A Flexible Real-time Hierarchical Multimedia Archive

Maria Eva Lijding, Pierre Jansen, and Sape Mullender

Fac. of Computer Science, University of Twente
P.O.Box 217, 7500AE Enschede, The Netherlands
lijding@cs.utwente.nl

Abstract. We present a hierarchical multimedia archive that can serve complex multimedia requests from tertiary storage. Requests can consist of multiple request units of streamed and non-streamed data. The request units can have arbitrary synchronization patterns.
Our scheduler *Promote-IT* promotes data from tertiary to secondary storage with real-time guarantees. Promote-IT uses an on-line heuristic algorithm to compute feasible schedules and a separate ASAP dispatcher to increase the efficiency of the resource usage. The heuristic algorithm runs in polynomial time. Schedules are optimized to give short response times to incoming requests.
Three major problems complicate this scheduling problem. First, the fragments of requested real-time data and their synchronization are unpredictable. Second, the medium switching times in tertiary storage are high, and the number of drives and robots is low compared to the number of removable media. Third, the shared resources in the tertiary storage system create resource contention problems.

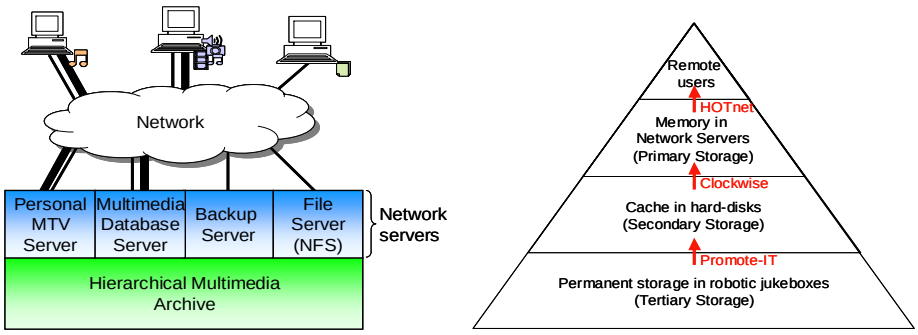
1 Introduction

We present a flexible *hierarchical multimedia archive (HMA)* that can serve complex requests for the real-time delivery of any combination of media files it stores. Such requests can originate from any system that needs to combine multiple, separately stored media files into a continuous presentation. The HMA can also be used for the more simple case of a Video-on-Demand (VoD) application, where the requests are generally for only a single media file—a movie—to be played from beginning to end. To the best of our knowledge there is no other hierarchical storage system that provides flexible requests and time constraints.

A request can consist of multiple streams and non-streamed data that are synchronized sequentially or concurrently in arbitrary patterns. Examples are queries to multimedia databases to assemble a TV documentary, or a computer generated play list for a huge library of music videos and advertisement that produces a MTV-like program.

The multimedia data is stored in a tertiary-storage jukebox. Tertiary storage can store large amounts of data in a cost-effective way, which makes it eminently suitable for applications handling continuous-media files, large databases and backup data. A jukebox is a large tertiary storage device whose *removable storage media (RSM)*—e.g. CD, DVD, magneto-optical disk, tape—are loaded and unloaded from one or more drives by one or more robots.[1]

[1] The acronym RSM stands both for the singular and the plural. In the literature, a jukebox is sometimes called a Robotic Storage Library.



(a) System overview. (b) Storage hierarchy.

Fig. 1. System overview with various application specific network servers and storage hierarchy showing the promotion mechanisms.

The problem with tertiary storage is that RSM switching times are high and the number of available drives and robots is low compared to the number of RSM. The RSM switching time in a jukebox is in the order of seconds or tens of seconds. This implies that multiplexing between two files stored in different RSM is many orders of magnitude slower than doing the same in secondary storage. Therefore, it is very important to schedule efficiently the use of the jukebox resources. On the other hand, the bandwidth offered by the devices in a jukebox is generally much higher than the one required by the end users. Thus, it makes good sense to stage data in secondary storage buffers from where it is delivered to the applications.

