Lars C. Wolf, Wolfgang Burke, Carsten Vogt:
[WBV96] *Evaluation of a CPU Scheduling Mechanism for Multimedia Systems*. In: *Software Practice and Experience*, vol. 26, no. 4, p. 375--398, April 1996.

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 26(4), 375–398 (APRIL 1996)

# Evaluation of a CPU Scheduling Mechanism for Multimedia Systems

LARS C. WOLF
*IBM European Networking Center, Vangerowstraße 18, D-69115 Heidelberg, Germany*
*(e-mail: lwolf@vnet.ibm.com)*

WOLFGANG BURKE
*Universität Karlsruhe, Am Fasanengarten 5, D-76128 Karlsruhe, Germany*
*(e-mail: burke@ira.uka.de)*

AND

CARSTEN VOGT
*Fachhochschule Köln, Betzdorfer Str. 2, D-50679 Köln, Germany*
*(e-mail: vogt@fh-koeln.de)*

SUMMARY

**Multimedia applications handling audio and video data have to obey time characteristics of these media types. Besides a basic functionality to express time relations, correctness with respect to time constraints requires mechanisms which lead to favoured processing of multimedia operations. CPU scheduling techniques based on the experience from real-time operating systems offer a solution and provide multimedia applications with the ability to meet time-related quality of service requirements. This paper discusses mechanisms to express time in multimedia systems and describes an implementation of a CPU scheduler designed to run under IBM's UNIX derivate AIX. The evaluation of the implementation based on measurements shows that the scheduler is able to support the time requirements of multimedia applications and that such mechanisms are indeed necessary since otherwise deadline violations occur.**

KEY WORDS multimedia; real time; scheduling; operating system support

## INTRODUCTION

Owing to the periodicity of continuous-media data, the processing of audio and video data must occur also in this fashion. Moreover, the execution of these operations has to be finished within certain deadlines to serve the real-time characteristics of these media. Owing to these real-time characteristics of audio and video data, multimedia systems have to provide mechanisms to support time-related *quality-of-service* (QoS) guarantees.

Sometimes, multimedia systems for single-user, and especially for single-task, machines provide only simple mechanisms to provide time-based operations, e.g. for delaying program execution, but no real-time support. It is often argued that this approach is sufficient for these systems since the CPU is used mostly for the multimedia

CCC 0038–0644/96/040375–24
© 1996 by John Wiley & Sons, Ltd.

*Received 7 December 1994*
*Revised 12 July 1995*

application during its run time. In those situations where the user has another time-consuming application running, it is easy for him to abandon that application. For multi-user and server systems such as video-on-demand servers, this assumption is not valid. Other user applications can disturb multimedia applications in such a way that the perceived QoS is not acceptable. Real-time CPU scheduling techniques which serve multimedia application processing with respect to their time-criticality provide a solution to these problems.

This paper first discusses various methods to express time in multimedia systems. Then a real-time scheduling algorithm and its implementation for IBM's AIX Version 3 operating system is described. Work for OS/2 has been discussed in Reference 1. This is part of our work on the transport system HeiTS (Heidelberg Transport System)[2] which offers real-time communication support for distributed multimedia applications.

Our goal is to show how a general-purpose operating system that is widely available on the market can be used for the processing of multimedia applications without a modification of its kernel structures. We did not intend to develop a new real-time system that is specifically tailored to multimedia requirements.

## EXPRESSING TIME IN MULTIMEDIA SYSTEMS

Various ways to express time in multimedia systems exist which place different amounts of burden on the application programmer and provide different kinds of real-time support. For simple programs, the mechanisms described in the next two sections are sufficient. In those cases, the timely operation is either hidden in high-level functions or is based on I/O adapter characteristics.

Programs which perform more complex tasks than just moving data from an input device to an output device often have to execute certain operations after some specified time. Hence, an operating system driven control of timing is needed. Consider, for instance, a server which reads data from a disk and sends it over a network periodically. In this scenario, neither the input device, the disk, nor the output device, the network, operate periodically. Therefore, the operating system must provide appropriate control mechanisms. Such mechanisms are explained after the next two sections.

### Hidden timing

Often, multimedia environments for stand-alone computer systems offer library functions which allow a programmer to play audio or video data in a simple way, e.g. consider a function such as `play_audio_file`. These functions perform the steps necessary to present the information to the user and hide the actual handling of time in their program code. Their use is simple for the application programmer, however, they serve only a closed set of applications. The functions cannot be adapted to specific application needs, e.g. to retrieve continuous-media data from the system, perform some application-dependent processing and present the result directly to a human user. Internally, these functions are based on some of the methods described in the following sections.

### Adapter-based timing

For some simple programs, e.g. programs playing audio data stored on disk via an audio adapter, it may not be necessary to use explicit programming techniques to pro-

```
do {
    read_audio_buffer();
    write_buffer_to_adapter();
} while (more data available);
```

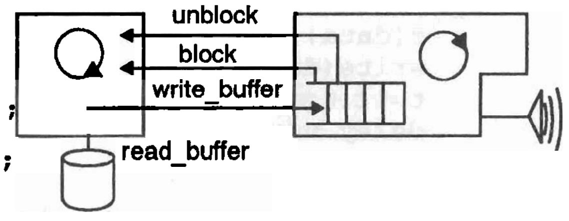

*Figure 1. Adapter based timing*

vide timely operation. Such programs can rather be based on the characteristics of the output device, e.g. the audio adapter. This adapter provides some buffer space in form of a FIFO queue into which the program writes data to be played. The adapter's processor reads data from the buffer and converts it into sound waves. If the buffer is full, the program is blocked from further processing. If the adapter's processor has removed data from the buffer space the process can be unblocked and continue processing as detailed in Figure 1.

## Loop/delay

The methods described in the last two sections are only sufficient for simple applications. Now, mechanisms for more complex programs are discussed.

An approach often taken by application programmers for writing periodic programs is the use of `loop/delay` constructs. These constructs rely on mechanisms provided by most operating systems to delay the execution of a program for a certain amount of time, such as UNIX's `sleep` function. They are used in the following way where `f` denotes some function for processing the data:

```
get_time(t);
while (read(data) != eof) {
    f(data);
    write(data);
    t = t + period;
    get_time(t1);
    delay(t-t1);
}
```

While such constructs avoid the influence of the actual execution time of the operations, in a preemptive multi-tasking environment—or due to interrupt processing—this approach is not correct since the statements are not executed atomically. Any preemption between the calculation of the length of the delay period and its use in the delay operation leads to time drifts since in that case the computed value is too large and the delay too long. Using a function which delays the execution until a specified point in time instead of delaying a specified amount of time can solve this problem:

```
get_time(t);
while (read(data) != eof) {
    f(data);
    write(data);
    t = t + period;
    delay_until(t);
}
```

The same problem has been recognized in Ada83, and the Ada9X revision took this into account by the introduction of the above-mentioned delay until primitive.[3] A functionally similar primitive can also be found in Chimera II.[4]

Although this approach leads to correct timing, it places the burden of time handling and temporal state information management on the application programmer. It is better, and safer, to have the underlying operating system do this work.

## Asynchronous events

Notifications by asynchronous events may be used by programs dealing with devices where the event indicates that some change in the device state has occurred. Similarly, this may be used to indicate timer expirations. Processing these events leads to the execution of specific program sequences, often called signals handlers or callback functions, to reflect the changed system state.

This method separates time handling and functional specification. However, often only subsets of functions may be executed in an event handler, which means that only state changing operations such as setting flags may be performed but no regular application processing. Furthermore, mutual exclusion between the application program and the event handler during access to shared data structures has to be enforced to avoid state inconsistency. Owing to the non-determinism of asynchronous events it is a general fact that those programs are hard to analyse and to debug because in subsequent executions the order of events can be different.

## Periodic threads

*Periodic threads* are threads which perform their operations at fixed periodic points in time without explicitly specified intervention by the application programmer. Periodic threads are described for example in References 5, 6 and 7. Using periodic threads, the programmer specifies some characteristics of the thread such as **estimated processing time**, **period** and **entry_point**. In each period, a thread is created which calls the specified entry point, executes the given functions, which take about the estimated time and exits:

```
create_periodic_thread(entry_point,
                       processing_time, period);
...
entry_point()
}
    /* the actual computation */
    f();
}
```

This approach separates functional and temporal specification in a similar fashion as the asynchronous event handling approach does. However, the functionality provided by the periodic thread and the asynchronous event handling method are different, i.e. in the periodic approach the tasks are inherently periodic in nature and the limitations of the event handling approach do not apply to the periodic thread model. Additionally, the periodic thread model provides the underlying system components with information about resource usage.

