Coordinated CPU and Event Scheduling for Distributed Multimedia Applications

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

Distributed multimedia applications require support from the underlying operating system to achieve and maintain their desired Quality of Service (QoS). This has led to the creation of novel task and message schedulers and to the development of QoS mechanisms that allow applications to explicitly interact with relevant operating system services. However, the task scheduling techniques developed to date are not well equipped to take advantage of such interactions. As a result, important events such as position update messages in virtual environments may be ignored. If a CPU scheduler ignores these events, players will experience a lack of responsiveness or even inconsistencies in the virtual world. This paper argues that real-time and multimedia applications can benefit from coordinated schemes for CPU and event scheduling and we describe a novel event delivery mechanism, termed ECalls, that supports such coordination. We then use ECalls to reduce variations in inter-frame times for media streams, and to reduce event response times for real-time applications like distributed virtual worlds.

1 Introduction

The achievable end-to-end Quality of Service (QoS) for distributed multimedia applications depends on the servers, networks, and end systems involved in media manipulation. This paper addresses the end (or client) systems, for which it is well known that a lack of QoS support from their respective operating systems can be a detriment to the efficient implementation of applications like video conferencing, distributed virtual environments, or video-on-demand. For instance, digital audio and video must be played out continuously, where latency, data loss, and jitter [1] are bound in order to prevent audible gaps in audio streams or choppy replay of video streams. While buffering can help solve these problems for play-out of stored media, this solution is not easily applied to real-time multimedia applications like distributed virtual worlds and multi-player games, which have to exchange position updates with other participants within time bounds determined by player and game speeds.

The demanding requirements of multimedia applications have been addressed by numerous research contributions, including the development of architectures that improve the performance of existing operating systems [2, 3, 4], efficient OS support for digital audio and video [5, 6], novel multimedia and real-time schedulers [7, 8], and lightweight mechanisms for user/kernel communication [9, 10, 11]. Considerable research has focused on scheduling techniques for multimedia systems [5, 6, 12, 8] based on priorities, reservations, or proportional share resource allocations [13].

This paper proposes to extend multimedia scheduling by also explicitly taking certain events of importance to applications into account. Examples of events are notifications of changes in kernel state like the receipt of a network packet, a response by a kernel service to a request issued earlier by the application, or an exception experienced within the kernel. We show that the timely delivery and processing of such events is particularly important for time-constrained multimedia applications like virtual environments, interactive distributed simulations [14], or distributed games that integrate streaming audio and video [15, 16]. Consider, for instance, a distributed game for which (1) jitter in the replay of media streams should be minimized, and (2) game events like position updates and certain actions of avatars must be delivered in a timely fashion. Techniques like proportional share scheduling of tasks and communications can reduce jitter for continuous media streams. However, the coordination of task scheduling with important game events can further reduce variations in inter-frame times and increase responsiveness to player actions.

Our goal is to improve the performance of multimedia and real-time applications by making these applications and the system services they use *event-aware*. Toward this end, we (1) provide an efficient operating system mechanism, termed ECalls (Event Calls) [11], for the exchange of events between applications and the OS services they utilize, and (2) develop policies for coordinating the scheduling of event delivery with the scheduling of operating system services. The particular service addressed in this paper is task scheduling.

To illustrate the performance advantages derived from event-awareness realized with ECalls, consider a distributed video player (e.g., Realplayer[1]), which uses *timed waits* to achieve the *inter-frame times* necessary for its desired frame rates. In other words, this application *sleeps* for a certain amount of time, and when it *wakes up*, it is placed back onto the run queue of the CPU scheduler. However, the delay between the point when this application becomes schedulable (i.e., wakes up) and when it begins to run (i.e., enters the 'running state') (see Figure 1) varies depending on the scheduling policy implemented, the scheduling attributes assigned to this and other schedulable applications, and the current CPU load. These delays – termed *run queue delays* in the remainder of this paper –

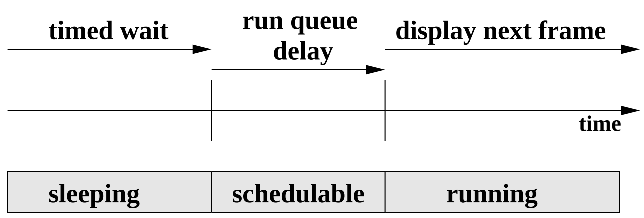

Figure 1: Run queue delays depend on the CPU scheduler, the scheduling attributes of all schedulable applications, and the CPU load.

can increase latencies and jitters for continuous media streams, and they can reduce the responsiveness of real-time applications like distributed games. For instance, when running on a general-purpose operating system like Linux, even a single video player experiences run queue delays due to the coarse granularity of the system's time base, which is 10ms on Intel-based Linux systems. When it has to compete with other video players for the same CPU, run queue delays increase substantially, resulting in significant variations in inter-frame times even for a small number of video players, as shown in Figure 2 below.

We reduce run queue delays by associating *application-level events* like position update messages with *kernel-level events* that act to minimize run-time delays for the tasks that process these messages. Specifically, the receipt of a position update message triggers a kernel-level event, which then uses the ECalls mechanism to notify the application and also to adjust CPU scheduling to ensure the timely delivery of the application-level event. We use ECalls in place of the comparatively expensive user/kernel communication mechanisms currently supported in general-purpose operating systems [9, 17, 18, 4] because it (a) offers lightweight alternatives to the user/kernel and kernel/user communications currently offered in Linux, such as system calls and signals, (b) reduces the number of

