Lightweight Kernel/User Communication for Real-Time and Multimedia Applications

Christian Poellabauer
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
chris@cc.gatech.edu

Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
schwan@cc.gatech.edu

Richard West
Computer Science Department
Boston University
Boston, MA 02215
richwest@cs.bu.edu

ABSTRACT

Operating system enhancements to support real-time and multimedia applications often include specializations and extensions of kernel functionality, as with the kernel HTTP daemon (khttpd) in Linux, for instance. To enable efficient and flexible interactions of such extensions with user-level functionality, we have developed ECalls, a lightweight, bidirectional kernel/user event delivery facility, which not only supports the timely delivery of events, but it also reduces the cost and frequency of kernel/user 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 its utility, this paper implements an I/O event delivery mechanism using ECalls. This mechanism is shown to improve the performance of two applications: a distributed video player and a web server.

1. INTRODUCTION

Multimedia and real-time applications often require support from the underlying operating system to achieve their real-time and QoS guarantees. This has led to the development of operating system services that are responsible for process scheduling [18, 21, 23] and resource management tasks [13, 27]. When using such kernel-level services, applications interact with them via system calls and/or signals. Since such interactions can be costly, researchers have sought ways to control call overheads [7, 16], and they have attempted to reduce the frequency of system calls, e.g., by extending kernels with appropriate application-specific functionality [4, 8, 9]. In addition, to better integrate the kernel- and user-level actions carried out for certain requests, efficient upcall primitives have been developed [6, 12], with real-time variants addressing the specific needs of multimedia and real-time applications [1, 10, 11].

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 [30], 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 [14] or even scientific applications [28]. 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 [26].

This paper presents *ECalls (Event Calls)*, a novel mechanism for coordinating kernel/user actions and for sharing information between both. Event-based approaches to communication have been proven useful in many areas, e.g., in GUI environments, virtual environments, or active database systems. ECalls is an event mechanism that offers different methods for the linkage of the execution of certain kernel services with corresponding user-level functionality. In addition, unprivileged instructions use ECalls to access the services offered by kernel extensions, thereby supporting efficient kernel/user interactions for both statically and dynamically created kernel functionality. Finally, by separating its facilities for information sharing versus coordination across the user/kernel boundary, ECalls provides degrees of flexibility in both data and control passing not offered by the system call facilities existing in general purpose operating systems [5].

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 prescribe 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 [11, 33].

The results shown in this paper exploit ECalls' ability to exchange lightweight events between applications and kernel, and to share application-level information with the kernel, based on which the kernel can change its provided service. In addition, we alter the kernel-level scheduling of applications in response to the communications they receive. Using ECalls, multiple applications that receive and play out video streams are scheduled such that desired levels of quality are maintained even in overload situations. Our measurements demonstrate that in high load scenarios, the cooperative application-kernel behavior supported by ECalls is superior to simpler solutions in which applications attempt to self-control their execution via timed waits (e.g., the Realplayer[2] approach). In other measurements, we have attained 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. Using ECalls we have observed improved 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. These actions can improve throughput by more than 50% and response times by more than 200%.

[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.
[2]http://www.realplayer.com

2. ECALLS SOFTWARE ARCHITECTURE AND FUNCTIONALITY

2.1 Software Architecture

The ECalls mechanism – simply referred to as *ECalls* in the remainder of this paper – has been implemented as a kernel-loadable module in Linux 2.2.13 and allows user-level applications and kernel-level functionality to exchange events and to share information. Events are raised via the *Event Raiser* (see Figure 1), and they are delivered via the *Event Dispatcher*. A kernel extension registers with ECalls

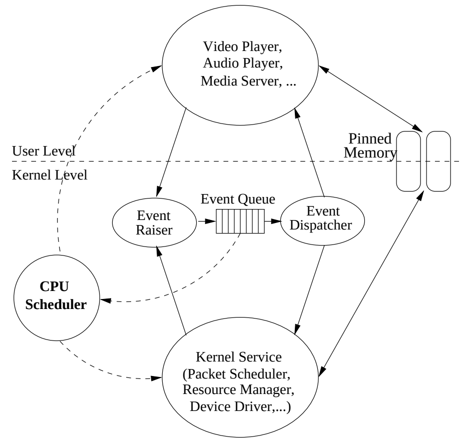

Figure 1: Architecture of ECalls.