### Periodic processes

Periodic processes are similar to periodic threads. Since process creation is much more expensive than thread creation, the process does not exit after the execution of each period's operations but calls a function `schedule_me` to wait for the beginning of the next period.* Thus, the execution scheme differs slightly:

```
p = create_new_process();
inform_scheduler(p, processing_time, period);
...
while (TRUE){
    schedule_me();
    /* the actual computation */
    f();
}
```

The principal characteristics of this approach are the same as of the periodic thread model, and differences are mostly related to operating system abstractions.

Note that the function `schedule_me` can be easily implemented using the `delay_until` primitive. However, with `schedule_me` the programmer need not care about the calculation of times; this is done by the operating system.

### Evaluation

As the summary in Table I illustrates, the first three methods do not support a separation of temporal and functional specification and place burden on the application pro-

Table I. Methods to express time

| Scheme | Functional and temporal specification separated | Applicability of real-time methods |
|---|---|---|
| Hidden timing | | |
| Adapter-based timing | | |
| Loop/delay | | |
| Asynchronous events | ✓ | |
| Periodic threads | ✓ | ✓ |
| Periodic processes | ✓ | ✓ |

* It is, of course, possible to use this approach with threads as well and may, therefore, be viewed more as an implementation than a modelling detail. Also, thread implementation techniques exist which reduce the cost for thread creation such as 'reclaiming threads'.[8]

grammer. Asynchronous events provide this separation, but the programming model is complicated. The weakest point of these models is that they do not provide any information about timing and requested resource usage to the system scheduler. This means that no real-time scheduling mechanisms can be applied. For instance, no schedulability tests can be performed to check whether the system is able to support all requests, and, thus, overload situations may occur leading to unacceptable QoS. Therefore, these models can be seen as simple *ad hoc* methods but are not applicable in general multimedia systems where mechanisms for reliable QoS provision have to be available.

As a result, only the last two methods, i.e. periodic threads and tasks, are general enough to be used in multimedia systems supporting reliable QoS. In particular, they provide timing information about the tasks to the system scheduler by their initialization calls. This information enables the scheduler to check the schedulability of the tasks and schedule them accordingly so that their QoS can be guaranteed. Methods for scheduling periodic tasks have been devised and implemented within the realm of real-time systems as discussed in the subsequent chapter.

The second advantage of periodic threads and processes is that their functionality is not limited as is the case with asynchronous event handlers. Hence, they can execute all the functions required to process a multimedia stream.

Aside from research operating systems, most commercially available systems do not offer the needed periodic thread or process model, but only provide basic real-time mechanisms to implement these models. The Posix threads extension,[9] for example, provides the possibility to associate scheduling attributes with each thread defining the scheduling policy and the thread's fixed priority, e.g. priorities assigned based on rate monotonic scheduling. The timing of the thread's operations, e.g. suspending a ready thread till its next period, has to be implemented by the programmer. The timer operations offered by the Posix real time extension[10] can be used for that purpose, in a similar way as is done in the rest of this paper. SunOS 5.0[11] offers the ability to preempt a thread after its time quantum has expired, but the preempted thread is put back on the dispatch queue. Therefore this approach implements some kind of round-robin scheduling but no real-time scheduling. Since the preempted thread has missed its deadline, rather some exceptional operations should be performed.

## SCHEDULING ALGORITHM

As the discussion in the previous section illustrated, mechanisms to express periodicity in the multimedia systems require real-time CPU scheduling mechanisms in form of periodic threads or processes. These have to be provided by the operating system. This section discusses the model used in HeiTS to specify the system workload of periodic data streams, shows how the various processes are prioritized and describes the scheduling algorithm used.

### QoS and workload model

QoS management in multimedia systems is based on two models. The workload model is used to describe the load an application will place onto the system. The QoS model is used by an application to define its performance requirements and by the system to return corresponding performance guarantees. Of course, the workload model can be regarded as being a part of the QoS model since one important QoS requirement of applications is that the system is able to process their workloads.

The QoS model used in HeiTS has three parts:

1. The throughput part describes the bandwidth required for or granted to a multimedia connection. It consists of the three parameters of the workload model described below.
2. The delay part defines the maximum delay a multimedia packet can experience on its way from the source to the sink of the connection.
3. The reliability part describes how packet losses and bit errors within packets are handled. They can be ignored, indicated or corrected.

The workload for multimedia systems is periodic by nature—consider for instance an application presenting audio or video data where data packets must be transmitted at certain instants. To describe the load induced into the system, HeiTS uses the *linear bounded arrival process* (LBAP)[12] as its workload model. The LBAP model assumes data to be processed as a stream of discrete units (*packets*) characterized by three parameters: $S$, the maximum packet size, $R$, the maximum packet rate, i.e. the maximum number of packets per time unit, and $W$, the maximum workahead.

The workahead parameter $W$ allows for short-term violations of the rate $R$: According to the LBAP definition in any time interval of duration $t$ at most $W + tR$ packets may arrive. This is necessary to model input devices that generate short bursts of packets, e.g. disk blocks that contain several continuous-media data frames. Furthermore, the notion of workahead is needed to account for any clustering of packets during the various processing stages before they are finally presented to the user. A useful concept of the LBAP is that of the *logical arrival time* $l(m_i)$, which is defined as:

$$\begin{aligned} l(m_0) &= a_0 = \text{actual arrival time of the first packet} \\ l(m_{i+1}) &= \max\,\{a_{i+1}, l(m_i) + 1/R\} \end{aligned}$$

The concept of logical arrival time essentially acts as a smoothing filter for the traffic streams. It ensures that no particular stream hogs a resource at the expense of other streams given their declared workload characteristics. A packet whose logical arrival time has passed is called *critical*, otherwise it is referred to as *workahead*.

The output stream of a resource or the processing stage serving an input LBAP, e.g. CPU or intermediate network, is itself an LBAP. Its parameters depend on the parameters of the input LBAP and the maximum and minimum delay within the resource. Their computation has been described in Reference 13. For an end-to-end connection passing a periodic stream through various processing stages, e.g. input device → CPU on sending host → intermediate network → . . ., this enables one to 'push' the LBAP workload model from the origin to the destination through all stages.

In addition to the three LBAP parameters defined above, the user must also specify for each resource the maximum processing time per packet to ensure that resource capacities can be correspondingly reserved.

The QoS and workload models described above were chosen because they are conceptually simple, though they describe the requirements of a multimedia stream in sufficient detail, and can be directly used as a basis for QoS management. However, research has brought up a number of other QoS models that can be used in multimedia and real-time processing and communication systems. Examples for alternative models can be found in References 14, 15, 16 or 17. Some of them define the workload by stochastic processes rather than by processes with fixed periods. This is done to reflect the requirements of streams with a variable bit-rate; although these can also be handled in a periodic framework, as shown in Reference 18. Some models account for the packet loss rate and others quantify the jitter of a stream, i.e. the variance of the interarrival times of the stream packets.

## Ordering of priorities

Not all users need the same *degree* of QoS. For some users it is important to get the specified quality all of the time without any degradation, others may accept some temporary quality degradation, especially if the cost for using this service decreases accordingly. The first degree of QoS ('guaranteed' QoS) is necessary for production-level applications, e.g. in a movie studio. The second degree of QoS ('statistical' QoS) is especially useful for playback consumer applications or video conferences, provided that degradations do not occur too often. Based on these QoS classes, different methods for resource reservation can be used.

For guaranteed QoS *pessimistic* resource reservation has to be done. The resource capacities reserved are those needed in the worst possible case, and the QoS requirements will be satisfied under all circumstances. Reserving such large amounts of resources, however, can be rather costly. A cheaper alternative is statistical QoS using an *optimistic* approach. Here, less resources are reserved, for example those needed in the average case. This implies that the QoS requirements will be met in general, but temporary QoS violations may occur. Figure 2 illustrates these different approaches.

In addition to the differentiation between processes serving applications with *guaranteed* and *statistical* QoS, Reference 19 suggests a method of *deadline-workahead* scheduling which dynamically classifies messages with respect to whether they are currently critical or workahead. Within the workahead class guaranteed and statistical streams may be separated, however, for simplicity they are combined into one class. This yields the following multi-level priority scheduling scheme:

1. Critical guaranteed processes.
2. Critical statistical processes.
3. Processes not performing multimedia operations (e.g. application processes).

