Real-Time Scheduling of Tertiary Storage

María Eva M. Lijding

University of Twente
P.O. Box 217 - 7500 AE Enschede - The Netherlands
telephone +31-53-4893690 / fax +31-53-4892927

Composition of the Graduation Committee:

Prof. Dr. S.J. Mullender, University of Twente (promoter)
Dr. D.L. Pressotto, Lucent Technologies, USA
Prof. Dr. S. Luitjens, Philips Nat.Lab.
Prof. Dr. G.J. Woeginger, University of Twente
Ir. P.G. Jansen, University of Twente
Prof. Dr. S.L. van de Velde, Erasmus University Rotterdam
Prof. Dr. C. Hoede, University of Twente (chairman)

The research reported in this thesis was partly financed by the NWO (the Netherlands Organization for Scientific Research) through the Advanced Multimedia Indexing and Searching project (AMIS), under grant 612-21-201.

IPA Dissertations Series No. 2003-07
The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

CTIT Ph.D.–thesis Series No. 03-48
Centre for Telematics and Information Technology (CTIT)
P.O. Box 271, 7500 AE Enschede, The Netherlands

This thesis was typeset with LaTeX in Times and Courier. Figures are drawn with PowerPoint and exported to EPS and PDF. *Powerpoint* is a registered trademark of Microsoft. EPS and PDF are registered trademarks of Adobe. The cover was designed by Hartmut Benz.

Printed by Ipskamp PrintPartners, Enschede, The Netherlands.
ISBN 90-365-1890-3
ISSN 1381-3617 (CTIT Ph.D–thesis Series No. 03-48)

REAL-TIME SCHEDULING OF TERTIARY STORAGE

DISSERTATION

to obtain
the doctor's degree at the University of Twente,
on the authority of the rector magnificus,
prof.dr. F.A. van Vught,
on account of the decision of the graduation committee,
to be publicly defended
on Thursday, May 1, 2003 at 16:45

by

María Eva Magdalena Lijding

born on May 14, 1970
in Buenos Aires, Argentina

This dissertation is approved by the promoter Prof. Dr. S.J. Mullender.

Acknowledgements

Although my experience with my own Dutch family had shown since early in life that not all Dutch people fit the image that is created about them in some other latitudes of the world—of being cold and dull as their weather—I still feared my family could be an exception, resulting from having stayed in warmer places than the Netherlands for extended periods of time. Luckily, my fears were wrong. It is hard to imagine people that enjoy life so much and are willing to joke and laugh about nearly everything as Dutch people do. My colleagues at the University of Twente received me in a warm and friendly fashion and showed incredible patience with my initially feeble attempts to hold full conversations in Dutch. Furthermore, the Dutch openness and readiness to discuss all type of topics enriched me with very interesting conversations during coffee time and lunch. I thank all my colleagues for their support, friendliness and their contribution to improve my Dutch.

I am very grateful to my promoter Sape Mullender, who trusted me to do a PhD and gave me the freedom to decide the direction of the research. He has a superb way of providing advice for researching and writing in the form of simple guidelines that are easy to follow. Most of all, I want to thank him for his friendliness and openness since I first got in contact with him five years ago.

Pierre Jansen became my unofficial daily supervisor after some informal discussions about our research interests. Since then he has been a great discussion partner. He also proposed the periodic scheduler to compare against Promote-IT. Additionally, I immensely enjoyed our conversations about history and world politics, interests that we both share.

The practical aspects should have been very hard to deal with without the help and support of Ferdy Hanssen. With great patience he maintained my system, installed software, helped me with LaTeX, and helped me fight some battles against a misbehaving jukebox. Ferdy programmed the first version of the periodic scheduler. He also greatly improved the Dutch version of the abstract. Moreover, he has been a great colleague, reliable and always ready to lend a helping hand.

To my good fortune, Sandro Etalle found the scheduling problem of the jukebox challenging and programmed the optimal scheduler, which I use in this thesis as a benchmark for the heuristic schedulers. I thank Pieter Hartel for his advice on how

to present the formalization of the scheduling problem and the solution. He also gave me useful comments regarding the presentations of the results of my research.

Gerhard Woeginger helped me feel at ease in the world of scheduling theory and become more confident in my own research. He also suggested me to look into the vehicle routing problem.

Thijs Krol, Hans Scholten, and Albert Schoute were very good discussion partners for some details regarding the hardware model. Additionally, through the interesting conversations during lunch they helped me understand better the Dutch culture. I thank Lodewijk Smit for our conversations on how to write a thesis. Thanks also to my former colleagues, Ties Bos and Peter Bosch.

I am profoundly grateful to Hartmut Benz for his contribution to this work, my happiness, and well-being. Our discussions on software design, design patterns and programming in Java helped me build and improve my software. He also performed a careful proofreading of the dissertation that resulted in many important fixes and helped me prepare the camera-ready version. Last, but not least I thank him for being a wonderful husband, listening to my 'loud-thinking', and comforting me and supporting me when the workload and the task ahead seemed too heavy.

I thank Dave Presotto for helping me to improve the grammar of the dissertation. He kindly offered to read the text again and point out ugly and incorrect sentences. The reader will probably regret that due to real-time deadlines in the printing process I could not profit more of his knowledge as native English speaker.

Many people encouraged me to pursue a PhD. I especially thank Claudio Righetti and Irene Loiseau, who encouraged me early in 1993 to apply for a research grant for students from the University of Buenos Aires, and since then have supported me in my career. They also encouraged me to ask for the post-graduate grant from the Fundación YPF that financed my first year in Europe. Thanks to Leandro Navarro who invited me to work with him in the Polytechnic University of Catalonia. He was not only a good mentor, but also a supportive friend. I thank as well the Fundación YPF, especially Alejandra Tomassini for her support and care.

Thanks as well to all the people that made my life easy and pleasant in the northern hemisphere. To my family in Alkmaar and England who gave me a warm welcome to this part of the world. To Marianne Benz for her support, listening and care. To Elisa Moreno, who gave me the great gift of her friendship. To the friends here, in Germany and Barcelona, who made me feel at home very soon.

Finally, I thank my parents and my brother who constantly support me and encourage me from the distance. They know that even if a long geographical distance separates us now, they are always near in my heart. I especially appreciate that they could understand that I had to follow my heart and pursue a PhD here. They have shown a big capacity for sharing my happiness and sorrows from a distance. I thank also the friends and family in South America for their support and encouragement.

Abstract

Today multimedia data is generally stored in secondary storage (hard disks) and is from there delivered to the users. However, the amount of storage capacity needed for a multimedia archive is large and constantly growing with the expectations of the users. Tertiary-storage jukeboxes can provide the required storage capacity in an attractive way if the data can be accessed with real-time guarantees.

A jukebox is a large tertiary storage device that can access data from a large number of removable storage media (RSM, for example DVDs or tapes) using a small number of drives and one or more robots to move RSM between their shelves and the drives. A central problem with this setup is that the RSM switching times are high, in the order of tens of seconds. Thus, multiplexing between files may be many orders of magnitudes slower than on a hard drive, where it takes only a few milliseconds. The second important problem is the potential for resource-contention that results from the shared resources in the jukebox.

Our *hierarchical multimedia archive (HMA)* is a service that provides flexible real-time access to data stored in tertiary storage. The HMA can serve complex requests for the real-time delivery of any combination of media files it stores. Such requests can for instance result from a database query to compile a historical background for news on-the-fly, or from a personalized entertainment program consisting of music video clips.

The HMA uses secondary storage as a buffer and cache for the data in its tertiary-storage jukeboxes. The *jukebox scheduler* is the key component of the HMA that guarantees the in-time promotion of data from tertiary storage to secondary storage. Apart from providing real-time guarantees, the scheduler also tries to minimize the number of rejected requests, minimize the response time for ASAP requests, minimize the confirmation time, and optimize hardware utilization.

The first step in order to build an efficient scheduler is to understand the scheduling problem thoroughly. On the one hand, we *model the hardware* and identify the parameters that define the hardware behaviour. Our model is flexible and can represent any present and expected future jukebox hardware. On the other hand, we *formalize the scheduling problem* using scheduling theory so that its characteristics and complexity can be analyzed, and the problem can be classified and compared

with other scheduling problems. Given the complexity of the scheduling problem we are dealing with, there are many different ways in which it can be modelled. We present a hierarchy of models and analyze their advantages and disadvantages.

The most important of the scheduling-problem models is the *minimum switching model,* which models the problem as a *flexible flow shop* with three stages—load, read, unload. The model requires that once an RSM is loaded in a drive, all the requested data of the RSM is read before the RSM is unloaded. Thus, the schedules that can be built with this model have a minimum number of switches, which in turn results in good resource utilization.

Promote-IT is an efficient heuristic scheduler based on the *minimum switching model*. It can deal with a wide variety of requests and jukebox hardware. Promote-IT provides short response and confirmation times, and makes good use of the jukebox resources. It separates the *scheduling* and *dispatching* functionality and effectively uses this separation to dispatch tasks earlier than scheduled, provided that the resource constraints are respected and no task misses its deadline. Promote-IT can use different scheduling strategies that vary in the way schedules are built.

To prove the efficiency of Promote-IT we implemented alternative schedulers based on different scheduling models and scheduling paradigms. The evaluation shows that Promote-IT performs better than the other heuristic schedulers. Additionally, Promote-IT provides response-times near the optimum in cases where the optimal scheduler can be computed. We developed a toolbox called *JukeTools* to easily implement, evaluate and compare jukebox schedulers.

Samenvatting

Tegenwoordig wordt multimedia data meestal in secondaire opslag (harde schijven) bewaard en vandaar aan de gebruikers geleverd. Maar de opslagcapaciteit die nodig is voor een multimedia archief is groot en groeit voortdurend mee met de verwachtingen van de gebruikers. Jukeboxen voor tertiaire opslag kunnen op een aantrekkelijke manier de benodigde capaciteit aanbieden, indien toegang tot de data met real-time garanties mogelijk is.

Een jukebox is een systeem voor tertiaire opslag, dat toegang heeft tot data, opgeslagen op een groot aantal uitwisselbare media (RSM, bijvoorbeeld DVDs of tapes) door middel van een klein aantal drives en een of meer robots die RSM tansporteren van de bewaarposities naar de drives en vice versa. Een centraal probleem met deze configuratie is dat de wisseltijd van de RSM hoog is, in de orde van tientallen seconden. Dus, het wisselen tussen bestanden kan vele malen langzamer zijn dan wisselen tussen bestanden op een harde schijf. Een tweede belangrijk probleem is het potentieel voor *resource-contention*, veroorzaakt door het feit dat de jukeboxcomponenten gemeenschappelijk zijn.

Ons *hiërarchisch multimedia archief (HMA)* is een service die flexibel real-time toegang tot tertiaire opslag biedt. Het HMA kan complexe verzoeken voor de real-time aflevering van elke willekeurige combinatie van in het archief opgeslagen data honoreren. Een dergelijk verzoek kan bijvoorbeeld van een database-query komen om historische informatie voor een nieuwsprogramma op dat moment samen te stellen, of van een persoonlijk muziekprogramma, waar videoclips gecombineerd kunnen worden.

Het HMA gebruikt secundaire opslag als buffer en cache voor zijn jukeboxen. De *jukebox scheduler* is de belangrijkste component van het HMA, die garandeert dat data op tijd van tertiaire naar secundaire opslag wordt gepromoveerd. De scheduler geeft niet alleen real-time garanties, maar probeert ook zowel het aantal afgekeurde verzoeken, als ook de respons- en de confirmatietijd te minimaliseren en de hardware utilisatie te optimaliseren.

De eerste stap voor het bouwen van een efficiënte scheduler is het doorgronden van het probleem. Daartoe *modelleren* wij *de hardware* op een flexibele manier zodat elk type huidige en toekomstige jukeboxen kan worden gerepresenterd. Ver-

der *formaliseren* wij *het schedulingprobleem* met schedulingtheorie om de eigenschappen en complexiteit te analyseren en om het probleem te classificeren en te vergelijken met andere schedulingproblemen. Er zijn er meerdere manieren om het probleem te modelleren. Wij presenteren een hiërarchie van modellen en analyseren hun voor- en nadelen.

Het belangrijkste model is het *minimum switching model*. Het modelleert het probleem als een *flexible flow shop* met drie stages (laden, lezen, ontladen). Wanneer een RSM in een drive wordt geladen, wordt alle benodigde data van de RSM gelezen voordat de RSM wordt terug gegeven. Dus de op dit model gebaseerde roosters hebben een minimum aantal wisselingen, zodat de hardware efficiënt wordt gebruikt.

Promote-IT is een efficiënte, op heuristiek gebaseerde scheduler afgeleid van het *minimum switching model*. Promote-IT kan elk type verzoek en jukeboxhardware aan. Het biedt korte respons- en confirmatietijden aan de gebruiker en gebruikt jukeboxcomponenten efficiënt. Promote-IT verdeelt de verroostering- en dispatchingfunctionaliteit en gebruikt deze verdeling op effectieve wijze om taken vroeger dan gepland af te handelen, zolang de beperkingen op het componentengebruik worden gerespecteerd en geen deadlines van taken worden overschreden. Promote-IT kan verschillende strategieën gebruiken, die variëren op de manier waarop de taken in het rooster zijn toegevoegd.

Om de efficiëntie van Promote-IT te tonen, hebben wij alternatieve schedulers geïmplementeerd, die op verschillende schedulingmodellen en paradigma's zijn gebaseerd. De evaluatie laat zien dat Promote-IT beter presteert dan de andere, op heuristiek gebaseerde schedulers en dat de responstijd dicht bij de optimale oplossing ligt in de gevallen waar een dergelijke oplossing berekend kan worden. Wij hebben een toolbox gebouwd, *JukeTools* genaamd, om gemakkelijk en snel jukeboxschedulers te kunnen implementeren, evalueren en vergelijken.

Contents

Chapter 1

Introduction

Parkinson's law of data states that:

> 'Data expands to fill the space available for storage.' Buying more memory encourages the use of more memory-intensive techniques. It has been observed since the mid-1980s that the memory usage of evolving systems tends to double roughly once every 18 months. Fortunately, the memory density available for the same price also tends to double once every 18 months (Moore's Law).

This law is applicable to all kind of storage, not only RAM. Therefore, the need for large storage devices, such as jukeboxes, has not decreased with the advent of bigger and cheaper hard disks and RAM. There are three main forces driving the need for larger storage capacity.

First, there is an ever increasing amount of data that needs to be stored. This data is the product of scientific measurements and simulations, videos, music, images and all imaginable type of multimedia data. Conservative estimates state that there are more than 6 million hours of video stored worldwide and this number grows at a rate of about 10% per year [71]. The output of scientific measurements and simulations amounts to many terabytes of data daily [11].

Second, our expectations about the data quality also grow. We want to have more audio channels, more colours, better definition, wider screens, etc. We can take as an example the move in video quality going from VHS to DVD to HDTV (high definition TV).

Third, the number of computer and Internet users grows world-wide. According to statistics, the number of on-line users worldwide is 605 million at the end of 2002 [76]. As the number of users grows, so does the amount of data that a system has to store. On the one hand, if the system provides storage capacity to the users, there is a quite direct relation between the number of users and the storage capacity needed. On the other hand, if the goal the system is to provide contents, e.g., Video-on-Demand or on-line news, having larger and more complete archives attracts a larger and more varied user population.

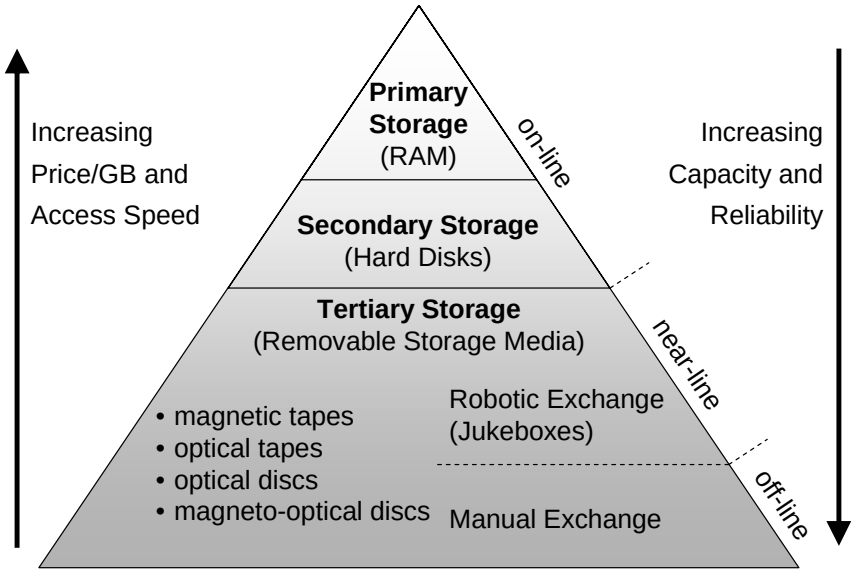


Figure 1.1: Storage hierarchy.

Additionally, the users' expectations toward the availability of the data also grow. First, the data must be accessible to a large number of people, possibly distributed world-wide. Second, the methods to access the data must permit the combination of data in a flexible way, independently of its specific storage location and type, e.g., to present the result of a database search, or to compose a multimedia presentation. Third, the data should be promptly available, in order to provide on-line services.