(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. Once a kernel extension has registered with ECalls, applications are free to register for these kernel services (using the function ECalls_register()). Along with the registration, an application also determines *how* it wishes to be notified of events generated by the kernel extension. 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. 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., determined by the application or kernel service raising this 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. Finally, 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. Unless accessed by multiple threads, memory accesses need not be synchronized, since each memory area is shared between only one reader and one writer (i.e., an application and a certain kernel functionality).

2.2 Basic Functionality

The following sections describe the multiple interaction models supported in ECalls, which are useful for efficient user/kernel and kernel/user interactions for real-time and multimedia applications.

2.3 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.

2.3.1 User-to-Kernel 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 ser-

System Calls

```
trap_to_kernel;
save_all_registers;
call_syscall_function;
if (bottom_halves_pending)
    call_bottom_halves;
if (need_resched)
    call_scheduler;
if (signals_pending)
    call_signal_handler;
return;
```

Fast User−ECalls

```
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.

vice as parameter. ECalls then redirects the system call to the corresponding kernel extension, which executes a system call function (useful when it is not desired to implement new system calls for each new kernel extension).

2.3.2 *Kernel-to-User Events*

A common problem in multimedia (and in communication) systems is the need for user-level applications to be notified of important kernel-level events. A kernel extension can use ECalls to generate an event for an application, which may trigger one or more of the following actions:

Kernel Handler Function. ECalls can invoke handler functions (residing in the kernel) on behalf of the application receiving the event. This function can be provided by the kernel service itself, as well as by the application, e.g., in a kernel-loadable module.

User Handler Function. ECalls can invoke a handler function, which resides in user space and is pinned in memory to prevent paging (only available to privileged applications in Linux).

Kernel Handler Thread. ECalls can execute a function in the context of a kernel thread, which is allowed to block (unlike Kernel and User Handler Functions).

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 [1]. 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.

These actions can be combined, e.g., an event may cause the execution of a Kernel or User Handler Function, which then decides if a Real-Time Signal to the applications has to be raised. This allows for *preprocessing* or *filtering* of events.

2.4 ECalls-based CPU Scheduling

Using ECalls, the scheduling of applications can be influenced by kernel-level knowledge and/or by application-level knowledge made available to the kernel via ECalls' pinned memory areas. Past work has shown that such coordinated kernel/user behavior is particularly important for real-time and multimedia applications [11, 33]. The example of coordinated behavior explored in this paper concerns multiple applications receiving video frames to be played out. By associating events with such communications, scheduling may be changed according to the desired event rates and the number of events pending for applications. In the current implementation, ECalls supports the cooperation with both the standard Linux scheduler and a novel real-time CPU scheduler based on the DWCS scheduling algorithm [32].

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 (processes in the 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 numerator 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 numerator is decremented, resulting in a tighter window-constraint. Details about DWCS can be found in [32, 34].

