Quality-Assuring Scheduling — Using Stochastic Behavior to Improve Resource Utilization

Claude-Joachim Hamann Jork Löser Lars Reuther Sebastian Schönberg
Jean Wolter Hermann Härtig

Dresden University of Technology
Department of Computer Science
e-mail: drops@os.inf.tu-dresden.de

Abstract

We present a unified model for admission and scheduling, applicable for various active resources such as CPU or disk to assure a requested quality in situations of temporary overload. The model allows us to predict ***and*** *control the behavior of applications based on given quality requirements. It uses the variations in the execution time, i.e., the time any active resource is needed.*

We split resource requirements into a mandatory part which must be available and an optional part which should be available as often as possible but at least with a certain percentage. In combination with a given distribution for the execution time we can move away from worst-case reservations and drastically reduce the amount of reserved resources for applications which can tolerate occasional deadline misses. This increases the number of admittable applications. For example, with negligible loss of quality our system can admit more than two times the disk bandwidth than a system based on the worst-case.

Finally, we validated the predictions of our model by measurements using a prototype real-time system and observed a high accuracy between predicted and measured values.

1 Introduction

The design of systems that behave predictably in situations of overload has obtained a lot of attention in the real-time, networks and operating systems community in the last decade. This interest has been sparked by so-called soft real-time applications that can tolerate occasional misses of deadlines, but cannot afford the waste of resources coming along with absolute guarantees for worst case load situations. The most fashionable examples are media applications where it is widely accepted to drop parts of a stream instead of repeating outdated data packets again. In addition to such an application driven scenario, it became apparent that particularly modern hardware architectures exhibit a widening gap between normal and worst case behavior of systems and hence behave badly with respect to absolute time guarantees [10].

Early work on reservation-based real-time systems focused on the CPU as the resource to be shared. RT Mach's capacity reserves are an example where a process can reserve a certain number of cycles per period. But to simplify the design of a complete system, a sufficient model should not only cover CPU but all resources of interest, such as disk bandwidth or network. Hence, systems such as "Resource Kernels" [15] have extended the initial work. They all use deterministic reservation times, which leads to over allocation of resources since the worst-case must also be covered.

To handle overload situations, the Imprecise Computations Model [11, 13] was introduced. It splits an application into two parts based on importance. The *mandatory part* must be completed, the *optional part* is nice-to-have and improves quality. Another interesting line of results extended deterministic mathematical models by probabilistic elements. An example is Statistic Rate Monotonic Scheduling (SRMS) [2]. It replaces worst-case execution times[1] by probabilistic execution times and introduces a probability for each job by which the deadline is requested to be met.

These approaches have a severe shortcoming in common: they are based on deterministic duration of the resource usage (which leads to a poor resource utilization), or they cannot guarantee a desired quality. Therefore, we propose a scheduling method that enables a better resource utilization *and* that assures a requested quality. Our approach combines the distinction between mandatory and optional parts with varying execution times and includes a quality parameter (i.e., the percentage of optional parts of a periodic task which meet their deadlines). From that an admission algorithm based on a probabilistic model computes a reservation time to schedule each task so that the actual achieved

[1] By execution times we mean the time any active resource is used

quality is the requested quality. This approach fundamentally differs from all known algorithms for imprecise computations. Furthermore, we present an implementation of resource managers for CPU and a SCSI disk subsystem following the model.

The remainder of the paper is organized as follows. Section 2 compiles related work. Section 3 gives an overview about requirements to both, the scheduling and admission model, and the system. The generic model of "Quality-Assuring Scheduling" is introduced in Section 4. The next section describes how we apply the model to two components, a real-time scheduler for CPU, and one for SCSI disks. It encloses a comparison of model predictions and system behavior for each of these resources based on random synthetic load and load generated from a real world example. We conclude the paper with a overhead analysis and an outlook for future work.

2 Related Work

Imprecise computations were invented in the real-time community about ten years ago as a scheduling model for so called flexible computations [12]. Such computations are designed for graceful degradation in result quality or in timeliness for real-time applications, for which a timely result of a poorer quality is better than a late result of the desired quality. The idea is that a mandatory part provides a minimum level of quality. An optional part will then improve the quality if enough resources are available. Scheduling is done such that *all* mandatory parts meet their deadlines while the remaining time is used by the optional parts as far as possible according to an optimal algorithm (i.e., an algorithm that finds a feasible schedule whenever the given set of jobs or tasks has a feasible schedule). Since the problem to find optimal algorithms is NP-hard for most practical cases, a lot of heuristical algorithms were proposed in the literature [13, 12]. Almost all of them have two disadvantages: they are based on deterministic duration of the resource usage, and they cannot assure a desired result quality. Only a few papers study tasks with random execution times, arrival times [4, 13] and even deadlines [3] using a queueing-theoretical approach. For that reason, all these times are assumed to be exponentially distributed - a quite unrealistic assumption in hard real-time systems, in particular for periodic tasks. Furthermore, only statements about mean values result from queueing theory.

Finally, while imprecise computations have been subject to considerable theoretical study in the real-time community and concrete real-time systems apparently have been built on that basis, we know only of one short note [7] to build support for it into and use it as the structuring paradigm for an operating system. And we know of no attempt to use imprecise computations for any other resource than CPU cycles.

