Real-time scheduling of a tertiary-storage jukebox

Maria Eva Lijding, Pierre Jansen, Sape Mullender
Distributed and Embedded Systems
University of Twente, Netherlands
lijding@cs.utwente.nl

Abstract— **We present a jukebox scheduler for real-time data. The scheduler is part of a hierarchical real-time file system to be used over a network. A jukebox is a large tertiary storage device whose removable media (e.g. cd-rom, dvd-rom) are loaded and unloaded from one or more drives by a robot. The problem with tertiary storage is that media exchange times are high and the number of drives is limited. This makes scheduling tertiary storage complicated. The storage media switching time in a jukebox is in the order of tens of seconds. Therefore multiplexing between two files stored in different media is many orders of magnitude slower than doing the same in secondary storage.**

The goal of the scheduler is to schedule the use of the jukebox devices (arm and drives) in such a way that the system can guarantee the deadlines while minimizing the response time. The problem is similar to that of scheduling multiple processors with the additional difficulty of having to deal with the high switching times and the use of a shared resource (the arm).

Finding an optimal schedule is an NP-hard problem. We provide a near-optimal polynomial solution by using heuristics to prune the tree of solutions. The scheduling time is in average less than 100 ms. The incoming requests are scheduled on-line.

Keywords—**scheduling, real-time, tertiary-storage, multimedia file system**

I. Introduction

We present a jukebox scheduler for real-time data. A jukebox is a large tertiary storage device whose removable storage media (CD-ROM, DVD, magneto-optical disk, tape) are loaded and unloaded from one or more drives by a robot. Tertiary storage can store large amounts of data which makes it eminently suitable for continuous-media, images and backup.

An area of interest for the use of tertiary storage is video on demand. In [6], [5] Lau et al. present a hierarchical multimedia storage server, specially designed for video on demand applications. An original contribution from our work is to use tertiary storage efficiently for a general file system with real-time guarantees.

The problem with tertiary storage is that media exchange times are high and the ratio of drives to data is low. The media switching time in a jukebox is on the order of seconds or tens of seconds. This implies that multiplexing between two files files stored in different media is many orders of magnitude slower than doing the same in secondary storage. That is a reason why data stored in tertiary storage is generally said to be *near-line*, as opposed to *on-line* data. Scheduling tertiary storage is, therefore, more complicated. 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. It makes good sense to read data into a secondary storage buffer from where it is provided to the applications.

The jukebox scheduler knows what data the users will need and when they will need it. With this information we build a real-time schedule that guarantees that all the data will be in the buffer by the time the user needs it. The schedule is computed *on-line* every time a new request arrives. We construct the schedule using a policy that we called *latest deadline last (*LDL*)*. We also experimented with some other policies as *earliest deadline first (*EDF*)*, but LDL proved to be more efficient in all the tests we performed. When building the schedules with LDL each task is scheduled as late as possible, while when using EDF it is scheduled as early as possible.

A schedule built with LDL has got idle periods (holes) in which the resources are not assigned to any task. Building schedules in this ways helps finding a feasible schedule, but it leaves resources idle. It is a very bad policy to leave resources idle when there are tasks that need to be executed, because there will be more tasks to schedule when new requests arrive. The dispatcher uses the schedule built by the scheduler in a flexible way, dispatching the tasks as early as possible to idle devices while keeping the guarantees that the original schedule has. In this way the dispatcher also makes good use of the fact that some resources are available earlier than estimated, because the schedules are built with worst case estimates.

The rest of the paper is organized as follows. In Section II we present an overview of the file system

and the scheduler. In Section III we present a formalization of the scheduling problem. In Section IV we discuss the principles of our scheduling algorithm. In Section V we present an evaluation of the scheduler. In Section VI we discuss future work and we finish the paper in Section VII with some concluding remarks.

II. System Overview

The main goal of the jukebox scheduler is to guarantee that data is buffered into secondary storage by the time applications need it (i.e., real-time deadlines).

The requests to the system have the following structure:

$$r_i = \{d_i, \{ru_{i1}, ru_{i2}, \ldots, ru_{il_i}\}\}$$

The deadline, d_i, of the request indicates the latest time the user must be guaranteed access to the data. When a request is submitted, d_i may be fixed and the scheduler will either schedule the request so all deadlines are met, or report the request inadmissible; d_i may also be set to the special value ASAP *(as soon as possible)* which will cause the scheduler to reset d_i to the earliest value that allows all deadlines to be met. A request may be formed by any number of request units, $ru_{ij} \mid j \leq l_i$.