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

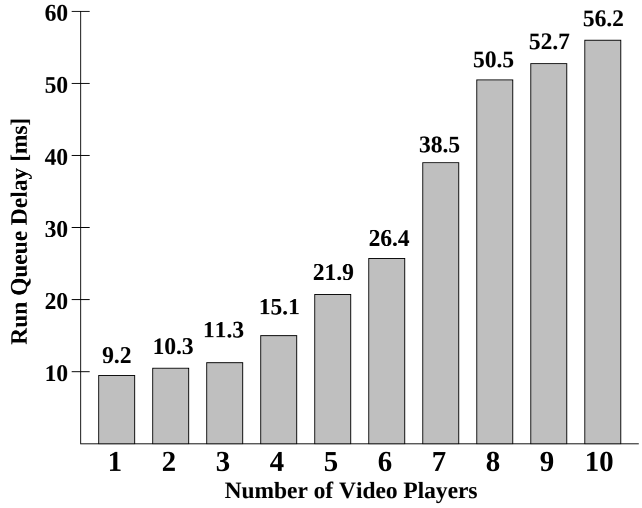

Figure 2: Run queue delays and consequently inter-frame times increase with the number of video players competing for the same processor.

user/kernel boundary crossings through *event batching*, (c) allows for the sharing of application-specific information between kernel and user via pinned memory areas, and (d) can affect CPU scheduling decisions.

In the remainder of this paper, we first provide an overview of the ECalls mechanism and its realization for the Linux operating system (see Section 2). In Section 3, we describe the rules used to coordinate event delivery with CPU scheduling in detail, using a novel hard real-time CPU scheduler, called DWCS (Dynamic Window-Constrained Scheduler) [19]. Coordinated scheduling is performed such that task responsiveness to events is maximized while minimizing the variations of inter-frame times experienced by the application's media streams. Section 4 demonstrates experimentally the importance of event-awareness and shows how ECalls supports its implementation. Section 5 concludes the paper with a summary and an overview of future work.

2 Event Delivery with ECalls

ECalls is an event-based mechanism that links user-space applications with certain kernel-level services, allowing them to share information and events in an inexpensive and flexible way. ECalls is implemented as an extension to the Linux 2.2.13 kernel, and it supports real-time and multimedia applications by (1) delivering events between applications and kernel services in a timely fashion, (2) enabling both to efficiently share relevant information, and (3) influencing process scheduling in response to the receipt of new events and/or pending events.

ECalls offers several methods of event generation and event handling, thereby giving event sources and sinks the flexibility to select a method that is appropriate for the specific needs of an application. For instance, an event source can choose the method of event generation on a **per-event** basis. An event sink must register for a certain way of event handling, but can change this registration when desired. For the efficient transfer of data between kernel and user, ECalls allocates two separate memory buffers for each pair of application and kernel service. One buffer is used for data transfer from the application to the kernel, the other for data transfer in the opposite direction, thereby reducing the need for synchronization for single-threaded applications. The sizes and contents of these memory buffers are determined by the kernel service, which supplies this information to the application via a C header file. Buffers are pinned into memory to prevent paging.

User-to-kernel events. An application uses one or more of the following methods to generate an event to a kernel service:

(a) Generic System Calls
(b) Fast User-ECalls
(c) Deferred User-ECalls

A *Generic System Call* can be used to produce an event directed at any existing or new kernel service. The memory buffer is used to transmit the system call attributes, including an identifier for the kernel service targeted by the system call. ECalls then uses this identifier to redirect the system call to the corresponding kernel service, which, in response, invokes a system call function. This removes the need to implement new system calls each time a new kernel service is added to the operating system. *Fast User-ECalls* and *Deferred User-ECalls* are both lightweight versions of traditional system calls, i.e., they reduce call overheads by minimizing the actions performed when returning from the call function, which include signal handling, interrupt handling, and a possible scheduler invocation. Both calls are intended for simple short-running and non-blocking kernel calls, such as updating QoS parameters or polling status flags in the kernel. The overhead for the invocation of Fast User-ECalls is approximately 50% smaller than for regular system calls. Deferred User-ECalls reduce this overhead even more in high load situations through the *batching* of events, i.e., the handler invocation is deferred until the next time the scheduler runs. The advantages are that (1) the function is invoked when the application is already in the kernel, thus removing the need for a user/kernel boundary crossing, and (2) several Deferred User-ECalls between two invocations of the scheduler cause only a single invocation of the handler function, thereby further reducing the overhead. This is useful for situations where events can be aggregated or where only the last event is of use.
Kernel-to-user events. If a kernel extension raises an event to an application, one or more of the following actions may be performed:

(a) Real-Time Signals
(b) Kernel Handler Functions
(c) User Handler Functions
(d) Kernel Handler Threads
(e) ECalls-based CPU Scheduling

From kernel to user, ECalls can be used to raise real-time signals to applications, or to invoke handler functions on behalf of applications, where these functions can reside either in user space (*User Handler Function*) or in kernel space (*Kernel Handler Function*). If these functions are non-blocking and short-running, they can even be invoked in interrupt context, otherwise a kernel handler function has to be run in the context of a kernel thread (*Kernel Handler Thread*), which is taken from a thread pool. This approach is similar to the functionality of optimistic message handlers [20]. Finally, ECalls is able to cooperate with the CPU scheduler, currently including both the standard UNIX scheduler and a novel hard real-time scheduler, termed DWCS [19], to influence scheduling decisions in conjunction with event communications. This feature of ECalls, called *ECalls-based CPU Scheduling*, is the topic of the remainder of this paper.

3 Coordinated Scheduling of CPU and Events

Most systems deploy CPU schedulers that ignore important application-level events like message arrivals. ECalls offers the basic functionality needed for creating event-aware systems, by (1) associating kernel-to-user events with important application-level events and (2) using these kernel-level events to affect CPU scheduling decisions. The effect is that processes acting as sinks of application-level events are favored over other processes whenever they receive events. Thus, ECalls may be used to implement policies by which applications cooperate with system services like CPU scheduling. One purpose of such cooperation is to minimize the end-to-end delays experienced for the delivery and processing of important application-level events mentioned in Section 1 above.