The problem of expressing and assuring different levels of quality was addressed first by algorithms using **time-value functions** [8, 16, 9]. Such functions express the gained quality depending on of the required resources for that quality. The solutions are of a rather static nature, because the different achievable levels of quality are considered only at admission time of jobs. That means, the solutions do not consider the jitter in resource needs during execution and they ignore changing load situations due to parallel best-effort applications.

Locke [8] uses time-value functions to represent the benefit of different computation times of a task when providing different levels of quality. Rajkumar's **Q-RAM** approach [16] includes quality dimensions other than timeliness. Multiple non-discrete and concave functions can be specified. They are used to maximize a global objective with given resources. By defining a minimal required level of quality, the mandatory parts of the imprecise computation model can be emulated. Rajkumar relaxes the assumptions on the time-value functions in [9]. He supports discrete QoS operating points which may be obtained by measurements, and are therefore more appropriate than the artificial time-value functions. All these solutions utilize the varying execution times of jobs but changing system load is not taken into account.

A system adapting to changing system load is presented in [1]. The authors describe a CPU-related approach to QoS negotiation. The application specifies a set of utility-gains and resource needs, and the system does admission on this. The set is accepted if, among other conditions, the system can guarantee to fulfill at least one tuple of the set. Each tuple is assigned to a set of tasks that must be executed if the tuple is selected for execution. However, the system targets at long-lived real-time services, and switching between QoS-levels is done upon explicit request after admitting new tasks. Thus, this system also assumes constant resource needs of one task.

An approach that shares our view of quality as miss-rate of jobs is presented in [18, 17]. While the papers focus on packet scheduling for network traffic, the authors believe, that their algorithm is feasible for scheduling other resources as well [18]. The authors address the quality of optional parts by limiting the number of late or missed packets within a time window respective within finite numbers of consecutive packets. This allows them to dynamically adapt to changing resource utilization and to guarantee a specified level of quality [17]. However, in contrast to our approach, the scheduling does not deal with semantic dependencies between packets of one stream.

A probabilistic extension of the classical rate monotonic scheduling approach is presented in [2]. Statistical Rate Monotonic Scheduling (**SRMS**) replaces the worst case execution times by values derived from statistic distributions. The main difficulty is that it relies on the knowledge of the actual execution time of a periodic task at the beginning of each period. This seems impractical, especially if disk accesses are part of the resource admission process.

Regarding resource specification, the work most closely resembling the ideas presented here is the **resource kernel** [14, 15], where a uniform interface to resource usage is presented. Call back mechanisms can be used to notify applications if resources with soft reservations are not available. However, there are two fundamental differences to our work. Firstly, a resource kernel provides access to all resources and manages them. In contrast, various servers can be added to or removed from the system to manage the resources in our approach. Secondly, the resource kernel follows the traditional scheme used in the real-time community to allocate a resource for a certain fraction of a period of time. Overload is not addressed, and nothing is said about the achieved quality or amount of optional resources. In contrast, our approach specifies the varying resources needed to accomplish a job. Based hereof, the needed resources are calculated *a priori*.

3 Overview

Before we describe and discuss the scheduling model and the exemplary implementation for two resources, we describe the requirements to the model and the system.

- An application must be able to specify which parts are critical and must maintain a certain amount of quality (e.g., decoding I and P frames of a MPEG Group of Picture), and which parts are non-critical or optional to improve quality. Some applications require multiple optional parts (e.g., the more B frames are completely decoded the better the quality). We refer to *quality* the fraction of completely executed optional parts of a periodic task.
- All active resources are handled by the same abstraction and a unified model. This allows us to some resource specific extent to use a single set of formulae for admission and scheduling of applications and resources.
- The system must be designed to assure the requested percentage of successful operations.
- The implemented system must allow us to enforce the assignment of resources and to revoke non-guaranteed resources.

The scheduling model is based on periodic use of active resources. The distribution of the variation in the execution time must be known before admission. Applications may split resource requests into several parts which may depend on each other and give a value for the importance. Furthermore, the model provides fixed priorities. Finally, a resource is allocated to a task only for a certain time span called reservation time. The basic idea of our approach is to compute the reservation time of a task in an admission test before the task is started. So the scheduling overhead is kept low and only the requested quality has been achieved. The approach is applicable to all types of flexible computations in the sense of Liu [12]; for lack of space it is demonstrated here for imprecise computations by periodic tasks using the sieve method (i.e., optional parts are discarded in their entirety).

The implementation for CPU is straight forward. Optional parts are scheduled only, if their reservation time is not yet fully used. For CPU, variation is mostly due to the jitter in processing time (e.g., variation in decoding time of video frames) and to a lesser extend to hardware reasons such as cache or TLB impacts.

In contrast to the CPU, the reason of the variance of disk execution times is based on the hardware and not on the requests coming from the application. A disk job is a sequence of single disk requests. The omission of optional parts corresponds to skipping disk requests. That leads to a filesystem implementing "optional streams." Here, blocks of optional streams are dropped and not delivered instead of postponed. Figure 1 illustrates such a behavior.

