A Case Against Periodic Jukebox Scheduling

Maria Eva Lijding, Ferdy Hanssen, Pierre G. Jansen
Distributed and Embedded Systems Group (DIES)
Fac. of Computer Science, University of Twente
P.O.Box 217, 7500AE Enschede, The Netherlands
lijding@cs.utwente.nl

Abstract

This paper presents the *jukebox early quantum scheduler (JEQS)*. JEQS is a periodic jukebox scheduler for a Video-on-Demand system. JEQS uses the jukebox robots in a cyclic way and the time is divided in constant units called *quanta*. A quantum is the maximum time needed to unload and load all the drives. An RSM is loaded in a drive for a fixed period of time, corresponding to the time needed to switch the media on the other drives. During this time the drive can read data from it. JEQS is based on the scheduling theory on *early quantum tasks (EQT)*. An early quantum task executes its first instance in the next quantum after its arrival and the rest of the instances are scheduled in a normal periodic way with the release time immediately after the first execution.

Although JEQS is an efficient periodic scheduler, that can guarantee the execution of most tasks in the next cycle after the requests arrive, we show that using JEQS results in much longer response times than using aperiodic schedulers. Furthermore, we show that the bad performance of JEQS is intrinsic to any periodic jukebox scheduler. The only advantage of using a periodic scheduler is that the scheduling algorithms are less complex. However, the simplicity of the algorithms clearly does not outweigh the unacceptably long response times.

1 Introduction

This paper presents the *jukebox early quantum scheduler (JEQS)*, a jukebox scheduler for a Video-on-Demand (VoD) system. JEQS schedules the jukebox resources in a periodic way using early quantum tasks. It guarantees that the requested data is promoted from tertiary-storage to secondary-storage in time.

A jukebox is a large tertiary storage device whose *removable storage media (RSM)*—e.g. CD-ROM, DVD-ROM, magneto-optical disk, tape—are loaded and unloaded from one or more drives by one or more robots. A jukebox can store large amounts of data in a cost-effective way, which makes it eminently suitable for applications that handle large amounts of continuous-media files, large databases

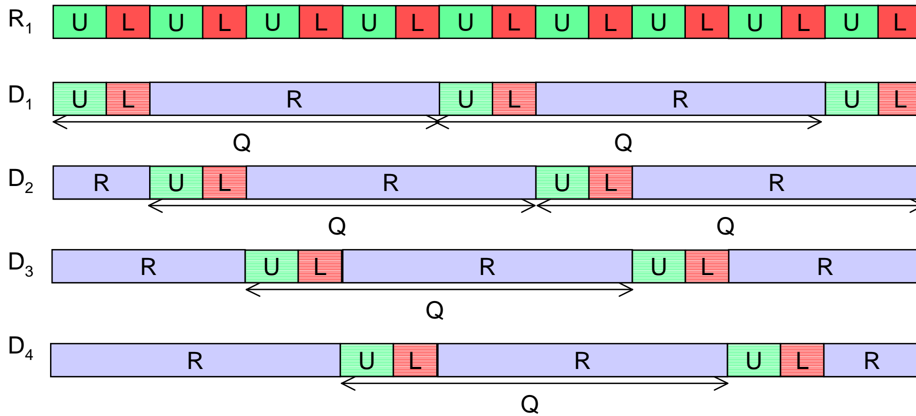


Figure 1: Use of the resources in a cyclic way.

and backups. In this paper we focus on the use of jukeboxes to provide storage for a Video-on-Demand system.

In order to use a jukebox effectively it is important to schedule the jukebox resources. On the one hand, a jukebox is not a random-access device: the RSM switching times are in the order of seconds or tens of seconds, which implies that multiplexing between two files stored in different RSM is many orders of magnitude slower than doing the same in secondary storage. On the other hand, the resources in the jukebox—robots, drives and RSM—are shared and require exclusive use, which creates the potential for resource-contention problems.

JEQS uses the jukebox robots in a cyclic way and the time is divided in constant units called *quanta*.[1] A quantum Q is the maximum time needed to unload and load all the drives. An RSM is loaded in a drive for a fixed period of time, corresponding to the time needed to switch the media on the other drives. During this time the drive can read data from it. Figure 1 shows the cycle for a jukebox with four drives ($D_1, \ldots, D_4$) and one robot (R_1).

A request is treated as a periodic task. The period of the task must guarantee that enough data is available in the buffer for the user to consume the data at the bandwidth specified in the request. The processing time of the task is always Q. The period of the task is obtained from computing how often the buffers need to be filled so that the user does not run out of data. The period depends on the bandwidth required by the request and the bandwidth offered by the drive.

JEQS uses the scheduling theory on *early quantum tasks (EQT)* presented in [13]. An early quantum task is a task whose first instance is executed in the next quantum after its arrival and the rest of the instances are scheduled in a normal periodic way with the release time immediately after the first execution. The role of the early quantum tasks is to serve incoming tasks as early as possible when

[1]In the rest of this paper we assume there is only one robot. In the case of multiple robots we use a separate cycle for each robot and define which drives and RSM will be served by it.

these tasks have a clear initialization phase, as pre-filling a buffer. An application of EQTs is to guarantee the in-time filling of buffers in a continuous-media file-system. In [13] we present such a system and discuss buffer administration issues.

JEQS builds the schedules assuming the existence of buffers that need to be filled in time for the user to access the data. However, if a drive is not 100% utilized, the schedule for the drive will have idle times. The dispatcher uses these idle times to fill the buffers of the active tasks earlier than scheduled. Thus, the data of the requests can be read in fewer instances than initially computed. This allows the tasks to leave the scheduler earlier, thus increasing the bandwidth available on the drive schedules for future requests. The dispatcher reads the data as soon as possible, whenever there is enough bandwidth available in the jukebox.

The goal of developing JEQS was to be able to compare Promote-IT [16], our aperiodic jukebox scheduler, with a periodic scheduler. To our best knowledge, JEQS is the only correct periodic jukebox scheduler. The other periodic jukebox schedulers proposed in the literature do not deal with the resource-contention problem correctly (see Section 2). Thus, they cannot guarantee that the deadlines are always met. Although JEQS performs worse than Promote-IT, because it does not use the jukebox resources efficiently, it has the advantage that it uses a very simple scheduling algorithm. In Section 8 we show that the comparatively bad performance of JEQS is intrinsic to any periodic jukebox scheduler.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 gives an overview of the system. Section 4 discusses some issues about the jukebox hardware. Section 5 provides a formal description of the scheduling problem. Sections 6 and 7 explain how the schedules are built and how the tasks are dispatched, respectively. Section 8 compares the performance of JEQS, Promote-IT and a First-Come-First-Serve scheduler. Finally, Section 9 concludes the paper.

2 Related Work

In the framework of our research on jukebox scheduling algorithms, we also developed *Promote-IT* [16], an aperiodic scheduler for a multimedia hierarchical archive. Promote-IT can serve complex requests for the real-time delivery of any combination of media files it stores. A request can consist of multiple streams and non-streamed data that are synchronized sequentially or concurrently in arbitrary patterns. In Promote-IT the scheduling problem is modeled as a *flexible flow shop* with three stages, one for loading, one for reading and one for unloading. Promote-IT builds near-optimal schedules using a heuristic algorithm, because finding an optimal solution is an NP-hard problem. Promote-IT uses the jukebox resources in a more efficient way than JEQS and, thus, provides shorter response times. However, the scheduling algorithm of Promote-IT is far more complicated than the one used by JEQS.

Lau et al. [14] present an aperiodic scheduler for VoD systems, which can use two scheduling strategies: *aggressive* and *conservative*. When using the aggressive strategy each job is scheduled and dispatched as early as possible, while when using the conservative strategy each job is scheduled and dispatched as late as possible. In [16] we present a comparison between Promote-IT and Lau's strategies and show that Promote-IT performs considerably better. As JEQS, Lau's strategies couple the unload and load of an RSM. Performing the unload and the load together as one switch operation simplifies the scheduling problem, but considerably affects the performance, because the drives stay loaded even when they are idle.