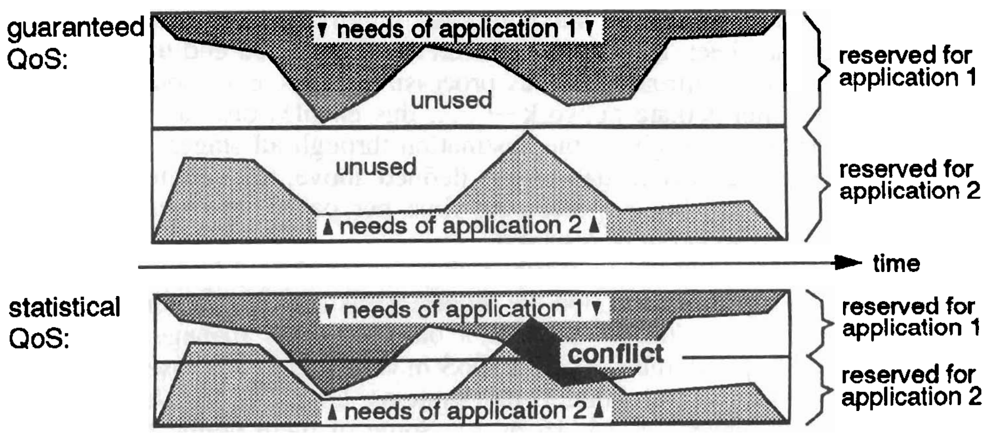
guaranteed QoS:
▼ needs of application 1 ▼
unused
unused
▲ needs of application 2 ▲
reserved for application 1
reserved for application 2
time
statistical QoS:
▼ needs of application 1 ▼
conflict
▲ needs of application 2 ▲
reserved for application 1
reserved for application 2

*Figure 2. Guaranteed vs. statistical QoS*

4. Workahead processes (both guaranteed and statistical).

The scheduling within these priority classes is (preemptive) deadline based (except for class 3 where any strategy can be used). The logical arrival time of a packet plus the guaranteed (or statistical) delay bound calculated for this connection serves as its deadline. A process moves from priority class 4 to 1 or 2, respectively, as soon as it becomes critical, which possibly entails the preemption of the currently executing process.

Because guaranteed processes are in priority class 1 and statistical processes in class 2, this scheme has the drawback that it prefers a guaranteed connection over a statistical connection even if the statistical connection has a closer deadline than the guaranteed connection. Hence, even a statistical connection that does not exceed the specified workload bounds might experience delays larger than those calculated by the QoS optimization function.[20] Although the notion of statistical connections allows such a behavior—because they may temporarily suffer from QoS violations—it is questionable whether this distinction is fair. To avoid this problem, a variant of the algorithm could be considered where priority classes 1 and 2 are combined and a new priority class for statistical processes, which have consumed their specified processing time, is introduced. This leads to the following priority scheme:

1. Critical processes (guaranteed and statistical).
2. Critical processes that have used up their processing times as specified by their workload descriptions, but require further processing.
3. Processes not performing multimedia-operations.
4. Workahead processes.

Within this scheme, a statistical process executes in priority class 1 until it has consumed its specified processing time. Then it is moved to priority class 2, which may lead to a preemption in favor of a different process from class 1. The detection that a process has consumed its processing time requires the supervision of execution times, which is not possible in the chosen operating system (AIX). An efficient implementation of such a mechanism would require a kernel modification which is not possible by the kernel modification facilities in AIX. Additionally, assume the case of communication processing where packets of one connection may be served by different processes. In this case, the above scheme would require special attention on proper sequencing of packets, i.e. a newly arriving packet is not allowed to be processed in priority class 1 while an older packet of the same stream is waiting in class 2. Owing to these reasons the first priority scheme is used, despite its described drawback.

Another item is the trade-off between the gain of processing workahead packets prior to their logical arrival times and the overhead of changing the priorities between critical and workahead status. Since this overhead can be significant, the 'standard' version of the scheduler used in HeiTS does not perform processing of workahead packets but leaves workahead packets unprocessed till their logical arrival time; yet, it is possible to compile a version including workahead processing.

Note that the scheduling approach described here is rather simple: The partitioning of priorities into a four-level scheme could be regarded as somewhat rigid. Also, more sophisticated approaches to support the scheduling of aperiodic requests, e.g. data stream management and non-multimedia computations, are possible. There exist solutions to reserve a specified capacity of the bandwith to those aperiodic tasks, e.g. the sporadic server[21] or the slack stealing algorithm.[22]

However, an efficient implementation of such schemes requires a modification of kernel data structures and the ability to supervise and stop the execution of a process. See, for instance, the implementation of the sporadic server in Reference 23. As our intention was to build a scheduler on top of a general-purpose operating system, we had to refrain from using those elaborate approaches.

## Schedulability test and priority assignment scheme

The target operating system for the implementation is AIX, IBM's UNIX derivate. In addition to the well-known *multi-level-feedback*[24] (MLFB) scheduling it provides a set of *fixed* priorities at the highest priority levels (priorities 0–15), which are even higher than the AIX scheduler's priority. Unlike the other (MLFB) priorities these priorities are not modified by the AIX scheduler and can be used for real-time processing.

Assigning priorities to processes produces a considerable overhead that cannot be neglected. Therefore, we do not use a *dynamic* scheme such as earliest deadline first (EDF) but use a *static* priority assignment scheme according to the rate monotonic (RM) algorithm[25] where a process with a short period (i.e. a high rate) receives a high priority. Priorities are computed at application establishment time and are not changed dynamically during application lifetime. Only when a newly established application needs a priority level that is already in use are the existing priorities shifted to make room for the new application handling process. With the priority scheme described in the previous section, the priorities are ordered in such a way that guaranteed processes possess the highest priorities and statistical processes use the lower part of the real-time priorities. All processes not subject to real-time constraints are handled by the AIX system scheduler and use priorities below the real-time priorities.

RM scheduling has also the advantage that a simple schedulability test exists. A set of real-time applications can be accepted with respect to the CPU load, i.e. no overload condition occurs, if the following inequality holds:[25]

$$\sum_{i=1}^{n} R_i P_i \leq U_n = n(2^{1/n} - 1)$$

The parameters of this inequality and their meanings are as follows: index $i$ runs through all $n$ real-time application handling processes $[T_1, \ldots, T_n]$, $R_i$ denotes the maximum rate of $T_i$, and $P_i$ specifies the processing time per packet of $T_i$. $U_n$ is a non-negative real number of value at most ln(2) ($\approx 0.69$) for RM scheduling of processes with arbitrary rates. The limit of $U_n$ (for $n$ approaching infinity) is $U = \ln(2)$ ($\approx 0.69$).

If the sum on the left hand side, i.e. the load generated by all real-time processes, does not exceed $U$, the processing of all packets is guaranteed to terminate within their respective deadlines $1/R_i$. If the sum is greater, this may still be the case, but no guarantees can be given with this test.

The processing times $P_i$ include the scheduling overhead, i.e. the overhead for inserting, choosing and removing a process from the run queue, and the overhead for blocking and awakening a workahead packet. A tool[26] has been developed to measure these times. The total of the overheads can be incorporated into the processing time $P_i$ in a similar way as is described in Reference 27.

Of the described time handling methods, only the last two, periodic threads or processes, can provide the necessary information to the scheduler. Thus, the other methods are not usable if timing guarantees have to be given.

It should be noticed here that the schedulability boundary $U$ can be relaxed in certain cases. If the periods of the processes have a certain ratio, $U$ can be larger than $\ln(2)$. For instance, if the periods are (integer) multiples of the smallest period in the process set, then $U = 1$ can be chosen. Also, Reference 28 showed that the maximum CPU load which can be accepted for RM scheduling is, in the average case, notably larger than $\ln(2)$.

However, the restriction of the maximum CPU utilization $U$ for multimedia processing to a value smaller than 1 is not such a strong limitation as it might seem. In any case, some CPU capacity has to be reserved to processes other than multimedia related processes. Owing to this reason and the simplicity and efficiency of the Liu/Layland schedulability test we do not use the more advanced analysis from Reference 28. Although the latter would enable the system to accept a greater number of real-time applications simultaneously, the Liu/Layland scheduling bound usually suffices for our purposes.

The Liu/Layland schedulability test can be applied not only to a static task set but also to dynamically arriving tasks, as was shown in Reference 29. More advanced tests, for instance those as described in Reference 30, are not used due to the above reasons.

### Buffer space

One issue that has to be considered in multimedia QoS management is the reservation of buffer space for the packets waiting to be processed. The longer packets must wait for a resource, the more memory space must be available to avoid losses. A detailed discussion of this problem is beyond the scope of this paper. For a stream defined by the LBAP model, the amount of buffer space to be reserved on a node can be calculated from the LBAP parameters and the maximum delay of packets on the node, for details see Reference 31. Reference 18 shows that even the requirements of VBR streams can be satisfied with a moderate amount of storage space, because the maximum delay bound to be enforced for multimedia streams is rather small.

## IMPLEMENTATION

The functionality of the real-time CPU scheduler in HeiTS consists mainly of two parts, the management of the information needed for proper scheduling and the actual scheduling of processes.