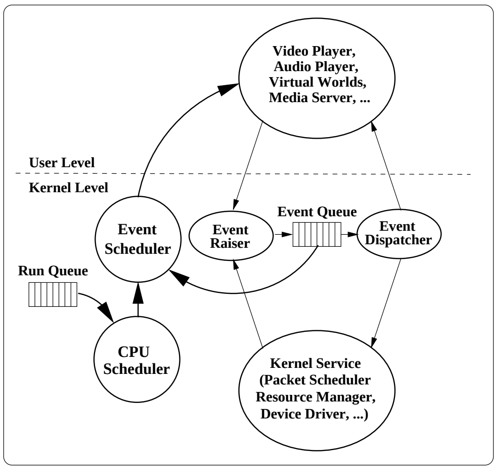

Figure 3: ECalls-based CPU Scheduling.

Implementation approach. ECalls' implementation presented in this paper permits both user-level applications (e.g., multimedia applications and media servers) and kernel-level services (e.g., device drivers and resource managers) to be sources and/or sinks for events generated in user and/or kernel space. Both applications and kernel services can generate events via the *event raiser* (see Figure 3), and these are delivered to the sink by the *event dispatcher*. In addition, ECalls can add events to the *event queue*, which is used by the *event scheduler* to revise the decisions of the CPU scheduler if and when necessary. The event scheduler takes the first event from the event queue, determines the sink of this event, and compares the scheduling attributes of the sink with the scheduling attributes of the process selected by the CPU scheduler. The event queue is ordered according to *time stamps*, which decide *when* events have to be delivered. ECalls ensures that an event with a time stamp in the future will not be considered by the event scheduler. A time stamp can be supplied by the event source; if it is missing, ECalls assumes that the event has to be delivered immediately and uses the *current* time as time stamp.

Coordinated scheduling support for processes and events can be implemented for any ECalls-enabled CPU scheduler. Our current implementation supports two such schedulers: the traditional UNIX scheduler and the DWCS hard real-time scheduler. Note that event and CPU schedulers are separated, thus permitting the event scheduler to utilize any appropriate CPU scheduler. We next describe the coordinated scheduling implemented for each of the two task schedulers we used.

3.1 Coordination Using the Traditional UNIX Scheduler

The cooperative scheduling policies implemented via the event scheduler and the UNIX scheduler are the following:

Rule 1: If a real-time process (a process in either the **FIFO** or the **round robin** run queue) is the sink of an event in the event queue, then it will be given preference over other processes that reside within the same or in lower priority classes.

Rule 2: If only non-real-time processes are currently schedulable, then a process that is the sink of an event in the event queue will always be given preference.

These rules ensure that non-real-time applications that are event sinks are given preference over all other best-effort applications, while real-time applications that are event sinks will be given preference over all other real-time applications within the same or in lower priority classes.

3.2 Coordination for DWCS CPU Scheduling

The DWCS CPU scheduler. The traditional UNIX scheduler has been shown to have unacceptable performance for multimedia applications [21]. For example, an application with a fixed real-time priority could have precedence over all other applications at all times, and therefore, starve best-effort applications. This has led to the development of new scheduling approaches, including those based on reservations and on proportional share resource allocations [22]. To efficiently support real-time applications, we use a hard real-time CPU scheduler, called DWCS (Dynamic Window-Constrained Scheduler[2]), which is based on the DWCS packet scheduler presented and evaluated in two other papers [19, 23]. DWCS assigns each process the following attributes: a period T, a service time C, and a window-constraint x/y. Using these attributes, DWCS **attempts** to service a process for at least C time units in a period of T time units, and it **guarantees** that it will service a process in $y - x$ periods in a window of y periods if the *CPU utilization* is less than or equal to 100%. Thus, the minimum CPU utilization consumed by a process i is determined by

$$U_i(min) = (1 - x_i/y_i) * C_i/T_i.$$

The period T_i of a process i is used to set a deadline until the scheduler has to service process i for at least C_i time units. If the process misses its deadline more than x_i times in a window of $T_i * y_i$, the scheduler *violated* the real-time guarantees to this process. Each process can be scheduled once in its period, unless it is marked as *work-conserving*, in that case it is possible to schedule this process several times within its period as long as CPU utilization allows.

Scheduling attributes are adjusted dynamically to reflect the progress of a process. We distinguish between the *original window-constraint* x/y and the *current window-constraint* x'/y', where the latter is modified dynamically according to the following rules:

Rule a: If the scheduler allocates C_i time units to process i within a period T_i, the window-constraint is relaxed by decrementing the window denominator. If the denominator and the numerator of the window-constraint are equal ($y'_i = x'_i$), both are decremented until they reach zero, at which they are reset to their original values:

if $(y'_i > x'_i)$ then $y'_i = y'_i - 1$;
else if $(y'_i = x'_i)$ and $(x'_i > 0)$ then
  $x'_i = x'_i - 1$; $y'_i = y'_i - 1$;
if $(y'_i = x'_i = 0)$ then $y'_i = y_i$; $x'_i = x_i$;

This ensures that a process that has been serviced already within its period, will relax its window-constraint.

Rule b: If a process misses to be scheduled within its current period, the window-constraint is adjusted to reflect an increased urgency:

[2] http://www.cc.gatech.edu/~west/dwcs.html

if $(x'_i > 0)$ then $x'_i = x'_i - 1$; $y'_i = y'_i - 1$;
if $(y'_i = x'_i = 0)$ then $x'_i = x_i$; $y'_i = y_i$;
else if $(x'_i = 0)$ then $y'_i = y_i + 1$;

This gives the process a tighter window-constraint and therefore an increased probability of being scheduled in the near future. Note, that if the window numerator is zero and a process misses to be scheduled within its period, a *violation* has occurred.