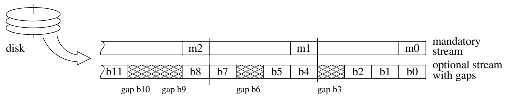

Figure 1. File system delivering a stream with one mandatory part and four optional parts per period. The quality parameter is 0.67

4 Scheduling and Admission Model

4.1 General Approach

The goal is to schedule active resources such that under overload a periodic task need not meet its deadline (the end of period) in each period but only in a user-specified fraction. More precisely, a periodic task $\mathcal{T}_i$ is a sequence of jobs J_{ij} where J_{ij} is to execute in the j^{th} period of $\mathcal{T}_i$. Each job consists of a mandatory part and one or more optional parts; the number of optional parts is fixed for each task. Each of these parts is considered as *successful* if it is completely executed. This condition must hold for all mandatory parts of a task, but it is sufficient that a given percentage (probability) of optional parts is successful. Since usually ([13, 12, 2]) an optional job is not completed only if it exceeds the end of period it may occur that an optional job which has a high priority but a low quality parameter (e.g., 40%) exceeds its requested quality immense (nearly 100%). Such jobs needlessly restrict the set of feasible tasks in overload situations. Therefore, a resource scheduler must enforce the reservation times for tasks as following: A constant amount of time is assigned to each task $\mathcal{T}_i$ in each period to execute the optional parts (analogously the mandatory part). When an optional part has consumed that amount it is aborted (if possible) and later optional parts of the same job are not started (similarly at the end of period).

Formally, for any $r \in \mathbb{R}$ and any $i \in \mathbb{N}$, let $p_i(r)$ denote the probability that an optional part of $\mathcal{T}_i$ is successful. Hence, the reservation time r_i of task $\mathcal{T}_i$ is the smallest time r where $p_i(r)$ is at least the given percentage q_i of successful optional parts:

$$r_i \quad = \quad \min(r \in \mathbb{R} | p_i(r) \geq q_i) \;\; \forall i = 1, ..., n \qquad (1)$$

n: number of tasks

(the achieved percentage of successful parts may exceed the requested percentage because the random variables have only discrete values).

Thus, the general admission criterion is that all mandatory parts of task $\mathcal{T}_i$ must meet their deadline, and the system of equations in Formula (1) is solvable.

Next, we give a general formal description of a task. Then we describe how the reservation times of a task are calculated and how the priorities are derived. We start with a specific situation with regard to the SCSI experiments (see Section 5.2): all tasks have the same length of period, and the resource is not preemptible. Finally, we give an outlook to the general situation.

4.2 The Task Model

Each task $\mathcal{T}_i$ is a sequence of jobs J_{ij} to be processed periodically:

$$\mathcal{T}_i = (J_{ij})_{j=1,2,...} \;\; i = 1, ..., n \qquad (2)$$

where $n \in \mathbb{N}$ denotes the total number of tasks in the task set $\mathcal{T} = \{\mathcal{T}_1, ..., \mathcal{T}_n\}$.

Each job J_{ij} consists of a mandatory part M_{ij} and c_i optional parts $O_{ij1}, O_{ij2}, ..., O_{ijc_i}$. The $M_{ij}, O_{ij1}, O_{ij2},...$ parts are started in this order within each period, but the second optional part O_{ij2} is only started if the first optional part O_{ij1} was successful (and so on).

The execution time of the mandatory and the optional parts of task $\mathcal{T}_i$ varies. In addition, no mandatory part exceeds the worst case execution time w_i. In other words, the execution time of the mandatory parts M_{ij} and the optional parts O_{ijk} are non-negative random variables X_{ij}, Y_{ijk}, respectively ($k = 1, ..., c_i$). We assume that all random variables of all tasks $\mathcal{T}_i$ are pairwise independent. Furthermore, for each task $\mathcal{T}_i$ the random variables X_{ij} are assumed to be identically distributed as well as all the Y_{ijk}.

Finally, an application may specify a probability that an optional part of task $\mathcal{T}_i$ is successful. In summary, the following definition describes a task.

Definition. A task $\mathcal{T}_i$ is a tuple

$$\mathcal{T}_i = (X_i, Y_i, c_i, w_i, q_i, t_i), \;\; i \in \mathbb{N} \qquad (3)$$

where

X_i non-negative random variable; execution time of the mandatory part;

Y_i non-negative random variable; execution time of an optional part;

c_i positive integer; number of optional parts;

w_i real number less or equal t_i; worst case execution time of the mandatory part, i.e., $\mathbf{P}(X_i \leq w_i) = 1$;

q_i real number $0 \leq q_i \leq 1$; quality parameter, percentage of successful optional parts;

t_i positive real number; length of period.

Note that $X_i \equiv 0$ enables us to consider tasks consisting of optional parts only (similarly of the mandatory part only). For simplicity, we identify the parts with their random variables and consider each mandatory part M_{ij} as a realization of the random variable X_i and the O_{ijk} as a realization of Y_i.

4.3 Nonpreemptible Resources, Uniform Periods

Let us now assume a task set with uniform periods (i.e., $t_1 = t_2 = ... = t_n =: t$) and let us consider non-preemptible resources, that means optional parts which cannot be aborted during its execution (e.g., a single disk access).

4.3.1 Calculating Reservation Times

Each mandatory part of a task $\mathcal{T}_i$ precedes its optional parts and must meet its deadline even in worst case situations. Therefore, we will give (see Section 4.3.2) the mandatory parts X_i of a set of tasks $\mathcal{T}$ a priority higher than the priority of any optional part. Thus, all mandatory parts meet their deadline if and only if

$$\sum_{i=1}^{n} \frac{w_i}{t} \quad \leq \quad 1 \qquad (4)$$

and w_i is the reservation time of the mandatory part X_i of task $\mathcal{T}_i$. To derive the reservation times r_i of the optional parts of $\mathcal{T}_i$ it is sufficient to consider the first period $[0, t]$ under the assumption that all tasks are ready at instant zero; obviously, it holds $r_i \leq t$ for all i. Due to the priorities of X_i all the mandatory parts of $\mathcal{T}$ are scheduled before any optional part is scheduled. Hence, let X describe the sum of the execution times of all mandatory parts X_i.