Therefore, there is a strong motivation to use *hierarchical storage* in order to store large amounts of data. Additionally, the data stored in each level of the hierarchy must be available in a timely manner and be accessible in a flexible way.

1.1 Hierarchical Storage

Figure 1.1 shows the storage hierarchy. The storage capacity and reliability of the storage increases toward the bottom of the pyramid. The access speed and cost per gigabyte increases toward the top of the pyramid.

Following Miller [72] we classify data storage devices according to the ease with which storage media can be switched. At the top of the hierarchy we have RAM as primary storage, which can be further divided into CPU registers, cache memory and main memory. The secondary storage level contains devices that require one device per storage medium, such as magnetic disks and solid state disks.

Tertiary storage devices allow the easy removal of media from the drives. Therefore, one device can read any number of *removable storage media* (RSM [1]). Examples of RSM are optical disks (e.g., CD-ROM, DVD-ROM, DVR[2]), magneto-optical disks, magnetic tapes, and optical tapes.

Tertiary storage can be further classified by the automation level to switch the media. *Jukeboxes* or *robotic storage libraries* have one or more *robots* to move the RSM between the *drives* and the *shelves* where they are stored. The RSM can also be kept in shelves and require *human intervention* to be loaded in a drive.

The access speed and the capacity to access the data in a random manner is higher at the top of the hierarchy. The time to switch RSM in a jukebox is 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. In turn, using a jukebox is much faster than having humans do the switching.

Therefore, another common classification of the storage levels is by the availability of data—*on-line*, *near-line*, *off-line*. The time to access the data is a function of this classification. The top two levels provide on-line storage, while jukeboxes provide near-line storage. When human intervention is required, we talk about off-line storage.

The availability of data comes at a price. The price to store a gigabyte of data in main memory is higher than doing the same in hard disks, and still higher than storing it in a jukebox. A factor frequently overlooked when determining the price of storing data on hard disks is that hard disks require valuable resources (e.g., controllers, power) to keep the data on-line. When using jukeboxes, there are only a limited number of drives that are shared to access two or three orders of magnitude more RSM.

The storage capacity of the devices increases as we descend in the storage hierarchy. At present, a jukebox can store few terabytes of data and there is virtually no limit to how much off-line data may be stored in warehouses. Also, the reliability of the storage increases as we descend in the hierarchy, which is especially interesting for long-term storage of data, as backups and archives.

As happens very frequently when developing services and systems, there are conflicting requirements to meet. On the one hand, we want large and cheap storage, while on the other hand, we want to access it promptly and in a flexible manner.

Jukeboxes provide cost-effective near-line storage for large amounts of data. They are specially suited to store bulky data—video, audio, large databases, and backups—on a permanent basis. However, jukeboxes are not random-access de-

1 The acronym RSM stands both for the singular and the plural.

2 DVR stands for Digital Video Recorder. This type of disk is also called ‘Blue-ray disc’

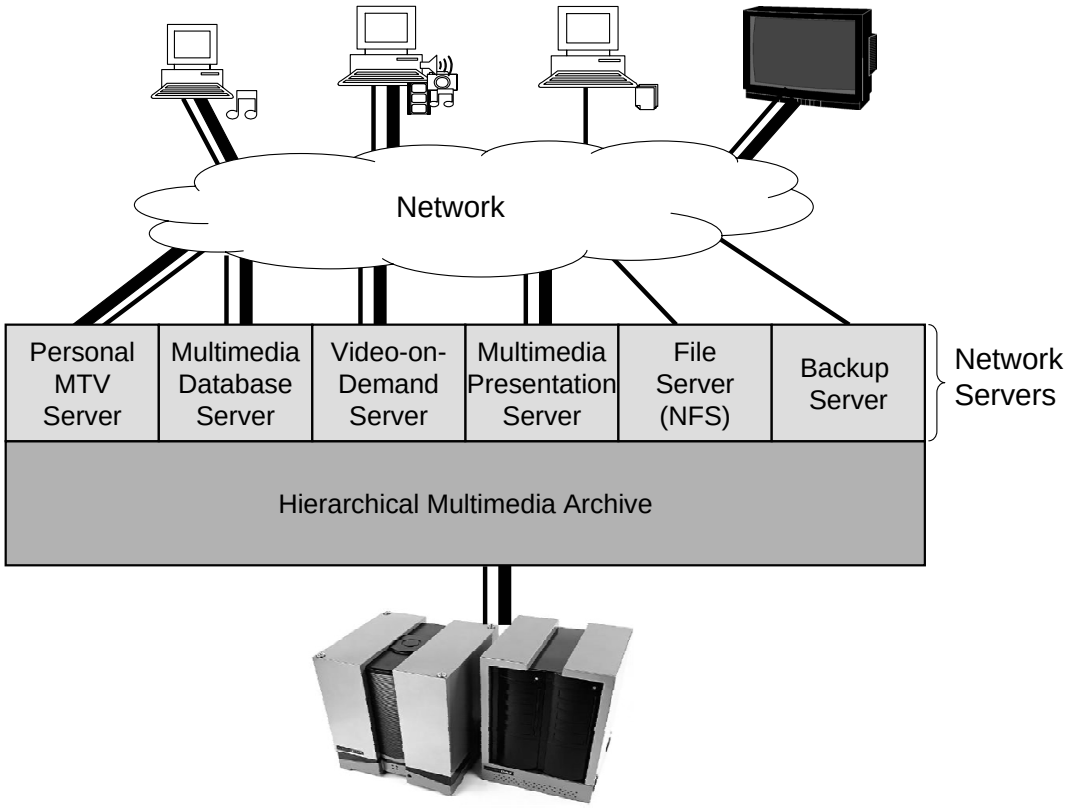


Figure 1.2: System overview with various application specific network servers.

vices and the resources in the jukebox—robots, drives and RSM—are shared and require exclusive use. This creates the potential for resource-contention problems. Therefore, in order to use a jukebox effectively it is important to schedule the jukebox resources.

In this dissertation we present a good compromise between the restrictions of the jukebox and the requirements of the users. We extend the use of jukeboxes to become a vital part of a real-time file system. We provide access to data stored in jukeboxes in a timely manner, guaranteeing real-time deadlines. We define a protocol that permits to access the data in tertiary storage in a flexible way as long as the user defines the data to access and the pattern to access the data in advance.

1.2 Hierarchical Multimedia Archive

Long gone are the days when computers were seen exclusively as working tools. Computers have found their way into our living rooms as the new generation en-

tertainment centre. Time-shifted TV, Video-on-Demand (VoD), and systems that automatically record TV-programs the user may like are some examples of this trend for digital video contents in the home. Also frequent activities are listening to music streamed through the Internet by traditional radio broadcasters and (semi)personalized radios that try to adapt to the musical taste of the listener.

In order to build a multimedia archive it is important to make the combination of different files and media in arbitrary patterns easy. For media providers it is important to have fast access to large multimedia archives and databases, and be able to efficiently combine the data on-the-fly—for example to produce documentaries or provide historical background for news. Consumers, for example, can easily create personalized music programs from huge archives of music and video clips.

A *hierarchical multimedia archive (HMA)* is a service that provides flexible access to hierarchical storage. The HMA 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 and maybe separately stored media files into a continuous presentation. The HMA can also be used for the more simple case of a Video-on-Demand application, where the requests are generally for a single media file—a movie—to be played from beginning to end.

The hierarchical multimedia archive acts as a real-time file system [33] 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 1.2.

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.

In this dissertation we are concerned with the *in-time promotion* of multimedia data from tertiary storage to secondary storage. In turn, Clockwise [14] can provide real-time access to data stored in secondary storage, which is used in HMA as cache. *RT-net* [41] can provide real-time guarantees for the use of a local area network.

The HMA uses secondary storage as a buffer and cache for the jukebox. On the one 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. On the other hand, the popularity distribution of the data is generally very skewed—following a Zipf-like distribution. Thus, some data will be requested very often and should be kept in secondary storage to avoid the repeated long retrieval from tertiary storage.

A *request* to the HMA can consist of multiple streams and non-streamed data that are synchronized sequentially or concurrently in arbitrary patterns. The request

defines the timeline with which the data must be available. Additionally, the users can define a deadline for the request that indicates the latest time by which the data must be available and indicate if the data should be available *as soon as possible (ASAP)*.

If 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* at which the user can start consuming the data 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. Under high load, the HMA may reject requests. The goal of the HMA is to provide access to the data as fast as possible (short *response time*) and to confirm the requests promptly (short *confirmation time*).

In a commercial scenario for exploiting an HMA, the user may get a price reduction on the payment if the response time is longer than a certain threshold—probably as a function of the response time. This is a good economic incentive for the provider to minimize the response time of ASAP requests. The system could also privilege those clients that make requests in advance, for example making a request for a movie one hour earlier than the required starting time. The more users, the more income. Therefore, it is profitable to maximize the number of simultaneous users and minimize the number of rejected requests. Furthermore, providing a fast response time and having a low rejection ratio, is a good parameter of the quality of the systems.

The VoD scenario illustrates the goal to minimize the confirmation time. After the user makes a request, the system should give a confirmation promptly, so that the user knows if the request was accepted, and in the case of an ASAP request, also the time when the video will be available. In this way, the user can plan what to do until the movie begins. Another possible scenario is that the application providing the VoD server uses the information contained in the confirmation to fill up the time until the movie begins with a short video-clip or advertisement. By knowing in advance the response time, the application can choose one or more videos that accurately fill in the gap.

1.3 Jukebox Scheduling

The jukebox scheduler is the key component of the HMA that provides the desired quality of service. The scheduling problem to solve presents an interesting challenge due to its complexity and conflicting goals.

The main goal of the jukebox scheduler is to guarantee that the data is promoted to 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 re-

jected requests, minimize the response time for ASAP requests, and minimize the confirmation time.

In this dissertation we use a new design of jukebox schedulers, where the *scheduling* and *dispatching* functionality are clearly separated. This separation allows us to improve the performance of the system, because the optimality criteria of both functions are different. The goal of the *schedule builder* is to find feasible schedules for the requested data. Thus, the scheduler tries to build schedules as flexible 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. We introduce the concept of *early dispatching*, by which a dispatcher can dispatch the tasks earlier than scheduled as long as the resource constraints are respected and no task misses its deadline.

In order to build efficient schedulers it is necessary to understand the scheduling problem thoroughly. The first step is to *model the hardware* and identify the parameters that define the hardware behaviour. The hardware model must be flexible to cover the existing and future hardware. It is used to predict and simulate the jukebox behaviour. The model plays a crucial role when constructing the jukebox schedules.

The second step is to *formally model the scheduling problem*. Given the complexity of the scheduling problem there are many different ways in which it can be modelled. However, each model puts restrictions on the original scheduling problem to make the problem manageable. We present a *hierarchy of scheduling-problem models* and analyze the advantages and disadvantages of each model in the hierarchy. We also provide schedulers that implement the models in the hierarchy. Even if the models in the hierarchy are simplifications of the original scheduling problem, they are also $\mathcal{NP}$-hard. Therefore, the algorithms used by the on-line schedulers are heuristic algorithms, which can compute schedules in polynomial time.

The most important of these models is the *minimum switching model*, which models the problem as a *flexible flow shop* with three stages.[3] The model uses shared resources to guarantee mutual exclusion in the use of the jukebox resources. This model puts only a small restriction on the utilization of the resources, which additionally results in better use of the resources and system performance. The model requires that once an RSM is loaded in a drive, all the requested data of the RSM is read before the RSM is unloaded. Thus, the schedules that can be built with this model have a minimum number of switches.

Promote-IT (Promote In Time) is the scheduler that we propose in this dissertation as the scheduler to use in an HMA. Promote-IT is based on the minimum switching model. For every incoming request it builds a new schedule that includes all the previously scheduled request units plus the request units of the new request. It uses

3 The scheduling concepts and terminology are explained in the next chapter.

an efficient heuristic algorithm to find a solution to an instance of the minimum switching model on-line. Promote-IT can deal with any type of request and jukebox hardware. Additionally, it provides short response times and confirmation times, and makes good use of the jukebox resources.

We defined different scheduling strategies for Promote-IT, which vary in the way in which the jobs are added to the schedule. These strategies can be classified as *Front-to-Back* and *Back-to-Front*. When using Front-to-Back each job is scheduled as early as possible, while with Back-to-Front each job is scheduled as late as possible. When using Back-to-Front, Promote-IT profits strongly from the separation of scheduling and dispatching. The scheduler creates schedules with idle times that are used by the dispatcher to dispatch tasks early. This combination proves useful in many cases, especially when the use of a shared robot is the bottleneck in the system.

To prove the efficiency of Promote-IT, we implemented alternative schedulers based on different scheduling models and scheduling paradigms. On the one hand, we designed two new schedulers: the *jukebox early quantum scheduler (JEQS)* and the *optimal scheduler*. On the other hand, we extended some heuristic schedulers proposed in the literature: the *extended aggressive strategy*, the *extended conservative strategy* and *Fully-Staged-Before-Starting (FSBS)*.

The *jukebox early quantum scheduler (JEQS)* is a periodic scheduler. The basic heuristic used by a periodic scheduler is to represent the requests as periodic tasks. A restriction of periodic schedulers is that they can be used only for some special use cases of HMA, as Video-on-Demand, because they are unable to deal with complex requests. Additionally, periodic schedulers have serious problems in avoiding resource-contention problems. JEQS solves these problems by using the robots and drives in a cyclic way. The robot exchanges the contents of each drive at regular, fixed intervals. This results in a cyclic use of the drives, which are dedicated to reading data of an RSM while the other drives are being served by the robot. Although, JEQS is generally able to start incoming requests in the next cycle of a drive, its performance is much worse than that of Promote-IT. We show, however, that this poor performance is characteristic of any periodic jukebox scheduler.

The *optimal scheduler* is a scheduler that computes the minimum response time for each incoming request. The objective of this scheduler is to be used as a baseline for evaluating the quality of the heuristic schedulers. The optimal scheduler cannot be used in a real environment due to its computing-time requirements. The computing time increases exponentially with the complexity of the requests and the system load. Therefore, we can only use it for evaluation of small test sets and relatively low system load. The comparisons we performed show that the performance of Promote-IT is near the optimum, at least under these special testing conditions.

We developed a toolbox called *JukeTools* to easily implement, evaluate and compare jukebox schedulers. We implemented and evaluated the aforementioned heuristic schedulers using JukeTools. The evaluation shows that Promote-IT performs better than the other heuristic schedulers.

1.4 Summary

Summarizing, the results of the dissertation are

- The hierarchical multimedia archive, which is a new application for tertiary-storage jukebox that provides real-time access to any combination type of media stored in the jukebox
- A new design of jukebox schedulers, where the scheduling and dispatching functionality are clearly separated
- A hardware model that can represent virtually any type of jukebox
- A thorough study of the scheduling problem and the resulting hierarchy of scheduling-problem models
- Promote-IT, an efficient heuristic scheduler based on the minimum switching model, which can deal with any type of jukebox and request
- Alternative schedulers to evaluate the performance of Promote-IT: JEQS, an optimal scheduler, extended versions of the aggressive and conservative strategies of Lau et al., and a Fully-Staged-Before-Starting scheduler
- JukeTools, a toolbox for implementing and evaluating jukebox schedulers
- A performance comparison of the different schedulers, which shows that Promote-IT performs better than the other heuristic schedulers, and additionally provides response-times near the optimum in cases where the optimal scheduler can be evaluated

1.5 Outline of the Dissertation

Chapter 2 presents related work. It reviews other jukebox schedulers and discusses their strengths and weaknesses. It also gives an overview of scheduling in related

environments and shows common features with our scheduling problem and solution. This chapter also gives a short overview of scheduling theory, both aperiodic and periodic, and describes the notation used in the rest of the dissertation.

Chapter 3 describes the HMA in detail and discusses some usage scenarios. It formally defines the user requests. It describes the architecture of the HMA and describes the separation between schedule building and dispatching. The system architecture serves also as road map for the following chapters.

Chapter 4 presents the hardware model. It gives an overview of jukebox technology, specially focusing on optical jukeboxes. It provides a comprehensive model that is used to build the schedules and to simulate the jukebox hardware. It presents a short validation of the hardware model for the hardware available in our laboratory. This chapter also formalizes the functionality of the jukebox controller.[4]

Chapter 5 gives a thorough analysis of the scheduling problem and presents a variety of scheduling-problem models. These models include the models underlying the most relevant schedulers discussed in Chapter 2. It analyzes the strength and weakness of each model and provides a justification for choosing the minimum switching model as the model for Promote-IT.

Chapter 6 describes in detail the scheduling algorithm used in Promote-IT. It presents the different scheduling strategies of Promote-IT: earliest deadline first, earliest starting time first, latest deadline last and latest starting time last. It also discusses the early dispatcher that makes the Back-to-Front strategies so interesting.

Chapter 7 presents the other two new jukebox schedulers: the jukebox early quantum scheduler (JEQS) and the optimal scheduler. It also presents the extensions to some existing jukebox schedulers, which we use to evaluate the performance of Promote-IT.

Chapter 8 describes the implementation and simulation environment of the jukebox schedulers. It presents JukeTools, the toolbox used for implementing and comparing the jukebox schedulers.

Chapter 9 provides a comparison of the different jukebox schedulers. It shows that Promote-IT is able to provide shorter response times than the other heuristic schedulers and that the response times of Promote-IT are near those of the optimal scheduler.

Finally, Chapter 10 presents some conclusions that can be drawn from this research and briefly discusses possible directions for future research.