Table 1 shows the precedence rules used by DWCS to find the next process to be scheduled.

Table 1: Precedence rules amongst processes

`Earliest deadline first (EDF)`
`Equal deadlines, order tightest window-constraint first`
`Equal deadlines and zero window-constraints, order highest window-denominator first`
`Equal deadlines and equal non-zero window-constraints, order lowest window-numerator first`
`All other cases: first-come first-serve`

The simplified pseudo-code for DWCS is as follows:

```
while (TRUE) {
  find process i according to the rules in Table 1;
  adjust window-constraints for process i (Rule a);
  for (each process j<>i missing its deadline) {
    adjust window-constraint for j (Rule b);
    adjust deadline for j;
  }
  schedule i;
  adjust deadline for i;
}
```

In a different paper [24], we investigate the DWCS packet scheduler and demonstrate its real-time characteristics, which are also applicable to CPU scheduling:

(a) DWCS is able to give firm bounds for the maximum delay of service to a given process on the run queue in both under-load and over-load situations.

(b) The least upper bound on the system utilization is 1.0, if $C_i = k$ and $T_i = nk$ $\forall i$ and $k, n \geqslant 1$.

The remainder of this section describes the cooperation between ECalls' event scheduler and DWCS, where the goal is to maximize event responsiveness without compromising the hard real-time guarantees of DWCS.

Event scheduling with DWCS. If ECalls has one or more events in the event queue, the event scheduler is invoked each time the CPU scheduler runs. After the CPU scheduler selects the next process, the event scheduler compares the scheduling attributes of this process with the attributes of the sink for the first event in the event queue.

Assume that process i is the process selected by DWCS and process j is the sink of the first event on the event queue. The event scheduler applies the following five rules to processes i and j:

Rule 1: If $j = i$ (i.e., DWCS already selected the sink process), the only action the event scheduler has to perform, is to remove the first event from the event queue.

Rule 2: If task i is a best-effort task, ECalls replaces i by j and removes the event for process j from the event queue. DWCS schedules best-effort processes only if all runnable real-time processes have been serviced within their respective periods and none of them is a work-conserving process. That means further that process j receives an additional time unit in its current period, so that it is able to react to an event immediately. No real-time guarantees are compromised since all real-time processes have been serviced in their corresponding periods.

Rule 3: If process i is a work-conserving process that received at least C_i time units of CPU time in its current period T_i, the event scheduler replaces i with j and removes the event for process j from the event queue. The real-time guarantees of i are not compromised in this case, since process i received C_i time units in its current period already.

Rule 4: Assume that both processes i and j have not been serviced in their current periods yet, and both have the same deadline. Further assume, that DWCS selected process i as the next running process due to its tighter window-constraint compared to process j. ECalls' event scheduler gives process j preference over process i, if this does not lead to a missed deadline for i (i.e., $T_0 + T_i - C_j > 0$, where T_0 is the current time). In other words, process i will be delayed by C_j, but since its deadline has not expired, it will be the highest priority process at the next invocation of the DWCS scheduler, and its real-time guarantees are not violated.

Rule 5: In addition to the rules above, we introduce the notion of a *task server*, which is a pseudo process with the attributes x_{ts}/y_{ts}, T_{ts}, and C_{ts} determined in the following way:

$$x_{ts}/y_{ts} = 0/y_{max}, y_{max} = max\{y_i\}.$$

This assigns the task server the tightest window constraint possible. The service time C_{ts} is the same as the service time of the sink process of the first event in the event queue, or 1 otherwise. 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_{ts}:

$$T_{ts} = C_{ts}/U_r.$$

Each time the first event in the event queue changes (i.e., a new event has been placed at the front of the queue or the first event has been delivered), the attributes for the task server have to be re-calculated if the service time of the sink process of the new first event is different from the service time of the previous sink process. Each time the task server becomes the highest priority task, the process which is the sink of the first event in the event queue is scheduled instead. If there are no events pending, available best-effort tasks can be scheduled. The purpose of the task server is to *reserve* the remaining CPU time, depending on the current utilization, for processes that have events pending.

The simplified pseudo-code for the event scheduler is as follows (i is the process selected by DWCS):

```
determine j by inspecting first event in event queue;
while (TRUE) {
  if (i == j) schedule i;
  else if (i is best-effort task) schedule j;
  else if (i is work-conserving and has been serviced
    in its current period) schedule j;
  else if (deadline(i) = deadline (j) and a delay of i
    does not cause a violation for i) schedule j;
  else if (i is task server) schedule j;
  else schedule i;
}
```

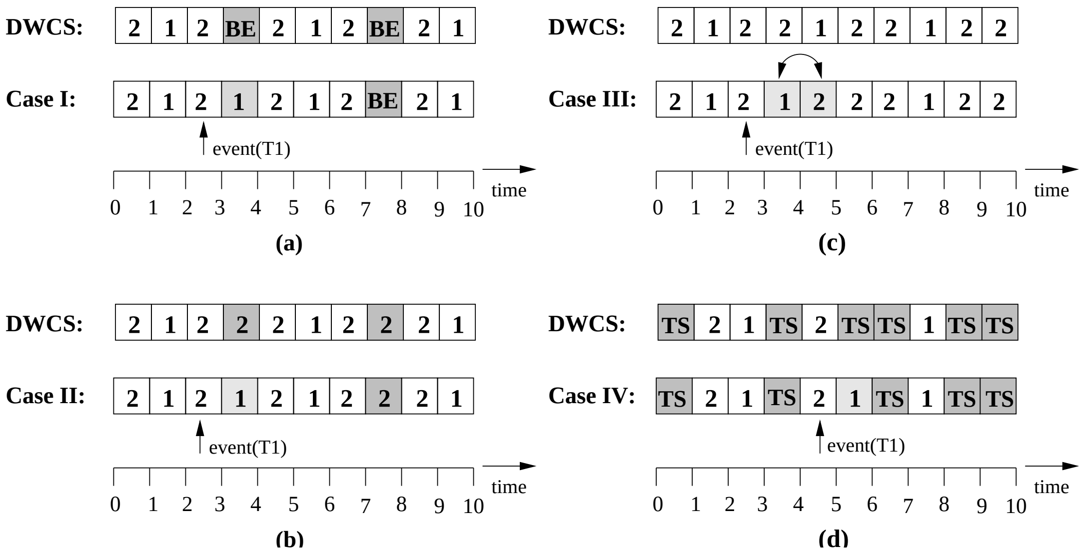

Figure 4: Examples for Rule 2 (a), Rule 3 (b), Rule 4 (c), and Rule 5 (d).