Without loss of generality, we assume that $\mathcal{T}$ is ordered according to the priorities of the optional parts. All optional parts of task $\mathcal{T}_i$ have the same priority. So Y_1 has highest priority (but lower than the priority of any X_i) and so on. An optional part of $\mathcal{T}_i$ is started if the end of the reservation time or the end of the period is not yet reached and it will be successful regardless whether it exceeds these deadlines or not. Therefore, q_i is the percentage of optional parts of task $\mathcal{T}_i$ which can be started during a period of $\mathcal{T}_i$. Let denote

r_i reservation time of the optional parts of task $\mathcal{T}_i$;

A_i number of optional parts of task $\mathcal{T}_i$ which can be started during a period of $\mathcal{T}_i$; random variable, their values are non-negative integers; A_i depends on $r_1, ..., r_{i-1}$.

Then the specified admission criterion for Formula (1) is

$$\exists r_1, ..., r_n \forall i = 1, ..., n : r_i = \min(r | EA_i \geq q_i c_i), \quad (5)$$
$$EA_i = \sum_{k=1}^{c_i} \mathbf{P}(A_i \geq k).$$

For $i = 1$ (the highest prioritized task) it holds (see Figure 2):

$$\begin{aligned} \mathbf{P}(A_1(r) \geq k) &= \mathbf{P}(X + (k-1)Y_1 < t \wedge (k-1)Y_1 < r), \\ & \quad k = 2, ..., c_1 - 1; \\ \mathbf{P}(A_1(r) = c_1) &= \mathbf{P}(X + (c_1 - 1)Y_1 < t). \quad (6) \end{aligned}$$

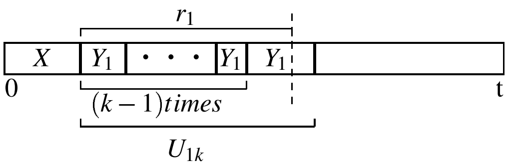

Figure 2. Computing the reservation time r_i and the total execution time of an optional job

To derive the reservation time r_2 we determine the random variable

U_1 total execution time of an optional job of task $\mathcal{T}_i$.

For that reason we define random variables $U_{11}, ..., U_{1,c_1}$ (see Figure 2); their values are at least r_1 (U_{1,c_1} has arbitrary values) and it holds for U_{1k}:

$$\begin{aligned} 1 \leq k < c_1 : &\quad X + (k-1)Y_1 < r_1 \wedge X + kY_1 \geq r_1 \\ k = c_1 : &\quad X + (c_1 - 1)Y_1 < r_1. \quad (7) \end{aligned}$$

Then

$$U_1 = \sum_{k=1}^{c_1} w_{1k} \cdot U_{1k},$$

$$w_{1j} = \mathbf{P}(A_1(r_1) \geq j) - \mathbf{P}(A_1(r_1) \geq j + 1). \quad (8)$$

Now we substitute X by $X + U_1$ and compute r_2 in the same way as r_1 and so on.

4.3.2 Deriving Priorities

As explained in Section 4.3.1, we give X_i an arbitrary but high priority. Next we assign priorities to the optional parts such that they are less than all priorities of all mandatory parts. Additionally, if an optional part has a lower quality parameter than another optional part also receives a lower priority. We call this priority assignment "Quality Monotonic Scheduling" (QMS) by analogy with the well known Rate Monotonic Scheduling (RMS).

4.4 Generalizations

So far we have considered nonpreemptible resources and uniform periods. We generalize that now.

4.4.1 Preemptible Resources

If the optional parts can be aborted during its execution then obviously it holds (see Figure 2)

$$U_1 \quad = \quad \max(X + (c_1 - 1)Y_1, r_1) \quad (9)$$

and so on.

4.4.2 Harmonic Periods and one Optional Part

Let us now consider tasks with one optional part only ($c_i = 1$ for all i) but with harmonic periods (any longer period must be a multiple of all shorter periods). Considering only the mandatory parts RMS is known to be optimal, and $\{X_1, ..., X_n\}$ is feasible if and only if Formula (4) holds with t_i instead of t. Unfortunately, including the optional jobs neither RMS nor QMS is optimal. Hence, two different ways (both based on QMS) to assign priorities are checked during admission. The task set $\mathcal{T}$ is not admitted if and only if the solution of Formula (5) fails in both cases.

4.4.3 Arbitrary Periods

The basic procedure is the same as described in Section 4.3.2. Priorities are assigned strictly according to QMS to all optional parts of any task. Now we use a simulator tool what was developed to verify the analytical results. The reservation times r_i belong to the inputs. The q_i-quantile s_{ik} of Y_{ik}

$$s_{ik} = \min(s \in \mathbb{R} | \mathbf{P}(Y_{ik} \leq s) \geq q_i \quad (10)$$

is a good approximation of r_i for high-prioritized optional parts because they seldom exceed the end of the period. Now we check the feasibility of $\mathcal{T}$ starting the simulation based on these values for all optional parts. If $\mathcal{T}$ is feasible it is admitted. Otherwise, let Y_{jl} ($1 \leq l \leq c_i$) be the highest prioritized optional part which did not reach its quality parameter q_j. Then we determine an approximated best value of r_j by binary search between s_{jl} and t_j (greatest possible value of r_j). Only if we can calculate the reservation times of all optional parts the task set $\mathcal{T}$ is admitted.