### Management of scheduling information

A 'scheduling cache' is used to store all information needed for scheduling the processing of the individual streams.* Several functions for management of cache entries are provided. During the creation of an application the information which characterizes the stream is inserted in the scheduling cache by means of the function `rms_cpu_create_entry` and can be freed during connection release by the function `rms_cpu_release_entry`. Since QoS parameters may be changed during the lifetime of an application, e.g. the rate is lowered, there must be a possibility to report this

* Each entry in this cache is associated with one process and can be compared to the attributes object associated with a thread in the Posix threads extension.[9]

change to the scheduler. This can be achieved by calling the function `rms_cpu_change_entry`.

## Scheduling of processes

The actual scheduling is performed through a set of kernel functions (AIX provides mechanisms for adding such system calls) that must be called by the process that wants to be scheduled. This is more efficient than implementing the scheduler as a separate process (like the AIX system scheduler) because it saves the context switch between the process to be scheduled and the scheduler process itself.

Requiring that the process calls the scheduler function explicitly leads to 'voluntary scheduling' and may seem dangerous. However, all code allowed to run in an environment where it is possible to use real-time priorities has to be established by an authorized user. Thus, only approved code will be subject to real-time scheduling and, therefore, especially with reflection on the performance gain, this approach can be regarded as secure.

### *Process structure*

To achieve proper scheduling of real-time processes some assumptions about the structure of the processes have to be made. As shown in Figure 3, it is assumed that after creating an application the process responsible for handling the data of this application is performing a program loop and processes one data packet (e.g. a video frame) in every iteration. This continues until the application is finished and the process is not subject to real-time scheduling any more.

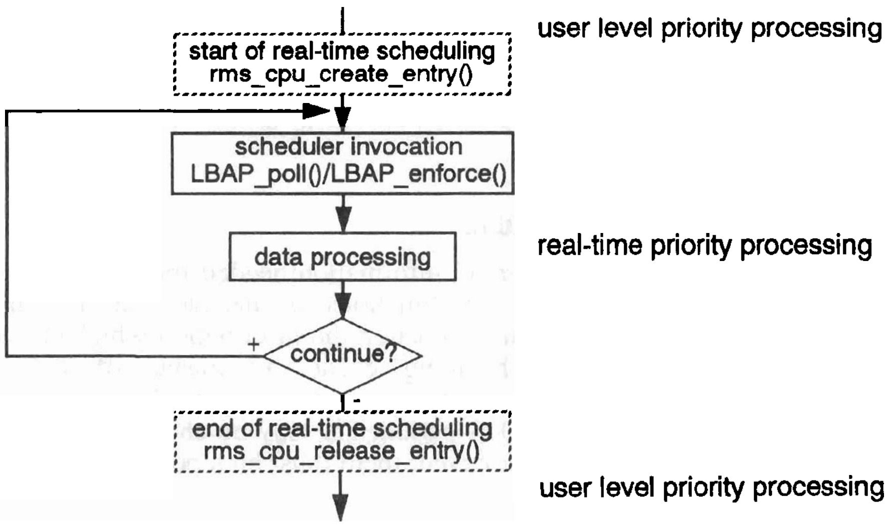
start of real-time scheduling
rms_cpu_create_entry()
user level priority processing
scheduler invocation
LBAP_poll()/LBAP_enforce()
data processing
real-time priority processing
+
continue?
-
end of real-time scheduling
rms_cpu_release_entry()
user level priority processing

*Figure 3. Processing structure*

*Enforcing LBAP property*

Before processing a newly arrived data packet the scheduler must check whether accepting this packet would violate the LBAP characteristic (i.e. the workload specification) of the data stream. This check can be done in a blocking or a non-blocking way. The blocking test is performed by the function **`LBAP_enforce`** and enforces the observance of the LBAP property of the data stream: the process is left in a wait state until the logical arrival time of the packet is reached.*

In the non-blocking test implemented in the function **`LBAP_poll`** the scheduler simply returns the calculated logical arrival time of the data packet and the information whether accepting this packet violates the LBAP properties of the data stream or not.† For all non-blocking tests of the LBAP property, it is the process' responsibility to take proper action if the packet violates the LBAP properties of the application (one possibility would be to call **`LBAP_enforce`**).

Non-blocking functions are provided to support the usage of monotonic computations where the quality of intermediate results does not decrease as it executes longer, e.g. compression algorithms as JPEG[32] or MPEG-II.[33] Such algorithms produce a preliminary result after a certain time which can be improved through further computations. Therefore, after an intermediate result has been reached, it has to be checked in a non-blocking way whether there is enough time left for further operations. If there is time available, the result will be improved otherwise the current result will be used, e.g. transferred to the consumer.

This is comparable to the milestone method mentioned in Reference 34. It should be noted that the used schedulability test further supports this type of algorithms since it leaves a portion of the processor capacity that can be used for the improvement of intermediate results.

The watchdog mechanism of the scheduler is especially useful for program development. It provides a method to get the system under control if some real-time process hangs in an endless loop. The watchdog also checks whether a process does not call the scheduling functions. Calling with a higher rate than specified is not possible since the scheduler code blocks the process until the logical arrival time. Calling never or with a much lower rate than specified is an indication that either the specification was substantially wrong (and should be changed, e.g. via **`rms_cpu_change_entry`**) or the process does not behave correctly and some management action has to be taken.

## EVALUATION

To show the effect of using the scheduler for different multimedia applications a series of measurements were performed. They should answer the following question: in which way does the use of the scheduler influence the behavior of the application and the system as a whole, i.e. are deadline violations indeed avoided and to what extent? Qualitative aspects such as expressing time characteristics are not considered in this section since they have been discussed already before.

* This is true for the standard version where the scheduler does not process workahead packets. If workahead is allowed the process is blocked till it can accept another workahead packet, i.e. the time till the logical arrival time is equal to or less than the time needed to process the maximum allowed workahead minus one period (needed to process this newly arrived data packet).

† If workahead packets are not processed the priority is left unchanged because it is the goal to avoid the overhead of priority changes. If workahead processing is performed the priority of the process is set to the workahead priority until the logical arrival time of the packet is reached.

## Measurement setup

The CPU scheduler function `LBAP_enforce` was instrumented in such a way that it generates events describing the laxity of the calling process, i.e. the time until the process reaches its deadline. Positive values indicate that the process still has time before the deadline is reached; therefore, it is operating correctly. Negative values indicate that the process violated its deadline; it is not able to perform its function in time.

In those cases where several real-time processes were running concurrently the events are given in generation-time order, i.e. they are not ordered by processes unless otherwise stated. The charts shown in Figures 4–9 below are extracts from much longer measurement series to increase readability. Each of them shows 200 values which have been taken from the middle of the sequence of values, the generation of measurement values having started later than the processes under consideration to reduce start-up effects. Each point in a graph represents a single event. The measurement values are given in seconds.

All measurements were performed on a mostly idle workstation, an IBM RISC System/6000, Model 360 with AIX 3.2.4, which was not modified during the measurements, e.g. simple applications such as mail, etc. were running as usual. However, none of these programs used much CPU processing time. These types of applications are running during normal workstation operation periods as well, thus, disabling them during the measurements might lead to slightly more regular measurement results but not to results which are better applicable to real-world scenarios.

The measurements were performed with a varying system load. The system load was generated artificially by synthetic, non real-time, computation processes performing simple integer calculations. Hence, in principle these processes were always ready to run, which also led to low priority due to UNIX scheduler characteristics.[24] Therefore, normal, user-created system load might be even harder than this synthetic load. 0, 1, 2, 3, 4, or 16 of these load processes have been used during the measurements. Running 16 processes leads to a heavily loaded system, the other loads resemble loads easily created during normal workstation operation.

The measurements were performed with programs using the CPU scheduler's real-time characteristics followed by measurements with the same programs without performing real-time scheduling using the time provision mechanisms of the scheduler, i.e. executing with the specified rate. The load generated by the programs is the same in both cases—since the static RM scheduling algorithm without workahead scheduling is used, no additional costs for the real-time processes during run time occur.

## Considered scenarios

Two basic application scenarios with different setups were investigated:

1. an endsystem application,
2. a video-on-demand server application.

In the first scenario, usually relatively few processes are running, performing operations such as software compression and decompression. For instance, in a video conference, one participant has to compress its own image before this is transmitted to the peers and it has to decompress the images received from the other peers. Hence, for a conference with *n* participants *n* processes for software compression and decompression exist on each workstation. Since compression algorithms for video conferencing such as

H.261[35] usually possess symmetric processing requirements, in the following it is not distinguished between compression and decompression processes. Another example of the endsystem scenario is a playback application presenting a video decompressed in software to the user; there only one process exists.