The following examples illustrate Rules 2-5 in more depth:

- **Case I:** According to Rule 2, the event scheduler in ECalls replaces a best-effort process that has been selected by DWCS, with a real-time process that is the sink for the first event from the event queue. Scenario I in Table 2 shows a simple set of three tasks and their scheduling attributes (x/y, T, and C). In this first example, all processes are non-work-conserving, i.e., once a process has been serviced in its period, it will be taken from the run queue until its current period expires. The first diagram in Figure 4(a) demonstrates how DWCS will schedule this task set according to their respective scheduling attributes. When no real-time tasks are schedulable, DWCS selects best-effort tasks (BE) if available. Now consider the situation when an event for task T1 is generated at time 2.5. This causes the event scheduler to run at the next invocation of DWCS, which is at time 3, and to revise DWCS' decision. In this example, the best-effort task selected by DWCS is replaced by task T1, which is able to receive and handle the event. This allows T1 to react to the new event 2 time units earlier than without the event scheduler.

Table 2: Task Sets

	Scenario I			Scenario II		
	x_i/y_i	T_i	C_i	x_i/y_i	T_i	C_i
Task T_1	1/2	2	1	1/2	3	1
Task T_2	1/4	4	1	1/4	3	1

- **Case II:** According to Rule 3, the event scheduler in ECalls replaces a real-time process i with another real-time process j that is the sink for the first event from the event queue, if i is work-conserving and has already been serviced for at least C_i time units in its period T_i. Consider the same task set as used in case I, but now assume that all tasks are work-conserving, i.e., they are allowed to run several times within their periods. The first diagram in Figure 4(b) shows a similar schedule as before, with the difference that when all real-time processes have been serviced in their corresponding periods, DWCS selects the work-conserving real-time process with the tightest scheduling attributes. Again, an event for process T1 is being generated at time 2.5. At the next scheduling point (t=3) the scheduler selects process T2 as the next task to be run. According to Rule 3, a process receiving an event is favored over a work-conserving process that has been serviced in its current period already, therefore, T1 is scheduled by the event scheduler at t=3. As in case I before, process T1 receives the event 2 time units earlier than without event scheduling.

- **Case III:** Consider scenario II shown in Table 2, both tasks T1 and T2 have the same period and both tasks are work-conserving. Figure 4(c) demonstrates how DWCS schedules this task set. Again, an event for process T1 is generated at time 2.5. At the next scheduling point (t=3), the scheduler selects process T2 as the next process to be run. Both processes T1 and T2 have the same deadlines at this point, but since T2 has the tighter window-constraints, DWCS prefers process T2. According to Rule 4, the event scheduler revises the decision made by DWCS and schedules T1 instead of T2, allowing process T1 to learn about the event 1 time unit earlier than without event scheduling. As it can be seen from the diagram, this causes processes T1 and T2 to *swap* their time slots compared to the schedule made by DWCS.

- **Case IV:** The last rule, Rule 5, states that a task server can be introduced, whose attributes are determined by the utilization of the system. In this example, the utilization is 42%, the rest utilization (58%) is used to determine the period of the task server:

  $$T = C/Ur = 1/0.58 = 1.7 => T = 2.$$

  Note that we have to round up the period to a full time unit according to the time base used. Further, the service time C of the task server is 1 and the window-constraint x/y is 0/4. This task is added to the task set and Figure 4(d) demonstrates how DWCS schedules this new task set. The task server is added to the schedule and whenever it becomes the highest priority task, the event scheduler selects the sink process for the first event in the event queue. If there are no events pending, the task server is replaced by a best effort task. However, if an event is generated (e.g., at time 4.5 in Figure 4(d)), the scheduler uses the next task server invocation to schedule the sink process for this event. In this example, task T1 receives the event two time slots earlier than without the support of the event scheduler.

  A side effect of the task server is that if no events are pending, the task server is replaced by best-effort tasks, which are now scheduled **before** the real-time tasks. This minimizes the delay best effort tasks will experience, without violating the real-time guarantees of other processes.

4 Experimental Evaluation

The following experiments have been performed on a workstation with a Pentium II processor with 300MHz and 256MB RAM, running Linux 2.2.13.

Basic overheads. In Linux, the CPU scheduler is invoked frequently, e.g., each time a timer interrupt occurs (100 times/second on Intel architectures), the scheduler can decide to select a different process to run. Therefore, it is important to reduce total scheduling overhead. The overhead caused by the event scheduler in ECalls depends on the CPU scheduler used, e.g., with DWCS, the event scheduler requires 3.5μs to execute. This overhead is independent of the length of the event queue since the event scheduler only considers the first entry in the queue. The costs for generating an event contribute an overhead of 6μs. This overhead increases minimally with the length of the event queue (e.g., 6.2μs for an event queue with 100 entries).

Scheduling video services. In the following experiments, a number of video players request video streams from video servers, which run on two Ultra 30 with 248MHz and 128MB RAM each. The Linux workstation and the video servers are connected via a switched 100Mbps Ethernet.