5 Implementation in DROPS

The section describes the implementation of Quality-Assuring Scheduling for two resources, namely CPU and SCSI. Both resource schedulers have been implemented in the Dresden Real-Time Operating System DROPS based on the scheme of cooperating resources managers [5].

5.1 CPU Scheduling

5.1.1 CPU Reservations

The underlying microkernel of the DROPS system uses a fixed priority scheduling with time slices and round-robin scheduling among tasks within the same priority class. On top of that it provides support for periodic threads and reservations. A user level scheduler can assign tuples *(priority, time slice)* to a thread. A thread owning such a reservation runs on the assigned priority until it either voluntarily releases the reservation or the reserved time slice expires. In both cases, the kernel assigns new scheduling parameters using the next reservation or the normal scheduling parameters of the thread. If a thread is periodic, the kernel automatically reassigns the reservations at the begin of each period. To allow threads to handle error situations, the kernel generates messages to the thread's associated scheduler if a reserved time slice has expired or a new period starts before all reservations have been consumed.

Based on these mechanism, DROPS provides a simple programming model for periodic real-time tasks. Figure 3 shows a code fragment. Real-time threads choose a period, make one or more reservations and enter an endless loop. `rts_begin_period()` releases all active reservations and blocks until the beginning of the new period. Then it starts its work, picks its next reservation using `rts_next_reservation()` and finally starts the loop again. If the thread consumes more than the reserved time or misses its deadline (the end of the period), the framework uses the message generated by the kernel to raise an exception.

The usage of reservation is not restricted to periodic tasks. It can also be used to guarantee a certain amount of CPU time at a high priority for known entities within the system like interrupt handlers. At the same time we specify an upper bound for the CPU consumption of these tasks.

5.1.2 Evaluation

To demonstrate the usability of our approach, we adapted an MPEG player to DROPS. It guarantees a minimum visual quality by decoding all I and P frames (i.e., at least each third frame is encoded). It tries to improve this basic quality by decoding as many B frames as the system load permits, but at least the requested amount.

Periodic Handler:

```
rts_set_period (period);
rts_reserve_time (mand_time, mand_priority);
rts_reserve_time (opt_time, opt_priority);
do {
    rts_begin_period ();
    try {
       do_something()
    } catch {
       exceeded:
          adjust_quality ();
    };
    rts_next_reservation ();
    try {
       do_something_else()
    } catch {
       exceeded:
           discard_result ();
    };
} while (!end);
rts_end_period ();
```

Figure 3. Code fragment for CPU reservation

To admit a new MPEG decoder, we need the period and distribution of decoding times for the stream to be played. Using the MPEG typical "group-of-picture", the length of the period follows from the (in general) fixed amount of I, P, and B frames. The player tries to reserve time for a mandatory job in which it decodes I and P frames and an optional job to decode the B frames. Here, decoding one B frame corresponds to one optional part. The distribution of decoding times depends on the hardware and the video and has to be measured once before playing the MPEG stream. The distribution of decoding time we used for our measurements is shown in Figure 4.

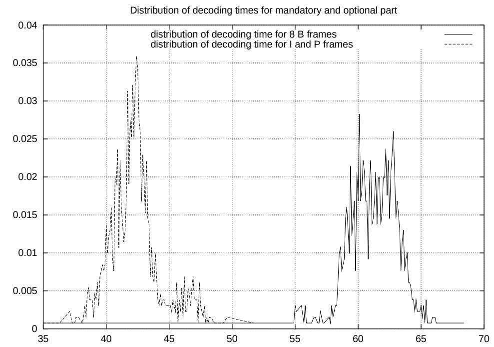

Figure 4. Measured distribution of total decoding times for I and P frames (left graph) and B frames (right graph) per group-of-pictures

Due to the complex decoding algorithms, the current version of our MPEG decoder is currently not able to cancel the decoding process at arbitrary points in time. Therefore, it cannot correctly handle incoming exceptions raised by the kernel after the decoder has exceeded its reserved time or has reached its deadline (the end of period). To overcome this, the decoder sets a flag when an exception arrives and checks this flag before starting to decode the next frame. Therefore, it can happen that the decoder continues the B frame from the previous period within the reservation for the mandatory part of the new period. To handle this correctly, we use the model described in Section 4.3 for non-preemptible resources and add the worst case decoding time of one B frame to the worst case execution time of the mandatory part. An improved version will be interruptible and able to handle all required exceptions at their time.

To validate our approach we admitted the decoder with different quality parameters and measured the actually achieved quality by counting the decoded and dropped frames. Table 1 illustrates the dependency between requested quality, model-derived reservation time and the achieved measured quality of the optional parts. The results are within five percent of the expected quality. This points to some hidden overhead within the implementation. But the results look promising especially since we did not remove sources of simple errors such as inaccurate timing yet.

The system was not in an overload situation during the measurements. To simulate a loaded system we restricted the MPEG decoder to its reserved time by assigning it no normal scheduling parameters. Therefore it blocked until the beginning of the next period after its last reservation was expired. By assigning the appropriate normal scheduling parameters other policies like time sharing with normal application or running at higher or lower priority than time sharing applications would be possible.