Chan et al. [5] stage a movie completely in secondary storage before it is displayed to the user, because their goal is to provide interactive VoD services. The movies are staged First-Come-First-Serve (FCFS), which in general provides bad response times. However, as we show in Section 8 the response times of FCFS, are in many cases better than that of a periodic scheduler. Chervenak et al. [7] also propose to stage a movie completely in secondary storage or to stream the movie directly from the jukebox drives to the user [6]. This last approach makes even worse use of the jukebox resources, because the bandwidth offered by the drives is in general much higher than the bandwidth requested by the users.

We now discuss briefly other approaches that, although they are interesting, do not deal with the RSM contention problem, which means that they cannot guarantee that an RSM is not assigned to two different drives during the same time period. Therefore, these schedulers cannot be used for jukeboxes with multiple drives and are not suitable to be used with most commercial big jukeboxes, which have multiple drives. Lau et al. [15] propose two algorithms, the *round-robin* and the *least-slack* algorithm, which break up the requests into time-slices and try to build a schedule with the time-slices of the different requests. Golubchik et al. [8] propose a periodic scheduler called *Rounds*. Cha et al. [4] use a jukebox scheduler based on a periodic EDF scheduler, which additionally does not deal with the robot-contention problem.

At first glance the least-slack algorithm looks similar to the scheduler presented in this paper. However, the least-slack algorithm does not use periodic scheduling theory to give guarantees about the schedulability of the request. Instead, it transforms the problem into an aperiodic scheduling problem where instances of the same request have to be scheduled separately. The scheduler does not determine a priori how the shared robot is used. It solves the robot-contention problem by incorporating into the system utilization the worst-case robot-contention scenario, which is having to wait for the robot to finish moves on all the other drives. This may result in poor utilization of the jukebox resources. The biggest disadvantage of the least-slack algorithm is that it does not deal with RSM contention, thus, it may occur that the two tasks that need to read data from the same RSM are assigned to different drives simultaneously. Executing such a schedule will either lead to missed deadlines or to a system crash.

Prabhakar et al. [19] and Triantafillou et al. [20] schedule requests without real-time deadlines in order to minimize the mean response time. The conclusion

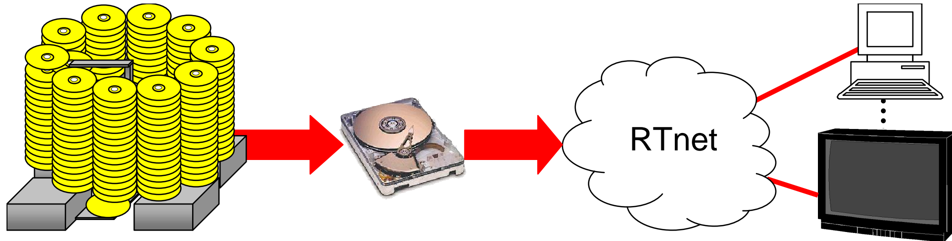


Figure 2: Flow of data from the hardware to the end users.

of their work is that as much data as possible should be read from an RSM when the RSM is loaded in the drive. Hillyer et al. [10, 11, 12] compare different scheduling algorithms for retrieving data from a magnetic tape without real-time requirements.

3 System Overview

The goal of our Video-on-Demand system is to provide fast and timely delivery of video and audio to users distributed in a local area network. The user can request any file or part of a file stored in the jukebox to be consumed with any desired bandwidth, as long as the bandwidth is less than the bandwidth offered by a drive in the jukebox. The data is not necessarily restricted only to video, it can also be music.

Figure 2 shows the flow of data from the jukebox to the end users. We must guarantee that the data is promoted from one storage level to the next in time. JEQS guarantees that the data is promoted from tertiary storage to secondary storage in time. In turn Clockwise [3] provides real-time access to data stored in secondary storage, which is used as cache, and finally *RTnet* [9] provides real-time guarantees for the use of a local area network. The VoD system is a special case of the *hierarchical multimedia archive (HMA)* presented in [16]. The VoD system can handle only requests with one request unit.

Once the system accepts and confirms a request, it is committed to provide the service requested by the user. The confirmation includes the *starting time* assigned to the request. The user can start consuming the data at the starting time, with the system's guarantee that the flow of data will not be interrupted. The request and the confirmation are the contract between the user and the system.

We define as *response time* the time between the arrival of the request and the starting time assigned to the request. The *confirmation time* is the time between the arrival of the request and the time at which the user gets a confirmation. We denote a request as u_k, the starting time of the request as st_k, and the response time as rt_i.

The main goal of the jukebox scheduler is to guarantee that the data is buffered in secondary storage by the time applications need it and guarantee uninterrupted access to the data. Beyond this, the scheduler tries to minimize the number of

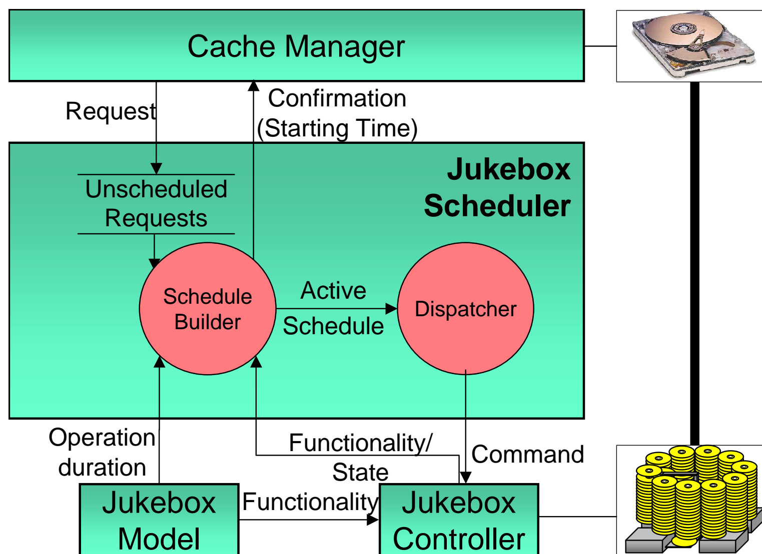


Figure 3: Architecture of the jukebox scheduler.

rejected requests, minimize the response time, maximize the number of simultaneous users, and minimize the confirmation time. The scheduling problem to solve is $\mathcal{NP}$-hard. Therefore, it is not possible to find an optimal solution on-line.

Figure 3 shows the architecture of the jukebox scheduler. The cache manager filters out the requests for data that is already in the cache or scheduled for staging. The schedule builder schedules the requests on-line, re-computing the schedule every time a request arrives. It generates a new schedule to replace the currently *active schedule*. The dispatcher uses the active schedule to send commands to the jukebox controller, which move RSM and stage data into secondary storage.

The schedule builder guarantees that including the new request does not lead to missed deadlines and the dispatcher guarantees that the commands are sent to the controller in time. The dispatcher may modify the schedules as long as no task in the schedule is delayed and the sequence and resource constraints are respected.

The schedule builder of JEQS uses a variant of EDF (earliest deadline first) [18], which we call *Quantum Earliest Deadline First (QEDF)*, to determine if a new task can be accepted. The feasibility analysis is simple and fast. In turn the dispatcher guarantees that all the tasks meet the deadlines, by dispatching the instances of the tasks in an *earliest deadline first (EDF)* way. When the cycle of a drive begins, the dispatcher determines all the tasks which are ready to execute on the drive. It chooses from those the task with the smallest relative deadline, following the EDF principle, and executes it.