The Linux version on the workstation has been modified as follows:

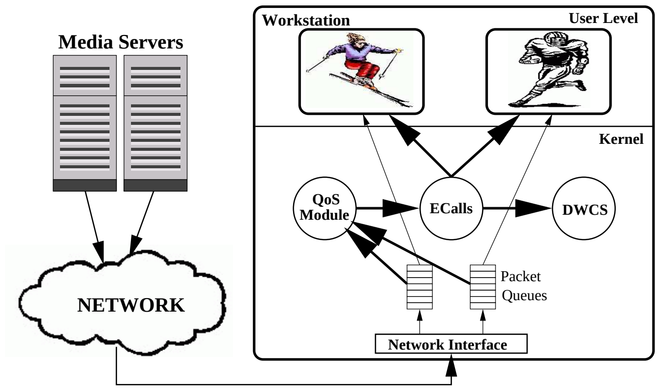

Figure 5: Video play-out with ECalls support.

(a) The traditional UNIX scheduler has been replaced by DWCS to support real-time and multimedia applications.

(b) The ECalls mechanism has been added for the support of event communication between applications and kernel services, as well as to support coordinated scheduling of events and processes with DWCS.

(c) A QoS module has been implemented, which generates events to applications depending on application-specific information. For instance, this QoS module can generate events when new packets arrive on a socket, thereby replacing costly **select** system calls. In our experiments, an application informs the QoS module via ECalls about the desired frame rate. The QoS module uses this information to generate periodic events in case that new frames can be read from the socket queue of the video application (Figure 5). Using the terminology introduced in Section 1, a timer raises periodic **application-level events** in case there are frames available to display, and these event generate **kernel-level events** via ECalls, which notify the application and affect the scheduling decisions of DWCS.

The gray bars in Figure 6 show the run queue delays for a simple distributed video player based on the Berkeley MPEG Player[3], which uses timed waits. These delays cause variations in inter-frame times. Specifically, when the video player runs as the only real-time process, the variations are close to 10ms and they increase rapidly when multiple real-time video players are run concurrently. In this experiment, all video players are run as work-conserving tasks and have periods of $T = 100ms$, window-constraints of $x/y = 1/5$, and service times of $C = 20ms$. Next, we use the QoS module to perform the timing in the kernel via timer queues and to raise events when these timers expire. Furthermore, we use the ECalls module to deliver these events to the application. The black bars in the same graph show how the run queue delays improve when we use the QoS module to generate events according to the desired frame rates (10 frames per second for each video player). However, in this test, the event scheduler in ECalls has been disabled, i.e., the only action triggered by a kernel-level event is that the sink process (the video player) is made schedulable.

The following experiments demonstrate how coordinated event/CPU scheduling can reduce run queue delays. For this purpose, separate tests are performed for each scheduling rule described above, whereas the event scheduler uses only one rule to decide whether the scheduling decision of DWCS has to be revised. This allows us to isolate the effects of the scheduling rules from each other and to show in which scenarios each rule becomes active. The real-time application used in these experiments is the distributed video player described above, however, the measured results are also applicable to other real-time and multimedia applications such as distributed games or virtual worlds.

[3]http://bmrc.berkeley.edu/frame/research/mpeg

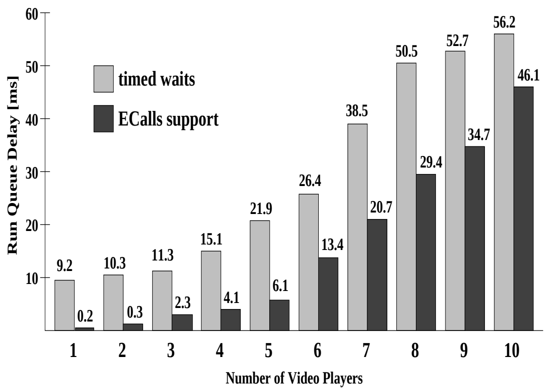

Figure 6: Comparison of run queue delays (= variations in inter-frame times) between the original video player using timed waits and the modified player using ECalls and the QoS module.

Event Scheduling (Case I)

Rule 2 in the event scheduling algorithm states that best-effort processes can be replaced by processes that are sinks for events on the event queue. Here, we run one video player as a non-work-conserving process with a period of $T = 100ms$, a service time of $C = 20ms$, and a window-constraint of $x/y = 1/5$. As a best-effort process we use a simple application that runs an endless **for** loop. In this case, 16.7% of all events cause ECalls' event scheduler to give preference to the video player instead of the best-effort task, leading to a reduction in run queue delays (and variations in inter-frame times) of 14%. Table 3 summarizes the results of this experiment.

Table 3: Event Scheduling (Case I)

Events causing process swaps (in %):	16.7
Run queue delay w/o event scheduling (avg.):	42.5ms
Run queue delay w/ event scheduling (avg.):	36.7ms
Run queue delay w/o event scheduling (max):	559ms
Run queue delay w/ event scheduling (max):	160ms
Improvement in avg. run queue delay (in %):	14%

Event Scheduling (Case II)

In this example, the event scheduler uses only Rule 3, which states that if DWCS selects a work-conserving process i that has been serviced for at least C_i time units in its current period, this process can be replaced by a process receiving an event. All video players run as work-conserving processes with the following scheduling attributes: $T = 100ms$, $C = 10ms$, and $x/y = 1/5$. Figure 7 compares the run queue delays (= variations in inter-frame times) for the two scenarios described above and shows that event scheduling can improve on these variations by approximately 10%.

Table 4 summarizes the percentages of events that cause the event scheduler to revise the scheduling decisions of DWCS (*swaps*). For instance, with 10 players running, more than 60% of all events cause the scheduler to give preference to the sink process for the first event on the event queue, thereby allowing the player to display the next frame earlier. In the remaining 40%, the process selected by DWCS and the process selected by the event scheduler are identical, so no swaps are necessary.