4 Chapter 4 is a relatively stand alone chapter. It can easily be used as a reference to jukebox hardware, independently of the scheduling problem presented here. The details about the hardware technology can be skipped without missing the important concepts of the scheduling problem.

Chapter 2

Background and Related Work

This chapter provides a framework for our work by introducing the main concepts of scheduling theory, both aperiodic and periodic, and discussing related work. It discusses systems that schedule jukeboxes in real-time, and shows that so far only aperiodic schedulers have succeeded in effectively providing real-time guarantees. The periodic schedulers suffer from resource-contention problems. Thus, they cannot guarantee that the deadlines are met when a contention problem is encountered during the execution of a schedule.

This chapter presents jukebox schedulers with no real-time goals. The research on this topic shows the importance of reading all the requested data from an RSM at once. These results support the approach we use in Promote-IT. Also, the scheduling of a single medium is discussed inasmuch as the techniques can be used for reading the data once an RSM is loaded in a drive.

We also present research in scheduling automated storage/retrieval systems, because these systems are similar to tertiary storage jukeboxes. Therefore, our results can be applied in the production environment. Finally, we discuss scheduling in logistic applications that have some important features in common with our scheduling problem.

2.1 Scheduling Theory

This section shortly introduces scheduling theory and defines the vocabulary and notation used in this dissertation. Scheduling theory can be classified into two main streams: aperiodic scheduling and periodic scheduling. Aperiodic scheduling is in general referred to simply as ‘scheduling theory’ and is mainly used in manufacturing and logistics, while periodic scheduling is strongly related to real-time computer systems.

The notation of both types of scheduling sometimes conflicts. In this dissertation we are mainly concerned with aperiodic scheduling and use the corresponding notation. We will clearly specify the use of periodic-scheduling notation.

2.1.1 Aperiodic Scheduling

Scheduling problems are characterized by three sets: set $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ of n *tasks*, set $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$ of m *processors (machines)*, and set $\mathcal{R} = \{R_1, R_2, \ldots, R_s\}$ of s types of *additional resources*.

Scheduling is to assign processors from $\mathcal{P}$ and resources from $\mathcal{R}$ to tasks from $\mathcal{T}$ in order to complete all tasks under the imposed constraints. A *schedule* is such an assignment as a function of time. Each task is processed by at most one processor at a time and each processor is capable of processing at most one task at a time.

A schedule is called *preemptive* if each task may be preempted at any time and restarted later with no additional cost. If preemption is not allowed, the schedule is called *non-preemptive*

A *scheduling problem* Π is defined by a set of parameters grouped in a triplet $\alpha \mid \beta \mid \gamma$ [37, 13]. The α field describes the processor environment, the β field describes task and resource characteristics and the γ field denotes the optimality criterion. For example, $1 \mid r_j\ \tilde{d}_j \mid -$ represents the scheduling problem of finding a feasible schedule for tasks with deadlines and arbitrary release times in a uniprocessor environment. We describe the possible values of the parameters in the rest of the section.

An *instance I* of problem Π is obtained by specifying particular values for all the problem parameters.

A *scheduling algorithm* is an algorithm which constructs a schedule for a given problem Π.

Processor Environment

The processors are characterized as *parallel* or *dedicated* depending on the capacity of the processors to perform the same functions (parallel) or if the processors are specialized for the execution of certain tasks (dedicated). There may of course be a single processor indicated by $\alpha = 1$.

When using dedicated processors the tasks form n subsets, each subset called a *job*. Each job, J_j is divided into n_j tasks in the following way $J_j=\{T_{1j}, T_{2j}, \ldots, T_{n_jj}\}$. Two adjacent tasks are to be performed on different processors. A set of jobs is denoted by $\mathcal{J}$.

The parallel processors are further classified depending on their speeds according to the criteria shown in Table 2.1. The dedicated processors are classified according to the way in which the sets of tasks are processed (described in Table 2.1).

A flexible flow shop is a generalization of the flow shop and the parallel machine environments. Instead of m processors in series there are s stages in series with a number of processors in parallel at each stage. The parallel processors can be either

Notation	Description
1	*single processor*
P_m	*m identical parallel processors* with equal processing speeds
Q_m	*m uniform parallel processors* with different speed, where the speed is independent of the processed tasks
R_m	*m unrelated parallel processors* with the processing speed dependent on the processed tasks
O_m	*open shop* with *m* dedicated processors, where the number of tasks is equal to the number of processors and each task T_{ji} must be processed by processor P_j
F_m	*flow shop* with *m* dedicated processors in series with the restrictions of open shop, plus the restriction that each task $T_{i-1,j}$ must be processed before task T_{ij}
J_m	*job shop* with *m* dedicated processors, where the number of tasks per job (n_j) is arbitrary, but each task $T_{i-1,j}$ must be processed before task T_{ij}
FF_s	*flexible flow shop* with *s* stages in series with parallel machines at each stage

Table 2.1: Parameters of the processor environment (α)

identical, uniform or unrelated. The minimum switching model we propose uses a flexible flow shop with three stages and unrelated processors at each stage (see Section 5.3).

When working with dedicated processors, it is usually assumed that there are buffers of unlimited capacity among processors and a job may wait after completion on one processor before its processing starts on the next one. If the buffers are of zero capacity, a *no-wait property* is assumed.

Blocking is a phenomenon that may occur in flow shops when there is a limited buffer between two successive processors. When the buffer is full, the upstream processor is not allowed to release a job. The job has to stay in the upstream processor, preventing—or blocking—the processor of working on another job.

Task and Resource Characteristics

The tasks T_j of $\mathcal{T}$ are characterized by the data detailed in Table 2.2, which is used to indicate for example if the tasks have deadlines or priorities, if the tasks have different processing and arrival times, etc.

Additionally, the tasks may have resource and precedence constraints. The *resource constraints* define additional scarce resources the tasks may need. There are different types of resources: *renewable*, *nonrenewable* and *doubly constrained* [12, Chapter 7]. Renewable resources are classified by parameters $\lambda\,\delta\rho$ that denote, re-

Notation	Description
p_j	*vector of processing times* $[p_{1j}, p_{2j}, \ldots, p_{mj}]$, where p_{ij} is the time needed by processor P_i to process T_j
r_j	*arrival time* or *ready time* indicating the time at which the task is ready for processing
d_j	*due date* indicating the time limit by which T_j should be completed and to which penalties are associated
$\tilde{d}_j$	*deadline* indicating a hard real-time limit by which T_j must be completed
w_j	*weight* or *priority* of T_j
s_{jk}	*sequence dependent setup times* between T_j and T_k (if the setup times depends on the machine, then the subscript i is included, i.e., s_{ijk}); s_{0k} denotes the setup time for T_k if T_k is first in the sequence and s_{j0} the clean-up time after T_j
M_j	*machine eligibility restrictions* to process the task T_j

Table 2.2: Parameters of the tasks and resource characteristics (β)

Notation	Name	Computation
C_j	*completion time*	
F_j	*flow time*	$F_j = C_j - r_j$
L_j	*lateness*	$L_j = C_j - d_j$
D_j	*tardiness*	$D_j = max(C_j - d_j, 0)$

Table 2.3: Computable parameters of the scheduled tasks

spectively, the number of resource types, resource limits and the maximum resource requirement of each task. The *precedence constraints* define the requirements that one or more tasks may have to be completed before another task is allowed to start processing.

Optimality Criteria

There are different criteria to evaluate schedules. The criteria presented in Table 2.4 are the most commonly used. The optimality criterion is computed using the computed parameters for the tasks in the schedule, which are described in Table 2.3.

A schedule for which the value of a particular performance measure γ is at its minimum is called *optimal*, and the corresponding value of γ is denoted γ^*.

If minimizing any of these parameters is the goal, the scheduling problem is an optimization problem. However, it can also be viewed as a decision problem to decide if for a given set of deadlines, there is a schedule with no late tasks. This is denoted as $\gamma = -$.

Notation	Description
L_{max}	*maximum lateness* computed as $L_{max} = max\{L_j\}$; when $L_{max} \leq 0$ then the schedule has no late tasks
U	*number of tardy tasks* computed as $U = \sum_{j=1}^{n} U_j$, where $U_j = 1$ if $C_j > d_j$, and 0 otherwise
U_w	*weighted number of tardy tasks* computed as $U_w = \sum_{j=1}^{n} wj\, U_j$
C_{max}	*schedule length* or *makespan* computed as $C_{max} = max\{C_j\}$
–	*feasibility*

Table 2.4: Optimality criteria (γ)

Notation	Name	Description
C_i	*processing time*	expected worst-case execution time of an instance of the task
T_i	*period*	how often is the task activated?
D_i	*relative deadline*	deadline of the task relative to the period
ϕ_i	*phase*	activation time of the first instance of the task
ρ_i	*shared resources*	set of shared resources that must be obtained and released in every instance of the task

Table 2.5: Parameters of a periodic task τ_i

In this dissertation we are mainly concerned with *feasibility* as optimality criteria. We consider a scheduling algorithm optimal, if it is able to produce a schedule for the most restrictive set of deadlines that are schedulable. That means that the algorithm will always find a solution when a solution is possible.

2.1.2 Periodic Scheduling

Periodic tasks consist of an infinite sequence of identical activities, called *instances* or *jobs*, that are regularly activated at a constant rate. A periodic task is generally denoted as τ_i and a set of periodic tasks is denoted as Γ.

Table 2.5 defines the parameters of a periodic task. T_i, C_i and D_i are considered to be constant for each instance. In general, the deadline of the task is equal to the period ($D_i = T_i$). Figure 2.1 shows an example of a periodic task.

In turn, we can also refer to the parameters of a particular instance τ_{ik} of the task τ_i. The release time of the instance is denoted as r_{ik}, and the absolute deadline of the instance is denoted as d_{ik} and computed as $d_{ik} = \phi_i + (k-1)T_i + D_i$. The activation time of τ_{ik} is computed as $\phi_i + (k-1)T_i$.

There are three important concepts defined for a periodic task set: the processor utilization fraction U, the processor demand $H(t)$ and the slack $S(t)$.

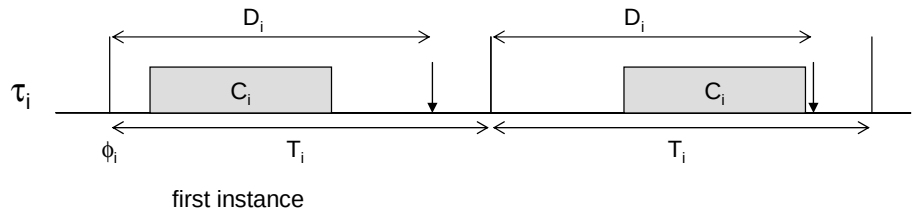


Figure 2.1: Example of a periodic task.

Given a set of *n* periodic tasks, the *processor utilization factor U* is the fraction of processor time spent in the execution of the task set [68]. The utilization is computed as

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{2.1}$$

If the utilization is greater than 1, the task set cannot be scheduled by any algorithm. An important result in periodic scheduling theory is that if the tasks are scheduled using *earliest deadline first (EDF)*, a task set Γ is schedulable if and only if $U \leq 1$ [68]. However, this important result can only be used for preemptable tasks with deadline equal to the period and no shared resources.

The *processor demand* $H(t)$ determines the accumulative load that needs to be resolved so that all tasks in Γ meet their deadlines. In order for all tasks to meet their deadlines $\forall\, t : H(t) \leq t$. The processor demand is defined as follows:

$$H(t) = \sum_{i=1}^{n} \lfloor \frac{t - D_i + T_i}{T_i} \rfloor C_i \tag{2.2}$$

The *slack time* $S(t)$ is the difference of the processor capacity and the processor demand at time t. If the processor is not fully loaded, the slack will be greater than 0. The slack time is useful to execute some tasks with a higher priority or to execute 'best-effort' tasks as soon as possible. The slack time is defined as:

$$\forall\, t : S(t) = t - H(t) \tag{2.3}$$

2.1.3 Problem Complexity

Most of the scheduling problems are $\mathcal{NP}$*-hard*. As defined in [9, page 27], $\mathcal{NP}$-hard is the complexity class of decision problems that are intrinsically harder than those that can be solved by a non-deterministic Turing machine in polynomial time. When a decision version of a combinatorial optimization problem is proven to belong to the class of $\mathcal{NP}$-complete problems (e.g., travelling salesman problem), then the optimization version is $\mathcal{NP}$-hard.

There are different approaches to reduce the complexity of a scheduling problem. One approach is to relax some constraints imposed on the original problem and solve the relaxed problem, e.g., assuming unit-length tasks, allowing preemption. Another approach is to use *heuristic algorithms*, which try to find an optimal schedule but do not always succeed. The advantage of the heuristic algorithms is that they are polynomial. If the heuristic algorithm can be evaluated for its accuracy, it is called an *approximation algorithm*. The accuracy or 'goodness' of an approximation algorithm is measured by the difference between the value of the solutions it produces and the value of the optimal solutions. The values to take into account when measuring the accuracy are the mean behaviour and the worst-case behaviour. In general the mean behaviour of the approximation algorithm is much better than the worst-case behaviour, thus justifying its use. The method generally used for evaluating a heuristic algorithm is to compare its solutions with those produced by the optimal algorithm for a large sample of instances.

2.2 Scheduling of Tertiary Storage

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.

Subsection 2.2.1 present schedulers that can be used to deal with the type of complex requests proposed for the HMA, i.e., the requested data can be stored in multiple RSM and be a combination of continuous and discrete data. Subsection 2.2.2 discusses schedulers for continuous-data stored in one RSM. Subsection 2.2.3 presents some schedulers that have unsolved contention problems. They cannot guarantee that an RSM is not assigned to two different drives during the same time period and, thus, cannot guarantee that the real-time deadlines are always met. Subsection 2.2.4 presents schedulers for systems where the requests do not have real-time constraints. Subsection 2.2.5 discusses work on retrieving data of a single medium, as a tape or a disk.

2.2.1 Schedulers for Complex Requests

In this subsection we present two schedulers that are capable to deal with complex requests as the ones used in the HMA. Although the scheduler of Lau et al. was designed to serve requests for Video-on-Demand, we can easily extend it to more flexible requests, because the data of the video can be stored in multiple RSM in arbitrary ways.

Lau et al. [61] present an aperiodic scheduler for VoD systems that 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 EDF and LDL strategies that we use in Promote-IT.

Their scheduler is part of a hierarchical multimedia storage server, which consists of a tape-jukebox with identical drives for storing video on a permanent basis and a disk array for caching. The authors are concerned mainly with Video-on-Demand applications and assume that the video objects are consumed with the same bandwidth as they were recorded. Thus, the user cannot request to view the video at a higher speed for viewing the video in fast-forward.

The system has full knowledge of the data layout in tertiary storage. A file is divided into pages of equal size. The pages are further divided into fragments that are striped through the disk array in secondary storage. Due to compression, the fragments may contain different numbers of video frames. The system keeps a database with a mapping of video frames to fragments, so that it can provide constant frame rate at the display with variable bit rate I/O retrieval.

Although there is a difference between their system and ours in the service provided and the way the data is stored, their scheduling algorithm can also be used for more flexible requests. We have implemented extensions of both the aggressive and conservative strategy to operate within the HMA, and compare them with Promote-IT.

An important difference between the strategies of Lau et al. and Promote-IT, is that they dispatch the tasks in the same sequence and time as assigned in the schedule. Thus, the conservative strategy performs poorly, because it leaves the resources idle, even when there are tasks that need executing.

Another important difference is that the algorithm handles the jobs to include in the schedule as formed by a *read task* and a *switch task*. The switch task is scheduled as a unity, although it involves unloading the RSM loaded in the drive and loading the new RSM. In Chapter 5 we present the formal model underlying their approach, which is basically a flexible flow shop with two stages (FF_2). In the first stage there is one processor—the robot—and in the second stage there are m identical processors—the drives.

Lau et al. assume that all the drives are identical and that the switching time is constant, independently of the drive and shelf involved. The former assumption is reasonable in many jukeboxes, but makes the algorithm difficult to generalize to the case with non-identical drives. The latter assumption is not reasonable in most of the large jukeboxes and forces to use worst-case switching times when building the schedules. Using more accurate switching times provide better schedules.

There are several important problems when modelling the scheduling problem as a FF_2. On the one hand, that the drives are left loaded until a new request arrives that uses the drive. This means that even if the robot and the drive are idle, the robot will not unload the drive. Our experience has shown that this is a bad approach, because the probability that new data will be requested from an RSM that is loaded at the time of computing the schedule is low. On the other hand, the scheduler cannot make use of small holes in the robot schedule.

A natural consequence of assuming identical drives and constant switching times is that the algorithm only tries assigning a task to one drive. If it fails, it concludes that the task cannot be scheduled. In the case of non-identical drives and/or variable switching times, this approach is a clear over-simplification as we show in Section 6.5.1.

Additionally, the conservative strategy suffers from the problem that it cannot be generalized to the case where the switch time is not constant and still be based on a FF_2. The conservative strategy schedules the unload of a tape whose identity is unknown, because every time it schedules a read it also schedules a switch. The switch includes unloading the tape that is loaded in the drive before the one being scheduled. But the schedule is built Back-to-Front and so the scheduler still does not know which tape is loaded in the drive. In Section 7.3.2 we propose an extension to the conservative strategy based on the minimum switching model, which uses a three-stage flexible flow shop (FF_3).