Within the second scenario, a video-on-demand server, several processes are active in the system, one for each data stream served. However, the processing requirement of such a process is lower than for a software decompression process.

In the following the results for the endsystem scenario are described first, then the measurements for the video-on-demand server scenario are discussed.

## End-system scenario

For the end-system scenario, a video playback program and a synthetic program have been examined. The video playback program reads compressed video data, decompresses the data in software, and presents the video frames via the X server to the user. The synthetic program performs simple calculations and data movements on arrays to resemble a playback program. The reason for using the synthetic program is that this has a more repeatable characteristic and allows for arbitrary modifications of processing time requirements. Hence, it provides a more stable environment and a broader range to study the behavior of the scheduler.

### *Video playback*

The video playback program uses one process for its operations, i.e. $n = 1$. The chosen video consists of 15 frames/s, i.e. 66·6 ms/frame, which was also set as the processing rate of the program. The processing time needed per period is on the average approximately 28 ms, which results in a total CPU usage of about 0·42.

The compressed data read by the program was stored in a local file which was cached into main memory by running the program first without measuring it. The file was small enough to fit into the cache.

Figure 4 shows the results for measurements with varying loads. If no load except the measured process exists in the system, no deadline violations occur even without using real-time scheduling.

If a load of medium size, i.e. three or more processes, is introduced into the system, the considered application is not able to provide acceptable service to the user. The last graph in the Figure illustrates that by using real-time scheduling, the application does not suffer from any deadline violations, even if a high load—up to 16 processes—has been introduced into the system.

### *Synthetic end-system program—one process*

The synthetic program operates with the same rate of 15 1/s as the video playback program; its processing time requirement of about 21 ms per iteration is lower than that of the video playback program. The reason is that the generated load of 21 ms $\times$ 15 1/s = 0·315 is lower and allows more concurrent processes to be measured, thus creating a heavier load.

The different CPU requirements have no major impact on the results, since the CPU utilization of the video playback program could be lowered to that of the synthetic

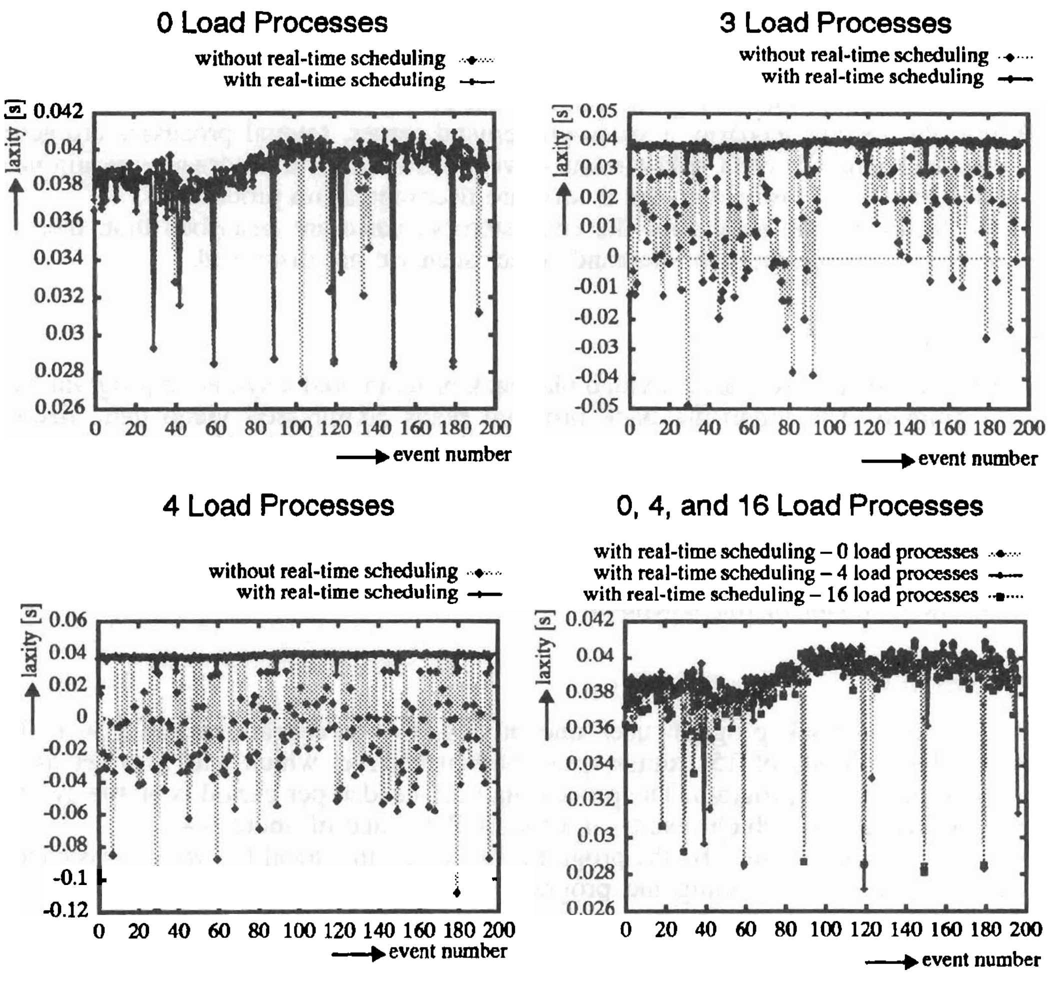

*Figure 4. Video playback application*

program, e.g. by reducing the frame size or using a different compression algorithm. To reduce the influence of other programs and system components the program performs no I/O. The synthetic program has been used since it has a more regular CPU utilization per iteration which increases the comparability of the values.

The achieved results are similar to the results for the video playback measurements as can be seen in Figure 5. The workstation can cope with the non-real-time program if the system is otherwise idle. Introducing an artificial load of three or more processes leads to deadline violations. The real-time program runs without any problem for all system loads; the laxity varies within tight bounds, all values except one are contained in an interval with a width of about 1 ms, the single value is outside of this interval by about 1 ms.

Reasons for the variations include interrupts and functions inside the operating system kernel which block timer interrupts leading to a delayed switch to the real-time process.

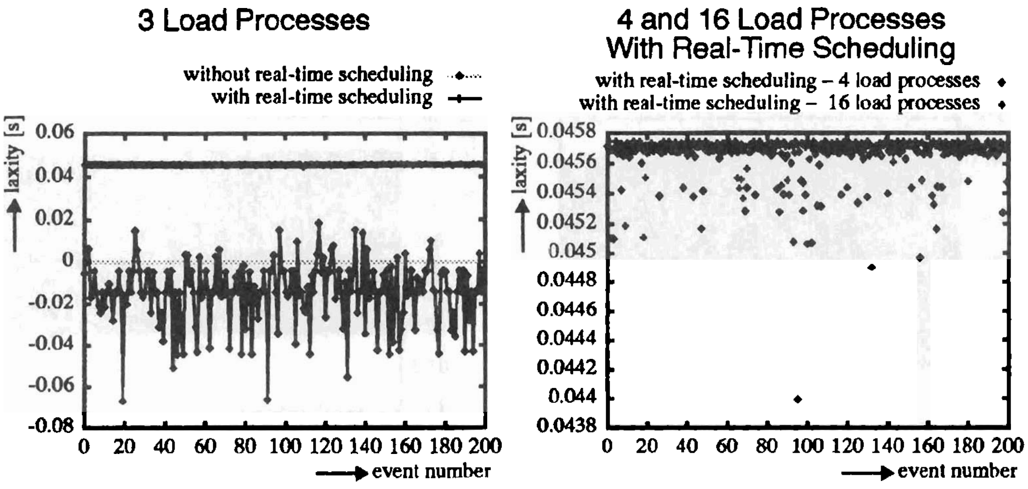

*Figure 5. Synthetic 'decompression' program, one process*

Sampling complete system traces including kernel functions introduces too much overhead for the measurements and modifies the behavior. Thus, we cannot give a complete explanation for the measured deviation. Many aspects in a general purpose computer system are difficult to predict; for instance, context switches influence cache performance.[36] However, we consider the reached accuracy as fully sufficient, for instance the synchronization requirement for audio and video, e.g. lipsynch, has been found to be 80 ms.[37]

*Synthetic end-system program—two and three processes*

Multimedia applications may use more than one decompression process, e.g. in a video conference between two persons one compression and one decompression process is running per system; for a conference with three participants on each system already a total of three (de)compression processes are running. Therefore, measurements for a system running two or three concurrent processes have been performed, each executing the synthetic program described above running with a rate of 15 1/s and a CPU utilization of about 0·315. The results are shown in Figures 6 and 7; since the processes are running at the same rate, the maximum acceptable CPU load under RM scheduling is 1.