Requested Quality	Required Time for Optional Parts/ms	Reservation for Optional Parts/ms	Achieved Quality
0.95	54.22	55	0.9506
0.90	51.84	53	0.8588
0.80	45.51	47	0.7875
0.70	38.85	39	0.6740
0.60	31.61	32	0.5804
0.40	22.52	23	0.4063
0.20	8.04	9	0.2451

Table 1. Requested quality, derived reservation time, and measured quality of the optional parts

5.2 Disk Scheduling

For media applications like video and audio streaming it is important to predict the bandwidth capacity of the disk system. In contrast to the CPU scheduling where the non-deterministic behavior is primarily induced by the application and less by hardware, for disks the non-deterministic behavior is mainly caused by the hardware. The processing time of a disk request mainly depends on rotational and seek delays. Using worst case assumptions for reservation based on maximum values for rotational and seek latencies and the transfer times of a disk leads to very low disk utilization. In fact, our experiments showed that the worst case processing time of a Seagate Cheetah ST39102LW disk drive is about 2.5 times the average processing time under random workload. Figure 5 shows the distributions of the processing times for some of our disk drives.

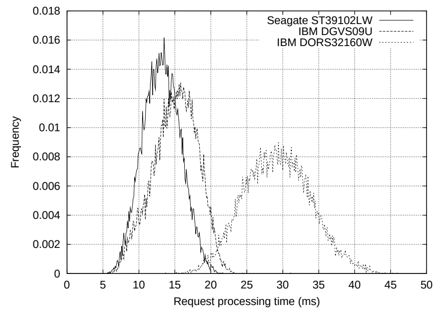

Figure 5. Distributions of disk request processing times (read requests, request size 64KB)

5.2.1 Resource Specification

Clients of the disk system (e.g., a real-time filesystem) request a certain bandwidth. This bandwidth requirement is translated into a number of disk requests per period[2]. Using the notation defined in Section 4.2, the client request can now be specified as a task with either a mandatory (if the requested quality is 1.0) or an optional job. The optional job consists of several optional parts, the number of parts is equal to the number of disk requests necessary to meet the requested bandwidth. As already mentioned, a disk request cannot be preempted once it is sent to the disk. Thus, the admission uses the algorithm described in Section 4.3.1 to calculate the reservation times.

5.2.2 Scheduling

The disk system defines the abstraction of a *stream*. Clients request a stream with a certain bandwidth and quality. Once the stream is established (i.e., the admission control successfully calculated the reservation time), the client must generate the disk requests in a timely manner. The requests must be available to the disk scheduler at the beginning of the period in which the request is to be processed. The deadline of the request is the end of that period.

The disk scheduler uses a fixed period, fixed priority scheme. At the beginning of a period the available process-

[2] The length of the period is defined by the disk system and must be chosen according to the available memory and latency requirements. In our experiments we used a period length of 500ms.

ing time for each stream client is set to its reservation time. Requests for a client are processed if

1. the client did not yet exceed its available processing time or its maximum number of requests per period;

2. no request of a client with a higher priority is available, and this client did neither exceed its available processing time nor its maximum number of requests in that period.

Requests without a reservation (*non-stream* or *best-effort* requests) are executed only if no stream request with the above-named properties is available and if sufficient time is left in the current period so that the request does not overlap with the next period.

5.2.3 Evaluation

We incorporated the disk scheduling algorithm into our SCSI device driver which runs on top of the Fiasco microkernel [6]. All experiments were done with a Seagate Cheetah ST39102LW (9.1GB, 10000rpm) disk drive. The disk was attached to a 200MHz PentiumPro PC via a NCR53c875 Ultra Wide SCSI adapter. Figure 6 shows the distribution of the "read-block" request processing times we used to calculate the reservation times. The mean value is 13.2ms with a standard deviation of 2.6ms, the maximum value is 33.2ms. The average bandwidth is 4852KB/s. Instead of a real client we used for all measurements a client which generates a randomly distributed workload to read disk blocks of 64KB size. For only few parallel active streams, the random load generates higher processing times than a filesystem does, since file blocks are often contiguously allocated. For multiple parallel active streams, the head has to "jump" between the streams which is closer to the generated used random load.

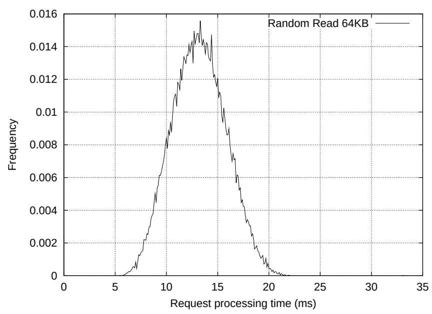

Figure 6. Distribution of the request processing time of a Seagate Cheetah ST39102LW disk drive

In our first experiment, we checked how accurate the disk system meets the predictions done by the admission. Table 2 shows an example configuration of this experiment. The client requests four different streams, each with bandwidth and quality requirements. The effective bandwidth is the bandwidth effectively to deliver to the client ($quality * requested\ bandwidth$). The last row shows the percentage of successfully processed disk requests. The results prove the practicability of the scheduling model for disk drives.