We could extend the aggressive strategy for other kind of jukebox architectures, while still using an FF_2. The extension can deal with non-identical drives, variable switching times, and multiple robots. However, the model still has restrictions regarding the scope and functionality of the robots, which originate in the fact that the unload and load operations are coupled. Moreover, in Chapter 9 we show that coupling the unload and load operations has a negative influence on the scheduler performance.

The argument given by the authors for and against these strategies is that the aggressive strategy makes better use of the jukebox resources, while leading to a higher page miss ratio, because the pages may be retrieved into the cache much earlier than needed. The conservative strategy instead makes bad use of the jukebox resources, leading to bad response times when the system load increases.

We show in Chapter 9 that the conservative strategy is unable to handle even reasonable loads, because it very soon creates a system overload. The reason for the overload is that the jukebox resources are left idle at moments when the system load is not high and that time cannot be recuperated later, when more requests arrive. In our tests the cache-hit rate of the conservative strategy is not better than that of the aggressive strategy.

Federighi et al. [29] use requests similar to those of the HMA. In their system the videos may be stored in multiple objects, with different sound tracks and subtitles corresponding to each video. The requests may consist of multiple objects (in the terms of HMA, multiple request units). The requests in their system have *soft deadlines*, e.g., the data should be available at around eight o'clock. Federighi et al. are mainly concerned about balancing the load on distributed video file servers, which are placed near the users [19]. The scheduler clusters the objects requested from the same tape. The request increases in priority as the time of the deadline approaches. Once a request has reached enough priority to stage the data, the corresponding tapes are scheduled for retrieval.

An important difference with our approach is that, even if the requests consist of multiple objects, the playback only begins once all the objects are available at the video file servers. We refer to this type of approach as *Fully-Staged-Before-Starting (FSBS)*. In the next subsection we discuss in more detail full staging, pipelining and direct access. All the schedulers that we propose in this dissertation use pipelining (directly or indirectly), which means that the data of a request can be consumed while other data of the request is being staged.

2.2.2 Schedulers for Simple Requests for Continuous Media

The schedulers we discuss in this subsection provide access to continuous-media stored in one RSM. The requests are much simpler than those of the HMA and the systems discussed in the previous subsection. The topic that is mainly discussed by the authors is (a) if the data should be streamed directly from tertiary storage to the end-users, called *direct-streaming*, (b) if the data should be *fully staged* in secondary storage before beginning to stream it to the user, or (c) if the data should be *pipelined*, i.e., overlapping staging on secondary storage and streaming to the user.

Chan et al. [21] stage a movie completely in secondary storage before it is displayed to the user, because their goal is to provide interactive VoD services. They are mainly concerned with providing the user the possibility to interact with the display by issuing fast-forward, rewind and pause commands. Thus, once the user is granted access to the movie, he must also be granted the possibility to interact with the movie presentation in real-time. Therefore, the user is given access to the movie only once the movie is completely staged.

Our approach is that it is the role of the network servers (or client applications) to deal with user interaction and not of the HMA. A network server can provide interactive VoD as proposed by Chan et al. by requesting all the data to be staged ASAP.

Whereas other VoD network servers can deal with user interaction by making a new ASAP request to the HMA to fulfil the new requirements of the user.

Chervenak et al. [22, 23] concludes that jukeboxes are not the appropriate devices to provide storage for a Video-on-Demand system. They suggest that the solution is to increase dramatically the number of drives in the jukeboxes or using a disk farm based on hard disks. However, they do not consider the possibility of using a real hierarchical storage system. Instead, their approach is to stream data directly from the drives to the user. What they propose as hierarchical storage is to store the most popular videos in secondary storage and the rest of the videos in tertiary storage. The videos stored in tertiary storage are either accessed directly from there [22] or fully staged into secondary storage [23]. Chervenak also considers that the data may be striped on multiple tapes to achieve the desired bandwidth when using direct-streaming, in which case more than one drive needs to be reserved for the whole duration of the display.

Pang [78] shows that, in most cases, providing direct access from the drives makes poor use of the jukebox resources, resulting in poor system performance. He proposes an algorithm called Asdac that intelligently decides when to stage data in cache and when to provide direct access. Pang is concerned with providing access to large continuous-media objects. However, he assumes that all the data must be staged in secondary storage before the user is given access to the data. We believe that if the user is given access to the data before the whole object is staged, for example by dividing the request into multiple request units as done in the HMA, then staging is always superior to direct access.

Triantafillou et al. [108] propose a combination between direct-access and staging in which some data is streamed directly to the user while other is first stored in secondary storage. They alter the sequence in which the videos are stored, so that in a periodic way the drive will be reading exactly the frames that can be streamed directly. Computing the new sequence strongly depends on the drives used. Therefore, all the drives must be identical and an update in the drive technology forces to regenerate all the sequences. Additionally, if the purpose is to cache the files in secondary storage, the data streamed directly to the user must still be stored in secondary storage. Thus, it does not lead to real savings in cache space.

Ghandeharizadeh et al. [35] show the benefits of using pipelining without making any assumptions on how the data is stored in the RSM. The only restriction is that the data should be stored in one RSM. They divide the object to retrieve in slices and compute the response time of a request by computing the time needed to retrieve the slices that are not in the cache. They provide a method to compute the response time once the RSM is in a drive that works for any relation between the bandwidth of the display and the bandwidth of the drive (i.e., the requested bandwidth may be lower or higher than the bandwidth of the drive).

2.2.3 Schedulers with Unsolved Contention Problems

We now discuss briefly other approaches that, although they are interesting, do not deal correctly with the resource-contention problem. Thus, they cannot guarantee meeting their deadlines, because in the case of a resource conflict the scheduler will either miss a deadline or crash. Therefore, these schedulers cannot be used for jukeboxes with multiple drives and, so, are not suitable to be used with most commercial big jukeboxes, which have multiple drives.

Boulos et al. [17, 16] extend the scheduler of Ghandeharizadeh we discussed in the previous subsection to objects stored in multiple RSM. They use an aperiodic scheduler to compute the starting time of a request. For each drive they keep a queue that contains the blocks that need to be read by the drive. Each element in the queue indicates the deadline and the time assigned to it. They incorporate new elements into the queue using an aperiodic EDF algorithm. Their approach does not have RSM-contention problems, because they retrieve all requested data from an RSM using the same drive. However, they cannot guarantee that there are no robot-contention problems. They consider that the robot is not a source of contention, and, thus, do not schedule its use. Our research clearly shows that the robot is in general a source of contention. The robot tends to be the bottleneck in the system, due to the slow switching speed when compared with the reading speed of the drives (see Chapter 4).

Lau et al. [62] present another approach to schedule the requests of the hierarchical multimedia storage server for Video-on-Demand introduced in Subsection 2.2.1. This approach, which they generically termed *time-slice algorithm*, consists of breaking up the requests into many tasks which are served separately in order of increasing task number.

Each request is assigned a time-slice period, which represents the duration of each task corresponding to the request. During this time-slice, the task must read the data and perform the required tape switch. Contrary to the approach we use in our periodic scheduler, JEQS (see Section 5.6), the size of the time-slices is not constant, but varies and must be computed for each incoming request.

Lau et al. deal with the robot contention that arises from having a shared robot by using the worst-case robot waiting time in the feasibility analysis. They assume that in the worst case, before executing each task there will be a robot contention. This considerably restricts the resource utilization that can be achieved.

However, they do not deal with the tape-contention problem. The algorithm does not take into account that multiple tasks may need to read data from the same tape. They assign drives to tasks on a per task basis, ignoring to which drives other tasks for the same tape were assigned. Thus, it can happen that two tasks that need to read data from the same tape are assigned to different drives during overlapping periods.

They propose two algorithms to schedule the tasks: the round-robin algorithm and the least-slack algorithm. When using the round-robin algorithm, the active requests are served in the same order in each round of service. A disadvantage of the round-robin algorithm is that it can only incorporate new requests to the system at the end of a round. The least-slack algorithm tries to incorporate new requests as soon as possible, without having to wait until the end of a round. This algorithm also adds a deadline to each task and computes a schedule with all the tasks. The algorithm can incorporate a new request to begin immediately if every task has enough slack.

The least-slack algorithm may look similar at first glance to the approach we use in JEQS, but it is really quite different. On the one hand, Lau et al. do not use periodic scheduling theory to give guarantees about the schedulability of the request. Instead, they transform the problem into an aperiodic scheduling problem where instances of the same request have to be scheduled separately. On the other hand, by not determining a priori how the shared robot is used, the computation of the maximum possible utilization has to take into account the worst-case scenario of having to wait for the robot to finish moves on all the other drives. Last, but not least, their algorithm cannot guarantee meeting real-time deadlines, while JEQS can.

A possible advantage of the least-slack algorithm over JEQS, is that the system could adapt the size of the slices when the load of the system increases, forcing to read more data every time a tape is loaded and, thus, increasing the efficiency in the use of the jukebox resources.

Golubchik et al. [36] propose a periodic scheduler called *Rounds*. The data is staged from a tape jukebox to secondary storage, from where it is streamed to the user. The usage environment is a multimedia storage-server application such as Video-on-Demand.

The entire database is stored in tertiary storage in *pages* of constant size. Golubchik et al. assume that each retrieval of a page requires a tape exchange—load, seek to the required page in the tape, the read itself, rewind and unload. The time required for these operations is termed *cycle time*.

Each request is a tuple (b, p), where b is the bandwidth and p is the number of pages. The bandwidth b is bounded by a minimum and maximum accepted bandwidth. The goal of the scheduler is to provide a low and predictable response time (or latency) and make good use of the hardware.

Golubchik et al. assume that the pages are randomly distributed among the tapes. Furthermore, they assume that the p pages of a request are stored in p different media. However, the way in which the pages are stored in the jukebox (randomly distributed) does not provide any guarantee that two pages in the same request will not correspond to the same tape. Moreover, they do not deal with the fact that there may also be conflicts on using the same tape by different requests.

When a new request arrives at the system, the tertiary storage subsystem needs to reserve periodic slots on the tape drives for the entire length of the request. The periodicity of slot reservation corresponds to the bandwidth of the request. Slots corresponding to different drives are *staggered* from each other by the 'robot-latency', because if the drives were synchronized all drives would require the robot to exchange tapes simultaneously. The staggering of the drives is the same approach we use in the periodic quantum model presented in Section 5.6.

Cha et al. [20] use a jukebox scheduler based on a periodic EDF scheduler [68], which deals neither with the robot-contention problem nor with the RSM-contention problem. Therefore, in the best case, this scheduler could be used for jukeboxes with one drive. However, even for these simple jukeboxes, the scheduler cannot guarantee meeting the real-time deadlines. The authors do not take into account that the tasks are non-preemptive, because every media switch certainly does not take negligible time. However, the EDF algorithm without additional modifications works only for preemptive tasks.

2.2.4 Schedulers for Requests for Discrete Data

We now present some schedulers for the retrieval of data from tertiary storage, when the requests do not have time constraints. The goal of these schedulers is to minimize the average response time. In all cases, the conclusion is that as much data as possible should be read from an RSM when the RSM is loaded in the drive. This serves as support for the *minimum switching model* that we propose in this dissertation.

Triantafillou et al. [107] propose using a hierarchical scheduling algorithm—CLUST—for scheduling non real-time requests on a tape library. The requests are simple: they consist of the tape identifier, the starting block and the number of blocks. CLUST groups the incoming requests according to some criterion, such as the tape needed. At the upper level CLUST uses an algorithm to determine what group to process first. At the lower level it uses an algorithm to schedule the tasks inside the group.

The upper-level algorithm of CLUST is a variation of the round-robin algorithm, which picks up the tape with the longest request queue. This algorithm can suffer from starvation. In practice, however, it achieves relative fairness and good performance, because it keeps popular tapes in the drives. The lower-level algorithm looks for the optimal schedule for reading data from the tape. A heuristic algorithm can also be used at this level.

Triantafillou et al. also present another algorithm called SATF (Shortest Access Time First), which is not hierarchical. SATF is a greedy algorithm that always picks up the requests that need the shortest access time. This algorithm may lead to starva-

tion, but when the system load is high it has the potential of minimizing the number of media switches.

The authors show that CLUST performs better than SATF. They also show that both algorithms perform better than the OPT(N,K) algorithm, which computes the optimal schedule for the first K requests in the queue. Their explanation for the bad performance of OPT(N,K) is that it only takes into account the first K requests. Computing the optimal schedule is out of question because the complexity of the problem is exponential.

Georgiadis et al. propose the Relief algorithm [34], which chooses the relief ratio of each request as the time the request has been waiting in the system divided by the time the system takes to serve the request. The system favours the requests that have either been waiting for a long time, thus, trying to reduce the average response time, or that can be served fast, for example, because the tape is already loaded. Relief is an improvement over another algorithm proposed in the same paper, called Bypass, which always favours requests for the tapes in a drive. Bypass is unfair and may lead to starvation.

Prabhakar et al. [88, 87, 86] show the benefits of reading all the requested data from an RSM before the medium is unloaded. Furthermore, they show that these results are valid both for optical and tape jukeboxes, even when the seeking time dominates over the switching time, as is the case with some serpentine tape-jukeboxes (see Section 4.1.2 for more details about tape technology).

They order the RSM by non-increasing ratio of number of requests to the sum of processing time and switch time. Their goal is to minimize the average response time. Their algorithm is optimal when there is only one drive, and performs very well when there are multiple drives. The scheduling problem for multiple drives is $\mathcal{NP}$-hard, thus, their algorithm is not optimal in that case.

Moon et al. [73] show that grouping all requests for an RSM results in a performance near the optimal. Their simulation environments consist of an optical and a tape jukebox with one drive and a small number of only 10 shelves. Their application environment is retrieving large database objects without real-time deadlines. Their goal is to minimize the average response time and the total response time. They evaluate three strategies for selecting the RSM: Round Robin (RR), Maximum Processing Time (MPT), and Maximum Number of Queries (MNQ). They compare these strategies against First-Come-First-Serve (FCFS) and an optimal scheduler.

The work of Moon et al. shows that scheduling tertiary storage is important, especially when the system load increases. It shows that the performance of the three strategies is near the optimal when compared against FCFS. The comparison among the three strategies shows that MNQ is the best, followed by MPT and RR.

More et al. [74] are concerned with performing queries on data that is stored in multiple tapes. Thus, a query may have multiple request units without real-time

constraints. Their goal is to minimize the response time of each query. They model the scheduling problem as a two-machine flow-shop with additional constraints. In their model, the unload and load of a tape are coupled. They propose the longest transfer-time first (LtF) algorithm that for each query starts reading first the data of the sub-queries that require the longest transfer time. If there are multiple sub-queries for the same tape they use the SORT algorithm proposed by Hillyer et al. [45] to decide the order in which the sub-queries should be read (see Section 2.2.5). The rationale behind the LtF algorithm is that while the data of the longest sub-query is being read, there is time to switch the tapes on the other drives and read the data corresponding to the shorter sub-queries. Through analytical analysis and simulations they show that LtF provides short response times.

In our HMA we can represent the type of requests More et al. are concerned with as multiple request units of one request, which all have the same delta deadline (see Section 3.2 for details about the request of the HMA). The strategies of Promote-IT that use the latest starting time as parameter to sort the jobs build similar schedules to those of LtF for this type of requests, even if the length of the transfer is not the scheduling parameter used by Promote-IT (see Section 6.2). Given a set of RSM with the same deadline and different transfer times, the ones with longer transfer times will have earlier latest-starting-times. Thus, these strategies of Promote-IT will also schedule the RSM to begin earlier.

Other large database systems (e.g., RasDaMan [93] and the high-performance digital library proposed by Grossman et al. [39]) rely on the scheduling of tertiary storage resources provided by a hierarchical storage system (e.g., UniTree or SAM-FS). The hierarchical storage systems, in general, cluster all requests for the same RSM to minimize the expensive exchange operations. Sarawagi [97] argues against relying on the scheduling provided by a hierarchical storage management system and advocates for reorganizing the queries and scheduling the resources in a global way so that the jukebox resources and the secondary storage cache can be used efficiently.

Johnson [52] presents an analytical model of robotic storage libraries. His requests are for multiple files that can be stored in different RSM. A request is satisfied when every file has been staged to secondary storage. The requests are served First-Come-First-Serve and the goal is to minimize the response time of the requests. The jukeboxes that he can model have only one robot.

Johnson concludes that adding drives to the jukebox can improve the performance of the jukebox, but after a threshold, adding drives does not significantly improve the performance. Then the robot becomes the bottleneck of the jukebox.

Johnson et al. [54] analyze the performance of different tape drives. They conclude that high-throughput drives need to read large amounts of data per tape to be effective.

Hierarchical Storage Management

Tertiary storage plays an important role in supercomputing environment and scientific computing. Essential to these environments is the capacity to deal with petabytes of data that must be easily accessible to geographically distributed scientists. The storage hierarchy that stores the data must be transparent to the users, except for the delays of accessing data in tertiary storage. Additionally, much of this data can only be collected once (e.g., measurements of the atmosphere or stars) and it is therefore important to provide safe backups mechanisms, as vaulting.

Therefore, much effort was put into developing *hierarchical storage management (HSM)* systems. The IEEE Mass Storage System Reference Model [24] describes the characteristics such systems should posses. Multiple HSM systems have been developed, both conforming to the reference model and prior to it. Some examples are the High Performance Storage System (HPSS) of the National Storage Laboratory [106], and the Storage and Archive Manager File System (SAM-FS) of Fujitsu [30]. Miller [72] and Setia et al. [100] provide an overview of different HSM.