The hierarchical multimedia archive acts as a real-time file system [8] and, thus, does not offer application specific services. We envision multiple *network servers* running on top of the HMA where each provides a specific service to the users as shown in Figure 1(a). In the Distributed and Embedded Systems group at the University of Twente, we are concerned with providing end-to-end quality of service to the users. Our solution is to provide the in-time promotion of data from each level of the storage hierarchy to the next. Figure 1(b) shows the storage hierarchy of the overall system and the mechanisms used to promote data between contiguous storage levels.

The focus of this paper is *Promote-IT (Promote In Time)*, the jukebox scheduler that guarantees the in-time promotion of multimedia data from tertiary storage to secondary storage. In turn Clockwise [2] provides real-time access to data stored in secondary storage, which is used in HMA as cache, and finally *HOTnet* [10] provides real-time guarantees for the use of a local area network.

The main goal of Promote-IT 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, Promote-IT tries to minimize the number of rejected requests, minimize the response time for ASAP requests, 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. So Promote-IT uses an heuristic algorithm that computes near-optimal schedules in polynomial time.

Promote-IT gains part of its efficiency by separating the scheduling and dispatching functionality, because their goals are different. Although separating scheduling and dispatching seems a natural design decision, it has not been used in any other jukebox scheduler. The goal of the scheduler is to find feasible schedules for the requested data. Thus, the scheduler tries to build schedules as flexibly as possible and is not concerned about the optimal use of the resources. The dispatcher, instead, is concerned about utilizing the jukebox resources in an efficient manner. Thus, it dispatches the tasks to the hardware controllers as soon as possible (ASAP). The dispatcher may modify the schedules built by the scheduler as long as no task in the schedule is delayed and the sequence and resource constraints are respected.

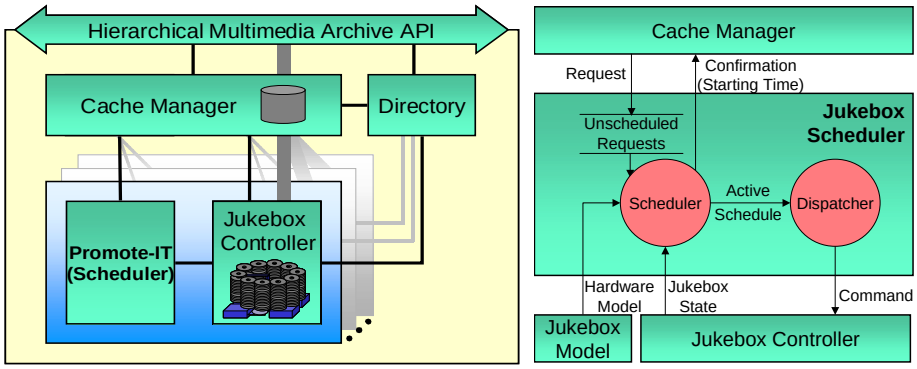
To achieve a high degree of flexibility when building the schedules, we model the scheduling problem in a novel way. The model is a *flexible flow shop* [16] with three stages. The first stage is to load the RSM into a drive, the second to read data from the RSM, and the third is to unload the RSM. The model uses shared resources to guarantee the mutual exclusive use of shared robots and RSM. Although solving this scheduling problem is far from trivial, it allows to build efficient schedules that deal with the resource contention problem correctly. Another advantage of this model is that it allows us to schedule for any type of jukebox architecture, e.g. one robot, multiple shared robots, dedicated robots, different drive models, etc.

The rest of this paper is organized as follows. Section 2 presents related work. Section 3 gives an overview of the system. Section 4 describes the jukebox scheduler. Section 5 discusses some important aspects of the implementation. Section 6 presents an evaluation of the system. Finally, Section 7 concludes the paper.

2 Related Work

Scheduling of tertiary storage has been studied mainly in the context of Video-on-Demand (VoD) systems and in systems with no time constraints. An assumption in typical VoD systems is that requests are for a single media file to be played continuously from beginning to end. To the best of our knowledge there is no previous work with flexible requests and time constraints.