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 [31], 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.

2.5 Protection

Although protection is not addressed in this work and will be topic of our future work, it is important to note that we do not expect protection mechanisms to significantly affect the performance and functionality of the interaction models supported in ECalls. The provision of safe extensions will be mainly supported at compilation time (i.e., as a one-time operation) through compiler-level safety checks (including pointer analysis), memory bounds, and type checking.

3. EXPERIMENTAL RESULTS AND DISCUSSION

3.1 Microbenchmarks

The measurements shown in Table 1 have been performed on an AMD Athlon computer with 550Mhz and 64MB of RAM running Linux 2.2.13 with ECalls support. All calls from user-level to kernel-level in this evaluation return the process ID of the calling process and all calls from kernel-level to user-level look up a certain address in memory and return its content. Fast User-ECalls require only 3.1μs, which is a gain of 1.8μs compared to standard system calls. The library function to generate a Deferred User-ECall consumes 700ns per event and each time the scheduler is invoked we have an additional kernel overhead of either 1.6μs (no events pending) or 3.0μs (one or more events pending). In other words, the costs for Deferred User-ECalls is 3μs plus 700ns for each new event, which is slightly more than those experienced with a comparative implementation using Fast User-ECalls. However, since the handler function is invoked only once even when several events have been generated, using Deferred User-ECalls is preferable to using Fast User-ECalls in high load situations. The ECalls-based implementation of coordinating user threads with communications is far better than one using standard Linux facilities like real-time signals. Real-time signals require 26μs, whereas the invocation of a handler function residing in kernel space requires 2.1μs and in user-space 3.3μs. To add an event to the event queue requires 200ns, while the scheduler overhead itself increases by 1.2μs if the event queue contains one or more entries.

3.2 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 [22], 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 (called *I/O module* in the following) 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. [3], which introduces a scalable event notification mechanism to

Table 1: Overhead of ECalls (in microseconds).

User-to-Kernel Communication	μs	Kernel-to-User Communication	μs
System Call (Linux)	4.9	Real-Time Signal	26.0
System Call (ECalls)	6.2	Kernel Handler Function	2.1
Fast User-ECall	3.1	User Handler Function	3.3
Deferred User-ECall (library call)	0.7	Event Queue Management	0.2
Deferred User-ECall (w/o events)	1.6	ECalls-based CPU Scheduling	1.2
Deferred User-ECall (w/ events)	3.0		

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.

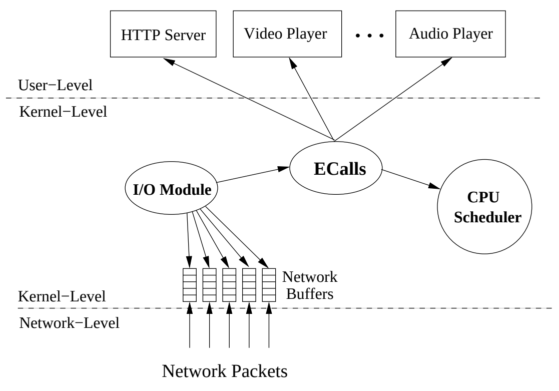

Figure 3: Notification of Socket Activity with an I/O Event Delivery Mechanism and ECalls

When the I/O module 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, called *flag*, is incremented each time the application submits a change in interest. The next entry, called *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, called *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, called *fd_active*.

3.3 Experiments with a Distributed Video Player

ECalls provides a flexible mechanism for coordination and information sharing for real-time and multimedia applications that use certain kernel services or that extend kernels with application-specific functions. We modified a distributed MPEG video player such that it uses ECalls to communicate with the I/O module describe above. In this experiment, a number of video players (running on a Pentium II with 450MHz and 512MB RAM) request video streams from several video servers (running on five Ultra 30 with 248MHz and 128MB RAM each).

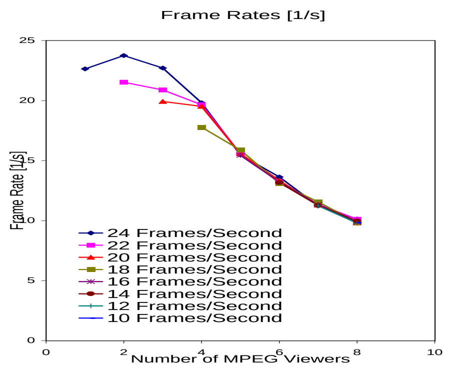