As Figure 6 shows, the workstation can handle two non-real-time (de)compression processes as long as either no load is introduced or only one other process is running. With only two load generating processes, the non-real-time decompression processes are no longer able to keep within their deadlines. As can be seen in Figure 6 the real-time processes perform their operations in time even for medium and high loads.

The reason for the regular patterns is that the plots show the laxity of all processes ordered by generation time. This way, events of processes with large laxity and with small laxity are mixed and, since the execution of the processes is ordered, the lines connecting single points lead to the patterns; see also the discussion for Figure 7, below, of the measurement of three processes.

If three non-real-time (de)compression processes are executed, already one load-gen-

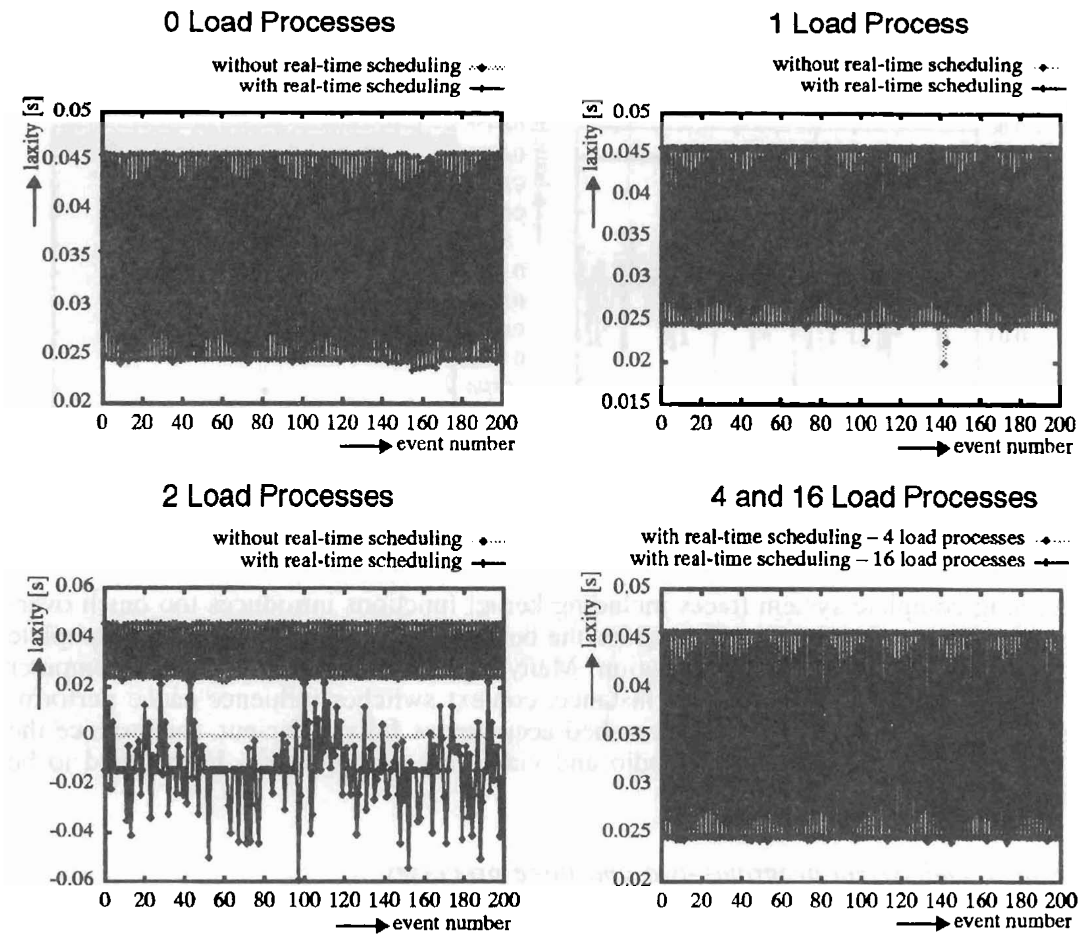

*Figure 6. Synthetic 'decompression' program, two processes*

erating process is sufficient that the system cannot provide its service in time, as can be seen in Figure 7. Since starting a process is a common operation in UNIX workstations it cannot be assumed to be avoidable, hence, it can be expected that users would not accept the offered presentation because deadline violations occur which lower the overall quality. Again, using real-time processes, the workstation provides correct service even for high loads.

The plot of the measurements for the three real-time processes running during medium and high additional workstation load without lines connecting the points in Figure 7 on the left side, bottom row, shows that the laxity of the processes is either 45·6 ms, 24·5 ms or 3·5 ms. The reason is that the real-time processes execute alternately and without interruption by each other. This is illustrated by Figure 7 on the right side, bottom row where the measurement for high load is plotted using a different pattern for each process.

From this graph it can be seen that in each iteration, the laxity of the first process is about 45·5 ms and those of the second and third processes are 24·5 ms and 3·5 ms,

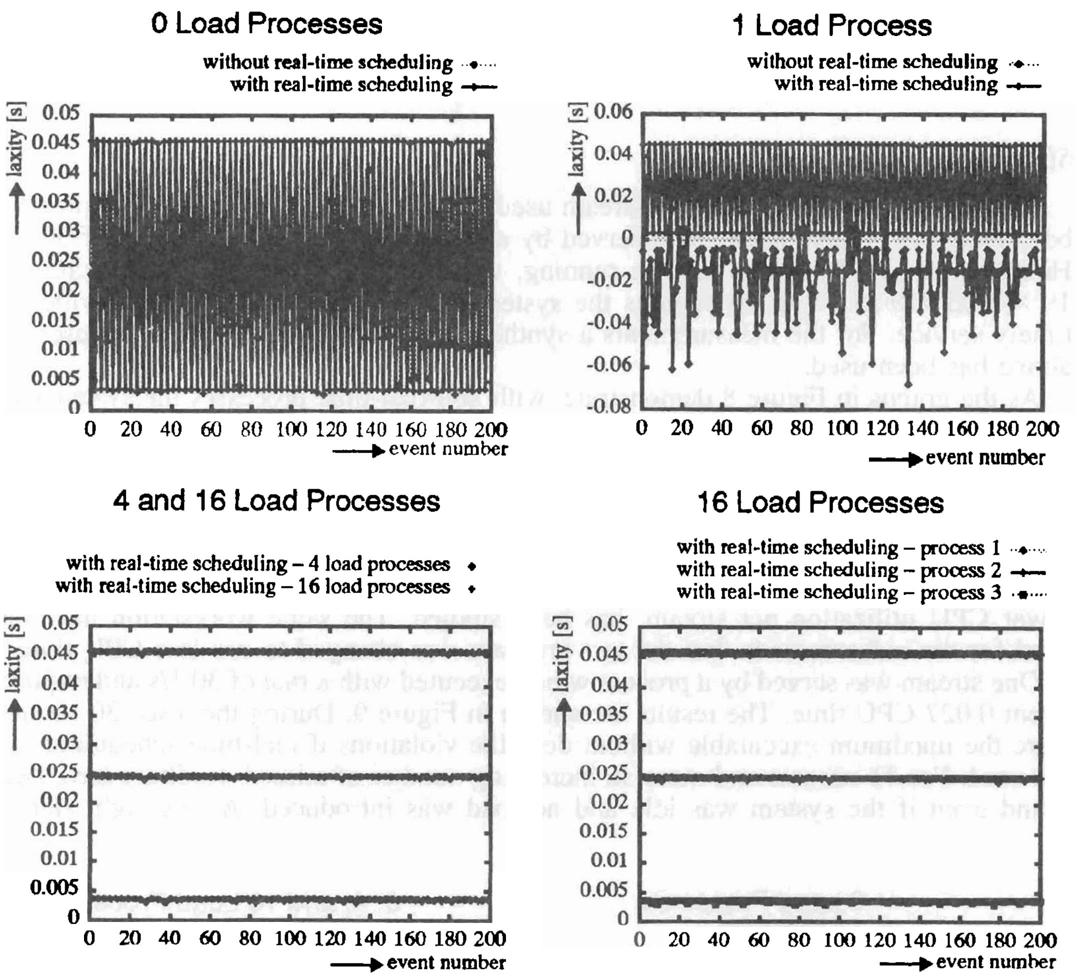

*Figure 7. Synthetic 'decompression' program, three processes*

respectively. The last 'segment' of 3·5 ms is not used by any real-time process, which means that 3·5 ms × 15 1/s = 0·0525 CPU time has been left.

This is in accordance with a per process CPU utilization of 0·315, which yields a total CPU utilization of 0·945. If the laxity of a process would alternate, the plot lines would cross the graph and yield a pattern as, for instance in the first graph seen on the left side, upper row.