The openness of the reference model permits to include specific real-time services as future interfaces [106]. However, no HSM so far supports real-time services. Our HMA can be incorporated in the reference model as a *Storage Server* component.

There are also many extensions to file systems, which incorporate tertiary storage. Plan 9 [82] uses tertiary storage to backup data from the file system allowing the users to access the state of the file system at any date in the past. Jaquith [75] uses a similar mechanism as Plan 9, but the system allows the user to decide which data should be stored in tertiary storage in a per file basis. Highlight [57] uses the same structure for the data stored in secondary and tertiary storage, allowing, for example, only some blocks of a file to be in tertiary storage.

2.2.5 Scheduling of a Single Medium

This subsection discusses scheduling mechanisms for accessing data stored in one medium. We present first some techniques for scheduling the use of a magnetic tape, which is the most used tertiary storage medium. We then present scheduling of discs. Most of the work on disks has been done for hard disks, which are non-removable. However, some of the techniques developed can be applied for removable disks.

Tapes

Hillyer et al. [45] present and evaluate different scheduling algorithms to retrieve data from tapes. Their work specially tackles the problem of randomly accessing

serpentine tapes. The problem they are concerned with is scheduling multiple requests to retrieve data blocks from a tape. The blocks to retrieve are stored in random parts of the tape. The requests represent, for example, database management workloads. The requests do not have deadlines.

A serpentine tape drive saves the data in a forward-backward manner. It records a track down the length of the tape and then reverses direction and shifts the head sideways to record another track (see Section 4.1.2). A characteristic of serpentine tapes is that the locate time is not a simple function of the distance in blocks between the source and the destination, but a more complex function that depends on the source and destination blocks [44]. Therefore, it is not enough to sort the requests by offset and read the data in that order.

Scheduling the read of data from a helical tape is straightforward, because the logical blocks correspond directly with physical tape positions. A helical tape uses rotary head technology like that of a video-cassette recorder. Therefore, the requests just need to be sorted by block number and retrieved in that order.

Hillyer et al. analyze the following algorithms for serpentine tapes:

READ reads the entire tape.

FIFO reads the data in the order that the requests arrive at the system.

OPT computes the optimal path to read the data by finding a solution to the asymmetrical travelling salesman problem. The computational complexity of OPT is exponential.

SORT orders the requests by segment number.

SLTF (shortest locate time first) reads the data in a greedy manner, jumping each time to the nearest request.

SCAN alternately shuttles up the tape, reading sections in forward tracks, and then back down the tape, reading sections in reverse tracks. This strategy tends to switch tracks more often than sort, but makes fewer passes up and down the length of the tape.

WEAVE is an approximation to SLTF that does not use the locate function.

LOSS is a greedy algorithm for the asymmetric travelling salesman problem [64].

The authors conclude that OPT can only be used for very small sets of up to 10 requests. If the number of requests is low or medium, LOSS behaves best. If the number of requests is high the best is to read the full tape. The performance of SLTF, WEAVE and SCAN is also good. As expected FIFO performs poorly.

Hillyer et al. [46] extended their study to a modified serpentine tape, which is designed for fast random access. In this case the performance of the SCAN, SLTF and OPT algorithms is nearly identical, because the locate function is simpler than that of a normal serpentine tape.

Disks

Scheduling the access to hard disks has been identified since long as an important issue in improving the performance of computer systems. Disk schedulers can be divided into non real-time and real-time.

Non real-time schedulers are not concerned with deadlines. These schedulers assume that requests, which simply indicate a contiguous disk block, arrive continuously and are put in a queue of requests. The goal is to minimize the average response time of each request and to avoid starvation of the requests. In 1967 Denning [26] proposed two algorithms: shortest-seek-time-first (SSTF) and SCAN. SSTF chooses the request in the queue that is closest to the current position of the head. SSTF can lead to starvation and provides poor response times to requests at the innermost and outermost tracks of the disk. SCAN reads the requested data as it moves the head in one direction until all requests in that direction have been processed. It then reverses the direction of the scan. Although this algorithm avoids starvation, the middle tracks receive better service, because they are read more often than the edges. There are numerous variations of the SCAN algorithm for non real-time environments, e.g., FSCAN, C-SCAN, LOOK, and C-LOOK. VSCAN [32] creates a continuum of algorithms with different levels of bias toward maintaining the direction of the seek or changing it. Worthington [116] evaluates the different algorithms.

Shastri et al. [101] adapt C-SCAN to read data from a CD-ROM in order to access continuous-media. Tsao et al. [109] use the same approach and propose reordering the blocks of the stream to serve multiple users. Tsao et al. consider the case of accessing a movie stored in more than one CD-ROM. Their concern is to provide a near-VoD server streaming the data directly from the CD-ROM drives. In both cases a disk is loaded in a drive on a permanent basis.

Real-time disk schedulers read data for multiple streams in a periodic way. During each period of a stream they read a relatively small amount of data, which is just enough to fill a buffer. Their design is based on the assumption that the medium is not removed, which is valid for hard disks, but not for optical disks in a jukebox. However, the principles of SCAN-EDF [91] can be used for non-periodic schedulers. In fact, the algorithm that we propose for building the medium schedules is similar to SCAN-EDF, although it is aperiodic (see Section 6.6).

The requests that the SCAN-EDF algorithm deals with are periodic requests with real-time deadlines and best-effort requests. The goal of SCAN-EDF is to be able to serve as many periodic requests as possible, while still providing a short response time to the best-effort requests. SCAN-EDF assigns a high priority to the best-effort requests and tries to read the data for them first. It schedules the requests using the EDF algorithm [68] and uses a SCAN algorithm for requests with the same

deadline. Reddy et al. [91] show that SCAN-EDF is a good combination of both algorithms, and performs well in achieving both algorithm goals. It can serve a high number of simultaneous periodic requests and provide short response times to best-effort tasks.

2.3 Scheduling of Automated Storage/Retrieval Systems

The scheduling mechanisms presented in this dissertation can also be used to schedule a *miniload automated storage/retrieval system (AS/RS)* in order to support *just-in-time (JIT) production*. A miniload AS/RS [38] is a storage device used in factories and warehouses to store small items, similar to a tertiary-storage jukebox. In factories they are used to store materials, intermediate products, spare parts, and tools.

An AS/RS consists of a series of storage aisles that are serviced by one or more *storage/retrieval machines (S/R)*. There is usually one S/R per aisle. An AS/RS has one ore more *pickup-and-deposit stations (P&D)* where the materials are delivered to and retrieved. In the case of a miniload AS/RS the materials are kept in *bins* or *drawers*. In order to retrieve elements from the AS/RS, a bin is brought by the S/R to a P&D where the desired elements are taken from the bin. The S/R then returns the bin to its location. For example, the bins may contain screws, nuts and bolts, where the elements of each bin have the same characteristics—size, material, etc. A request for 10 bolts of type X triggers the S/R to bring the appropriate bin to a P&D where a human operator extracts 10 bolts from the bin and S/R returns the bin with the remaining contents to its location in the aisle.

The goal of *just-in-time production* is to produce small lot-sizes and respond promptly to new demands from the users. Therefore, it is important that the different components are available in time for the production, so that once the production starts it is not interrupted and the amount of work-in-process is kept low.

The requests that we propose in this dissertation can be used to define the time at which the different components are needed for production. The scheduler for the storage system determines the earliest time at which the products can be available while respecting the timeline defined by the request. It also guarantees that the components are available at the pickup stations in time.

The hardware model for the jukebox can be easily used for an AS/RS. Only the details for computing the time to move the robots and pick up the components need to change. As a matter of fact, many features of the hardware model are more frequent on this type of storage than on tertiary-storage jukeboxes, e.g., multiple

robots, limited scope and different robot functionality. Some big semi-automated tape jukeboxes use a miniload to handle the storage and retrieval of the tapes, e.g., Odetics Storage Subsystem [89].

However, the normal utilization pattern of a miniload AS/RS gives raise to different scheduling problems than those present in jukeboxes. Mahajan et al. [70] assume that the time needed to retrieve elements from the bins is short compared to the time needed to move the S/R machine between locations. Therefore, it is important to pair unload and loads correctly in order to minimize the average response time of requests. The pairing of unload and loads is called *dual-command cycles*. Mahajan et al. propose using a nearest-neighbour retrieving sequence to pair the operations in order to minimize the distance travelled between finishing an unload operation and beginning a load operation. They show that their approach is 5 to 15% better than FCFS. In their hardware model the P&D are located at the same place and the picker can be seen as one machine with an outbound buffer of one unit.

The picker may also have a larger buffer. Park et al. [79] analyze the optimal size of the inbound and outbound buffers to maximize the throughput of a miniload AS/RS. In our hardware and scheduling-problem model we can model the buffers as drives. Instead of viewing a load as bringing a bin from a shelf to the beginning of the queue of bins that need to be processed and unloading it from the end of the queue to process, as done by Park, we handle the buffers in a cyclic way.

To the best of our knowledge no work has been done to schedule a miniload AS/RS in order to meet real-time deadlines. Given the strong similarities between the two applications, we will only refer to the production application when the difference is important or the solution is more applicable in this environment than in the HMA.

2.4 Scheduling in Logistics Applications

Our scheduling problem has some common features with the problems from the family of the *vehicle routing problem (VRP)*. However, there are important differences that prevent the solutions to be directly comparable. This section presents the types of VRP that are most closely related to our scheduling problem and presents some VRP-like problems that we use to model sub-problems of our jukebox-scheduling problem.

In the *vehicle routing problem with time windows (VRPTW)* a number of *identical vehicles* must be routed to and from a depot to cover a given set of customers, each of whom has a specified *time interval* indicating when they are available for service. Each customer also has a known *demand*, and a vehicle may only serve the customer on a route if the total demand does not exceed the *capacity of the vehicle*.

An extension of this problem deals with *non-identical vehicles* [40, 58]. There are numerous exact and heuristic solutions to the VRPTW. Larsen [60] provides an overview of different solutions.

An interesting special case of the VRPTW is the *asymmetric multi-travelling salesmen problems with time windows (m-TSPTW)*, which ignores the capacity of the vehicles. However, all the salesmen are identical, so the extension for non-identical vehicles is lost. The m-TSPTW is a generalization of the *asymmetric travelling salesman problem with time windows (TSPTW)*, which is in turn a generalization of the well known *travelling salesman problem*. Ioannou et al. [49] show the necessity to use different algorithms to solve the m-TSPTW than to solve the VRPTW, even if the former is a special case of the latter.

We model reading data from one RSM as a TSPTW (see Section 5.2.4). In the dedicated robots model presented in Section 5.7 we propose modelling a jukebox with one drive as a TSPTW. Furthermore, we model a jukebox with multiple drives as an extension of the m-TSPTW with non-identical salesmen and additional resource constraints. However, we show that even after making these extensions to the m-TSPTW, such a model can only be used for jukeboxes with dedicated robots.

The *inventory routing problem (IRP)* is interesting because it deals with a longer time horizon than the vehicle routing problem. The customers have a consumption rate and the delivery company decides how much to deliver to which customer each day, so that the customers do not run out of product. The IRP is concerned with the repeated distribution of a single product from a single facility to a set of customers over a long planning horizon. The customer consumption is not necessarily periodic, because it may stop during the weekends. Therefore, this is a good application for the type of flexible requests that we propose.

The solution to the IRP presented by Campbell et al. [6] has important points in common with our scheduler. On the one hand, it is similar to the approach we use to represent our scheduling problem as the *optimal model* (see Chapter 5) in the sense that it first clusters the customers and then it computes the schedules for each cluster. On the other hand, Campbell et al. tries to use full trucks and delivering to customers earlier than necessary. In Promote-IT we read as much as possible from an RSM once it is loaded in a drive (equivalent to full trucks) and dispatch as early as possible. However, in the IRP the customers have buffer limitations that do not always permit to deliver early.

2.5 Summary

This chapter presented different approaches to schedule tertiary storage. We categorized the schedulers according to the type of requests they can handle. The sched-

ulers for complex requests have the potential to handle complex real-time requests as the ones used in our HMA. In this case the data can be stored in multiple RSM, can be for continuous and discrete data and can be combined in any possible way. In Chapter 8 we present our extensions to the schedulers presented—the aggressive and conservative strategies of Lau et al., and the Fully-Staged-Before-Starting scheduler of Federighi—in order use them in our HMA.

The schedulers for simple requests for continuous-media can only provide real-time access to data that is stored in one RSM. We presented some schedulers that attempt to provide real-time guarantees to access continuous-media, but due to resource-contention problems they cannot guarantee meeting the deadlines. The discussion about schedulers for requests without real-time deadlines showed the benefits of reading all the requested data of an RSM once the RSM is loaded in a drive. We also discussed shortly the scheduling of single media as tapes and disks.

Additionally, we discussed how the concepts presented in this dissertation can be applied to schedule an automated storage/retrieval system. Using the type of requests and scheduling mechanisms proposed for an HMA, such a storage system can be used in a just-in-time production environment. Finally, we discussed the relation between our scheduling problem and scheduling problems in logistic applications.

Chapter 3

Hierarchical Multimedia Archive

The *hierarchical multimedia archive (HMA)* is a flexible storage system that can serve complex requests for the real-time delivery of any combination of media files. Such requests can originate from any system that needs to combine multiple media files into a continuous presentation. A request can consist of multiple streams and non-streamed data that are synchronized sequentially or concurrently in arbitrary patterns.

As described in Section 1.2, the HMA acts as a *real-time file system* and the *network servers* provide specific services to *remote users*. The remote users access the services through local *client applications*, e.g., a video player, a multimedia presentation tool, a database query interface.

Following the classification presented by Gemmel et al. [33], the HMA is *file-system oriented*, because it offers to its users (the network servers) simple operations such as open, read, close. The network servers may in turn offer *stream-oriented services* to their clients, offering operations such as play, pause, resume. Almeroth et al. [4] discuss different types of paradigms that can be used to provide access to video. They advocate limiting the VCR capability of the servers, and putting the burden of interaction on the client applications. This can be achieved by having the client applications buffer the data. This approach is used by time-shifted TV.

We first present some usage scenarios for the HMA. We then define formally the user requests. We present the system architecture, with special emphasis on the cache manager, which is not further described in the rest of the dissertation. We present the generic schedule builder that is the basis of all the heuristic jukebox-schedulers. Finally, we briefly discuss how to add new data into the jukebox.

3.1 Usage Scenarios

We present here some usage scenarios of the HMA. In the first scenario a user in the local network decides to watch a movie with English audio and Dutch subtitles. The movie should start ASAP. The client-application sends the request to the

VoD server. The server consults the archive directory to map the movie to the corresponding files and to enquire some other details as format and bandwidth. Let us assume that the video for the movie is stored in three different disks. Each disk contains also the audio and subtitles belonging to the respective part of the video. The server creates a request with request units corresponding to the video, the audio and subtitles of each part of the movie. The HMA schedules the request and immediately replies that the user can start consuming the data in 25 seconds. To fill in the time, the server streams a 30-second long advertisement and then starts streaming the movie.

In the second scenario, a user requests the *personal MTV server* to play clips from a play list in a random order ASAP. The play list involves songs from different albums stored in different RSM. The server consults the HMA about the songs that could be played immediately because they are already in cache. It then builds a request in which the first song is already in cache, so that the request can be served faster. This approach is similar to the *opportunistic scheduling* proposed by Aksoy et al. [2] for broadcasting readily available data first, and the *Storage Latency Estimation Descriptors (SLEDs)* of van Meter et al. [111]. This server can also be used by a video-clip broadcaster with multiple channels like MTV, where the viewers make requests for video clips in the form of votes. Periodically, e.g., every half hour, the broadcaster analyses the votes for each channel and makes requests to the HMA for the video clips that should be played in the next half hour.

In the last scenario, a *multimedia-database server* uses the HMA to manage the contents of the database. The front end of the database server provides multimedia searches, for which an on-line index is used. As a result of a search the server requests the HMA the contents the user wants to view. The order in which the data is available is not relevant in this case, only that it should be available ASAP. The client application shows the results in different colours, indicating the results already available and those that still need to be made available by the HMA. The user may start viewing the results as they become available.

3.2 User Request

A *request* r_i, which a user issues to the system, consist of a deadline and a set of l_i request units u_{ij} for individual files (or part of files). The request can represent any kind of static temporal relation between the request units. This type of request is called absolute expressions [56], because they represent time synchronization information as the order in which the events take place on an absolute time axis. Formally we express the user request structure in the following way:

$$r_i = (\tilde{d}_i, asap_i, maxConf_i, \{u_{i1}, u_{i2}, \ldots, u_{il_i}\})$$

$$u_{ij} = (\Delta\tilde{d}_{ij}, m_{ij}, o_{ij}, s_{ij}, b_{ij})$$

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 user may specify no deadline ($\tilde{d}_i = \infty$) if the only restriction is that the request should be scheduled ASAP.

The maximum confirmation time $maxConf_i$ 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 system must provide a confirmation before making the data available, so $maxConf_i \leq \tilde{d}_i$.