Request unitjs describe pieces of a real-time data stream or block. A request unit has got the following structure:

$$r_{ij} = \{m_{ij}, o_{ij}, s_{ij}, \Delta d_{ij}, b_{ij}\}$$

m_{ij} is the CD [1] where the data of the request unit is stored, o_{ij} is the offset in the CD, s_{ij} is the size of the data requested, Δd_i the relative deadline of the request unit, and b_{ij} is the bandwidth with which the user wants to access the data. If ru_{ij} is a *block*, that is when b_{ij} is 0, the deadline of the request unit, d_{ij}, is $d_i + \Delta d_{ij}$. When it is a *stream*, the deadline of the nth byte in r_{ij}, is $d_i + \Delta d_{ij} + n/b_{ij}$.

In Fig. 1 shows the graphical representation of the following request:

$$\begin{aligned} r_k \;=\; & \text{ASAP}, \\ & \{(\text{CD}1, 200\ \text{KB}, 115200\ \text{KB}, 0\ s, 128\ \text{K}bps), \\ & (\text{CD}2, 60\ \text{KB}, 614400\ \text{KB}, 0\ s, 1024\ \text{K}bps), \\ & (\text{CD}3, 62\ \text{KB}, 645120\ \text{KB}, 2400\ s, 1024\ \text{K}bps), \\ & (\text{CD}4, 54\ \text{KB}, 583680\ \text{KB}, 4920\ s, 1024\ \text{K}bps)\} \end{aligned}$$

[1]In the rest of the text we will call the removable storage media simply CD, but this does not imply that what is presented is limited to CD-ROM technology. We could have as well called it media, but this may provoke confussion with media types as audio, video, etc., with which we are also concerned.

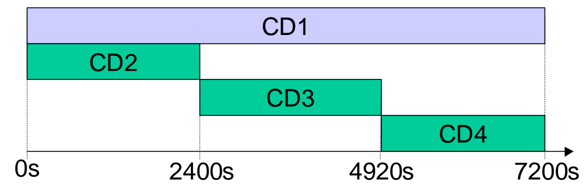

Fig. 1. Example of a request

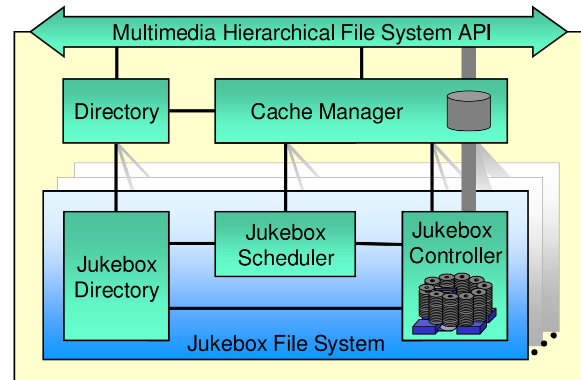

Fig. 2. Architecture of the file system

The request is for a film that has the audio stored in a separate file. The request specifies that the audio and the first file containing the video must be available ASAP. The second file containing video (CD 3) must be available 40 minutes later and the last file 82 after the begining of the film. The audio will be consumed at 128 Kbps and the video at 1 Mbps.

In Fig. 2 shows the architecture of the file system the jukebox scheduler forms part of. The data of the file system may be stored in multiple jukeboxes. Each jukebox has its own jukebox scheduler. The requests arriving at the jukebox scheduler are filtered first by the cache manager, which decides what data is already in the cache or being copied to the cache at that moment. The cache manager consults the directory to find out in which jukebox(es) the data requested is stored. The cache manager sends the filtered requests to the corresponding jukebox schedulers, one request to each scheduler from the jukeboxes that have data needed in the request. The cache manager not only handles the cache, but also the buffer[2].

The jukebox scheduler schedules incoming requests on-line, reconstructing the schedule when a request comes in. We call this process *schedule construction*, and the resulting schedule *active schedule*. This schedule is used by the *dispatcher* to send commands to the

[2]In general we will talk about the data being in the cache, but if there is a request that needs the data, then the data is really in a buffer.