## Server scenario

In a video-on-demand scenario two different interest areas exist. The client, typically using the system for playback, wants a reliable service from the server just as in the end-system scenario described above. The service provider, i.e. the owner of the server, wants to be able to serve as many streams as possible from one system without degradation of QoS since otherwise customers will be dissatisfied.

The measurements presented in the following show that using real-time processes instead of non-real-time processes enables a guaranteed service and a larger number of concurrent streams and, hence, lowers the costs per stream.

### *Slow server*

First, a scenario where a single stream used up about 0·05 of the total CPU time has been examined. Each stream was served by a process operating with a rate of 30 1/s. Hence, at most 19 streams can be running, which means that the CPU utilization is $19 \times 0{\cdot}05 = 0{\cdot}95$. With 20 streams the system is overloaded and cannot provide any timely service. For the measurements a synthetic program similar to the one described above has been used.

As the graphs in Figure 8 demonstrate, with non-real-time processes the system cannot serve 17 streams. Using real-time processes, all 19 streams can be served even if high additional load is introduced into the system.*

### *Fast server*

Finally, the behavior of a server which is able to serve more streams, i.e. with a lower CPU utilization per stream, has been studied. The same workstation has been used for the measurements but the test program was changed to use less CPU time.

One stream was served by a process which executed with a rate of 30 1/s and required about 0·027 CPU time. The results are shown in Figure 9. During the tests, 30 streams were the maximum executable without deadline violations if real-time scheduling was not used. For 31 streams and more an increasing number of missed deadlines have been found even if the system was idle and no load was introduced. As the right side of

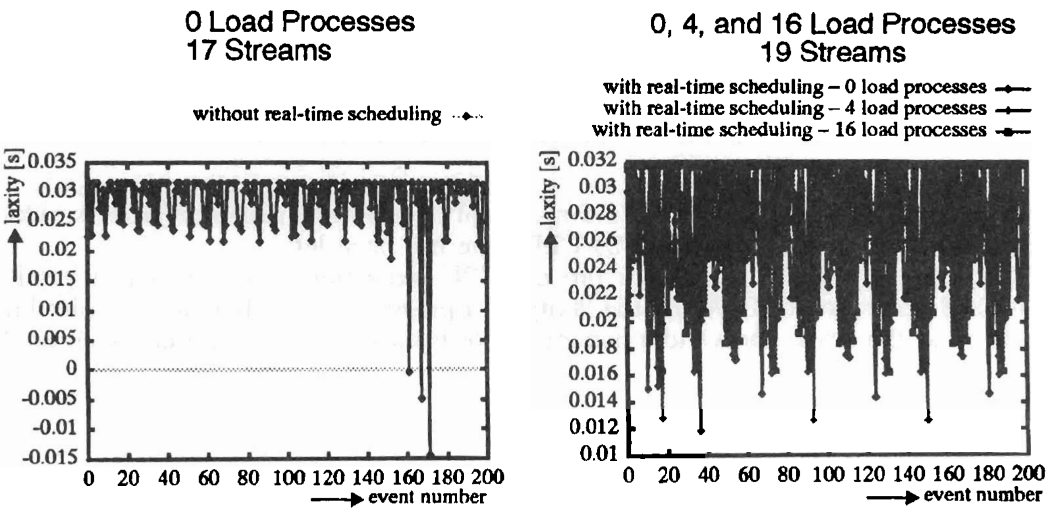

*Figure 8. Synthetic 'server' program, 17 and 19 streams*

* The execution sequence of the processes is not ordered, since the 19 processes must be mapped to fewer priorities, leading to switches between processes.

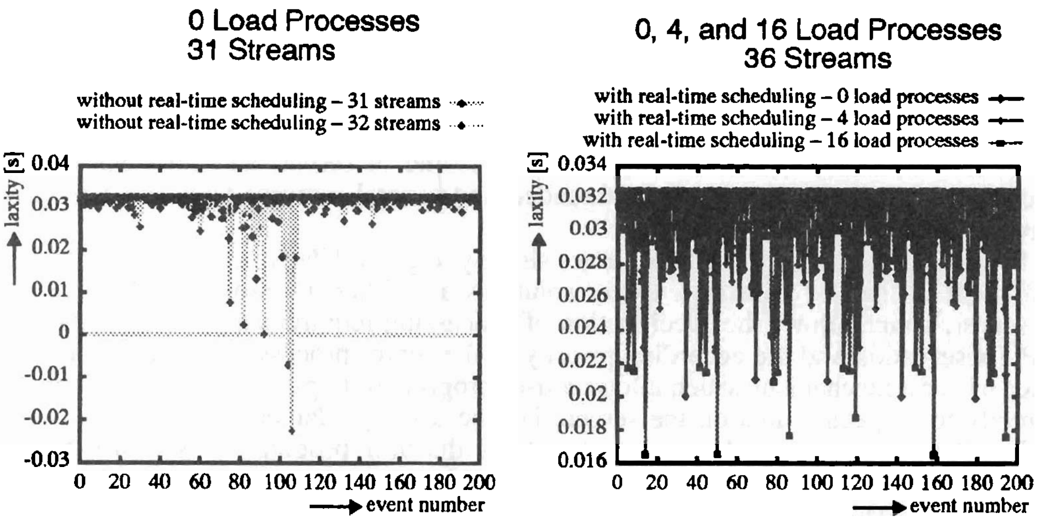

*Figure 9. Synthetic 'server' program on 'faster' server, 31, 32 and 36 streams*

Figure 9 demonstrates, using the real-time scheduler it was possible to execute 36 streams, yielding a total CPU utilization introduced through these real-time processes of 0·972, even if high load were introduced.

## RELATED WORK

Real-time mechanisms for multimedia systems are provided by several research systems. In most cases, these are based on a newly developed operating system kernel, and hence the problems of integrating the mechanisms into an existing kernel and the corresponding restrictions do not occur.

DASH[19] uses a deadline driven scheduling algorithm. As described before, due to the period-based process dispatching and the considerable overhead for priority changes, this approach is not useful for our scenario.

Sun's High Resolution Video (HRV) workstation project assumes that no deterministic bounds can be provided; thus, no guaranteed processing is available.[38] For several 'production-level' applications, we consider guaranteed processing to be so important that neglecting them is not acceptable.

In YARTOS (Yet Another Real-Time Operating System)[39,40] a new operating system kernel is designed. The task model is based on sporadic (instead of periodic) tasks. The schedulability test considers all accesses to shared resources, which are only available via kernel mechanisms, and avoids contention situations. Hence, the mechanisms are not usable in conjunction with standard kernels.

Reference 41 describes a system similar to ours which yields comparable results. Their work is based on Real-Time Mach,[42] hence, due to the micro-kernel their approach is not usable in our operating system environment.

Other work from the field of real-time systems has already been quoted in the sections describing the scheduling algorithm and its implementation.

## POSSIBLE CLIENT SYSTEM ENHANCEMENTS

Using another process to present images to the user can lead to problems if this process is not under the control of the multimedia system. This is, for instance, the case in X Windows; here, the X server process displays the images. Even if shared memory between server and client is used, a non-real-time X server can introduce deadline violations, especially if it is single threaded and several requests from other programs have to be executed.

Increasing the priority of the X server slightly, e.g. via UNIX 'nice' mechanism, was sufficient in the test scenarios. Better solutions are either the provision of a real-time X server, which allows the specification of processing requirements or the 'transfer' of CPU reservation and the according priority to the server process as suggested in Reference 41, or a mechanism which allows a user program to bypass the X server by writing directly to a specific area on the screen, i.e. the display adapter memory.

For the latter, the window manager allows the user program to write to that area where its window is mapped by attaching the memory to the program's address space via a special system call; other memory areas may still be protected. Any change in the visibility, size, or location of the window is known inside the window manager which can change or withdraw the memory from the program's address space accordingly.

## CONCLUSIONS

The inherent periodicity of continuous-media data requires operating system provided mechanisms for timely operation. Simple methods, e.g. functions which only delay the execution of certain functions, are not suitable for general-purpose multimedia systems. Information about the programs' time characteristics are needed to apply real-time scheduling techniques which are a prerequisite for reliable QoS provision.

This paper discussed several approaches for time handling and described a real-time scheduling method and its implementation for a standard operating system kernel. Several multimedia applications (e.g. a video server) have been implemented successfully using the described scheduler.

The experimental evaluation shows that real-time scheduling is indeed necessary for end-system and video-on-demand server applications. The measurements demonstrate that the scheduler is able to provide QoS guarantees even for highly loaded systems.

### ACKNOWLEDGEMENTS

The authors would like to thank the anonymous referees for their comments and suggestions.