The dispatcher of JEQS uses the idle times in the schedule to fill the cache of the active tasks. Thus, the data of the requests can be read in fewer instances than initially computed. This allows the tasks to leave the scheduler earlier, so

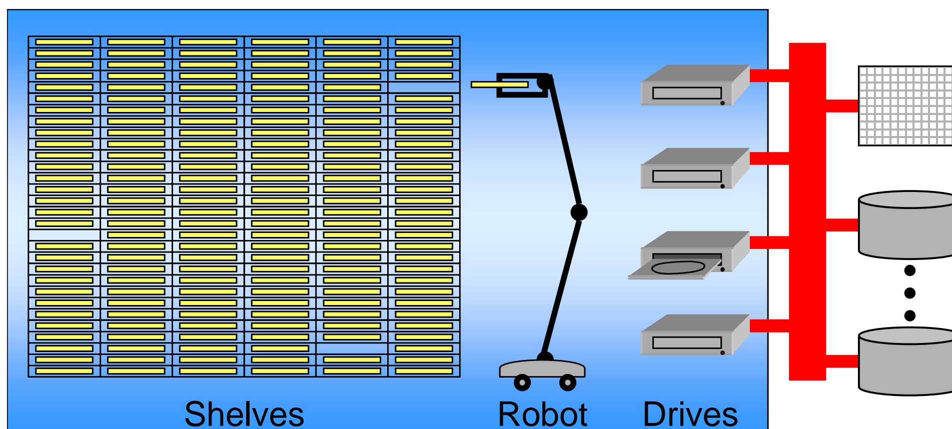


Figure 4: Jukebox architecture.

increasing the bandwidth available on the drive schedules for future requests. JEQS reads the data as soon as possible, whenever there is enough bandwidth available in the jukebox.

Caching is a key mechanism to improve the performance of a VoD system. On the one hand we need to keep in the cache a video during the duration of a task. On the other hand we need to store the popular videos in secondary storage, so that they do not need to be promoted from tertiary storage every time they are requested. The chances that a small set of videos is requested frequently are high, because the access pattern in VoD systems is in general very skewed, following a Zipf-like distribution [6]. Therefore, if the popularity of a video is high, the system will frequently receive requests for it. Thus, we achieve the goal of keeping the popular videos in cache by using a simple least recently used (LRU) cache administration policy.

4 Hardware Model

Tertiary-storage jukeboxes are composed of the following hardware: *m drives* to access the data in the RSM, *s shelves* where the RSM are kept and *r robots* to move the RSM from the shelves to the drives and vice versa. JEQS requires that all drives and RSM are identical. We assume optical or magneto-optical jukeboxes, because they provide lower and more predictable switching times that tape jukeboxes.

In big jukeboxes the number of shelves is at least two orders of magnitude larger than the number of drives and the number of robots. Our jukebox, for example, has 720 shelves, 4 drives and 1 robot. Jukeboxes are available for different types of RSM, for example CD-ROM, DVD-ROM, magnetic tape or magneto-optical disk. Figure 4 shows the architecture of a generic jukebox with four drives and one robot. The data from the drives can be transferred directly to secondary storage through a high-bandwidth connection.

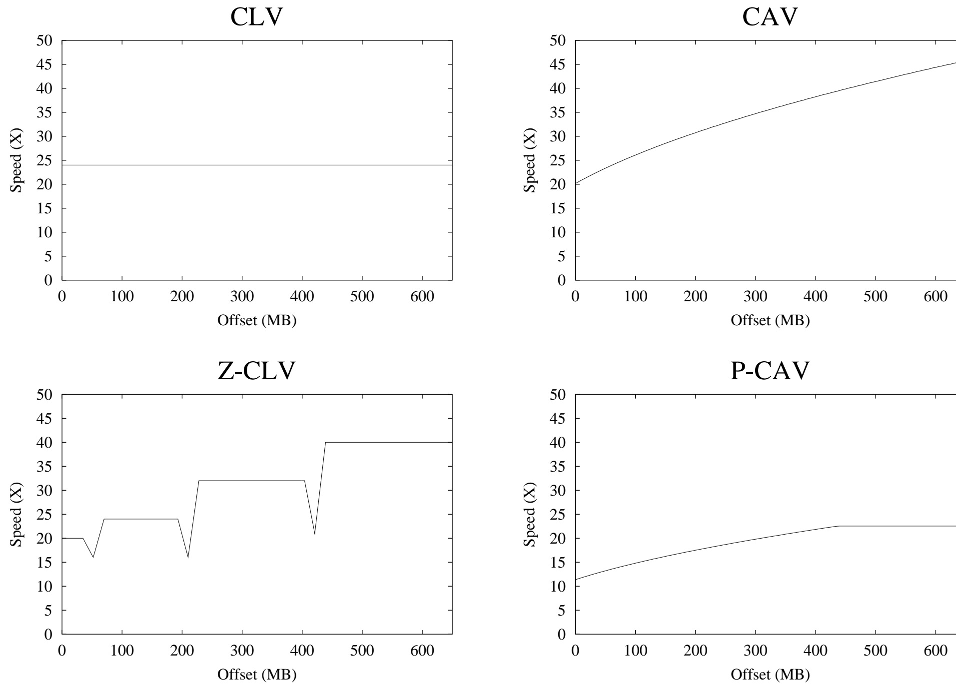


Figure 5: Different optical drive technology. The graphics show the transfer speed as a function of the track read.

We use a model of the hardware to predict the time that the system will need for operations on robots and drives. We have validated the model against our actual hardware and use it both for constructing the schedules and as a simulator in our experiments. In [17] we present the complete hardware model. Here we present some the characteristics of jukebox technology that help the better understanding of JEQS.

The time it takes to load and unload a drive depends on different factors: opening and closing time of the drive, spin-up/-down time, and the distance between the drive and the shelf where the RSM is kept.

To compute the time to read data we need to take into account two components: the transfer time and the access time. Except for drives using CLV technology, the transfer time varies considerably when reading data starting at different offsets. Figure 5 shows the transfer speed as a function of the offset for different optical technologies. The hardware model provides the function $s^{min}_{transfer}(t)$ that computes the worst-case amount of data that can be transferred in time t. The worst case is determined by reading data from the inner tracks starting at track 0. The bound provided by this function will be very far from the real value in CAV drives. The access time also depends the offset of the data to read, but the difference is not as big as with the transfer time. So the upper bound provided by the function t^{max}_{access} is not very far from the real time.

5 Scheduling Problem

This section presents the scheduling problem using a periodic quantum model. The robot is used in a cyclic way. The robot first unloads drive 1 and loads it with another RSM. It then does the same for drive 2, and so on until all drives have been served. It then starts again its cycle on drive 1. The jukebox has m identical drives.

We define a *quantum* Q as the time needed to complete a cycle:

$$Q = m\,(t_{load}^{max} + t_{unload}^{max}) \tag{1}$$

Using quantum tasks [2] has the advantage that although the tasks are non-preemptable, they can be treated as preemptable during the feasibility analysis. The only condition is that release times of the tasks scheduled on drive i always coincide with the beginning of a cycle for drive i. We can guarantee that the release times of the tasks always fall at the beginning of a cycle, if the release time of the first instance is at the beginning of a cycle, because the period of the tasks are multiples of Q. Therefore, a task which is executing in a drive never needs to be preempted.

An RSM is loaded in a drive for a fixed period of time. During this time the drive can read data from it. The model assumes that all drives are identical, so the time for reading data is the proportion of the quantum needed to switch the media on the other drives ($\frac{m-1}{m}Q$). During the time assigned to read data, the drive must first access the data and then transfer it. Given that the drive cannot predict the offset of the data to read, it uses the worst-case access time. Therefore, the remaining time for transferring data TT is:

$$TT = \frac{m-1}{m}Q - t_{access}^{max} \tag{2}$$

Any periodic model either needs to impose restrictions on the way the robot is used or has to take into account the worst-case robot-contention time in the execution time of the tasks. The first approach is the one used by Golubchik et al. [8] in their algorithm Rounds and in the periodic quantum model we present here. The second approach is used by Lau et al. [15] in the time-slice algorithm. The worst-case robot-contention time is $\frac{m-1}{m}Q$, which is the time needed to perform a switch in all the other drives. Furthermore, a periodic model needs to couple the unload and the load. In the case of a cyclic robot utilization the coupling is natural to the cycle. In the case of the worst-case robot-contention time, if the load and unload are not coupled, then the robot-contention time has to be taken into account twice, once before the load and once before the unload, which should result in adding another $\frac{m-1}{m}Q$ to the execution time.