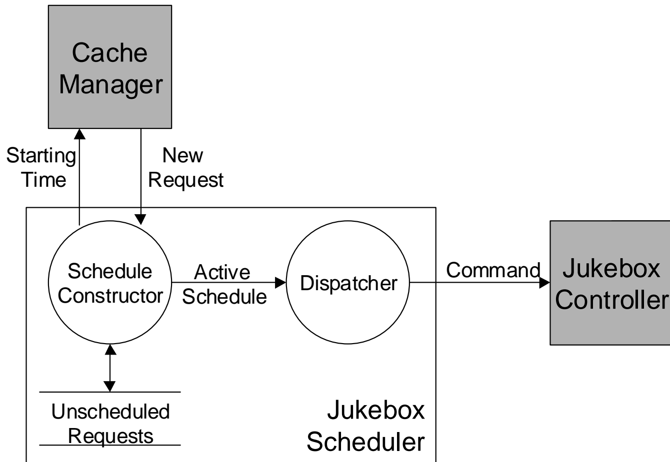

Fig. 3. Structure of the Jukebox Scheduler

jukebox controller so that media are moved and data is copied into secondary storage (see Fig 3). A new active schedule is generated only when we can guarantee that including the new request does not lead to missed deadlines. The new schedule then replaces the previous one.

If the incoming request has an ASAP deadline, then we assign to the request an early deadline, thus minimizing the *response time*. The jukebox scheduler reports the deadline it assigned to the request to the cache manager. Once the cache manager has the reply of all the involved jukebox schedulers, it combines them and returns the definite *starting time* to the user. At this moment, the up to now flexible deadline of the request becomes fixed. If instead the incoming request has a fixed deadline, the jukebox scheduler will report to the cache manager if the incoming request is schedulable or not. In this case a request will only be schedulable if all the jukebox schedulers could schedule the filtered requests they got.

Once the system confirmed the acceptance of the request to the user and returned the starting time, the system commits to keep this deadline. This implies that the user will be able to start consuming the data at that deadline and the flow of data will not be interrupted, guaranteeing a smooth access to continuous-data. The request and its reply is the contract between the user and the system. If the user tries to consume the data with a different pattern than the one specified in the request, by consuming data at a higher bandwidth for example, the system offers a best effort service for accessing the data.

We want to minimize the response time the requests with an ASAP deadline get and to minimize the number of rejected requests that have a fixed deadline. There is however a problem, because the scheduling problem to solve is NP-hard. We use an heuristic scheduling algorithm of polynomial complexity. We also want to give a reply as soon as possible about the schedulability of a request (or the starting time when the deadline is ASAP). We call this time the user has to wait for the reply of the system *scheduling time*.

III. Scheduling problem

We will now formalize the scheduling problem[3]. Let's say that at time t request r_k arrives at the jukebox scheduler. At time t the jukebox scheduler has a set of request units from previous requests. We will call this set $\mathcal{RU}$, formed by elements $ru_{ij}/i < k, j \leq l_i$.

In order to simplify the problem we assume that the user can start consuming the data of a request unit only once all the data of the request unit has been buffered. In this way for each request unit in a request we can compute a deadline, even for the request units for streaming data. This means that $d_{ij} = d_i + \Delta d_{ij}$.

The goal of the scheduling algorithm is to find a feasible schedule for the new set $\mathcal{RU}' = \mathcal{RU} \cup \{ru_{k1}, ru_{k2}, \ldots, ru_{kl_k}\}$. If r_k has got ASAP as deadline, we must also find the starting time $x \mid d_i = x$ makes $\mathcal{RU}'$ schedulable. In this case the resulting deadline of the request units is $d_{kj} = x + \Delta d_{kj}$.

A. Hardware characteristics

All tertiary-storage jukeboxes are composed of the following hardware: *drives* to access the data in the CDs, *slots* where the CDs are kept and *robotic arms* to move the CDs from the slots to the drives and viceversa. In big jukeboxes the number of slots is at least two order of magnitude bigger than the number of drives and the number of arms. Our DAX CD-ROM jukebox [3], for example, has 4 drives, 720 slots and 1 arm. The type of storage media in the jukeboxes can vary, being for example CD-ROM, DVD-ROM, magnetic tape, magneto-optical tape, etc.