Stream	1	2	3	4
Requested bandwidth (KB/s)	2560	1280	640	1280
Number of disk requests	20	10	5	10
Requested quality	**0.950**	**0.900**	**0.850**	**0.500**
Effective bandwidth (KB/s)	2432	1152	544	640
Reservation time (ms)	244.7	112.4	50.0	61.1
Achieved quality	**0.947**	**0.898**	**0.850**	**0.508**

Table 2. Test configuration with four streams

The next obvious question is about the benefit of the approach. Table 3 shows the dependency of the requested quality of a stream and the maximum available bandwidth for our disk drive. With a quality requirement of 100% (which means that all requests of a stream must be delivered and thus the reservation must be done with worst case times), only 40% of the actual bandwidth is available to stream requests. But already with a quality of 99.99% (which means that 64KB out of 625MB of disk data get dropped), 92% of the average bandwidth is available to stream requests. With a quality of about 95% practically the full available bandwidth can be used for stream requests. Furthermore, the potential number of concurrent streams increases.

Quality	Max. number of disk requests	Bandwidth (KB/s) requested	Bandwidth (KB/s) effective
1.0000	15	1920.0	1920.0
0.9999	35	4480.0	4479.6
0.9993	36	4608.0	4604.8
0.9961	37	4736.0	4717.6
0.9864	38	4864.0	4797.8
0.9691	39	4992.0	4837.7
0.9473	40	5120.0	4850.2
0.9246	41	5248.0	4852.3

Table 3. Dependency of available bandwidth and quality

We used these streams for another experiment. In addition to the stream request the client generates a non-stream (best effort) workload. Table 4 shows the results. The achieved overall bandwidth is about 1% below the average bandwidth of the disk (4852 KB/s). We suspect this is due to the special handling of the period end of the disk scheduling algorithm.

Quality	Bandwidth (KB/s)		
	Stream	Non-Stream	Σ
1.0000	1920.0	2872.2	4792.2
0.9999	4477.5	287.9	4765.4
0.9993	4602.3	159.5	4761.8
0.9961	4709.1	69.7	4778.8
0.9864	4773.7	17.4	4791.1
0.9691	4800.7	4.7	4805.5
0.9473	4812.3	2.2	4814.5
0.9246	4804.3	2.8	4807.1

Table 4. Concurrent stream and non-stream workload

5.2.4 Concluding Remarks

Our experiments proved that the predictions of the scheduling model can be achieved with a real disk system. However, the randomly generated workloads we used do not well reflect the actual behavior of real-world clients like a filesystem. Further tests can be made using e.g., traces of real filesystem workloads. But as shortly mentioned, here we expect more improved results. The disk scheduler currently uses a FIFO scheduling scheme, requests are processed in the order of their position in the stream. Since the deadline for all requests of a period is the end of the period, the requests of a period can be reordered to reduce seek overhead (e.g., using SCAN algorithms).

6 Admission and Scheduling Overhead

It is worth emphasizing that the scheduling overhead is negligible small in comparison to other scheduling methods for flexible applications as known to us (especially in comparison to SRMS where at each ready time of a job an on-line admission is required).

The overhead caused by the CPU scheduler mainly consists of additional manipulations of the ready queue when activating a reservation and therefore increasing the priority, and when a reservation is released or expired and the scheduler must restore the original priority. This happens once per period for each reservation. Another source of overhead is the notification of the user level scheduler in case of a reservation overrun. This notification is sent using L4's fast IPC.

The disk scheduler maintains a list for each period containing all clients which enqueued requests in that period. Each client descriptor contains the requests and the remaining reservation time for that client. The list is sorted according to the priority of the clients. Thus, picking out the next client can be done with virtually no costs.

The admission control is more expensive, but it is not time-critical because it is done before starting the application. It must solve the Formula (5) to calculate the reservation times for the optional parts. We use discrete random variables derived from our measurements for the execution times for the mandatory and optional parts X_i resp. Y_i. The most costly part of the calculation is the convolution of random variables which is required to calculate the distribution function of the sum of two random variables. The complexity of one convolution is $o(\nu^2)$ where ν denotes the maximum number of values of the random variables.

We need $n-1$ convolutions to compute X and $n \cdot (c_1 - 1)$ convolutions for $(c_1 - 1)Y_1$ (see Formulae (6), (7)). The complexity to compute $\mathbf{P}(A_i(r) \geq k)$, U_{1k} and U_1 is $o(c_1\nu)$, $o(c_1\nu^2)$ and $o(c_1\nu)$, respectively. So the complexity of the entire admission algorithm is $o(n \cdot c \cdot \nu^2)$ where

$$c = \max_{i=1,\dots,n} (c_i). \tag{11}$$

Based on the example described in Section 5.2.3, we studied the influence of ν on the admission time t_{Adm}, on the reservation times r_i, and on the achieved quality $q_{i,ach}$. Each stream consists of optional parts only. The number c_i of optional parts of stream 1,...,4 is 20, 10, 5, and 10 per period, respectively. The length of period t is 500ms. The number of values of the distribution functions varies in the range of 500 through 50,000 (i.e., the class width varies from 1ms through 0.01ms). The dependency of t_{Adm} on ν is shown in Table 5 and in a double logarithmic coordinate system (see Figure 7). The ascent of the straight line is about 2, that means, the admission time t_{Adm} depending on ν is approximately a function of the type $y = ax^2$. Furthermore, Table 6 shows that the reservation time r_i is the smaller (and hence, $q_{i,ach}$ is the more precise) the larger the number of values. The achieved quality (determined by simulation based on $\nu = 50,000$) does not remain under the requested quality q_i in all situations.