A shortcoming of this model (and any other periodic model) is that it needs to reserve the worst-case execution time of the operations. On the one hand it uses the maximum load and unload time to compute the quantum, because it must guarantee that all combinations of drives and shelves are schedulable. If the switch finishes earlier, the robot waits until the time of the worst-case to start unloading the next

drive. On the other hand it uses the minimum amount of data which can be read during a quantum, even if the amount of data that can be read in a quantum varies. In different instances of the task the data is read starting at a different offset. We define B to be the minimum amount of data that can be transferred in time TT. We compute B using a function $s^{min}_{transfer}$ that computes the amount of data that can be transferred in time TT in the worst-case scenario.

$$B = s^{min}_{transfer}(TT) \quad (3)$$

A request u_k has the following parameters: m_k, o_k, s_k and b_k that indicate the RSM where the data is stored, the offset in the RSM, the size of the data and the bandwidth, respectively. We denote the starting time assigned to the request as st_k

A request is treated as a periodic task τ_i. The period of the task must guarantee that enough data is available in the buffer for the user to consume the data at the bandwidth specified in the request. The processing time of the task C_i is always Q. The period of the task T_i is obtained from computing how often the buffers need to be filled so that the user does not run out of data. The period depends on the bandwidth required by the request and the bandwidth offered by the drive. Without loss of generality we can assume that the data is consumed with a constant bit rate, because the buffer size is large. Anastasiadis et al. [1] and Bosch [3] show that a variable bit-rate stream can be treated as a constant bit-rate stream when the buffer size is large enough.

This model creates as output a set $\Gamma = \{\tau_1, \ldots, \tau_n\}$ of periodic tasks to schedule. Each task τ_i needs to be executed only a finite number of times. We compute the number of instances required by a request as $\lceil t_{transfer}(o_i, s_i)/TT \rceil$. As the number of instances of each task is finite, there is no real need to use a periodic scheduler to build schedules for this model. An aperiodic schedule can also be used. We use a periodic scheduler, because it allows to use simple computations to decide if a task set is feasible.

The parameters of a task to schedule τ_i are the following:

Execution time (C_i) The execution time is always Q.

Period (T_i) The period of the task is always a multiple of Q. We compute the period of the task as:

$$T_i = \lfloor \frac{B}{b_i Q} \rfloor Q$$

Shared resources (ρ_i) Indicates the RSM on which the data of the request is stored. The RSM is used as a shared resource, so that tasks using the same RSM are not assigned to different drives simultaneously. If the task is being executed in a drive it also contains the drive D_j in which it is being executed, to guarantee that it is not assigned to another drive.

Next release time (r_i) Indicates the release time of the next instance of the task. If $r_i < t_0$, where t_0 is the present time, then the last instance of the task

has not yet been executed. Otherwise, the last instance has already been executed and the release time corresponds to the next instance of the task. These parameters are obtained from the active schedule, which maintains a history of the executed tasks.

Remaining instances (RI_i) Indicates the number of remaining instances for the task. For the correct functioning of the scheduler, once the last instance was executed for the last time, the task remains in Γ until the deadline of the last instance. Only then can the bandwidth utilized by τ_i be utilized by another task.

6 Schedule Builder

The schedule builder of JEQS uses a simple and fast feasibility analysis. The scheduler builds a separate schedule for each drive. When a new request u_k arrives the scheduler tries to include it in the schedule of one of the drives. The scheduler first tries to include the task as an *early quantum task (EQT)* in the schedule of one of the drives. An early quantum task is a task whose first instance is executed in the next cycle of a drive and the rest of the instances are scheduled in a normal periodic way with the release time immediately after the first execution. If the request can be incorporated as an EQT on drive i, then the first buffer will be filled at the end of the next cycle of drive i. At that moment the user may start consuming the data. If the task is scheduled on drive i without using an EQT, then the starting time is at the end of the first instance of the task. Thus, the starting time of the request st_k is the starting time of the next cycle of drive i, plus the period of the task T_k.

The scheduler guarantees that an RSM never needs to be loaded in different drives, by using the same drive to process all the tasks involving the same RSM. When a task is scheduled for the first time it is assigned a drive. All the instances of the task will be executed on the same drive. The scheduler distinguishes a new task from the others, because ρ_k will not indicate the drive to which the task is assigned.

As we have explained in the previous section, we can treat the tasks as preemptable. Therefore, we can use a simple variation of EDF, called QEDF, to determine if the task set is schedulable. EDF has the advantage that it is optimal and very simple to compute. These properties also hold for QEDF, because QEDF is a special case of EDF where all tasks have the same execution time $C_i = Q$. Furthermore, using quantum tasks has the advantage that under certain circumstances we can fill the first buffer of a stream and allow an incoming task to start executing as soon as possible.

6.1 Scheduling Algorithm

The algorithm assumes that the tasks are normal periodic tasks that run indefinitely, as is the case in classical real-time scheduling theory. It does not take into account that the number of instances of each task is limited. Instead, it waits until a task

finishes its execution and the last instance reaches the deadline, to remove the task from the task set and consider the bandwidth used by the task available.

Using this problem simplification has the advantage that determining if a request is schedulable is extremely simple. However, the scheduler is unable to efficiently schedule requests with starting times far into the future. The scheduler is only able to decide about the schedulability of a request at the starting time of the next cycle of each drive. Another problem is that the scheduler cannot schedule requests which need the bandwidth used at present by another task, even if that task will finish its execution soon. This last restriction prevents in many cases to provide an immediate confirmation to the user about the schedulability of the request.

The scheduler will try at most $2m$ possible starting times to schedule an incoming request. It first tries to incorporate the task to start in the next cycle of a drive, using an early quantum task. If this fails, the scheduler attempts to schedule it as a normal quantum task. The scheduler tries the drives in the order in which they will start the next cycle. The cycle of a drive begins with the unloading of the RSM loaded in the drive, the loading of the new RSM and finally the reading of the data. Let us assume that the next drive to start a cycle is drive 1 and that the cycle of drive 1 will start in time t_{next}. The starting times that the scheduler will attempt are:

$$\forall\, 1 \leq i \leq m \mid \{t_{next} + (i-1)\,(t_{load}^{max} + t_{unload}^{max}) + Q \wedge \\ t_{next} + (i-1)\,(t_{load}^{max} + t_{unload}^{max}) + T_k\}$$

When transforming the requests into tasks, we assign to τ_k the release time $r_k = st_k - T_k$. Thus, when using an EQT it seems that the first instance of τ_k is waiting for execution. This allows to represent that the first instance of the task has an 'almost immediate' deadline and the next instances behave normally.

Let us illustrate with an example how an incoming request u_k is transformed into the periodic task τ_k and the value of the candidate starting times for u_k. The request u_k represents a two hour long video with a bandwidth of $6Mbps$.[2]

Table 1 shows the relevant characteristics of the jukebox and the parameters of the scheduling problem. We compute the period of the task τ_k as:

$$T_k = \lfloor \frac{B}{b_k\, Q} \rfloor\, Q = \lfloor \frac{808.2MB}{0.75MBps\; 180s} \rfloor\, 180s = 5 \cdot 180s = 900s$$

Figure 6 shows the candidate starting times for u_k. The candidate starting times, shown as $st_k^1, \ldots, st_k^8$, are tried in that order. Figure 7 shows the task τ_k when using st_k^1 and st_k^5. In both cases the drive to use is D_1. The resulting task τ_k^1 is an early quantum task and as such it requires that the drive executes the first instance of the task $\tau_{k,1}$ in the first cycle to meet the deadline of the first instance. When using st_k^5 the drive can execute $\tau_{k,1}$ in any of the first five cycles.

For each candidate starting time st_k^j the scheduler executes the feasibility analysis for the set $\Gamma_i \cup \tau_k^j$, where τ_k^j is the task built by using st_k^j and Γ_i is task set of

[2]We express the bandwidth requested by the users in Megabits per second ($Mbps$) and the bandwidth offered by the drives in Megabytes per second ($MBps$).