The characteristic of the storage media in each case is that it is removable and that the contents of a file are stored sequentially in the medium. The latter greatly simplifies the estimation of the time needed to read a file (or part of a file). We only need two functions: $read(d, offset, size)$ and $seek(d, from, to)$ that return the time needed to read a block of data

[3] As the scheduling process that is carried out in each jukebox scheduler is independent from the scheduling in the other jukebox schedulers, we can ignore the existence of multiple jukebox schedulers.

and jump from one position in the CD to another[4]. CD-ROM readers for example are based on different technologies as CAV (constant angular velocity) and CLV (constant linear velocity). A drive using CAV is faster when reading data from the outer tracks of a CD-ROM than from the inner tracks, and thus a slower drive based on CLV technology may result faster when reading data from the inner tracks. In our model we assume that the particular storage medium where the data is stored does not influence the output of the functions *read* and *seek*. However some experiments we performed show that some CD-ROM readers show different performance when reading silver or gold CDs. We assume that all the CDs are from the same type, as is the case in our system.

The data on the CD can only be accessed when the storage media is loaded in a drive. The robotic arm needs time to load and unload drives. Once the CD is loaded in a drive, the drive must seek for the data to read, read and transfer the data.

The drives in a jukebox may also differ in the time it takes to open and close a drive, to spin-up and spin-down in the case of CD-ROM or DVD-ROM, or rewinding before unloading in the case of tapes. These characteristics of the drives influence the load and unload times. Another issue that influences the load and unload times is the distance to the drive of the slot where the CD is kept.

Based on the experience of Clockwise [2] we use a *model* of the hardware to predict the time the system will need for operations on robots, drives and media. We have validated the model against our actual hardware. We use this model both for constructing the schedules and as a simulator in our experiments.

To give a feeling of the scale of the times involved, we will mention here some values of the jukebox we are using. The jukebox has three 12X CD-ROM readers (1.75 MBps) and one 8X CD-ROM reader/writer (1.17 MBps). All the drives are based on CLV technology, with a constant bandwidth over all the tracks. The average seeking time of both models is 85 ms. In Table I shows the corresponding load and unload times. Though our jukebox is far from being state of the art, the relation between switching time and switching time is representative. We define that relation as

$$LostBW = AvgDriveBW \times AvgSwitchingTime$$

[4]For simplicity of the model, the functions take as parameter the drive involved, though the functions we use in our implementation really use the drive model.

TABLE I
LOAD AND UNLOAD TIMES OF A DAX JUKEBOX. TIMES ARE GIVEN IN SECONDS.

	Minimum	Maximum	Average
Load 12X	22.164	28.664	25.414
Load 8X	25.120	31.620	28.370
Unload 12X	17.566	24.066	20.816
Unload 8X	14.003	20.503	17.253

For our 12X readers the lost bandwidth with each switch is 80.9 MB and for a jukebox with 100 MBps drives and 1 sec switch time it is 100 MB.

IV. Outline of the scheduling algorithm

We believe that the best solution, in practical terms, is one in which all the data requested from one CD is read before unloading it, because with each switch effective bandwidth is lost. This is the approach we use in our algorithm. This solution requires using buffers in secondary storage, something that is needed as well to be able to take advantage of the bandwidth offered by the drives. In our model we assume that there are no restrictions in the amount of buffer space in secondary storage[5].

The optimal solution, however, may be one in which only some of the data requested from a CD is read and then it is unloaded. If the tasks to schedule are the elements of $\mathcal{RU}'$ it makes the scheduling problem intractable due to the number of posibilities to analyze. If instead the tasks to schedule are the CDs with request units, the problem is inherently *nonpreemtive* following the definition of a preemtive schedule of scheduling theory that states that a schedule is preemtive if each task may be preempted at any time and restarted later at no cost [1]. As we have exposed, switching CDs is clearly not free of cost.