Event Scheduling (Case III)

Rule 4 in the event scheduling algorithm states that a process j that is a sink for an event on the event queue can

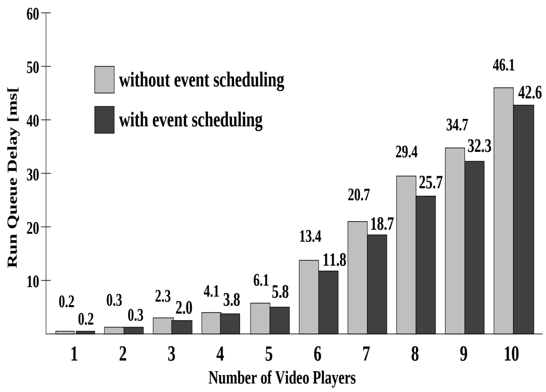

Figure 7: Run queue delays.

Table 4: Number of *swapped* processes

Video Players	Swaps (in %)
1	0
2	0
3	3.3
4	14
5	18
6	24
7	28
8	41.3
9	54.7
10	61.3

replace a process i, selected by DWCS, if the deadlines of i and j are equal and if this will not lead to a violation for i. In this example, we run 10 video players as work-conserving processes, each with a period $T = 20ms$, a service time $C = 10ms$ and a window-constraint $x/y = 1/5$. Here, 15% of all events cause ECalls' event scheduler to give preference to a process that is a sink for an event on the event queue instead of the process selected by DWCS. This leads to a reduction in inter-frame time variations of 24%. Table 5 summarizes the results of this experiment.

Table 5: Event Scheduling (Case III)

Events causing process swaps (in %):	15
Run queue delay without event scheduling (avg.):	20.8ms
Run queue delay with event scheduling (avg.):	16ms
Run queue delay without event scheduling (max):	216ms
Run queue delay with event scheduling (max):	56ms
Improvement in avg. run queue delay (in %):	24%

Event Scheduling (Case IV)

Rule 5 in the event scheduling algorithm introduces the notion of a task server, whose DWCS scheduling attributes are determined using the system utilization. This task server is inserted into the task set and whenever it becomes the highest priority task, the event scheduler can select a process receiving an event instead. In this example, we ran up to 4 video players, with a period of $T = 100ms$, a service time of $C = 10ms$, and a window-constraint of $x/y = 1/5$ each. The task server has a service time of $C = 10ms$, a window-constraint of $x/y = 1/5$ and a period, which is computed the following way (in the case of 4 video players):

$$U_r = 100\% - 4 * (1 - x/y) * C/T = 68\%$$

$$T = C/U_r = 10ms/0.68 = 14.7 = 20ms.$$

The period T for the task server results in 14.7ms, i.e., 20ms for our experiments, since we have to set the period in multiples of the internal time base for the Linux system, which is 10ms.

Figure 8 shows that the improvements achievable vary between 9% and 36% depending on the system load. The number of *swaps* is reflected in Table 6.

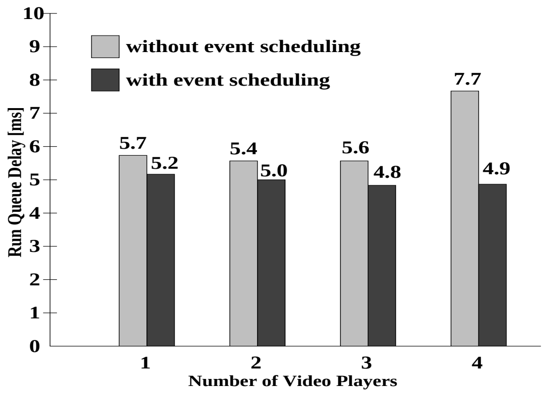

Figure 8: Run queue delays.

Table 6: Number of *swapped* processes

Video Players	Swaps (in %)
1	96
2	88
3	71
4	62

Analysis

All experiments use different task sets and attributes, because each rule of the event scheduler focuses on a different scenario:

Case I: Using Rule 2, the event scheduler replaces best-effort processes with real-time processes that have events pending. This is useful in situations where (a) the CPU load is less than 100% and (b) there are no work-conserving real-time tasks schedulable.

Case II: According to Rule 3, the event scheduler prefers processes with events pending over other work-conserving processes, if they have already been serviced for their respective service times in their periods. That is, this rule becomes active only when (a) the CPU load is less than 100% and (b) there are real-time processes that are work-conserving.

Case III: If two processes have the same deadlines, a process that receives an event from the event queue can be favored over another real-time process, if this does not lead to a violation for the other process (Rule 4). This rule can apply anytime, even in over-load situations (i.e., when CPU load > 100%).

Case IV: The task server can only be put into the task set if the utilization is less than 100%. In that case, it will act as a *place holder* for processes that are sinks for events on the event queue and best-effort processes if no events are pending.

The different rules for the event scheduler can contribute to reductions in run queue delays in different scenarios as described above. Figure 9 shows a snapshot of a final experiment, where all rules described above are activated and the system is overloaded (i.e., CPU load > 100%).

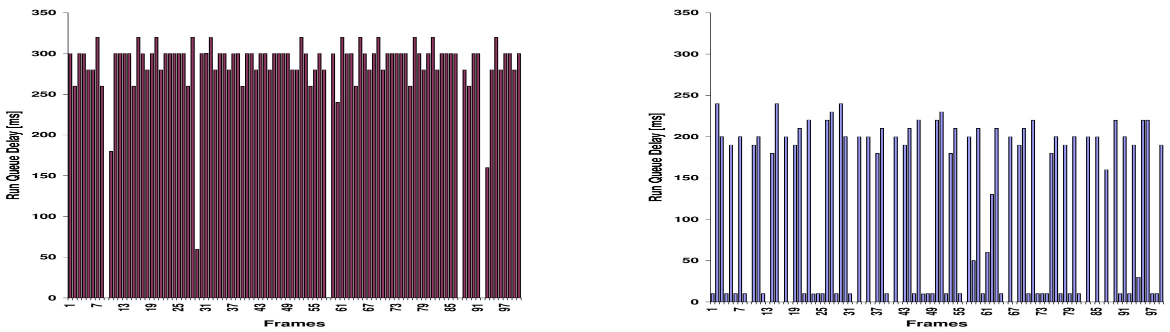

Figure 9: Variations in inter-frame times for 100 frames.