Type RSM	Double-layered DVD-ROM
Drive technology	CAV
m	4
$t_{load}^{max} + t_{unload}^{max}$	$45\,s$
Q	$180\,s$
Transfer Speed	$[5.11, 12.2]\,MBps$
t_{access}^{max}	$0.31\,s$
B	$808.2\,MB$

Table 1: Jukebox specification and scheduling problem parameters for example.

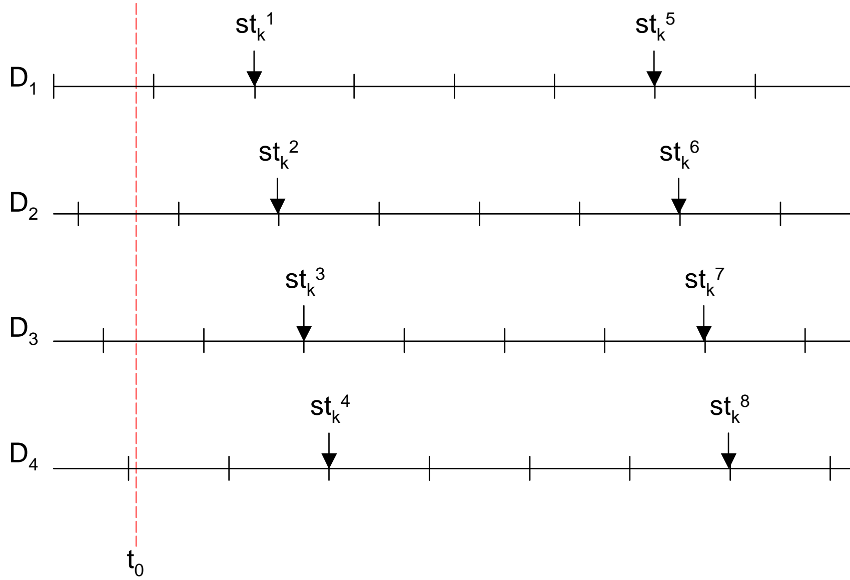


Figure 6: Candidate starting times for u_k.

the drive corresponding to st_k^j. If the feasibility analysis succeeds, then the τ_k^j is incorporated to Γ_i.

The first step of the feasibility analysis on the set Γ_i is to determine if there is enough bandwidth available on drive i to schedule all the tasks. We do this using the feasibility analysis for EDF as defined by Liu and Layland [18]. The task set Γ_i is schedulable if:

$$U = \sum_{j=0}^{n} \frac{C_j}{T_j} \leq 1 \tag{4}$$

If there is enough bandwidth to schedule all the tasks and the release time of the first request is before t_0, the scheduler checks if it can use an EQT. The condition to use an EQT is that enough time has passed since an EQT was used last. The scheduler, thus, keeps a record of when an EQT was last incorporated

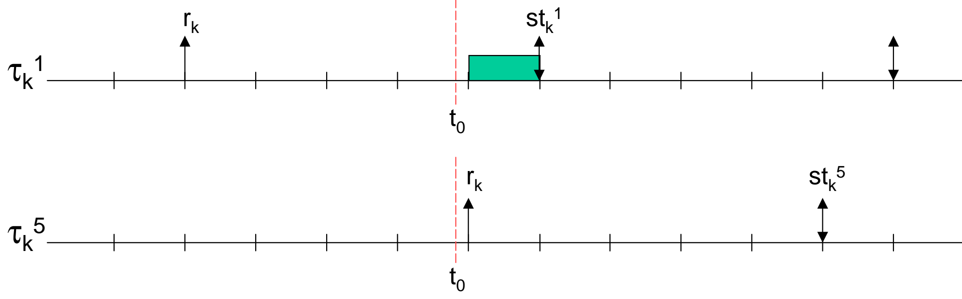


Figure 7: Tasks resulting of scheduling the request with the first and fifth candidate starting time. Top: Resulting task τ_k^1 when using st_k^1. Bottom: Resulting task τ_k^5 when using st_k^5.

in each drive. If the last instance has occurred at least $\lceil \frac{1}{1-U^i} \rceil$ time units earlier, then the new request can be accepted as an EQT. U^i is the utilization of drive i at the beginning of the next cycle of drive i, without considering the bandwidth requirements of u_k.

Condition 1 *Given a quantum task set Γ with utilization U, a new task τ_k, t_n the time at which the new cycle begins and t_l the last time a task was incorporated in Γ as an early quantum task, τ_k may be incorporated in Γ as an early quantum task if:*

$$T_k \geq \lceil \frac{1}{1-U} \rceil \ \wedge$$
$$t_n \geq t_l + \lceil \frac{1}{1-U} \rceil$$

The first condition is needed to guarantee that the task can be incorporated into the set as a normal quantum task. It derives from the fact that the set is scheduled with QEDF. The second condition guarantees that enough time has passed to accept the task τ_k as an early quantum task.

There is a trade-off between optimizing the response time and the confirmation time. The scheduler may accept a request as normal quantum task in order to provide a fast confirmation time at the cost of a worse response time, or delay the confirmation of the request until it can incorporate it to the schedule as an early quantum task. The response time of a request accepted as normal quantum task will never be better than waiting to schedule the request until it can be accepted as an early quantum task.

A request u_k can only be scheduled in drive D_i if the utilization needed by the request is smaller or equal to the remaining utilization available on the drive ($T_k \geq \lceil \frac{1}{1-U^i} \rceil$). Therefore, if the request is accepted as a normal quantum task the

starting time of the request is at least ($st_k \geq t_0 + \lceil \frac{1}{1-U^i} \rceil$). As $t_0 \geq t_l$, the minimum possible starting time is waiting until it can be accepted as EQT. Additionally, the utilization of the drive may decrease if one of the active tasks leaves the system, in which case waiting to schedule the request may be even more profitable. Therefore, if increasing the confirmation time is not a problem, the scheduler may only try to schedule the requests as early quantum tasks. If the request cannot be incorporated as early quantum task it is placed in the queue of requests awaiting scheduling until a drive can accept the request as EQT.

However, when the load of the system increases and requests arrive with a shorter inter-arrival time than the time until an EQT task can be incorporated into the system, the performance of the scheduler that only incorporates tasks as EQT quickly degrades. The reason for this is that the queue of requests awaiting scheduling grows very fast, because whenever a request is scheduled the time to wait until the next request can be accepted as EQT is set further into the future. Thus, waiting to accept all requests as EQT is only profitable if the length of the queue is in average 1 and this is only possible when the system load is low. Incorporating some requests as normal quantum tasks alleviates the pressure on the scheduler. In a sense the requests that are 'unlucky' to arrive at a time when the scheduler cannot accept a request as EQT, pay the cost of an overall better scheduler performance by being incorporated as normal quantum tasks.

In Section 8 we compare the performance of both approaches and show the point at which using normal quantum tasks is more beneficial than using only EQTs.

7 Dispatcher

At the beginning of a cycle for drive i the dispatcher decides what must be executed during that cycle. The dispatcher uses the EDF rule to choose the task with the earliest deadline among the tasks with an instance awaiting execution. A task τ_j has an instance awaiting execution if the release time of the task is earlier than the present time ($r_j \leq t_0$). If there is no task with an instance awaiting execution, then according to the EDF rule, the next cycle of the drive would be idle, however the dispatcher uses the cycle to fill up the buffer. In this way, the dispatcher dispatches some instances early to be able to remove tasks from the scheduler as soon as possible and, so, make bandwidth available for new tasks.

The dispatcher uses the following rule to decide what buffer to fill. It first tries to go on reading data of the same task that was active in the drive, unless all the data of the task has been already buffered. If all the data corresponding to the task active in the drive has been read, but there is another task for the same RSM, it reads the data of this other task. If the drive can go on reading data from the same RSM, the RSM loaded in the drive does not need to be unloaded, and the next Q units of time can be fully utilized for reading data. If no data can be read from

Arrival (sec)	ID	Offset (MB)	Size (MB)	Bandwidth (Mbps)	Period (quanta)
24	1	2	5000	5	7
108	2	1500	3000	6	5
112	3	500	3272	9	3
137	4	2	8192	10	3
238	5	2000	657	1.8	19
300	6	4000	182	1.2	29
423	7	2	7200	5	7
460	8	100	1300	2.4	14
520	9	500	2500	3	11
546	10	750	3000	10	3
681	11	600	4000	3.8	9
865	12	10	8000	10	3
966	13	5000	100	0.125	287
1098	14	2	8000	15	2
1181	15	10	7000	15	2
1456	16	300	7200	10	3

Table 2: Example of requests which arrived at the system.

the RSM, it loads the RSM corresponding to another task. Only if no task has got pending instances, the drive is left idle.