### REFERENCES

1. Andreas Mauthe, Werner Schulz and Ralf Steinmetz, 'Inside the Heidelberg multimedia operating system support: real-time processing of continuous media in OS/2', *Technical Report 43.9214*, IBM European Networking Center, Heidelberg, Germany, 1992.
2. Lars C. Wolf and Ralf G. Herrtwich, 'The system architecture of the Heidelberg Transport System', *ACM Operating Systems Review*, **28**(2), 51–64 (1994).
3. John Barnes, 'Introducing Ada9X', *ACM Ada Letters*, **13**(6), 61–132 (1993).
4. David B. Stewart, Donald E. Schmitz and Pradeep K. Kosla, 'The Chimera II real-time operating system for advanced sensor-based control applications', *IEEE Trans. on Systems, Man and Cybernetics*, **22**(6), 1282–1295 (1992).

5. Karsten Schwan and Hongyi Zhou, 'Real-time threads', *ACM Operating Systems Review*, **25**(4), 35–46 (1991).
6. Hideyuki Tokuda and Clifford W. Mercer, 'ARTS: A distributed real-time kernel', *ACM Operating Systems Review*, **23**(3), 29–53 (1989).
7. Jun Nakajima, Masatomo Yazaki and Hitoshi Matsumoto, 'Multimedia/realtime extensions for the Mach operating system', *Proceedings of the Summer 1991 Usenix Conference*, Nashville, Tenn., 1991, pp. 183–198.
8. Brian N. Bershad, Edward D. Lazowska and Henry M. Levy, 'PRESTO: A system for object-oriented parallel programming', *Software—Practice and Experience*, **18**(8), 713–732 (1988).
9. IEEE Standards Project P1003.4a, 'Threads Extension for Portable Operating Systems, Draft 6', February 1992.
10. IEEE Standard for Information Technology, Std 1003.1b-1993, 'System application program interface (API)—amendment 1: realtime extension', September 1993.
11. Sandeep Khanna, Michael Sebree and John Zolnowsky, 'Realtime scheduling in SunOS 5.0', *USENIX*, Winter 1992.
12. R. L. Cruz, 'A calculus for network delay, part I: network elements in isolation', *IEEE Trans. Information Theory*, **37**(1) (1991).
13. Martin Andrews, 'Guaranteed performance for continuous media in a general purpose distributed system', *Masters Project Report*, University of California, Berkeley, October 1989.
14. American National Standards Institute, 'Integrated services digital network (ISDN)—digital subscriber signaling system No.1 (DSS1)—signaling specification for frame relay bearer service', *ANSI T1.617–1991*, June 1991.
15. Israel Cidon, Inder Gopal and Roch Guerin, 'Bandwidth management and congestion control in plaNET', *IEEE Communications Magazine*, **28**(10), 54–63 (1991).
16. Domenico Ferrari, Anindo Banerjea and Hui Zhang, 'Network support for multimedia: A discussion of the Tenet approach', *TR-92–072*, International Computer Science Institute, Berkeley, CA, USA, 1992.
17. Mark Moran and Bernd Wolfinger, 'Design of a continuous media data transport service and protocol', *TR-92–019*, International Computer Science Institute, Berkeley, CA, USA, 1992.
18. Carsten Vogt, 'Quality-of-service management for multimedia streams with fixed arrival periods and variable frame sizes', *ACM Multimedia Systems*, 3(2), 66–75 (1995).
19. David P. Anderson, 'Metascheduling for continuous media', *ACM Trans. Computer Systems*, **11**(3), 226–252 (1993).
20. Carsten Vogt, Ralf Guido Herrtwich and Ramesh Nagarajan, 'HeiRAT: the Heidelberg Resource Administration Technique—design philosophy and goals', *Kommunikation in Verteilten Systemen*, Munich, Germany, March 1993.
21. Brinkley Sprunt, Lui Sha and John Lehoczky, 'Aperiodic task scheduling for hard-real-time systems', *Real-Time Systems*, **1**, 27–60 (1989).
22. John P. Lehoczky and Sandra Ramos-Thuel, 'An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems', *Proceedings of the IEEE Real-Time Systems Symposium*, 1992, pp. 110–123.
23. Brinkley Sprunt and Lui Sha, 'Implementing sporadic servers in Ada', *Technical Report CMU/SEI-90-TR-6*, Carnegie-Mellon University, Software Engineering Institute, Pittsburgh, PA, USA, May 1990.
24. Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels and John S. Quarterman, *The Design and Implementation of the 4.3-BSD UNIX Operating System*, Addison-Wesley, Reading, Mass., 1989.
25. C. L. Liu and James W. Layland, 'Scheduling algorithms for multiprogramming in a hard-realtime environment', *Journal of the ACM*, **20**(1), 47–61 (1973).
26. Hartmut Wittig, Lars C. Wolf and Carsten Vogt, 'CPU utilization of multimedia processes: the HeiPOET measurement tool', *Proceedings of the Second International Workshop on Advanced Teleservices and High-Speed Communication Architectures*, Heidelberg, Germany, September 1994.
27. Daniel I. Katcher, Hiroshi Arakawa and Jay K. Strosnider, 'Engineering and analysis of fixed priority schedulers', *IEEE Trans. on Software Engineering*, **19**(9), 920–934 (1993).
28. John Lehoczky, Lui Sha and Ye Ding, 'The rate monotonic scheduling algorithm: exact characterization and average case behavior', *Proceedings of the Tenth IEEE Real-Time Systems Symposium*, Santa Monica, CA, USA, 1989, pp. 166–171.
29. Lui Sha, Ragunathan Rajkumar, John Lehoczky and Krithi Ramamritham, 'Mode change protocols for priority-driven preemptive scheduling', *Real Time Systems*, **1**(3), 243–263 (1989).

30. Ken Tindell, Alan Burns and Rob Davis, 'Fixed priority scheduling of hard real-time multi-media disk traffic', *Proceedings of the IEEE Workshop on Real-Time Issues in Multi-Media*, November 1993.
31. David P. Andersen, Ralf G. Herrtwich, Carl Schaefer, 'SRP: a resource reservation protocol for guaranteed-performance communication in the Internet', *TR-90–006*, International Computer Science Institute, Berkeley, CA, USA, Februray 1990.
32. Gregory K. Wallace, 'The JPEG still picture compression standard', *CACM*, **34**(4), 30–44 (1991).
33. ISO IEC JTC1/SC29/WG11, 'Generic coding of moving pictures and associated audio (MPEG-2)', *International Standard ISO/IEC IS 13818*, November 1994.
34. Jane W. S. Liu, Kwei-Jay Lin, Wei-Kuan Shih, Albert Chuang-shi Yu, Jen-Yao Chung and Wei Zhao, 'Algorithms for scheduling imprecise computations', *IEEE Computer*, **24**(5), 58–68 (1991).
35. Ming Liou, 'Overview of the p × 64 kbit/s video coding standard', *CACM*, **34**(4), 59–63 (1991).
36. Jeffrey C. Mogul and Anita Borg, 'The effect of context switches on cache performance', *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, April 1991. Also in *Operating Systems Review*, **25** (Special Issue), 75–84 (1991).
37. Ralf Steinmetz and Clemens Engler, 'Human perception of media-synchronization', *Technical Report 43.9310*, IBM European Networking Center, Heidelberg, Germany, 1993.
38. James G. Hanko, Eugene M. Kuerner, J. Duane Northcutt and Gerard A. Wall, 'Workstation support for time-critical applications', *Proceedings of the 2nd International Workshop on Network and Operating System Support for Digital Audio and Video*, Heidelberg, Germany, 18–19 November 1991.
39. Kevin Jeffay, Daniel E. Poirier, F. Donelson Smith and Donald L. Stone, 'Kernel support for live digital audio and video', *Proceedings of the 2nd International Workshop on Network and Operating System Support for Digital Audio and Video*, Heidelberg, Germany, 18–19 November 1991.
40. Kevin Jeffay, Donald. L. Stone and Daniel E. Poirier, 'YARTOS: kernel support for efficient, predictable real-time systems', *Proc. IFAC, Workshop on Real-Time Programming*, Pergamon Press, Atlanta, May 1991.
41. Clifford W. Mercer, Stefan Savage and Hideyuki Tokuda, 'Processor capacity reserves: operating system support for multimedia applications', *Proc. First International Conference on Multimedia Computing and Systems*, Boston, MA, USA, 17–19 May 1994.
42. Hideyuki Tokuda, Tatsuo Nakajima and Prithvi Rao, 'Real-time Mach: toward a predictable real-time system', *Proc. USENIX Mach Workshop*, Burlington, VT, USA, 4–5 October 1990, pp. 73–82.