Both graphs show the run queue delays (= variations in inter-frame times) for the display of 100 frames for a video player that competes for the CPU with a real-time *disturber* process and 9 other video players. The video players are work-conserving and have the following attributes: $T = 100ms$, $C = 10ms$, $x/y = 1/5$. The disturber process is a non-work-conserving process with a period $T = 20ms$, a service time of $C = 10ms$, and a window-constraint of $x/y = 1/5$. The first graph shows the run queue delay without the support of event scheduling, compared to the run queue delays with the support of event scheduling in the second graph. The average run queue delay in the first case is $250ms$, compared to $100ms$ in the second case. This is a reduction of 60%. The worst case run queue delays in the first case exceed 320ms, compared to 240ms in the second case.

5 Conclusions and Future Work

Extensions to general-purpose operating systems such as real-time schedulers succeed in improving the performance and quality of service of multimedia applications. However, typical CPU schedulers are not *event-aware*, that is, they are not cognizant of important events shared by distributed application components and/or of events shared by applications and kernel services. This can result in significant delays in event delivery. Specifically, the operating system's contribution to the delay between the time an important event is generated and the time the event is received and processed by the application, which we term *run queue delay*, can cause variations in inter-frame times for video applications, and it can lead to timing violations in real-time applications like distributed multi-player games.

This paper demonstrates the utility of event-awareness for distributed multimedia applications and in general-purpose operating systems like Linux. It presents ECalls, an event communication facility with which coordinated policies may be implemented to minimize run queue delays. Using ECalls and a set of coordination rules, we then demonstrate performance gains with a distributed video player application, realizing reductions in event delivery of up to 60%, especially in over-load situations (i.e., when CPU load > 100%).

Future work will further investigate the performance effects of event-awareness, realized with ECalls and the coordinated event/CPU scheduling policies we have developed and using a distributed multi-player game that integrates data, video, and audio streams. Furthermore, while this paper focuses on event-awareness on a single end-system, other work has investigated the use of intra- and inter-address space/machine events to create event-aware adaptive applications [25]. We will extend such work to create event-aware systems, by extending ECalls to operate across multiple operating system kernels linked via networks. In addition, by porting ECalls to a version of Linux running on a mobile platform, we plan to address the effects of user mobility, data loss, and limited battery power on the performance of multimedia applications.

References

[1] M. Claypool and J. Riedl, "End-to-End Quality in Multimedia Applications," in *In Chapter 40 in Handbook on Multimedia Computing, CRC Press, Boca Raton, FL, 1998.*, 1998.

[2] R. Govindan and D. P. Anderson, "Scheduling and IPC Mechanisms for Continous Media," in *Proc. of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

[3] V. Sundaram, A. Chandra, P. Goyal, P. Shenoy, J. Sahni, and H. Vin, "Application Performance in the QLinux Multimedia Operating System," in *Proceedings of the Eighth ACM Conference on Multimedia, Los Angeles, CA*, November 2000.

[4] 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.

[5] K. Jeffay, "On Kernel Support for Real-Time Multimedia Applications," in *Third IEEE Workshop on Workstation Operating Systems*, April 1992.

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

[7] M. B. Jones, D. Rosu, and M.-C. Rosu, "CPU reservations: Efficient Predictable Scheduling of Independent Activities," in *Proceedings of the* 16^{th} *ACM Symposium on Operating System Principles*, pp. 198–211, October 1997.

[8] 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*, pp. 107–121, USENIX, 1996.

[9] 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.

[10] 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.

[11] C. Poellabauer, K. Schwan, and R. West, "Flexible Event Delivery for Linux Kernel Extensions," Tech. Rep. GIT-CC-00-36, College of Computing, Georgia Institute of Technology, 2000.

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

[13] I. Stoica, W. Abdel-Wahab, and K. Jeffay, "O the Duality between Resource Reservation and Proportional Share Resource Allocation," *Multimedia Computing and Networking*, 1997.

[14] C. Shen, "Discrete-event Simulation on the Internet and the Web," in *The International Conference on Web-Based Modelling and Simulation*, January 1998.

[15] J. Bolot and S. Parisis, "Adding Voice to a Distributed Game on the Internet," in *In Conference on Computer Communications (IEEE Infocom), San Francisco, California*, March 1998.

[16] J. Lui, O. So, M. Chan, and T. Tam, "Dynamic Partitioning for a Distributed Virtual Environment," in *Proc. of the 3rd High Performance Computing Asia Conference (HPC Asia'98)*, 1998.

[17] 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.

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

[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.

[20] D. A. Wallach, W. C. Hsieh, K. L. Johnson, M. F. Kaashoek, and W. E. Weihl, "Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation," in *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pp. 217–225, July 1995.

[21] J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall, "SVR4 UNIX Scheduler Unacceptable for Multimedia Applications," in *Proc. of the 4th Int. Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '93), Lancaster, UK*, November 1993.

[22] T. Plagemann, V. Goebel, and P. Halvorsen, "Operating System Support for Multimedia Systems," *Computer Communications Journal, Special Issue on Interactive Distributed Multimedia Systems and Telecommunications Services (IDMS '98)*, 1998.

[23] 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.

[24] 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.

[25] 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*, June 2001.