ν	t_{Adm}/s
500	0.015
1,000	0.053
2,500	0.304
5,000	1.204
10,000	5.017
25,000	75.774
50,000	609.191

Table 5. Dependency of the admission time t_{Adm} on the number of values ν of the distribution functions

7 Conclusion and Future Work

We presented an approach for a unified admission and scheduling scheme to give probabilistic guarantees for periodic real-time use of active resources. Considering the known distribution of the variation of execution times allows us to admit far more applications than in systems based on worst-case admission.

Currently, we considered resources as independent and quality parameters must be derived for each resource separately. As future work, we also want to include resource

ν	r_1/ms	$q_{1,ach}$	r_2/ms	$q_{2,ach}$	r_3/ms	$q_{3,ach}$	r_4/ms	$q_{4,ach}$
500	246.00	0.954	114.00	0.914	51.00	0.872	62.00	0.507
1,000	245.00	0.951	113.00	0.907	50.00	0.857	60.50	0.507
2,500	245.00	0.951	112.40	0.903	49.80	0.854	61.40	0.513
5,000	244.70	0.950	112.50	0.903	50.00	0.857	61.10	0.511
10,000	244.65	0.950	112.10	0.901	49.65	0.852	60.30	0.507
25,000	244.64	0.950	112.08	0.900	49.54	0.850	59.70	0.503
50,000	244.64	0.950	112.09	0.901	49.56	0.851	59.80	0.504
q_i		0.950		0.900		0.850		0.500

Table 6. Dependency of the reservation times r_i and the achieved quality $q_{i,ach}$ on the number of values ν of the distribution functions

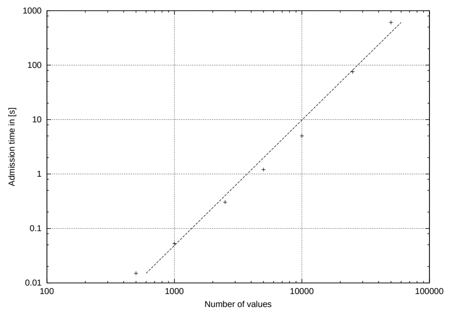

Figure 7. Regression test for admission times

dependencies such as the amount of CPU the disk scheduler requires. We also want to extend the system to derive all resource parameters from one application-specified quality parameter. Additionally, a precise scheduling model for harmonic and arbitrary periods should be found.

8 Acknowledgments

We appreciate the valuable feedback and comments from the reviewers to improve this paper. We would also like to thank DFG, Intel Research, and IBM for supporting this work.

References

[1] T. F. Abdelzaher, E. M. Atkins, and K. G. Shin. Qos negotiation in real-time systems and its application to automated flight control. In *Third IEEE Real-time Technology and Applications Symposium (RTAS)*, Montreal, Canada, June 1997.

[2] A. Atlas and A. Bestavros. Statistical Rate Monotonic Scheduling. Technical Report 98-010, Boston University, May 2, 1998.

[3] R. Baldwin, N. J. D. IV, J. E. Kobza, and S. F. Midkiff. Real-time queueing theory: A tutorial with an admisison control application. Technical report, 2000.

[4] E. K. P. Chong and Z. Wei. Performance evaluation of scheduling algorithms for imprecise computer systems. *The Journal of Systems and Software*, 15(3):261pp, July 1991.

[5] H. Härtig, L. Reuther, J. Wolter, M. Borriss, and T. Paul. Cooperating resource managers. In *Fifth IEEE Real-Time Technology and Applications Symposium (RTAS)*, Vancouver, Canada, June 1999.

[6] M. Hohmuth. The Fiasco kernel: System architecture. Technical report, TU Dresden, 2000. Unpublished manuscript.

[7] D. Hull, W. chun Feng, and J. W. S. Liu. Operating System Support for Imprecise Computation. In *AAAI Fall Symposium on Flexible Computation*, University of Illinois at Urbana-Champaign, 11 1996.

[8] E. D. Jensen, C. D. Locke, and H. Toduda. A time-driven scheduling model for real-time operating systems. In *Proceedings of the 6th IEEE Real-Time Systems Symposium*, 1985.

[9] C. Lee, J. Lehoczky, R. Rajkumar, and D. Siewiorek. On quality of service optimization with discrete qos options. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, 1999.

[10] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Third IEEE Real-time Technology and Applications Symposium (RTAS)*, pages 213–223, Montreal, Canada, June 1997.

[11] K. J. Lin, S. Natarajan, and J. W. S. Liu. Imprecise results: Utilizing partial computations in real-time systems. In *Proc. IEEE Real-Time System Symp.*, 1987.

[12] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[13] J. W. S. Liu, K. J. Lin, W. K. Shih, A. C. Yu, J. Y. Chung, and W. Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer*, 1991.

[14] S. Oikawa and R. Rajkumar. Linux/RK: A Portable Resource Kernel in Linux. In *19th IEEE Real-Time Systems Sumposium*, Madrid, Spain, Dec 2-4 1998.

[15] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource Kernels: A Resource-Centric Approach to Real-Time Systems. In *SPIE/ACM Conference on Multimedia Computing and Networking,*, Jan 1998.

[16] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for qos management. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, 1997.

[17] R. West and C. Poellabauer. Analysis of a window-constrained scheduler for real-time and best-effort packet streams. In *21st IEEE Real-Time Systems Symposium (RTSS)*, Orlando, Florida, USA, Nov. 2000.

[18] R. West, K. Schwan, and C. Poellabauer. Scalable scheduling support for loss and delay constrained media streams. In *Fifth IEEE Real-Time Technology and Applications Symposium (RTAS)*, Vancouver, Canada, June 1999.