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. These two strategies are similar to the strategies EDF (earliest deadline first) and LDL (latest deadline last) that we use in Promote-IT (see Section 4). However, there are very important differences. The most important difference is that their system dispatches the tasks exactly in the same way in which they are scheduled. Thus, they make bad use of the schedules built by the conservative strategy. The schedules they build are less efficient than ours, because they handle the RSM switch (load+unload) as one task and they try to schedule the reads only with the best drive. Furthermore, their scheduler can handle only jukeboxes in which all the drives are identical and the load and unload time are constant for every drive and every RSM. Promote-IT, instead, is able to handle any type of jukebox architecture, including jukeboxes in which the drives are different.



(a) Hierarchical Multimedia Archive. (b) Jukebox Scheduler.

Fig. 2. Architecture of the Hierarchical Multimedia Archive and the Jukebox Scheduler.

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. This type of algorithm is not appropriate for the flexible type of requests that the HMA handles, because a FCFS approach cannot deal with relative deadlines.

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. [9] 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.

Prabhakar et al. [17] and Triantafillou et al. [19] schedule requests without real-time deadlines in order to minimize the mean response time. The conclusion of their work is that as much data as possible should be read from an RSM when the RSM is loaded in the drive. This supports the approach we used in Promote-IT where we read all the data requested from an RSM before the RSM is unloaded. Hillyer et al. [13, 12, 11] compare different scheduling algorithms for retrieving data from a magnetic tape without real-time requirements.

3 System Overview

This section gives an overview of the hierarchical multimedia archive (Figure 2(a)). The data of the archive can be stored in multiple jukeboxes. Each jukebox has its own scheduler and controller, thus providing scalability to the system, because the complexity of the scheduler does not increase by incorporating more jukeboxes.

A request arriving at the system is filtered by the cache manager, which checks whether any of the requested data is already in the cache or scheduled for staging. It then consults the directory to find out in which jukebox(es) the remaining data is stored and sends the appropriate requests to them.

The cache manager may be physically distributed, as proposed by Brubeck et al. [3], to avoid becoming a bottle-neck. The directory is a database that contains meta-data about the contents of the jukeboxes and can easily be distributed or replicated. In this paper, we do not address the cache manager and directory any further, but consider only the scheduling of tertiary storage.

Figure 2(b) shows the architecture of *Promote-IT (Promote In Time)*, the jukebox scheduler of our system. Promote-IT schedules incoming requests on-line, recomputing the schedule every time a request arrives. It generates a new schedule to replace the currently *active schedule* only if it can guarantee that including the new request does not lead to missed deadlines. The dispatcher uses the active schedule to send commands to the jukebox controller to move RSM and stage data into secondary storage. The dispatcher guarantees that the tasks are sent to the controller in time.

Once the HMA accepts and confirms a request from a user, 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.

3.1 Requests

The requests consist of a deadline and a set of request units u_{ij} for individual files, or part of files. The requests can represent any kind of static temporal relation between the request units. Formally we represent a request r_i with l_i request units as:

$$r_i = (\tilde{d_i}, \mathit{asap}_i, \mathit{maxConf}_i, \{u_{i1}, u_{i2}, \ldots, u_{il_i}\})$$
$$u_{ij} = (\Delta\tilde{d}_{ij}, m_{ij}, o_{ij}, s_{ij}, b_{ij})$$

The deadline $\tilde{d_i}$ of the request is the time by which the user must have guaranteed access to the data. The flag asap_i indicates if the request should be scheduled as soon as possible. The maximum confirmation time $\mathit{maxConf}_i$ is the time the user is willing to wait in order to get a *confirmation* from the system, which indicates if the request was accepted or rejected. The relative deadline of the request unit $\Delta\tilde{d}_{ij}$ is the time at which the data of the request unit should be available, relative to the starting time of the request. The other parameters of the request unit m_{ij}, o_{ij}, s_{ij} and b_{ij} represent the RSM where the data is stored, the offset in the RSM, the size of the data, and the bandwidth with which the user wants to access the data, respectively.

The confirmation to the user indicates if the request is accepted or rejected. If the request is accepted, the confirmation contains the starting time st_i assigned to the request. The starting time must be less or equal to the deadline of the request ($st_i \leq \tilde{d_i}$). If the request is ASAP the scheduler tries to find the earliest value of st_i that will allow it to accept the request. The system must provide a confirmation before $\mathit{maxConf}_i$.