Figure 4: Achieved Frame Rates without ECalls.

Each video player writes the *desired frame rate*, a *frame counter*, and the *time stamp* of the last displayed frame into the pinned memory supplied by ECalls. The frame rate can be changed dynamically if desired (e.g., as image resolution or compression changes). The I/O module uses this information to compute the display time of the next frame. Incoming frames are monitored by the I/O module, 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

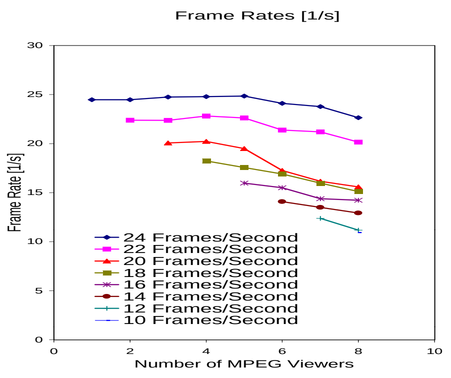

Figure 5: Achieved Frame Rates with ECalls.

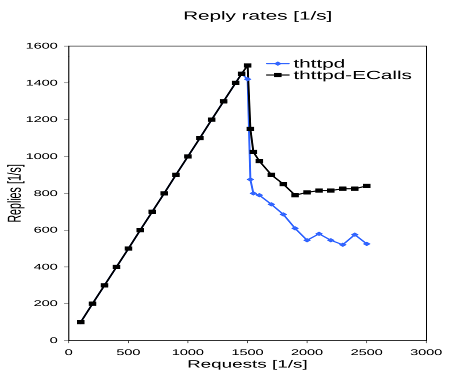

Figure 6: Reply rates for both the thttpd web server and the modified thttpd web server using ECalls and the I/O event delivery module.

the same priority, i.e., all players operate at the same priority level (future experiments will use DWCS [32] as CPU scheduler). Using ECalls-based CPU Scheduling, the players with the shortest deadlines are given preference by the scheduler. This experiment shows that the interaction between an application and a kernel-level service (using ECalls) allows the application to achieve its desired QoS. Figure 4 shows that the achieved frame rates drop rapidly when the number of players increases. Using ECalls we are able to maintain frame rates close to the desired frame rates (Figure 5). ECalls achieves that by delaying event notification for a period of time determined by the frame rate and by influencing the scheduling decisions such that the scheduler reorders the run queue to favor applications receiving these events.

3.4 Experiments with a Web Server

Our next experiments investigate 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 [20] performance tool. The HTTP server is a Pentium II with 450 MHz and 512MB RAM, running our modified thttpd application. 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.

In this experiment, the clients request a small static web page for a duration of 180s, each request with a timeout value of 1s. Figure 6 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).

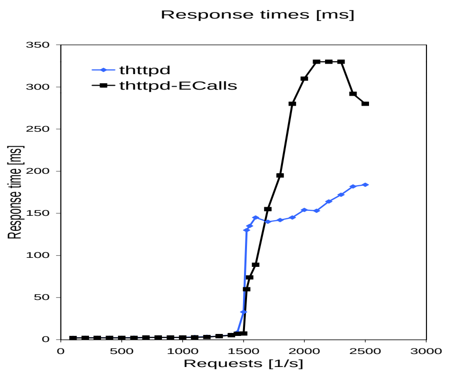

Figure 7: Response times for both the thttpd web server and the modified thttpd web server using ECalls and the I/O event delivery module.

Figure 7 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. [24, 25] analyzed the overload behavior for the thttpd and phhttpd web servers. In [24], 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

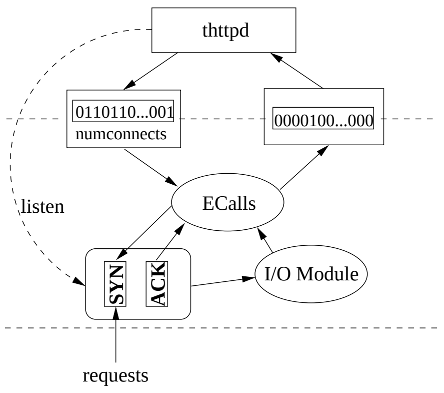

Figure 8: Modified thttpd web server using ECalls and the I/O module: ECalls monitors both the number of open connections (*numconnects*) and the buffer fill level of the listen queue with completed requests (ACK-queue) to determine the size of the listen queue with incomplete requests (SYN-queue).

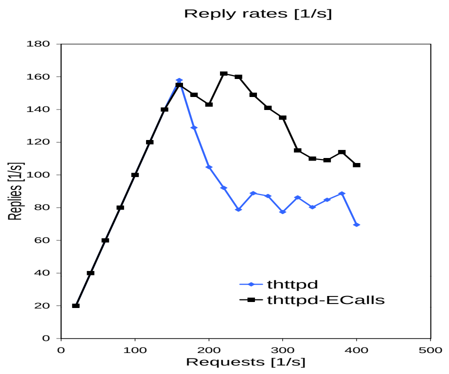

Figure 9: This graph shows the improved overload behavior of the web server when ECalls modifies the size of the *incomplete connection queue* of the listening socket according to the load.

used as indicator for server overload. The overload behavior shown in Figure 6 is referred to as *receive livelock*; Mogul et al. suggest to drop requests as early as possible to achieve more request completions [19]. While in [24] 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 8). 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 8), 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 9 and Figure 10 show 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, Figure 9 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. Figure 10 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).

4. CONCLUSIONS

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-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 [31].

In general, ECall-Scheduling can enhance the CPU scheduler with knowledge about event receipt, e.g., which processes have events pending, and it influences scheduling decisions correspondingly. This functionality builds on and extends earlier work. For example, in [17], Manimaran et al. propose an integrated framework for interacting process and message schedulers for distributed real-time systems. Similarly, Lee et al. [15] 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.