The parameters of the request units are

m_{ij} RSM where the data of the request unit is stored

o_{ij} offset in the RSM

s_{ij} size of the data requested

b_{ij} bandwidth with which the user wants to access the data—if $b_{ij} = 0$ then we say that u_{ij} represents a *block*, otherwise it represents a *stream*

$\Delta\tilde{d}_{ij}$ relative deadline of the request unit—this is the time by which the data of the request unit should be available for the user to access it, relative to the starting time of the request Formally we define $\tilde{d}^n_{ij}$ as the deadline of the nth byte in u_{ij}:

$$\tilde{d}^n_{ij} = \begin{cases} st_i + \Delta\tilde{d}_{ij} & \text{if } b_{ij} = 0 \\ st_i + \Delta\tilde{d}_{ij} + n/b_{ij} & \text{otherwise} \end{cases}$$

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 that the HMA 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 system tries to find the earliest value of st_i that will allow it to accept the request. The system must provide a confirmation before $maxConf_i$.

The request and its confirmation are the *contract* between the user and the system. The user can start consuming the data at the assigned starting time with the system's guarantee that the flow of data will not be interrupted.

The *response time* of a request rt_i is the time that elapses between the arrival of the request and the assigned starting time. The *confirmation time* ct_i is the time between the arrival of the request and the time at which the user gets the confirmation. When evaluating the schedulers we will also refer to the *computation time* of

a request, which is the CPU time used to compute a feasible schedule that permits to incorporate the request into the system. The computation time includes also the failed attempts to build such a schedule.

The users can also request files without specifying the offset and number of bytes, or for an offset in a file and define the number of bytes to read. However, all these requests are finally translated to the structure presented above.

An important design decision is to let the network servers decide how to build a request in order to achieve a good degree of pipelining. In the VoD scenario presented in the previous section, the VoD server really chops each file containing video, audio and subtitles into request units corresponding to approximately ten minutes of presentation. Generally, the presentation can start as soon as the first ten minutes worth of data are buffered. An alternative design would be to let the HMA do the pipelining and decide when to give access to the data. In theory, the HMA can compute the optimal size of the first request units that allow the request to start earlier, because the HMA knows the details of the jukebox hardware. However, finding an optimal solution for the scheduling problem is already $\mathcal{NP}$-hard without asking for that additional optimality criterion. Therefore, finding the perfect request-unit sizes is virtually impossible. Additionally, our approach of letting the network servers decide on the structure of the request makes modelling the scheduling problem simpler (see Section 3.5), and simplifies the evaluation of the different schedulers.

Another design decision is, not to provide an explicit OR semantic in the requests. Such a semantic should be useful to express alternatives in the timeline. However, the same results can be achieved by submitting multiple requests. In the database example of the previous section, the database server builds a request for each query result. It then sends all the requests to the HMA in the order resulting from the hit confidence of the results.

3.3 System Architecture

This section presents the architecture of the HMA and gives an overview of the main components of the system. The components are discussed in the following chapters. Figure 3.1 shows the system architecture. 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. The *cache manager* may be physically distributed, as proposed by Brubeck et al. [19], to avoid becoming a bottleneck. The *directory* is a database that contains metadata about the contents of the jukeboxes and can easily be distributed or replicated.

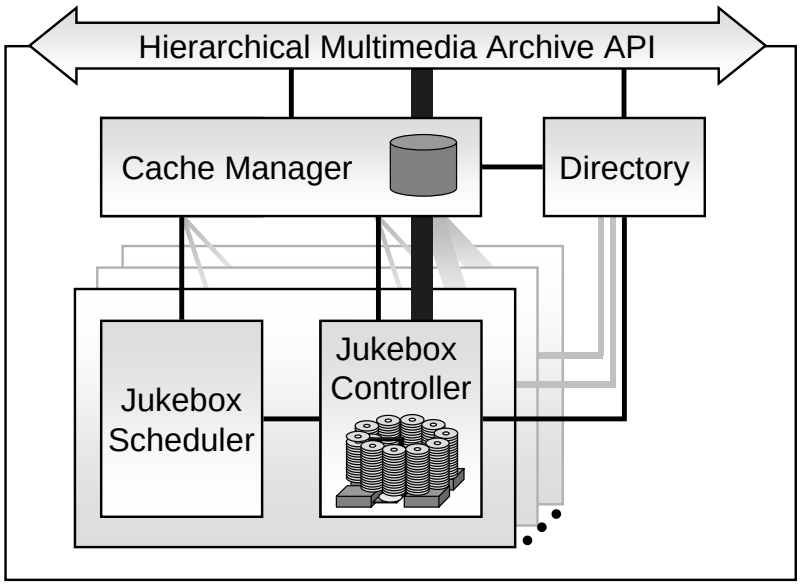


Figure 3.1: Architecture of the Hierarchical Multimedia Archive. The thick lines represent broadband connections for fast data transfer, and the thin lines represent service requests and replies.

When a new request arrives at the system, the *cache manager* filters out the parts of the request that refer to data that 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 the corresponding *jukebox schedulers*.

Figure 3.2 shows the jukebox-scheduler architecture. An important feature of the architecture is the separation of the schedule building and dispatching functionality. We separate both functionalities because their goals are different. Although separating schedule building and dispatching seems a natural design decision, it has not been used in any other jukebox scheduler. In this dissertation we show that this separation leads to a better performance of the jukebox scheduler and simplifies the development of new schedulers.

The *schedule builder*[1] schedules the filtered requests on-line, recomputing 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 to move RSM and stage data into secondary storage. Thus, basically, the goal of the schedule builder is to find a feasible schedule that permits to buffer all the requested data before its deadline, while the goal of the dispatcher

1 We will normally refer to the 'schedule builder' simply as the 'scheduler' and will use the term schedule builder when there is a need to distinguish it from the dispatcher. We also refer to 'schedule building' simply as 'scheduling'.

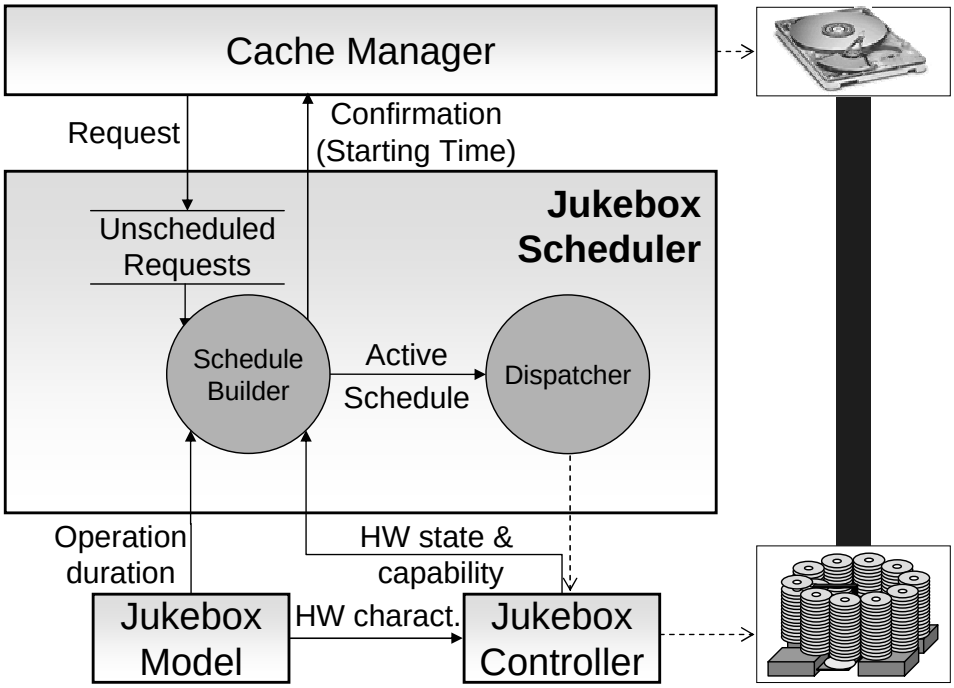


Figure 3.2: Architecture of the Jukebox Scheduler. The thick lines represent broadband connections for fast data transfer, the thin lines represent service requests and replies, and the dashed lines represent commands.

is to dispatch in time. Our functionality separation goes a step further in order to increase the efficiency of the system.

We make the dispatcher responsible for utilizing the jukebox resources in an efficient manner. We use an *early dispatcher* that dispatches the tasks to the jukebox controller as early as possible. The dispatcher may modify the schedules built by the scheduler as long as no task is delayed and the sequence and resource constraints are respected. By assigning this responsibility to the dispatcher, we reduce the job of the scheduler to finding feasible schedules, without having to optimize the use of the resources.

Additionally, using an early dispatcher allows the scheduler to use safe worst-case operation times, knowing that the dispatcher will use the idle times resulting from overestimating the operation times to dispatch other tasks earlier. However, not even an early dispatcher can compensate for using very bad estimates. Therefore, it is still important to use an accurate jukebox model and to build the schedules with the estimates of each operation, and not a generic worst-case operation time.

The combination between Back-to-Front scheduling and early dispatcher used in Promote-IT shows the efficiency of this functional separation. In this case the

schedule builder builds schedules with ‘holes’ (idle times), and the dispatcher uses this ‘holes’ to dispatch tasks early.

However, the early dispatcher cannot solve all deficiencies of a schedule builder. When the scheduler is based on an inappropriate scheduling-problem model such as a periodic model, the dispatcher is not able to eliminate all the unnecessary switches in the schedule.

As shown in Figure 3.2, the schedule builder uses the information provided by the *jukebox model* and *jukebox controller* to determine the parameters of the tasks to schedule. The *hardware model* provides information about the duration of the hardware operations and the capabilities of the different devices in the jukebox. The *jukebox controller* provides a consistent view of the state of the devices in the jukebox, and guarantees that all the data in the jukebox can be read with at least one drive. Additionally, the jukebox controller acts as a schedule verifier, because it only performs valid operations (see Section 8.5). We present the jukebox model and the jukebox controller in detail in the next chapter.

3.4 Cache Manager

The cache manager provides logical administration of the storage space in the cache. The cache manager distinguishes between two types of cache contents: buffered data and cached data. *Buffered data* belong to one or more requests in the system. The data is considered buffered as long as there are requests that own it. When no request owns the data it becomes *cached data*. The space used by cached data can be released whenever the space is needed to buffer data for an active request. The cache manager can use different administration policies, although a simple *least recently used (LRU)* policy performs well.

The storage capacity and bandwidth of the secondary storage must be big enough not to become a bottleneck in the system. The cache manager should be able to store between 10% and 20% of the data in the jukebox to achieve a system in which most of the requested data can be served directly from secondary storage. We derive these numbers from assuming that the data is requested following a Zipf-like distribution [117], because this type of distribution has been detected in most data-access systems. Chervenak [22] shows that the requests for multimedia databases and VoD follow a Zipf distribution, while multiple studies [3, 18] show that web access also follow a Zipf-like distribution.

To reach 10% cache capacity, generally more than one hard disk will be required, e.g., an ASM jukebox with 1744 DVD-ROM requires 1482 GB. This capacity is at least one order of magnitude bigger than the capacities offered by current hard disks. An array of disks can be used, or multiple independent disks, or distributed disks

as proposed by Brubeck et al. [19]. Taking as parameter current optical-disk and hard-disk technology, the bandwidth is not a source of bottlenecks. Managing the secondary storage devices and providing real-time access from secondary storage is out of the scope of this research. We refer to the work of Bosch [14] for a discussion on this topic.

3.5 Generic Schedule Builder

The generic schedule builder provides the structure and basic functionality present in all the heuristic schedulers presented in the dissertation. It finds an appropriate starting time for each incoming request, so that the request can be incorporated to the schedule. The new request and the previously accepted requests have to meet their deadlines. The specific scheduling problem model Π and the heuristic to determine when to stop searching for a solution depends on the scheduling algorithm used.

The scheduler views a request as a set of request units with fixed deadlines in the following way:

$$u'_{ij} = (\tilde{d}_{ij}, m_{ij}, o_{ij}, s_{ij})$$

We assume that the user can start consuming the data of a request unit only once all its data has been buffered, so that we can compute the deadline for each request unit in a request as $\tilde{d}_{ij} = st_i + \Delta\tilde{d}_{ij}$. Therefore, it is important that the network servers request big continuous-media files as multiple request units as discussed in Section 3.2.

The bandwidth does no longer appear in the definition of u'_{ij}, nor is it used in computing $\tilde{d}_{ij}$. However, we can use the bandwidth to divide the original request units into smaller request units or join them into bigger request units. We use this technique in the periodic quantum model (see Section 5.6). The bandwidth is used also by the secondary-storage file system to guarantee real-time access from secondary storage [14].

Let us say that at time t_0 request r_k arrives at a jukebox scheduler. At time t_0 the jukebox scheduler has a set $\mathcal{U}$ of request units from previous requests that have not yet been dispatched to the jukebox:

$$\mathcal{U} \subseteq \{u'_{xy} \mid x < k, y \leq l_x\} \tag{3.1}$$

The goal of the scheduler is to find a feasible schedule for the new set $\mathcal{U}'$ that includes the request units u'_{kj} corresponding to the incoming request r_k:

$$\mathcal{U}' = \mathcal{U} \cup \{u'_{k1}, u'_{k2}, \ldots, u'_{kl_k}\} \tag{3.2}$$

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 puts the request in the list of unscheduled requests until it can schedule it or $maxConf_i$ is reached and the request is rejected.

The scheduler uses an iterative algorithm to schedule a request. The algorithm keeps a list of candidate starting times that it 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 heuristic *guessST* to generate the candidate starting times. This heuristic also determines when to stop searching.

2. Incorporate the request units of r_k into $\mathcal{U}' = \mathcal{U} \cup \{u'_{k1}, u'_{k2}, \ldots, u'_{kl_k}\}$.

3. Model $\mathcal{U}'$ as an instance I of a scheduling problem Π.

4. Try to compute a valid resource assignment for I. If the scheduling algorithm succeeds, the output of this step is a feasible schedule S^x; otherwise $S^x = \varnothing$. The pair (S^x, st_k^x) is incorporated into the list of analyzed solutions.

5. Repeat from step 1 until the stop criteria of the heuristic *guessST* is fulfilled for the list of candidates.

6. Select the best solution. The best solution is the earliest candidate starting time for which step 4 could compute a feasible schedule ($min\{st_k^x \mid S^x \neq \varnothing\}$). 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.

Chapter 5 presents different ways of modelling $\mathcal{U}'$ into a scheduling problem Π. The most important of those models is the minimum switching model because it allows the representation of any kind of jukebox architecture and the creation of schedules that use the resources efficiently. Promote-IT is an efficient heuristic jukebox scheduler that uses the minimum switching model.

3.6 Storing New Data in a Jukebox

To store data in tertiary storage an extension to the HMA is needed that we do not present in this dissertation. We left the 'writing functionality' outside the research scope, because it makes the scheduling-problem models more complex, but does not enrich them in any valuable way.

There are two ways in which new data can be stored in the jukebox: (1) by writing data to a writable RSM that is already stored in the jukebox, or (2) by incorporating an already written RSM into the jukebox through its mailbox.

To handle the first case, the scheduler needs to load the RSM from a shelf into a drive, write the data to the RSM (by copying it from secondary storage) and move the RSM back to its shelf. The duration of the write task depends on the writing speed of the drive. Its computation needs to be incorporated to the hardware model as an extension. The load and unload times will most probably also be different (e.g., the time to recognize an 'empty' CD-R is different that the time needed to recognize a 'burnt' CD-R).

The scheduler must guarantee that the writing is not interrupted. When using the aperiodic schedulers, we only need to represent the writing as one request unit. When using an aperiodic scheduler (e.g., JEQS), the scheduler must assign the resulting task a period of 1, so that the RSM is not unloaded from the drive until the writing task finishes.

To handle the second case, the only difference with a normal read is that the source of the load is different from the destination of the unload, and that the data to read is the directory of the RSM and not a file. The operations to perform are loading the RSM from the mailbox into a drive, extracting the directory structure from the RSM into the directory database, and moving the RSM into an empty shelf. The model should include the estimated time to read the directory of an RSM. Given that the RSM is not yet known in the system, it should use the worst-case time.

3.7 Summary

This chapter presented the architecture of the HMA, with special emphasis on the jukebox scheduler. It described a generic schedule builder, which is present in all the heuristic scheduling algorithms presented in the dissertation. It defined the structure of the requests and the interface between the users and the system. It also presented some usage scenarios showing how the HMA acts as real-time file-system, while the network servers may provide stream-oriented services.

Chapter 4

Tertiary-Storage Hardware

In order to schedule the resources, we need to predict how long the operations with these resources will take. We predict operation times with a hardware model, which we also use to time the jukebox simulator. The scheduler basically needs to know the time required to load an RSM in a drive, read data from the RSM and unload it. Computing these times is not straightforward. Many factors are important: the type of RSM, the drive, the jukebox robotics, the number of robots in the jukebox and the location of the shelves and drives in the jukebox. Our hardware model has separate sub-models for the RSM, the drives, the robots and the jukebox. Together, the sub-models can describe any type of jukebox architecture.

To model a jukebox, we must understand how it behaves. Therefore, in the next section we give an overview of jukebox technology. Our focus is mainly on optical and magneto-optical jukeboxes, because, in contrast to magnetic tapes, these types of media allow easier random access to the data. We then present the hardware model. Finally, we briefly describe the jukebox controller and show how it uses the jukebox model.

4.1 Jukebox Technology

Tertiary-storage jukeboxes are composed of the following schedulable resources:

Removable Storage Media (RSM) where the data is stored. The RSM are stored in *shelves*, also called *slots*.

