991.

[9] *POSIX.4: Programming for the Real World.* O'Reilly, 1995.

[10] R. Gopalakrishnan and G. Parulkar, "Efficient User Space Protocol Implementations with QoS Guarantees using Real-time Upcalls," *IEEE/ACM Transactions on Networking*, 1998.

[11] R. Gopalakrishnan and G. Parulkar, "Real-time Upcalls: A Mechanism to Provide Real-Time Processing Guarantees," Tech. Rep. WUCS-9506, Department of Computer Science, Washington University, 1995.

[12] D. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole, "A Feedback-Driven Proportion Allocator for Real-Rate Scheduling," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, February 1999.

[13] R. Kravets, K. Calvert, and K. Schwan, "Payoff Adaptation of Communication for Distributed Interactive Applications," in *The Journal for High Speed Networking: Special Issue on Multimedia Networking*, Winter 1999.

[14] K. Schwan and W. Bo, "Topologies - Distributed Objects on Multicomputers," *ACM Transactions on Computer Systems*, vol. 8, pp. 111–157, May 1990.

[15] D. Rosu, K. Schwan, S. Yalamanchili, and R. Jha, "On Adaptive Resource Allocation for Complex Real-Time Applications," in *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS)*, (San Francisco, USA), December 1997.

[16] J. C. Brustoloni and P. Steenkiste, "Evaluation of Data Passing and Scheduling Avoidance," in *Proc. 7th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, 1997.

[17] R. West and K. Schwan, "Experimentation with Event-based Methods of Adaptive Quality of Service Management," Tech. Rep. GIT-CC-99-25, Georgia Institute of Technology, 1999.

[18] C. Poellabauer, K. Schwan, R. West, I. Ganev, N. Bright, and G. Losik, "Flexible User/Kernel Communication for Real-Time Applications in ELinux," in *Workshop on Real Time Operating Systems and Applications and Second Real Time Linux Workshop*, November 2000.

[19] R. West and K. Schwan, "Dynamic Window-Constrained Scheduling for Multimedia Applications," in *6th International Conference on Multimedia Computing and Systems, ICMCS'99*, IEEE, June 1999. Also available as a Technical Report: GIT-CC-98-18, Georgia Institute of Technology.

[20] R. West and C. Poellabauer, "Analysis of a Window-Constrained Scheduler for Real-Time and Best-Effort Packet Streams," in *Proceedings of the 21st IEEE Real-Time Systems Symposium*, (Orlando), IEEE, November 2000.

[21] R. West, K. Schwan, and C. Poellabauer, "Scalable Scheduling Support for Loss and Delay Constrained Media Streams," in *Proc. 5th Real-Time Technology and Applications Symposium*, (Vancouver, Canada), 1999.

[22] R. West, K. Schwan, and C. Poellabauer, "Scalable Scheduling Support for Loss and Delay Constrained Media Streams," Tech. Rep. GIT-CC-98-29, Georgia Institute of Technology, 1998. Fifth IEEE Real-Time Technology and Applications Symposium, June 1999, Vancouver, Canada.

[23] V. Pai, P. Druschel, and W. Zwaenepoel, "IO-Lite: A Unified Buffering and Caching System," in *3rd Symposium on Operating Systems Design and Implementation (OSDI '99) Proceedings*, February 1999.

[24] G. Banga, J. Mogul, and P. Druschel, "A Scalable and Explicit Event Delivery Mechanism for UNIX," in *Proc. USENIX Annual Technical Conference*, 1999.

[25] D. Mosberger and T. Jin, "httperf - A Tool for Measuring Web Server Performance," in *Workshop on Internet Server Performance*, June 1998.

[26] N. Provos and C. Lever, "Scalable Network I/O in Linux," in *FREENIX track Proc. of the USENIX Annual Technical Conference*, June 2000.

[27] N. Provos, C. Lever, and S. Tweedie, "Analyzing the Overload Behavior of a Simple Web Server," in *Proc. of the 4th Annual Linux Showcase and Conference*, October 2000.

[28] J. Mogul and K. Ramakrishnan, "Eliminating Receive Livelock in an Interrupt-driven Kernel," in *Proc. of USENIX Technical Conference*, January 1996.

[29] G. Manimaran, M. Shashidhar, A. Manikutty, and C. Murthy, "Integrated Scheduling of Tasks and Messages in Distributed Real-time Systems," in *IEEE Workshop on Parallel and Distributed Real-time Systems*.

[30] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar, "Predictable Communication Protocol Processing in Real-Time Mach," in *Proceedings of the Real-time Technology and Applications Symposium*, June 1996.

[31] D. Andersen, D. Bansal, D. Curtis, S. Seshan, and H. Balakrishnan, "System Support and Content Adaptation in Internet Applications," in *4th Symposium on Operating Systems Design and Implementation*, October 2000.

[32] D. Yau and S. Lam, "An Architecture Towards Efficient OS Support for Distributed Multimedia," in *Proc. IS&T/SPIE Multimedia Computing and Networking Conference*, January 1996.

[33] C. Small and M. Seltzer, "A Comparison of OS Extension Technologies," in *USENIX 1996 Annual Technical Conference*, January 1996.