We divide the original scheduling problem of finding a feasible schedule for $\mathcal{RU}'$ into smaller sub-problems. The first problem is how to schedule the reading of the data from a CD once the CD is loaded in a drive. The tasks to schedule are the request units for that CD. This problem is relatively easy to solve, though finding an optimal solution is an NP-hard problem. We call the schedule for each medium, *medium schedule (*MS*)*. The second problem is to make an assigment of the jukebox resources (drives and arms) to the stor-

[5]In the simulations we performed, 10% of the jukebox capacity in buffer+cache space (approximately 42 GB) proved to be more than enough.

age media so that the deadlines of the data in the CDs can be met. We call such a feasible assigment, *resources schedule (RS)*. The tasks to schedule in a RS are the CDs that have requests. This problem in NP-hard. If d_{r_k} = ASAP then we must also determine the starting time for r_k. The algorithm will only build nonpreemptive schedules because of the reasons already explained.

V. Evaluation

In this section we will show some results of the simulations we performed. We will mainly concentrate on showing the difference in performance between our scheduling policy LDL and EDF.

The implementation of our system is in Java. We performed the simulations using JDK1.3.1 under Linux. Each test we run consisted of a set of 1000 requests. All requests have ASAP as deadline. We generated the requests off-line and stored them, so that the tests were done with exactly the same set of requests. We generate the request sets in a random manner indicating the probability of requesting certain kind of data. The type of data that we handled are long videos (duration over 15 minutes), short videos, music and discrete-data (e.g. text, images). The requests can be for full albums, single files, parts of an album, multiple albums and a mix of files of different albums. The data requested is chosen using a Zipf distribution (i.e. 10% of the data is requested 90% of the times). This assumption seems to model correctly what has been observed in file systems, databases and video rental stores [4].

The hardware we are simulating is that of our DAX CD-ROM jukebox The space in secondary storage dedicated for cache and buffer space is 10% of the storage of the jukebox, approximately 42 GB. The cache remotion policy we used in all tests was *least recently used (LRU)*. Before starting the tests we preload the cache with the results of another simulated previous run. The requests are built independently than that of the current run, but with the same probability distribution.

We have simulated different loads of the system, by varying the interarrival interval of the requests. We generate the interarrival intervals randomly using a Poisson distribution. In order to see the difference in performance between the different scheduling policies, we performed simulations at a higher load than that expected in real operation.

In the following graphics we show the analysis for requests with a data type proportion of: 10% long videos, 40% short videos, 30% music and 20% discrete-data. We chose these data proportions because they allow in theory to make an efficient use of the jukebox resources. The bottleneck is the use of the arm. Under high loads conditions (78 req/hour) the arm is in use nearly 100% of the time. EDF proved unable to handle a load higher than 69 req/hour, while LDL could even handle 78 req/hour with quite a reasonable average response time of 70 sec.

The cache hit rate experienced was in all cases around 52%. To determine the cache hit rate, we measure the proportion of the data requested that is already in the cache or buffered.

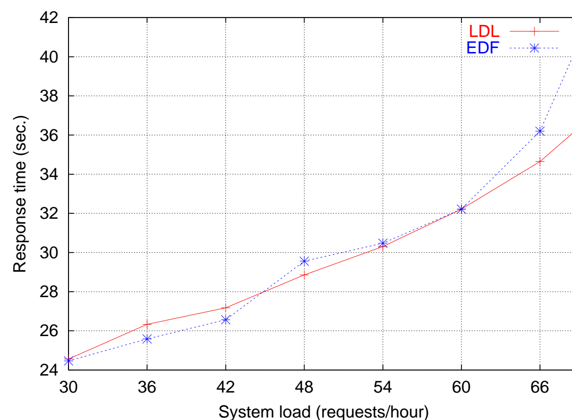

Fig. 4. Mean response time

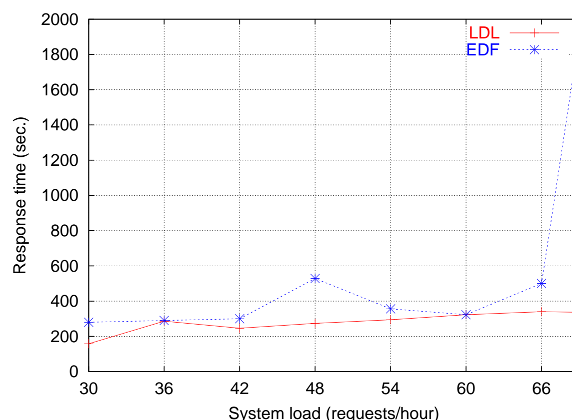

Fig. 5. Maximum response time

In Fig. 4 we show the mean response time of the system under different loads and in Fig. 5 the maximum response time. When the load is low, both policies offer similar mean response times, though LDL always has a better worse case (maximum response time). When the load increases, LDL clearly outperforms EDF. As we have already said, at even higher

system loads, EDF crashes, while LDL keeps offering service.