Drives to read the data from the RSM. The data from the drives can be transferred directly to secondary storage or to memory through a high-bandwidth connection (see [69] for an overview of storage interfaces). Although Figure 4.1 shows all the drives using the same bus, they could be attached to different buses.

Robots to move the RSM from the shelves to the drives and vice-versa.

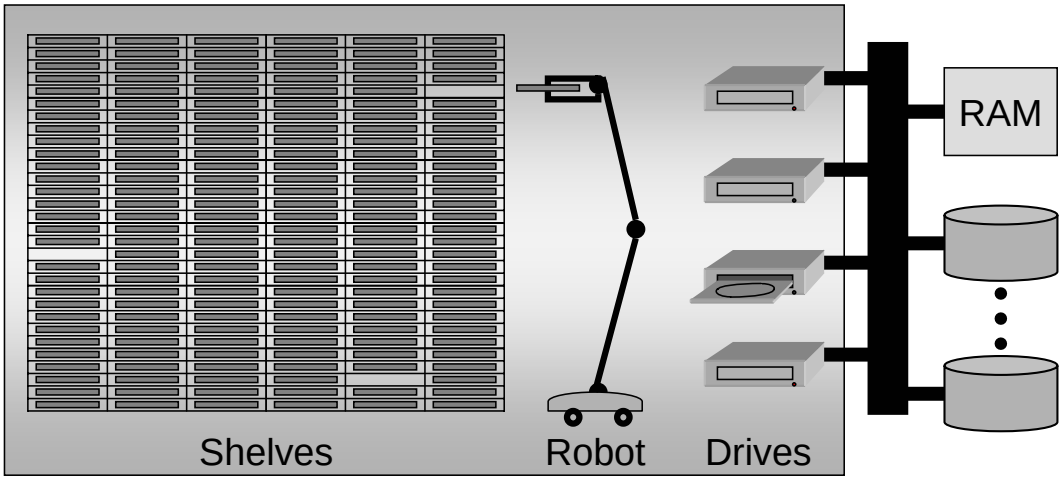


Figure 4.1: Architecture of a generic jukebox with four drives and one robot.

There are jukeboxes for different types of RSM. Older jukeboxes used mainly magnetic tapes. Newer jukeboxes use optical disks—CD-ROM, DVD-ROM, DVD-RAM—and magneto-optical disks. We are mainly concerned with jukeboxes for optical and magneto-optical disks, because (a) this is the type of hardware that we have available in our laboratory, (b) disks are better suited for random access than tapes, and (c) disks can be loaded and unloaded faster than tapes. Given that the requests for the HMA may include request units for small files or parts of files, it is important that we are able to access the data in an RSM in a random manner. Additionally, the faster we are able to switch RSM the better.

Most optical jukeboxes can store several types of optical disks at the same time and read all of them with the same drives. Most optical drives can read 'earlier' disk technologies, e.g., most DVD-ROM drives can read CD-ROM, and the DVD-RAM drives can read both CD-ROM and DVD-ROM. However, the performance of a drive differs among different types of disks. For example, the transfer and access speed of DVD-ROM drives is different when reading DVD-ROM and CD-ROM. Additionally, there may even be a difference when reading CD-ROM with different reflective layers and dyes.

A jukebox can be equipped with different drive types, leading to configurations in which some RSM can only be read by a subset of the drives. For example, in a jukebox with two DVD-ROM drives and two DVD-RAM drives the DVD-ROM stored in the jukebox can be read with any of the drives, but the DVD-RAM can only be read with the DVD-RAM drives. Such a situation can arise, for example, from gradual upgrades of the jukebox hardware.

The number of shelves is between two and three orders of magnitude bigger than the number of drives. The time needed to switch the RSM loaded in a drive is in the order of seconds. In optical jukeboxes, the average switch time is in the order of tens of seconds, while in tape jukeboxes it can reach hundreds of seconds.

Table 4.1 provides some characteristics of some jukeboxes available at present in the market. The data was taken from the product specifications of the manufacturers [7, 27, 42, 55, 59, 84, 102, 104, 105].

The switch time reported by the manufacturers does not take into account the time needed to spin down the RSM in the drive, eject it, close the drive once the new RSM is loaded, spin up, and recognize the RSM. However, the time needed to perform these activities can not be ignored. For example, most DVD-ROM drives available at present need approximately 12 seconds to close the drive and recognize the disk.

Some jukeboxes have a robot with a *dual-picker*, which can carry two disks at the same time. The robot can unload a drive and immediately load a new disk. Most of the optical jukeboxes in the table are capable of using double-sided DVD-ROM and DVD-RAM by turning them over. The turning is generally done while moving the disk between the slot and the drive and does not add a considerable extra delay. However, in order to read both sides of a disk, the disk needs to be unloaded, turned over by the robot and loaded again. In Chapter 5 we discuss the implications of not being able to read both sides of a disk without a robot intervention.

The way in which the shelves and drives are placed in the jukebox varies considerably between the different models. In some jukeboxes, the shelves and drives are placed as in a book shelf or matrix, e.g., ASM jukeboxes. In others, they are placed in a concentric fashion around the robotic arm, e.g., the DAX jukeboxes. Some jukeboxes are built like big PC-towers and the shelves and drives are in two columns at the front and back of the tower, e.g., the JVC jukeboxes. The robot mechanism also differs between different models, mainly dependent on the way the shelves and drives are placed. The drives are generally at the bottom of the jukebox.

In some jukeboxes, some space can be used either to put drives or slots, and therefore increasing the number of drives reduces the number of shelves. In the case of the NSM series, each additional drive reduces the number of shelves by 15. In the case of the ASM jukeboxes, the drives can only be added in fixed-sized blocks. A block with 6 drives reduces the number of shelves by 24. In the case of the JVC jukeboxes, the drives can be added in blocks of 3 drives replacing 50 shelves. The Pioneer jukebox can be configured with any even number of drives between 2 and 16 drives. Each pair of drives reduces the number of RSM by 50.

In most jukeboxes, the RSM are stored in *removable magazines*. The magazines vary in size from 10 to 50 disks, depending on the jukebox. Most jukeboxes provide the possibility to import and export one RSM through a *mailbox* as well.

Model	Producer	RSM Type	Robots	Shelves	Drives	Switch Time (sec.)
DVD 1400	ASM	CD-ROM DVD-ROM DVD-RAM	1[a]	1600	48	6–12
				1600	24	
				1720	18	
				1744	12	
MC-8600U	JVC	CD-ROM DVD-ROM DVD-RAM	1	600	6	avg. 10
				550	9	
				500	12	
NSM6000	DISC	CD-ROM DVD-ROM DVD-RAM	1	605	1	avg. 6
				515	7	
				410	14	
DRM-7000	Pioneer	CD-ROM DVD-ROM	1[b]	720	2	≤ 9
				570	8	
				370	16	
D480	Plasmon	CD-ROM DVD-ROM DVD-RAM	1[a]	440	2	avg. 4
					4	
					6	
smartDAX700	DAX	CD-ROM DVD-ROM DVD-RAM	1[b]	720[c]	4	avg. 7
SA-1600	Kubota	DVD-RAM (in cartridge)	1	436	2–16	6.15–10
				904	2–32	
				1840	2–64	
AV-1450	Asaca	CD-ROM DVD-ROM DVD-RAM	1[a]	1100–1450	1–24	avg. 3.5
Orion D1050	Disc	Magneto Optical	1[a]	910–1050	4–32	N/A
G638	Plasmon	Magneto Optical	1[a]	638	6	avg. 6.4
					10	
					12	
2200mx	Hewlett-Packard	Magneto Optical	1	238	4	avg. 6.5
					6	
					10	

Table 4.1: Hardware specification of some commercial jukeboxes. Notation: (*a*) The robot may use a dual picker. (*b*) The robot cannot handle double-sided RSM. (*c*) The jukebox does not provide removable magazines.

4.1.1 Optical and Magneto-optical Disks

The data on an optical disk is stored using indentations on a polycarbonate plastic layer and read using a very low power laser. The indentations are called *pits*, the areas around them are called *lands*. The pits and lands do not correspond directly to 1's and 0's to avoid having very small artefacts on the disk, which may produce read errors. Instead a modulation scheme is used that encodes a byte of data (8 bits) into a larger number of *channel bits*, allowing the pits and lands to be longer. The original scheme, called *eight-to-fourteen (EFM)*, encodes eight bits into fourteen channel bits. Additionally, three extra channel bits are added to the fourteen to ensure that the pit length is never shorter than three or longer than eleven channel bits. DVD-ROM uses an eight-to-sixteen modulation scheme.

The basic CD (compact disc) is 120 mm in diameter and a 1.2 mm thick sandwich of three coatings: a back layer of clear polycarbonate plastic, a thin sheet of aluminium and a lacquer coating to protect the disk from external scratches and dust. There are multiple extensions of this simple 'disk definition'. The two main extension are CD-ROM (compact disc read only memory) and DVD (digital versatile disc), with their write-once and rewrite versions. The potential successors of the DVD offer storage capacities between 20 and 27 GB per layer. The higher storage capacity is primarily the result of using blue laser [47].

The data is stored in a spiral track circling from the inside to the outside of the disk. With each rotation the radius of the spiral grows with a constant amount called *track pitch*. Figure 4.2 shows a representation of the spiral and a track in the disk.

A player reads information from the disk's spiral track of pits and lands, starting from the centre and moving to the outer edge. It fires a laser and interprets the reflected light. Light reflected from a pit is 180 degrees out of phase with the light from the land. The differences in intensity are measured by photo-electric cells and converted into electrical pulses.

A *magneto-optical disk (MO)* is a read/write disk that combines magnetic and optical technologies. It uses laser light to enable a relatively large magnetic write head at a large flying height to write small magnetic domains.

The magnetic coating used on MO media is designed to be extremely stable at room temperature, making the data unchangeable unless the disk is heated to above a temperature level called the Curie point (approx. $200°C$). Once heated, the magnetic particles can easily have their direction changed by a magnetic field that is generated by the read/write head. Information is read using a less powerful laser, making use of the Kerr Effect, where the polarity of the reflected light is altered depending on the orientation of the magnetic particles.

A *DVD-RAM* uses phase-change recording technology. The layer that records the data has the property to change between amorphous and crystalline states by

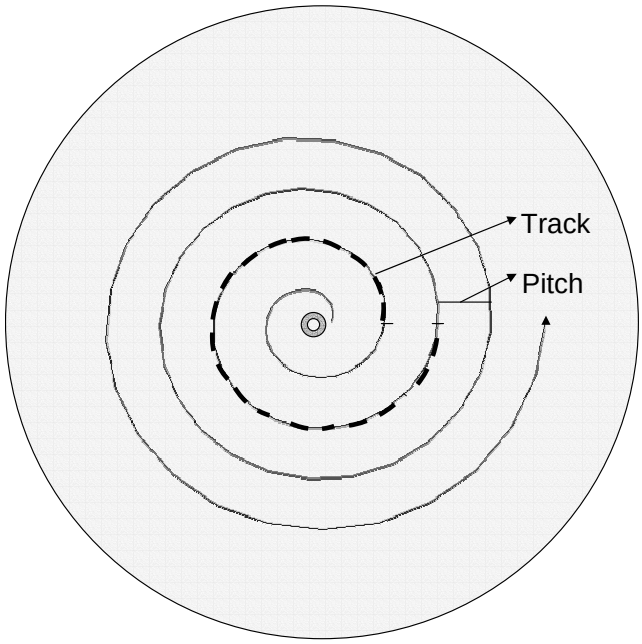


Figure 4.2: Spiral of an optical disk.

controlled heating and cooling [95]. The medium is in polycrystalline state during reading. Recording is done by rapidly heating the material with a laser and letting it cool quickly to the amorphous state. To revert to the polycrystalline state, the alloy is heated to a temperature just above its crystallization point.[1]

The type of media determines the size of the pits and the track pitch. The smaller these values are the more data can be stored on the same surface. Figure 4.3 shows the surface of a CD-ROM and a DVD-ROM. The pits in a CD-ROM are $0.5\,\mu m$ wide, between $0.83\,\mu m$ and $3\,\mu m$ long and $0.15\,\mu m$ deep, and the pitch is $1.6\,\mu m$. A DVD-ROM can store more data and, thus, the sizes are smaller. The pits in a DVD-ROM have a minimum length of $0.4\,\mu m$ and the pitch is $0.74\,\mu m$. These values are even smaller for Blu-ray Discs.

The official storage capacity of a CD-ROM is 650 MB, equivalent to 74 minutes of audio. However, through over-burning and reducing the track pitch, higher capacities can be obtained. Currently it is common to use 700 MB (80 minutes), 800 MB (90 minutes) and 870 MB (99 minutes) disks. There are other ways of increasing the capacity of CDs. The double-density CD-R/RW, for example, has a capacity of 1300 MB. In this case the track pitch is $1.1\,\mu m$ and the minimum pit length is $0.623\,\mu m$.

1 Except when explicitly mentioned we treat DVD-RAM as a normal part of the DVD-ROM family.

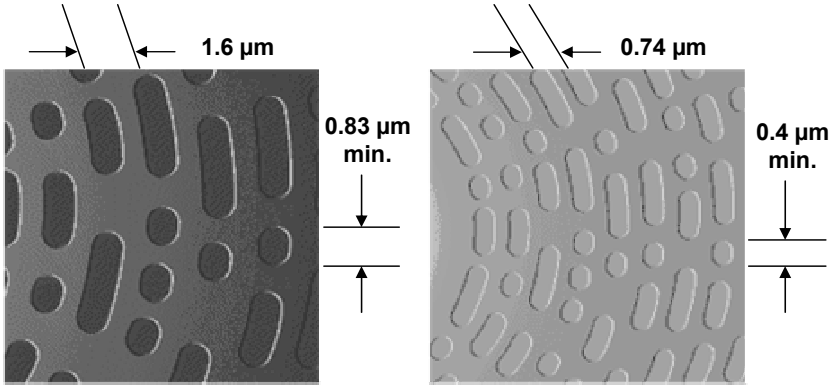


Figure 4.3: Surface of a CD-ROM (left) and a DVD-ROM (right).

DVD-ROM provides up to 17 GB of storage. The data on a DVD-ROM can be stored in one or two layers. A single layer disk can store 4.7 GB per side. A double-layer disk can store 4.25 GB on each layer, resulting in a storage capacity of 8.5 GB per side. When using two layers, instead of one, the pit length is slightly bigger, 0.44 μm, to avoid interference. The second layer can contain data recorded 'backward', or in a reverse spiral track. With this feature, it takes only an instant to refocus a lens from one reflective layer to another. Additionally a DVD-ROM can store data on both sides of the disk. The disadvantage of double-sided disks is that they have to be turned over.

Magneto-optical disks offer different capacities, varying from 128 MB to 9.1 GB. When using double-sided disks, these also have to be turned over. Magneto-optical disks have smaller seek times than optical disks, because the read/write head is lighter than that of an optical disk.

There are many simple technology improvements, which are not yet widely used. One of them is the Calimetrics' Multilevel (ML) Recording technology, which uses different depths of the pits in CD-ROM and DVD-ROM to store more information at the same location. On conventional optical disk only lands (0) and pits (1) are distinguished. At this moment ML recording uses 8 levels—or 3 bits—at each location. An important feature of this technology is that manufacturers only have to replace one chip in their drive designs. Conventional units have the laser pickup connected to a chip that converts the measured reflection to 0 or 1—the replacement chip converts the same input signal to 3-bit sequences.

Another technological improvement is to use fluorescent multilayer optical-data. The storage layers are coated with a fluorescent material. When the light hits the layer fluorescent light is emitted that can transverse adjacent layers undisturbed. In this way, many layers can be used in a disk, avoiding the problems of interference. Constellation 3D claims that up to a hundred layers are feasible [96].

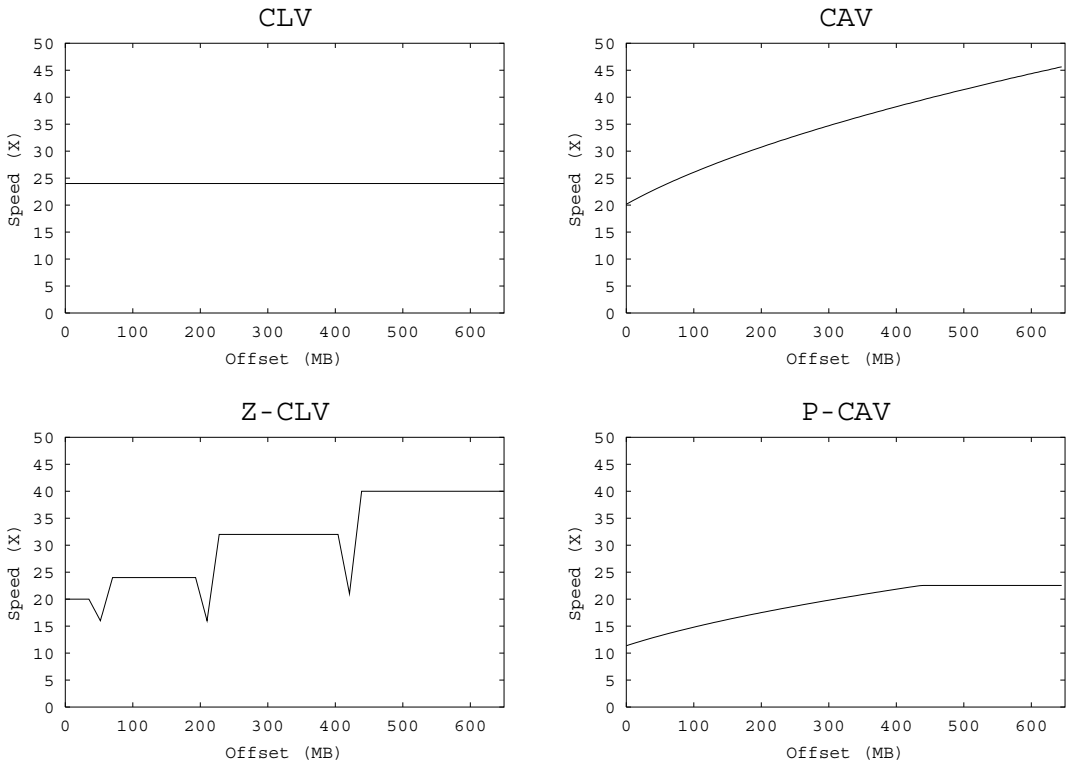


Figure 4.4: Optical-drive technology. The graphics plot the transfer speed as a function of track number. CLV stands for constant linear velocity, CAV stands for constant angular velocity, Z-CLV stands for zoned CLV, and P-CAV stands for partial CAV.