The dispatcher does not alter the functioning of the QEDF schedule. It only uses idle cycles when it has decided that those cycles should go unused otherwise.

The dispatcher also deals efficiently with situations in which a task has got a period of 1. The RSM is kept constantly in the drive until all the data has been read and only then unloaded. In such a case, the improvement in the bandwidth utilization of the drive is considerable compared to the original schedule.

The dispatcher does not know in advance the exact number of instances that need to be used for each task, because when there are idle slots it can advance in the execution of a task. If all the data of a task has been read once the deadline of the last executed instance of the task is reached, no new instance is released and the task is removed from the schedule.

7.1 Example

We illustrate with an example how the scheduler works. We use the same jukebox specification shown in Table 1. Table 2 shows the requests that arrived at the system since time 0. The data of each request is stored on different RSM ($\forall\, i,j \mid m_i \neq m_j$). The table also shows the period of the corresponding task. We assume that before these requests arrived, the system was idle.

Table 3 shows the starting time st_i and response time rt_i of each task. The response time is the starting time minus the arrival time. The table also shows for each drive: the utilization (U^i), the starting time of the next cycle (t_n^i), and the earliest possible time to accept an early quantum task ($t_l^i + \lceil \frac{1}{1-U^i} \rceil$). The last two

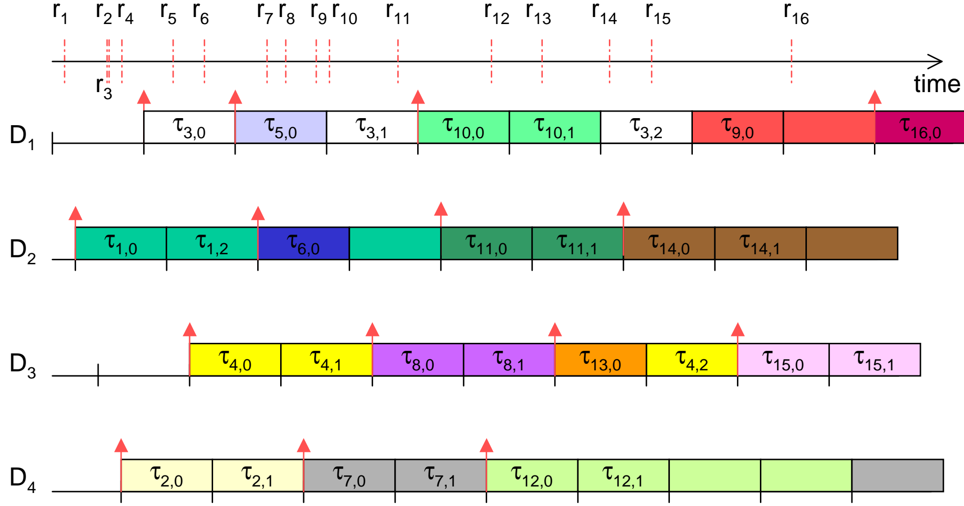


Figure 8: Schedule for the tasks of the example. The blocks without text are idle slots used for filling up the cache.

columns of the table show the maximum number of instances needed to read all the data of the request (RI_i) and the latest time by which all the data of the request will be cached (d_i^{last}). The exact time is the deadline of the last instance d_i^{last}, which is computed as $d_i^{last} = (RI_i - 1)\, T_i\, Q + st_i$. However, the dispatcher does not know in advance the exact number of instances that need to be used for each task, because when there are idle slots it can advance in the execution of a task. Therefore, the table provides an upper bound for both values.

At time 24 when r_1 arrives at the scheduler, the next drive to begin its cycle is D_2 so τ_1 is assigned to D_2 and it can start immediately. All tasks except τ_9 can be incorporated as early quantum tasks. Note that if τ_{16} would have arrived earlier, it would have been rejected, because there was not enough bandwidth in any drive until time 1440 when τ_3 left the system. Therefore, τ_{16} would not be schedulable without using the early dispatcher.

Figure 8 shows the first part of the schedule. The top line indicates the arrival time of the requests. The up arrows in the schedule indicate when an early quantum task begins. The blocks without tags represent work that has been dispatched early.

8 Performance Evaluation

This section presents a comparison between aperiodic scheduling and periodic scheduling. Aperiodic scheduling is represented by Promote-IT, while periodic scheduling is represented by JEQS. We analyze the performance of JEQS, both

Time	ID	D_1	D_2	D_3	D_4	st_i	rt_i	Drive	EQT?	RI_i	d_i^{last}
24	1	0 180 180	0 45 45	0 90 90	0 135 135	225	201	D_2	YES	≤ 5	≤ 5265
108	2	0 180 180	0.14 225 405	0 270 90	0 135 135	315	207	D_4	YES	≤ 3	≤ 2115
112	3	0 180 180	0.14 225 405	0 270 90	0.2 135 495	360	248	D_1	YES	≤ 3	≤ 1440
137	4	0.33 180 360	0.14 225 405	0 270 90	0.2 315 495	450	313	D_3	YES	≤ 7	≤ 3690
238	5	0.33 360 360	0.14 405 405	0.33 270 630	0.2 315 495	540	302	D_1	YES	1	540
300	6	0.38 360 720	0.14 405 405	0.33 450 630	0.2 315 495	585	285	D_2	YES	1	585
423	7	0.38 360 720	0.17 585 765	0.33 450 630	0.2 495 495	675	252	D_4	YES	≤ 6	≤ 6975
460	8	0.38 540 720	0.17 585 765	0.33 630 630	0.34 495 855	810	350	D_3	YES	2	3330
520	9	0.38 540 720	0.17 585 765	0.40 630 990	0.34 675 855	2520	2000	D_1	NO	≤ 3	≤ 6480
546	10	0.42 720 720	0.17 585 765	0.40 630 990	0.34 675 855	900	354	D_1	YES	≤ 3	≤ 1980
681	11	0.75 720 1620	0.14 765 765	0.40 810 990	0.34 855 855	945	264	D_2	YES	≤ 4	≤ 5805
865	12	0.75 900 1620	0.25 945 1125	0.40 990 990	0.34 855 855	1035	170	D_4	YES	≤ 7	≤ 4275
966	13	0.75 1080 1620	0.25 1125 1125	0.40 990 990	0.73 1035 1575	1170	204	D_3	YES	1	1170
1098	14	0.75 1260 1620	0.25 1125 1125	0.40 1170 1350	0.73 1215 1575	1305	207	D_2	YES	≤ 7	≤ 3465
1181	15	0.75 1260 1620	0.75 1305 2025	0.40 1350 1350	0.73 1215 1575	1350	169	D_3	YES	≤ 6	≤ 3150
1456	16	0.42 1620 1080	0.75 1485 2025	0.90 1530 3330	0.73 1575 1575	1800	344	D_1	YES	≤ 6	≤ 4500

Table 3: State of the scheduler at the arrival times and resulting starting time and response time when using the cached early quantum scheduler. The values shown for each drive are the utilization U^i, the starting time of the next cycle t_n^i and the earliest time at which an EQT can be accepted $t_l^i + \lceil \frac{1}{1-U^i} \rceil$.

using normal quantum tasks and scheduling only EQTs. We show also the performance of the FCFS scheduler proposed by Chan [5].