3.2 Hardware Model

Tertiary-storage jukeboxes are composed of the following hardware: *drives* to access the data in the RSM, *shelves* where the RSM are kept and *robots* to move the RSM from the shelves to the drives and vice versa.

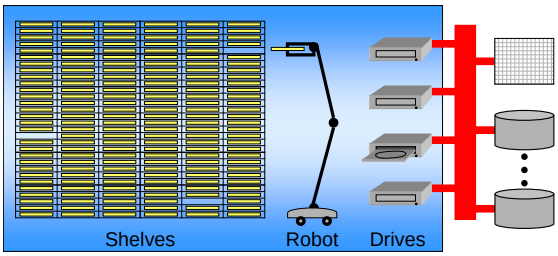


Fig. 3. Jukebox architecture.

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, DVD, magnetic tape or magneto-optical disk. Figure 3 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.

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 in the case of disks, rewind time for tapes, and the distance between the drive and the shelf where the RSM is kept.

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.

4 Promote-IT

Promote-IT is the jukebox scheduler of the hierarchical multimedia archive. It guarantees that the data is promoted to secondary storage in time. Promote-IT schedules the incoming requests on-line, building a new schedule that includes all the request units that still need scheduling plus the request units of the new request. Promote-IT provides short response times, short confirmation times and makes good use of the jukebox resources.

We now formalize the scheduling problem. Let us assume that at time t_0 request r_k arrives at a jukebox scheduler. At time t_0 the jukebox scheduler has a set of request units from previous requests, $\mathcal{U}$, that have not yet been dispatched to the jukebox. The goal of the scheduling algorithm is to find a feasible schedule for the new set $\mathcal{U}'$, which includes the request units of the new request r_k.

$$\mathcal{U} \subseteq \{u_{ij} \mid i < k, j \leq l_i\}$$
$$\mathcal{U}' = \mathcal{U} \cup \{u_{k1}, u_{k2}, \ldots, u_{kl_k}\}$$

The starting time of the request must not be later than its deadline, so $st_k \leq \tilde{d}_k$. If the request is ASAP, the scheduler assigns the request the earliest possible starting time st_k that will allow it to be incorporated into the system. Thus, the scheduler must find the minimum starting time st_k that makes $\mathcal{U}'$ schedulable. The scheduler tries different

candidate starting times st_k^x and selects the earliest feasible st_k^x. If the request is not ASAP, the scheduler assigns it the starting time corresponding to its deadline. If the deadline of the request cannot be met, then the scheduler rejects the request.

Determining if the set $\mathcal{U}'$ is schedulable is an $\mathcal{NP}$-hard problem and so is finding the the minimum starting time that makes $\mathcal{U}'$ schedulable. We use a polynomial heuristic algorithm to schedule the requests. In order to simplify the problem we assume that the user can start consuming the data of a request unit only once all its data has been buffered. In this way we can compute a deadline for each request unit in a request, namely, $\tilde{d}_{ij} = st_i + \Delta\tilde{d}_{ij}$.

The scheduler uses an iterative algorithm to schedule a request. The algorithm keeps a list of candidate starting times already analyzed and the schedules produced for them. The structure of the algorithm is the following:

1. Generate a candidate starting time st_k^x and update the deadline of each request unit so that $\tilde{d}_{kj} = st_k^x + \Delta\tilde{d}_{kj}$. The algorithm uses a variation of the bisection method for finding roots of mathematical functions.
2. Compute the *medium schedules* for the RSM corresponding to the request units in $\mathcal{U}'$. The medium schedule determines the sequence in which the requested data of an RSM is read.[2] We read all the requested data of an RSM at once. Therefore, there is at most one medium schedule per RSM. As there may be different drive models in the jukebox, we compute a medium schedule for each drive.
3. Model as a *flexible flow shop* scheduling problem with three stages (FF_3) [16]. The first stage is to load an RSM to a drive, the second stage is to read the data from the RSM and the third is to unload the RSM. The scheduling problem is represented as a set of jobs $\mathcal{J}$, where each job J_j must execute one task corresponding to each stage ($J_j = \{T_{1j}, T_{2j}, T_{3j}\}$). There is at most one job for every RSM, because the read task (T_{2j}) involves reading all the requested data from the RSM.
4. Compute the resource assignment. The algorithm must incorporate each job $J_i \in \mathcal{J}$ into the schedule. If the algorithm succeeds in finding a valid resource assignment, the output of this step is a feasible schedule S^x; otherwise $S^x = \emptyset$. The pair (S^x, st_k^x) is incorporated into the list of analyzed solutions.
5. Repeat from step 1 until the bisection stop-criteria is fulfilled for the list of candidates, i.e. the time difference between the last unsuccessful and first successful candidate is smaller than a threshold.
6. Select the best solution. The algorithm selects the earliest candidate starting time for which step 4 could compute a feasible schedule ($\min\{st_k^x \mid S^x \neq \emptyset\}$). If there is no such st_k^x the request r_k is placed in the list of unscheduled requests to be scheduled at a later time. Otherwise the scheduler confirms the starting time st_k to the user and replaces the active schedule with the new feasible schedule.