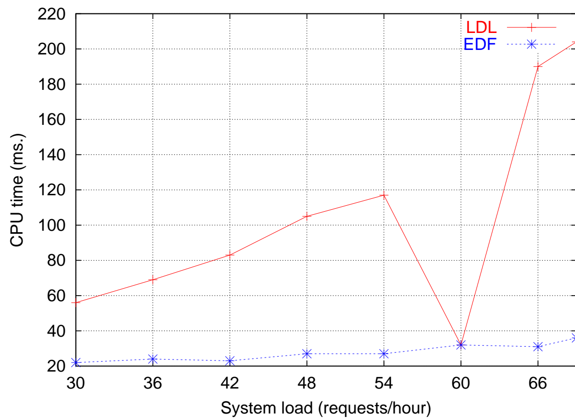

Fig. 6. Mean CPU time

The average CPU time needed by LDL is higher than that required by EDF. This is because the computations performed by LDL many times imply backtracking, while with EDF it is rarely so. In Fig 6 you can see the average CPU time used by the scheduler in each case.

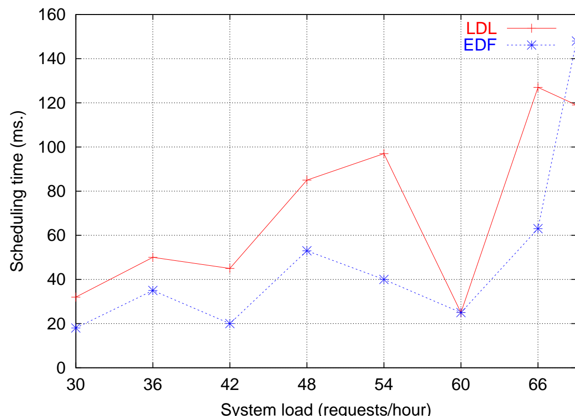

Fig. 7. Mean scheduling time

In Fig. 7 we show the mean scheduling time. EDF seems to give earlier replies to the user, though we do not have conclusive results on this issue yet. Something to be noticed, is that the average response time is around 100 ms.

VI. Future work

We are at present working on a new scheduling policy, *Mixed*, that makes use both of LDL and EDF, because there are cases in which the EDF scheduling policy produces a schedule that cannot be found with LDL. The new policy first tries to schedule the incoming request with LDL and if it does not succeed it tries with EDF. The preliminary results we obtained from experimenting with this policy, indicate that sometimes it performs better than LDL, but most of the times it performs worse.

We are also implementing an off-line optimal scheduler using constraint logic programming [7]. The idea is to be able to compare the performance of our scheduler against that of the optimal scheduler, even if that scheduler cannot be used on-line because of the computing time needed.

VII. Conclusions

We have presented the principles and evaluation of the jukebox scheduler for our hierarchichal real-time file system. The jukebox scheduler is so flexible that it allows the system to accept multimedia requests representing nearly every pattern.

We have showed that the scheduling problem is NP-hard, but we have an heuristic scheduling algorithm that solves the problem in polynomial time, finding good solutions. We are working at present to compare our solution against the optimal solution.

We have presented two scheduling policies LDL and EDF and we compared their performance. LDL shows a better performance in all the simulations we did, because it makes better use of the arm that is in general the bottleneck resource. Finally we have briefly discussed the new scheduling policy we are working in at present.

References

[1] Jacek Blazewicz, Klaus Ecker, Gnter Schmidt, and Jan Weglarz, editors. *Scheduling in Computer and Manufacturing Systems*. Springer Verlag, Berlin, second edition, 1994.

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

[3] Chess Engineering bv. *DAX Software Architecture Manual, Version 0.5*, March 1998.

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

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

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

[7] P. van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programmig Series. MIT Press, Cambridge, MA, 1989.