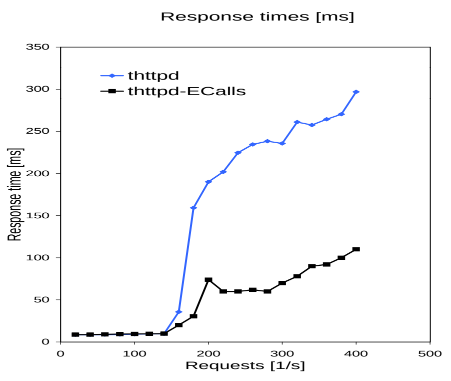

Figure 10: This graph indicates how this adaptive approach also improves the response time of requests.

Andersen et al. [2] 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 [7] and [35], 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 [3] 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. [22] introduce a new flexible and general I/O approach that avoids data copying with minimal overhead.

Future Work. Kernel extensions allow applications to specialize certain kernel functionality and achieve better performance, but they also introduce new threats to system security and reliability. Small and Seltzer [29] 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.

5. REFERENCES

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

[2] 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.

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

[4] 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.

[5] 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.

[6] D. Clark. The Structuring of Systems Using Upcalls. In *Proceedings of Tenth ACM Symposium on Operating Systems Principles*, pages 171–180. ACM, December 1985.

[7] 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.

[8] D. R. Engler, F. M. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.

[9] 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.

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

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

[12] N. Hutchinson and L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*,

17(1):64–76, January 1991.

[13] M. Jones. Adaptive Real-Time Resource Management Supporting Modular Composition of Digital Multimedia Services. In *Proc. of the 14th Int. Workshop on Network and Operating System Support for Digital Audio and Video*, November 1993.

[14] 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.

[15] 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.

[16] J. Liedtke. Improving IPC by Kernel Design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993.

[17] 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.*

[18] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reservation for Multimedia Operating Systems. In *IEEE International Conference on Multimedia Computing and Systems*, pages 90–99, May 1994.

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

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

[21] J. Nieh and M. Lam. SMART: A Processor Scheduler for Multimedia Applications. In *Proc. of SOSP 15*, December 1995.

[22] 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.

[23] X. G. Pawan Goyal and H. M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *2nd Symposium on Operating Systems Design and Implementation*, pages 107–121. USENIX, 1996.

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

[25] 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.

[26] 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.

[27] L. Schreier and M. Davis. System-Level Resource Management for Network-Based Multimedia Applications. In *Proc. of the 5th Int. Workshop on Network and Operating System Support for Digital Audio and Video*, 1995.

[28] K. Schwan and W. Bo. Topologies - Distributed Objects on Multicomputers. *ACM Transactions on Computer Systems*, 8(2):111–157, May 1990.

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

[30] 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.

[31] 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, November 2000. IEEE.

[32] 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.

[33] R. West and K. Schwan. Quality Events: A Flexible Mechanism for Quality of Service Management. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, May 2001.

[34] 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.

[35] 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.