We show that using periodic scheduling is a bad technique for scheduling a jukebox, because even the FCFS scheduler which is extremely simple—load the RSM, read all the data, and unload the RSM—behaves better than JEQS in most cases. We argue that the bad performance of JEQS is not a characteristic of JEQS, but is intrinsic to any periodic jukebox scheduler. As discussed in Section 5, a periodic scheduler either needs to use the robot in a cyclic way, or take into account the worst-case robot-contention time in the execution time of the tasks. Therefore, when using a periodic scheduler the best-case starting time for a request that does not produce a cache-hit is Q, even if the system load is very low and all drives are idle. In the same scenario the starting time for Promote-IT is in most cases just the time to load the RSM in the drive and read the data of the first request unit. Therefore, in a situation with low system load, the best-case response time for a periodic scheduler is around $\frac{m-1}{m}Q$ worse than the general case for Promote-IT. The difference gets even worse for the periodic schedulers as the system load increases, because the periodic scheduler wastes drive bandwidth with unnecessary switches.

The request set consists of 1000 requests that follow a Zipf distribution. Each request is for one full video file. The bandwidth of the videos is uniformly distributed in the range $[1, 8]Mbps$. The duration of the videos is uniformly distributed in the range from 15 minutes to 2.5 hours. The data in the jukebox is stored in double-layered DVDs. Each video is stored completely in one disc. However, one disc may store multiple videos.

We show simulation results for two jukebox architectures. In both cases the jukebox is a smartDAX (for hardware model see [17]) with four identical DVD drives. The load time is in the range $[21.8, 24.912]s$ and the unload time is in the range $[14.326, 17.438]s$. In the 'CAV-jukebox' the drives use CAV technology and the transfer speed is in the range $[7.96, 20.53]MBps$. In the 'CLV-jukebox' the drives use CLV technology and the transfer speed is $7.96MBps$.

The size of the cache is 10% of the jukebox capacity. The average cache-hit rate is 63%. The cache-hit rate is nearly the same, independently of the scheduler used or the system load.

The graphics show simulations with different inter-arrival rates. The requests arrive following a Poisson distribution. When using the CAV-jukebox the inter-arrival rate is between 60 and 120 requests per hour, while when using the CLV-jukebox it is between 5 and 60 requests per hour. The drives in the CLV-jukebox are slower, therefore, the jukebox can serve only requests at a slower rate.

Figure 9 shows the response time of the different schedulers. Apart from the mean response time, we also show the maximum response time for 90% of the requests, and the maximum response time, to pinpoint the difference between the version of JEQS that uses normal quantum tasks and the version that uses only EQTs.

In Figure 9 we can clearly see that the response time of Promote-IT is always much shorter than the response time of JEQS. As the system load increases, the performance of FCFS is also better than that of JEQS. When using the CAV-jukebox the performance of JEQS is proportionally worse than when using the CLV-jukebox, because the drives in the CAV-jukebox are faster. Therefore, when using the CAV-jukebox more drive bandwidth is wasted with each switch.

Figure 9 also shows that when the system load is high, the performance of the version of JEQS that only uses EQTs degrades very fast. Note that the plots of the version of JEQS using only EQTs are not complete, because this version of JEQS cannot cope with high loads. As the load passes a certain limit, the length of the waiting queues grows so much that no new requests can be scheduled.

To compare the performance of the two versions of JEQS, it is useful to analyze the maximum response time. Figure 9(e) shows that the maximum response time of JEQS is very high, but it is a somewhat exceptional case, because Figure 9(c) shows that the maximum response time for 90% of the requests is much lower. The high maximum response time for JEQS comes from incorporating to the schedule a normal quantum task with a long period. As we explained in Section 6.1, scheduling normal quantum tasks is a trade-off to make the scheduler able to cope with higher system loads. Figure 9(c) also shows that in 90% of the cases scheduling normal quantum tasks results in a better response time than scheduling only EQTs. Figure 10(a) clearly shows that scheduling only EQTs also results in long confirmation times.

Figures 10(c) and 10(d) show that building periodic schedules needs less computation than building aperiodic schedules. However, this fact is not directly reflected in the confirmation time, because Promote-IT and FCFS have shorter confirmation times than JEQS (see Figure 10(b)).

Figures 10(e) and 10(f) show that the robot utilization of JEQS is higher than that of Promote-IT and FCFS. The high robot utilization originates from the fact that multiple switches are performed for reading data from an RSM, while Promote-IT and FCFS use the minimum amount of switches. The robot utilization of 'JEQS only EQTs' is proportionally lower than that of JEQS as the load increases, because 'JEQS only EQTs' accepts new requests at a lower rate.

9 Conclusions

We presented JEQS a jukebox scheduler for a Video-on-Demand system. JEQS is a periodic scheduler that uses early quantum tasks to provide prompt service to the incoming requests. The scheduler uses the jukebox robots in a cyclic way and defines a quantum as the time to load and unload all drives. It solves the resource-contention problem by treating the RSM as shared resources and assigning all tasks for the same RSM to one drive.

The advantages of JEQS are its simplicity when compared to aperiodic schedulers, and its correctness when compared to the other periodic jukebox schedulers.

JEQS is the only periodic schedulers that deal with the resource-contention problem correctly. However, using periodic scheduling is a bad approach when scheduling jukeboxes.

References

[1] Stergios V. Anastasiadis, Kenneth C. Sevcik, and Michael Stumm. Server-based smoothing of variable bit-rate streams. In *Proceedings 9th ACM Multimedia Conference*, pages 147–158, Ottawa, Canada, October 2001.

[2] Sanjoy K. Baruah, N. K. Cohen, C. Greg Plaxton, and Donald A. Varvel. Proportionate progress: A notion of fairness in resource allocation. In *ACM Symposium on Theory of Computing*, pages 345–354, 1993.

[3] Peter Bosch. *Mixed-media file systems*. PhD thesis, University of Twente, June 1999.

[4] Hojung Cha, Jongmin Lee, Jaehak Oh, and Rhan Ha. Video server with tertiary storage. In *Proc. of the Eighteenth IEEE Symposium on Mass Storage Systems*, April 2001.

[5] Sheng-Han Gary Chan and Fouad A. Tobagi. Designing hierarchical storage systems for interactive on-demand video services. In *Proc. of IEEE Multimedia Applications, Services and Technologies*, June 1999.

[6] Ann Louise Chervenak. *Tertiary Storage: An Evaluation of New Applications*. PhD thesis, Dept. of Comp. Science, University of California, Berkeley, December 1994.

[7] Ann Louise Chervenak. Challenges for tertiary storage in multimedia servers. *Parallel Computing*, 24(1):157–176, January 1998.

[8] Lena Golubchik and Raj Kumar Rajendran. A study on the use of tertiary storage in multimedia systems. In *Proc. of Joint NASA/IEEE Mass Storage Systems Symposium*, March 1998.

[9] Ferdy Hanssen, Pieter Hartel, Tjalling Hattink, Pierre Jansen, J. Scholten, and Jurriaan Wijnberg. A real-time Ethernet network at home. In Michael González Harbour, editor, *Proceedings Work-in-Progress session 14th Euromicro international conference on real-time systems (Research report 36/2002, Real-Time Systems Group, Vienna University of Technology)*, pages 5–8, Vienna, Austria, June 2002.

[10] Bruce K. Hillyer, Rajeev Rastogi, and Avi Silberschatz. Scheduling and data replication to improve tape jukebox performance. In *Proc. of International Conference on Data Engineering*, pages 532–541, 1999.

[11] Bruce K. Hillyer and Avi Silberschatz. Random I/O scheduling in online tertiary storage systems. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 195–204, June 1996.

[12] Bruce K. Hillyer and Avi Silberschatz. Scheduling non-contiguous tape retrievals. In *Proc. of Joint NASA/IEEE Mass Storage Systems Symposium*, pages 113 – 123, March 1998.

[13] Pierre G. Jansen, Ferdy Hanssen, and Maria Eva Lijding. Early quantum task scheduling. Technical Report TR-CTIT-02-48, Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands, November 2002.

[14] Siu-Wah Lau and John C. S. Lui. Scheduling and replacement policies for a hierarchical multimedia storage server. In *Proc. of Multimedia Japan 96, International Symposium on Multimedia Systems*, March 1996.

[15] Siu-Wah Lau, John C. S. Lui, and P. Wong. A cost-effective near-line storage server for multimedia system. In *Proc. of the 11th International Conference on Data Engineering*, pages 449–456, March 1995.