The problem model we use allows us to to represent any kind of jukeboxes (e.g. jukeboxes with one shared robot, multiple shared robots, multiple dedicated robots). We use additional shared resources to guarantee the mutual exclusive use of the robots

[2] Finding the optimal sequence is equivalent to solving the *asymmetric traveling salesman problem with time windows* [1], and is thus an $\mathcal{NP}$-hard problem. We use an algorithm that finds a near-optimal sequence.

and drives. The model imposes the restriction that all the data requested from an RSM must be read before the RSM is unloaded. This restriction results in a good system performance, because with each switch effective bandwidth is lost. Prabhalkar [17] shows that reading all the requested data from an RSM at once is good even when the access time in a loaded RSM dominates over the switching time.

In step 4 we use a branch-and-bound algorithm to prune the tree of possible assignments of jukebox resources to the jobs in $\mathcal{J}$. The branch-and-bound algorithm uses the *best-drive heuristic* to choose which drive will be tried first to schedule a job and prune from the tree the branches corresponding to drives which offer a worse solution. When pruning the tree, the algorithm may be throwing away a feasible solution that an optimal scheduler would find. But searching the whole tree of solutions is computationally impossible. For comparison, we have also implemented an optimal scheduler, but it can take up to several days to compute a feasible schedule for one new request, in contrast to the few milliseconds needed by Promote-IT. The complexity of our algorithm is $O(m!\, n)$, where m is the number of drives and n is the number of jobs in $\mathcal{J}$.

We have defined four strategies to incorporate the jobs to the schedule: *earliest deadline first* (EDF), *earliest starting time first* (ESTF), *latest deadline last* (LDL) and *latest starting time last* (LSTL). They are classified by two axes indicating the way in which the jobs are incorporated to the schedule: *Front-to-Back* and *Back-to-Front*, and the parameter of the tasks to use for sorting the jobs:*deadline* or *latest starting time (LST)*. The LST of a task is the latest time at which a task should start in order not to miss its deadline. None of the strategies is absolutely better than the others, because each strategy can find schedules that cannot be found by the others. However, their performance varies considerably depending on the usage scenario.

Using the F2B strategy each job is scheduled as early as possible and the jobs are incorporated to the front of the schedule in increasing order of 'restrictiveness', in such a way that the most 'restrictive' jobs are incorporated to the schedule first. In addition the tasks are also incorporated F2B, scheduling first the load T_{1j}, then the read T_{2j} and finally the unload T_{3j}. When using the B2F strategy, each job is scheduled as late as possible and the jobs are incorporated to the back of the schedule in decreasing order of 'restrictiveness'. The tasks in turn are incorporated B2F, scheduling first the unload T_{3j}, then the read T_{2j} and finally the load T_{1j}. In both cases, the goal is that the most restrictive tasks should be at the front of the resulting schedule, while the less restrictive tasks should be at the back. We determine the 'restrictiveness' of a job using either the deadline or the LST of the read tasks.

Figure 4 shows an example of the schedules built for a set of four jobs with EDF and LDL, and a jukebox with one robot and two identical drives. Both drives and the robot are initially empty. For the sake of simplicity we assume in this example that the load and unload time is constant for all drives and shelves.

Both strategies can build feasible schedules, but these schedules are quite different. Many times LDL is more successful in finding feasible schedules than EDF because it delays unloads as much as possible. In that way, the unloads tend to interfere less with other jobs, as illustrated by the unloading of J_1 in the example. If J_2 would have only a slightly earlier deadline, EDF would no longer find a feasible schedule because its load would overlap J_1's unload. The LDL-schedule would not be affected by the change.

The graphic at the bottom shows the resource utilization resulting from using the LDL-schedule with the ASAP dispatcher. This schedule is very compact and makes excellent use of the resources, specially of the shared robot. The dispatcher can easily fill the holes in an LDL-schedule by successively moving tasks to the front. The holes in an EDF-schedule allow no such optimization, except by small amounts when a task finished earlier than predicted. In the dispatched schedule there are two places where the dispatcher created a 'jump', changing the order of the tasks in the robot schedule.

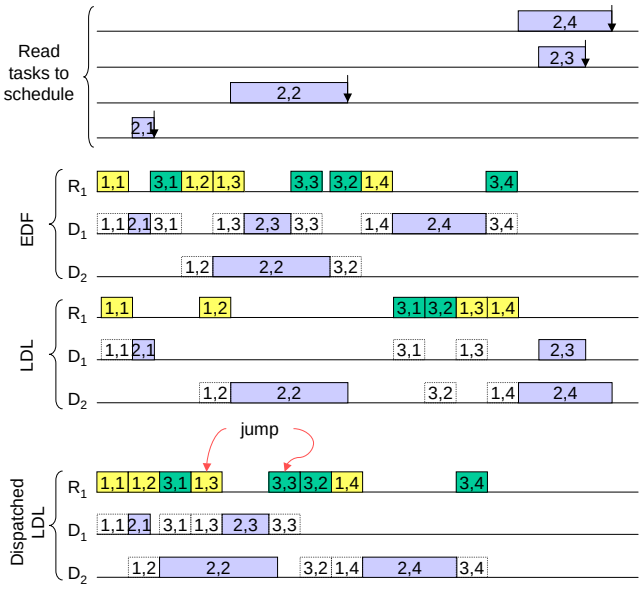


Fig. 4. Example of an EDF and LDL schedule. (i, j of T_{ij} shown in the boxes)

5 Implementation and Simulation

We have implemented the HMA in Java 1.3 on Linux. The system can run the same code when using a real jukebox and a simulation of a jukebox. When using a real jukebox, the jukebox controller sends commands to the jukebox and waits for the replies. When using simulated hardware, it places an event on the event queue simulator indicating the time at which it should be woken up. The drive controller uses the Java Native Interface (JNI) to call C functions in order to open and close the drives and get drive specific information. We use the same hardware models to simulate the behavior of the jukebox and to estimate the processing time needed for the different tasks to schedule.

At present the cache manager uses a Linux file system to store the data, which does not provide real-time guarantees. However, it seems to be good enough for the applications we have tested and it is of no relevance when simulating. The network servers use the Java Media Framework [18] to stream data to the remote users. Although JMF cannot provide real-time guarantees, it is useful for building prototype network servers. The next step is to integrate Clockwise and HOTnet into the HMA.

6 Evaluation

In this section we present a brief evaluation of the jukebox scheduler. We compare the different scheduling strategies used in Promote-IT and show the superiority of Promote-

IT over the scheduling strategies proposed by Lau et al. [14].[3] On one hand we show the effectivity of the ASAP dispatcher by comparing the LDL strategy against the conservative strategy. On the other hand we show that EDF performs better than the aggressive strategy. We also show how the performance of the system improves considerably by specifying concrete deadlines for the ASAP requests.