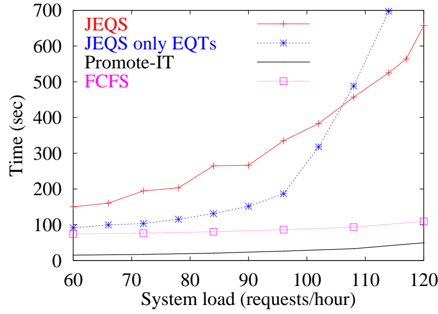
[16] Maria Eva Lijding, Pierre G. Jansen, and Sape J. Mullender. A flexible real-time hierarchical multimedia archive. In *Joint Int. Workshop on Interactive Distributed Multimedia Systems / Protocols for Multimedia Systems (IDMS/PROMS)*, page to appear, Coimbra, Portugal, Nov 2002. Springer-Verlag, Berlin.

[17] Maria Eva Lijding, Sape J. Mullender, and Pierre G. Jansen. A comprehensive model of tertiary-storage jukeboxes. Technical Report TR-CTIT-02-41, Centre for Telematics and Information Technology, University of Twente, October 2002.

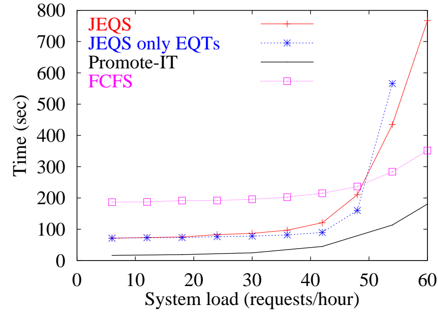
[18] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[19] Sunil Prabhakar, Divyakant Agrawal, Amr El Abbadi, and Ambuj Singh. Scheduling tertiary I/O in database applications. In *Proc. of the 8th International Workshop on Database and Expert Systems Applications*, pages 722–727, September 1997.

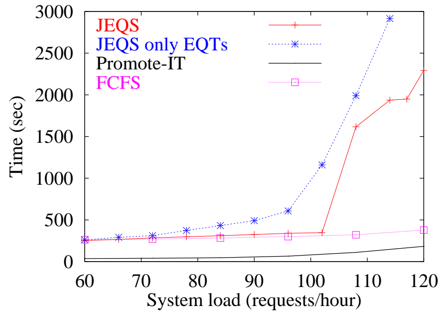
[20] Peter Triantafillou and I. Georgiadis. Hierarchical scheduling algorithms for near-line tape libraries. In *Proc. of the 10th International Conference and Workshop on Database and Expert Systems Applications*, pages 50–54, 1999.



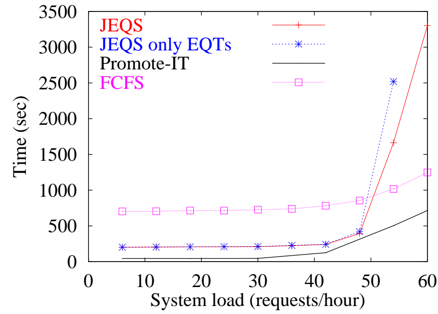
(a) Mean response time. CAV-jukebox.



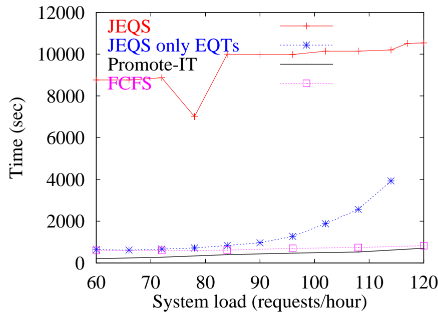
(b) Mean response time. CLV-jukebox.



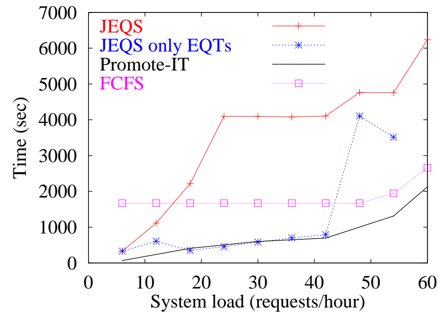
(c) Maximum response time for 90% of the requests. CAV-jukebox.



(d) Maximum response time for 90% of the requests. CLV-jukebox.

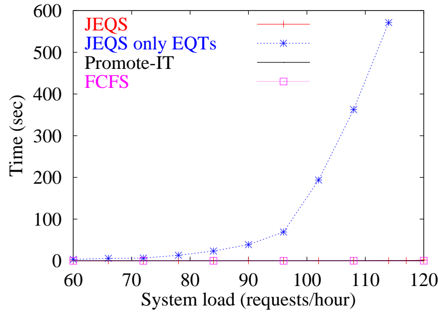


(e) Maximum response time. CAV-jukebox.

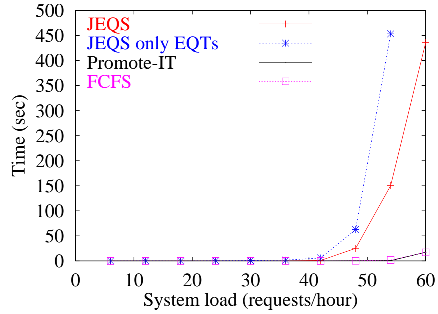


(f) Maximum response time. CLV-jukebox.

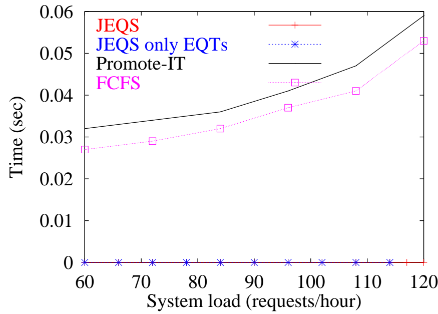
Figure 9: Response time for Promote-IT, JEQS and FCFS for the CAV-jukebox and the CLV-jukebox. The graphics are scaled for best resolution, thus the scales are different.



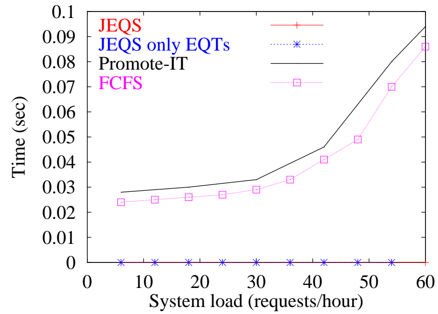
(a) Mean confirmation time. CAV-jukebox.



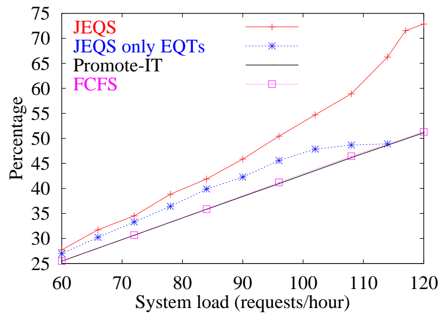
(b) Mean confirmation time. CLV-jukebox.



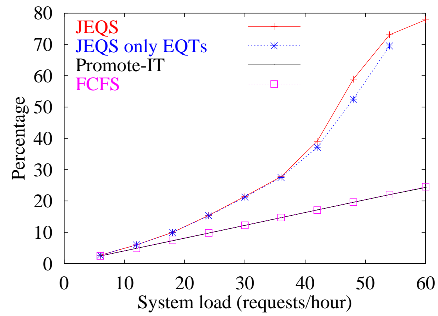
(c) Mean computing time. CAV-jukebox.



(d) Mean computing time. CLV-jukebox.



(e) Mean robot utilization. CAV-jukebox.



(f) Mean robot utilization. CLV-jukebox.

Figure 10: Confirmation time, computing time, and mean robot utilization for Promote-IT, JEQS and FCFS for the CAV-jukebox and the CLV-jukebox. The graphics are scaled for best resolution, thus the scales are different.