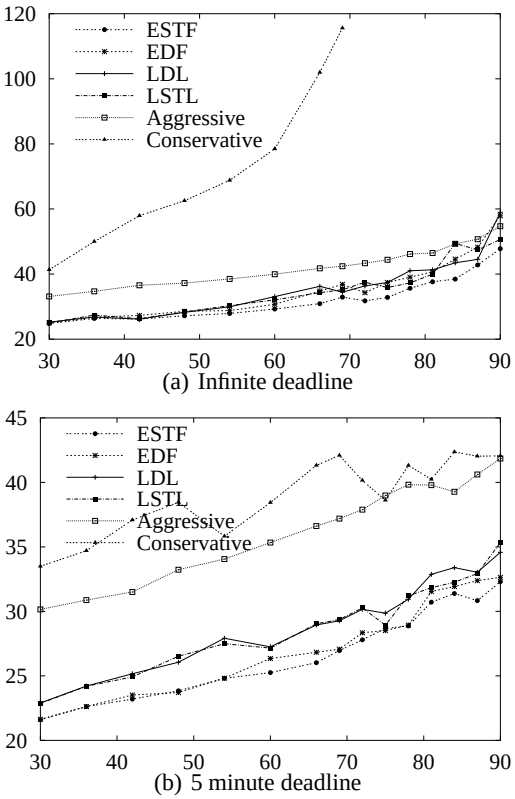


Fig. 5. Mean response time (y-axis; seconds) over system load (req./hour) with CD jukebox.

For Figures 5(a) and 5(b) we used the hardware model of our DAX jukebox [7], which has 720 shelves, 1 robot and 4 drives—three CD readers (1.75 MBps) and one 8X CD reader/writer (1.17 MBps). The average switch time is 44 seconds. For Figure 6 we assume that the jukebox has four 16X DVD drives. The DVD drives use CAV technology (constant angular velocity) and the transfer speed is 6.45 MBps at the inner track and 19.53 MBps at the outer track.

The test set used consist of 1000 ASAP requests for audio, video and discrete-data. The data requested follows a Zipf distribution, because this type of distribution represents correctly the pattern of requests on storage systems [6]. In the first case the deadline of the requests is infinite, so the system does not reject any request and the response times can become very high. In the second and third case the requests have a deadline of 5 minutes and a maximum confirmation time of 30 seconds, so if the scheduler does not succeed within that limit it rejects the request.

The cache manager uses *least recently used (*LRU*)* policy. The capacity of the cache is 10% of the jukebox capacity. The cache is preloaded with the results of another simulated run. The requested data produced in average 57% cache hits when using the CD jukebox and 60% when using the DVD jukebox. This did not change significantly when varying the system load or the scheduling strategy.

Figure 5(a) clearly shows that the conservative strategy performs very badly, even when the system load is not very high, because it makes a poor use of the jukebox resources by dispatching as late as possible. The graphic also shows that EDF performs

[3] We have extended their aggressive and conservative strategies for jukeboxes with different drive characteristics.

better than the aggressive strategy. This is mainly noticeable when the system load is low, because the aggressive strategy leaves the drives loaded until they are needed again by the system. Thus, when the system load is low, the robot is left idle while it could be unloading an RSM and freeing a drive. Leaving the drives loaded could be a good idea if the probability of receiving new requests for an RSM loaded in a drive would be high, but given the big number of RSM in the jukebox the probability is very low.

Figure 5(b) shows a considerable improvement in the response time of Promote-IT when the requests have deadlines and the scheduler is able to reject some requests. The improvement is considerable, even if the percentage of rejected requests is less than 2%. The aggressive strategy does not improve much, because it leaves the drives loaded. The rejection ratio of the conservative strategy is high (between 1 and 11%).

Figure 6 shows the performance of the different scheduling strategies of Promote-IT and the aggressive strategy when using the DVD jukebox. This graphic does not show the conservative strategy, because its rejection ratio is so high, that the mean response time is not comparable to the others. In this architecture the robot is the clear bottleneck in the system. The utilization of the robot reaches nearly 95%, while the utilization of the drive is less than 35%. The two Back-to-Front strategies show the best performance as the load increases, because Back-to-Front makes better use of the robot.

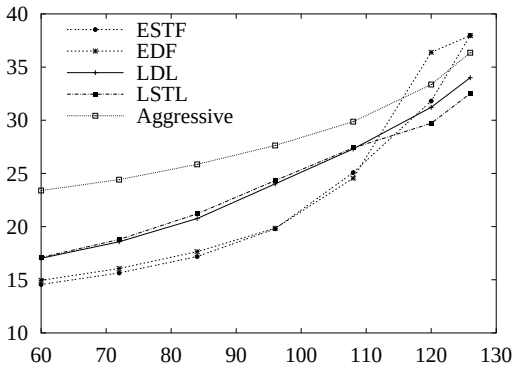


Fig. 6. Mean response time (y-axis; seconds) over system load (req./hour) with DVD jukebox and 5 min deadline.

7 Conclusions and Future Work

We present a hierarchical multimedia archive that provides flexible real-time access to large volumes of data stored in tertiary storage. Promote-IT guarantees the in-time promotion of data from tertiary to secondary storage. Promote-IT is a key component in the end-to-end real-time delivery system that spans from tertiary storage to end-user applications.

We describe a polynomial heuristic algorithm to solve the scheduling problem, whose optimal solution is $\mathcal{NP}$-hard to find. Promote-IT provides short response times while using the jukebox resources efficiently and performs better than comparable schedulers. At present we are comparing the performance of Promote-IT against that of an optimal scheduler. The results obtained so far indicate that Promote-IT performs very near the optimal. We are also comparing the performance of Promote-IT against a new periodic scheduler that uses the robot in a cyclic way, called *cached early quantum scheduler (CEQS)*. Promote-IT performs clearly better that CEQS with all jukebox architectures and request sets we have tested so far.

A distinguishing feature of Promote-IT is the separation of scheduling and dispatching. The scheduler can build flexible schedules with holes in which the resources are scheduled to be idle, while the ASAP dispatcher dispatches each task as soon as possible. This feature makes the Back-to-Front strategy, where each task is scheduled as late as possible, a competitive strategy.

References

1. N. Ascheuer, M. Fischetti, and M. Grötschel. Solving the asymmetric travelling salesman problem with time windows by branch-and-cut. *Math. Program.*, 90(3):475–506, 2000.
2. P. Bosch. *Mixed-media fi le systems*. PhD thesis, University of Twente, June 1999.
3. D. W. Brubeck and L. A. Rowe. Hierarchical storage management in a distributed vod system. *IEEE Multimedia*, 3(3):37–47, 1996.
4. H. Cha, J. Lee, J. Oh, and R. Ha. Video server with tertiary storage. In *Proc. of the Eighteenth IEEE Symposium on Mass Storage Systems*, April 2001.
5. S.-H. G. Chan and F. A. Tobagi. Designing hierarchical storage systems for interactive on-demand video services. In *Proc. of IEEE Multimedia Applications, Services and Technologies*, June 1999.
6. A. L. Chervenak. *Tertiary Storage: An Evaluation of New Applications*. PhD thesis, Dept. of Comp. Science, University of California, Berkeley, December 1994.
7. Chess Engineering bv. *DAX Software Architecture Manual, Version 0.5*, March 1998.
8. D. Gemmel, H. Vin, D. Kandlur, P. Rangan, and L. Rowe. Multimedia storage servers: A tutorial and survey. *IEEE Computer*, 28(5):40–49, November 1995.
9. L. Golubchik and R. K. Rajendran. A study on the use of tertiary storage in multimedia systems. In *Proc. of Joint NASA/IEEE Mass Storage Systems Symposium*, March 1998.
10. F. Hanssen, P. Hartel, T. Hattink, P. Jansen, J. Scholten, and J. Wijnberg. A Real-Time ethernet network at home. In M. G. Harbour, editor, *Research report 36/2002*, pages 5–8, Vienna, Austria, June 2002. Real-Time Systems Group, Vienna Univ. of Technology.
11. B. K. Hillyer, R. Rastogi, and A. Silberschatz. Scheduling and data replication to improve tape jukebox performance. In *Proc. of International Conference on Data Engineering*, pages 532–541, 1999.
12. B. K. Hillyer and A. 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.
13. B. K. Hillyer and A. Silberschatz. Scheduling non-contiguous tape retrievals. In *Proc. of Joint NASA/IEEE Mass Storage Systems Symposium*, pages 113 – 123, March 1998.
14. S.-W. Lau and J. C. 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. S.-W. Lau, J. C. 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. M. Pinedo. *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, 1995.
17. S. Prabhakar, D. Agrawal, A. E. Abbadi, and A. 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.
18. Sun Microsystems. *Java Media Framework API Guide*, November 1999.
19. P. 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.