Drive Technology

The first generation of CD-ROM drives used *constant linear velocity (CLV)* technology, which adjusts the spin speed of the disk to provide a constant transfer speed. This is identical to what the audio-CD players do.

The first generation of single-speed (1X) CD-ROM drives were based on the design of audio CD drives, employing *constant linear velocity (CLV)* technology to spin a disk at the same speed as an audio CD with a resulting bandwidth of 150 KBps. Variable spin is necessary to cope with audio data, which is always read at single-speed, whatever the transfer rate for computer data. While audio disks have to be read at single-speed, there is no limit for the CD-ROM. Indeed, the faster the data is read, the better.

The speed of a CD-ROM drive is normally denoted as ‘nX’ and it means it provides a maximum bandwidth of *n* times 150 KBps. The same terminology is used for DVD-ROM, except that in this case 1X is equivalent to 1250 KBps, which is needed to play MPEG-2 encoded video. The top speed of DVD drives in the mar-

ket at the end of 2002 is 16X, which is equivalent to a maximum transfer speed of nearly 20 MBps and an average transfer rate speed of 14 MBps, but this value grows every few month.

Since there are more sectors on the outside edge of the disk than in the centre, CLV uses a servo motor to slow the spin speed of the disk toward the outer tracks in order to maintain a constant data-transfer rate over the laser read head. As the speed of a CLV drive increases, access times often suffer as it becomes harder to perform the abrupt changes in spindle velocity needed to maintain a constant high data-transfer rate, due to the mass inertia of the disk itself.

The next step in the CD-ROM drive evolution was using *constant angular velocity (CAV)* to spin the disks. A drive using constant angular velocity transfers data at a variable rate while the drive spins at a constant rate. This results in increased data transfer rates and reduced seek times as the head moves toward its outside edge.

The first drives using CAV were using a mix of CLV and CAV called *partial constant angular velocity (P-CAV)*. In a P-CAV drive, CAV is used for reading close to the centre of the disc while the drive switches to CLV mode for reads closer to the outer edge. The main problem of P-CAV drives is the need to step the motor speed up and down in response to moving the head. Full CAV was not used at the beginning for faster drives, because the chips were unable to handle transfer speeds faster than 16X.

With the advent of full CAV drives, the speed boosted very fast. A CD-ROM 56X CAV drive, for example, spins the disk at 10000 rpm, resulting in a bandwidth of 8400 KBps in the outer tracks. A problem with spinning disks at such high speeds is excessive noise and vibration, often including loud hissing noise caused by air being forced out of the drive casing by the spinning disk. Because the disk is clamped at its centre, the most severe vibration occurs at the outer edges of the disk—at exactly the point where the decoding circuits have to handle the highest signal rate. Some drives however have spindle motors with anti-vibration mechanisms to try to solve this problem. Another problem of spinning disks at even higher speeds is that they disintegrate. The technicians of Atlas Copco AB [1] present very interesting experimental results showing how the disks explode at rotation speeds of 28000 rpm.

Another technology is *zoned constant linear velocity (Z-CLV)*, which splits the disk into several CLV zones with different rotation speeds. The advantage of this technology lies in better seek times, because the spindle motor does not need to spin up and down so much when seeking across the disk. This technology is especially popular for CD writers and DVD-RAM.

Figure 4.4 shows the transfer speed for the four drive technologies described.

True-X drives based on multi-beam technology achieve high transfer speeds using a CLV rotation system. Multi-beam technology uses multiple laser beams to read

multiple tracks of the disk at once, resulting in both higher transfer rates and lower rotational speeds. A diffracted laser is used to illuminate the disk with 7 discrete beams. The central beam is used for focus and tracking in the same way conventional players work. The 7 beams are spaced evenly on both sides of the central beam and allow a multi-beam detector to pick up the data from 7 tracks.

In 1998 Kenwood introduced the first True-X CD-ROM drives and they received very good reviews. However, Kenwood discontinued the production after many users' complaints. Another multi-beam drive was presented by Afeey on the Cebit 2001. It could read DVD-ROM at 25X and CD-ROM at 100X. However, no such drive has yet appeared in the market.

4.1.2 Magnetic Tapes

For decades tapes have been the main tertiary storage medium. Their high storage capacities and low cost per megabyte made them an appropriate media for backup, which was the main application of tertiary storage. However, tapes are ill suited to be used in applications where random access and real-time guarantees are needed, as is our case. In the following paragraphs we show why it is difficult to build a flexible multimedia archive based on tape technology.

The access time of tapes is very high, when compared to that of optical and magneto-optical disks—tens or hundreds of seconds, against tens of milliseconds. Additionally, accurately predicting the access time and transfer time is very complex, if not impossible. The time needed for loading and unloading a tape in a drive is much higher than for optical disks (generally tens of seconds). A reason for this is that many tapes have to be rewound before unmounting.

High bandwidths can be achieved with high-end hardware. However, the performance of this hardware does not justify its high price when compared to optical disks, especially because the access time is still orders of magnitude higher. Last but not least, the wear tapes suffer from random access is considerable.

We do not present a profound analysis of tape technology. Through the years much has been written about magnetic tape and their performance. Hillyer et al. [43] and Chervenak [22] provide a good overview of tape technology. Johnson et al. [53] present a benchmark methodology for tapes. The PC technical guide [80] gives an on-line overview of current technology and products in the markets.

Magnetic tapes store data as small magnetized regions. The tape is composed of magnetic material deposited on a thin flexible substrate. The tape is rolled along a reel. Therefore, the thinner the substrate the more tape can be stored per volume. However, a thin substrate is more prone to distortion and breaking when submitted to high accelerations and many start and stop operations.

Tapes are categorized by the track orientation and the tape width. The most important track orientations are *linear* and *helical*. In a linear tape the data is stored in longitudinal tracks through the length of the tape. A popular recording variety of linear tapes is *serpentine* tapes, in which the data is recorded in an 'S' shape. The data is recorded along the length of the tape, then the direction is reversed and the head shifts sideways to record another track. Helical scan tape drives wrap the tape around a cylinder that contains the read/write heads and rotates the cylinder rapidly while the tape is transported relatively slowly. Helical tapes provide large storage capacities, because the track and linear density are high. They also provide high transfer rates, because of the high relative speed of the head across the tapes.

Tapes are mainly designed to stream data. There is a start/stop problem whenever the application interrupts the stream and then tries to resume it. After the drive stops, it reverses direction to position the heads in the inter-record gap before it can resume the read. A solution that many drives provide to this problem is that they go on reading and caching the data into secondary storage, which implies that secondary storage buffering must be attached to the tape drive.

Estimating the locate (or seek) time is extremely difficult when working with tapes. One reason is that if there is an error while writing a block of data, the block is rewritten again in the next portion of the tape. This makes it difficult to predict the exact location of the data on the tape. Worse still, when the drive tries to read the faulty block it may try to reread it many times, before it concludes it is faulty and tries with the next block.

Hillyer et al. [44] present a model to estimate the locate time on serpentine tapes. They show that performing this estimate is complex. One reason for the complexity is that the lengths of the tracks are not equal. Another reason is that the drives store the location of particular blocks in the tape to which they can jump when performing seeks. The drive uses the high-speed seek to jump to a known block before the desired block and then reads the tape until it reaches the desired block. They show that an individual model needs to be made for each tape, which takes several hours. Using the model built for another tape can result in bad predictions. Therefore, there is not a unique locate time model for a given tape drive technology, but there must be an individual model for each tape in the jukebox. Estimating the locate time for linear tapes is easier, because the seek time is linear in the distance the tape must travel [53].

4.2 Hardware Model

We now present an analytical model to compute the time to perform each jukebox operation. Figure 4.5 shows the structure of the hardware model using a UML class

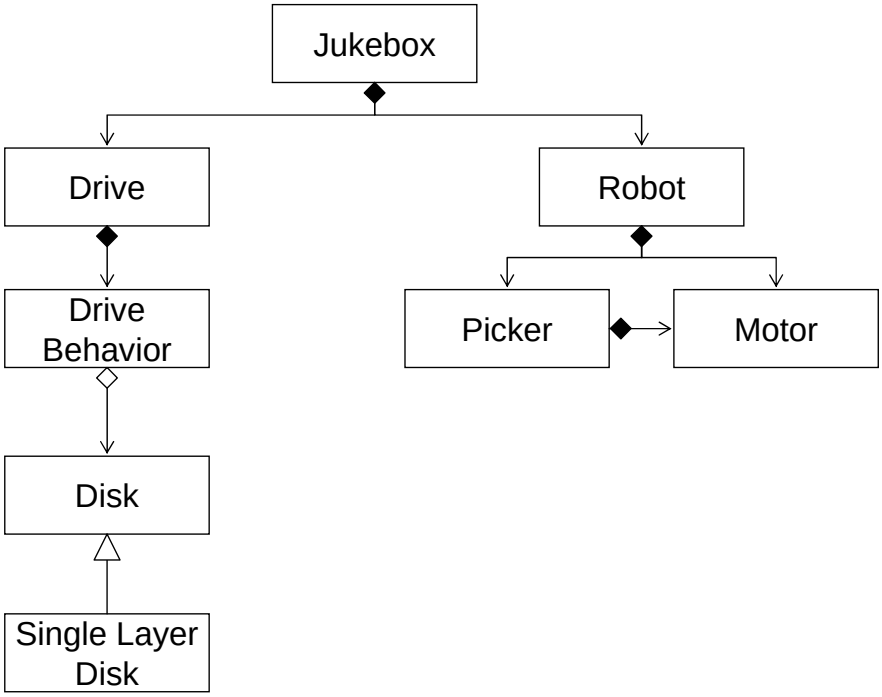


Figure 4.5: Structure of the hardware model. The different parts of the model are represented as classes in a UML class diagram.

diagram. At the higher level we have the jukebox model that provides the higher level functions that compute the time to load and unload a disk and to read data from a disk. These times are computed using the specific model of the drive and robot involved. As the behaviour of a drive varies depending on the type of disk loaded in it, the drive model uses different *drive behaviour* models to compute the times for each type of disk. We model a robot with independent motors, one for each axis of the robot, and with one or multiple pickers. We use separate models for the motors and the pickers.

The rest of the section describes each part of the model in a bottom-up fashion. Subsection 4.2.1 presents the disk model. The model of the disk is used in Subsection 4.2.2 to model the drives. Subsection 4.2.3 presents the essentials of the robot and jukebox model. These models are described in further detail for a specific jukebox model.

In the rest of this section we use a mix of $\mathcal{Z}$ notation [103] and normal mathematical formulae to specify the functionality provided by the model. The $\mathcal{Z}$ notation is used to describe typed parameters and the operations of the model, and also to describe logical relations among them. We use the normal mathematical notation for better readability of complex formulae.

4.2.1 Disk Model

This section presents the model of an optical or magneto-optical disk. The main goal of the model is to provide a mapping from an offset on the disk, given in bytes, to a track number. The drive model, in turn, uses the track number to compute the access time, transfer time, etc. The model is restricted to single-layer disks, because the extension needed to convert this model in a model for double-layer disks does not provide more insight into the issues that are relevant to the model.

SingleLayerDisk

$R_0 : PhysicalSize$

$R_f : PhysicalSize$

$p : PhysicalSize$

$BS : Size$

$N : \mathbb{N}$

$Tracks : \mathbb{N}$

$length : TrackNumber \rightarrow PhysicalSize$

$sumlength : TrackNumber \rightarrow PhysicalSize$

$TL : PhysicalSize$

$BL : PhysicalSize$

$block : Offset \rightarrow BlockNumber$

$track : BlockNumber \rightarrow TrackNumber$

$size : TrackNumber \rightarrow Size$

We model an optical disk by viewing the data as stored in concentric circles instead of a spiral and we show that this approximation is very accurate. Each circle represents a track. The exact model of the data stored is an Archimedes' spiral.

The model takes as parameters the inner and outer radius defining the area on the disk the data can be stored, represented as R_0 and R_f, respectively, the track pitch (p), the block size in bytes (BS) and the maximum number of blocks that can be stored in a disk (N).

We compute the number of tracks on the disk ($Tracks$) using the inner and outer radius of the recordable area and the track pitch. The length of a track i ($length(i)$) is a function of the distance from the inner radius. The length of the first x tracks ($sumlength(x)$) is simply the sum of the length of the tracks. Therefore, we can compute the total length of the recordable area of the disk (TL) as $sumlength(Tracks)$.

$$Tracks = \frac{R_f - R_0}{p} \tag{4.1}$$

$$length(i) = 2\,\pi\,(R_0 + i\,p) \tag{4.2}$$

$$\begin{aligned} sumlength(x) &= \sum_{i=0}^{x} length(i) = \sum_{i=0}^{x} 2\,\pi\,(R_0 + i\,p) \\ &= (x+1)\,2\,\pi\,R_0 + 2\,\pi\,p\,\frac{x\,(x+1)}{2} \\ &= \pi\,p\,x^2 + \pi\,(p + 2\,R_0)\,x + 2\,\pi\,R_0 \end{aligned} \tag{4.3}$$

$$TL = sumlength(Tracks) \tag{4.4}$$

We now show that the length of the data track computed with the concentric circles approximation is very near the exact length of the spiral when using CD-ROM. When using DVD-ROM the approximation is even better, because the track pitch is smaller. Applying the formulae to the CD-ROM specification ($R_0 = 22.5\,mm, R_f = 58\,mm, p = 1.6\,\mu m$), we obtains $5.611249 \times 10^9\,\mu m$ as value for *TL*, while the exact length of the spiral is $5.6113 \times 10^9\,\mu m$.

In most cases we need to know the track number of a given offset *x* in the disk. The smallest part of data that can be accessed independently is a *sector* or *block* [28]. Therefore, we compute the position of a particular byte as the position of the beginning of the block which contains the byte. Byte *x* belongs to block $\lfloor \frac{x}{BS} \rfloor$, where *BS* is the block size in bytes. We compute the block length *BL* using the total length of the disk *TL* and the maximum number of blocks *N* that can be stored in the disk using Equation 4.5 and compute the track to which byte *x* belongs as the track where the block containing *x* is as shown in Equation 4.8. In Equation 4.6 *s* is the block number corresponding to byte *x*. Finally, we compute the size of a track *x* in bytes (*size*(*x*)) using Equation 4.9.

$$BL = \frac{TL}{N} \tag{4.5}$$

$$s = \frac{sumlength(s)}{BL} \tag{4.6}$$

$$s\,BL = \pi\,p\,track(s)^2 + \pi\,(p + 2\,R_0)\,track(s) + 2\,\pi\,R_0 \tag{4.7}$$

$$track(s) = \left\lfloor \frac{-\pi\,(p + 2\,R_0) + \sqrt{(\pi\,(p + 2\,R_0))^2 - 4\,\pi\,p\,(2\,\pi\,R_0 - s\,BL)}}{2\,(\pi\,p + 2\,\pi\,R_0)} \right\rfloor \tag{4.8}$$

$$size(x) = \frac{length(x)}{BL}\,BS \tag{4.9}$$

4.2.2 Drive Model

The goal of the model is to predict the time needed to transfer data and access data on a medium using a given drive. It must also predict the time needed to open

and close the drive, both when the drive is loaded and empty. It also predicts the time it takes to spin-down the drive. This model covers the most important drive technologies—CAV and CLV. The other two technologies we discussed in Subsection 4.1.1 (P-CAV and Z-CLV) are covered by simple extensions to this model.

$$TechnologyType ::= CLV \mid CAV$$

The drives in a jukebox may be different and differ on the type of RSM they can handle. When a drive can handle multiple types of media, the performance with each type may be different. Therefore, we have a *DriveBehaviour* for each drive model when handling the possible RSM types.

Drive

$$Models : \mathbb{F}(MediaType \times DriveBehaviour)$$
$$behaviour : MediaType \rightarrow DriveBehaviour$$

The *DriveBehaviour* gets the information about the disk from the model presented in the previous subsection. If the drive uses CLV technology, the transfer speed ($s_{transfer}$) is constant. Instead if the technology is CAV, the rotation speed (s_{rotate}) is constant. In one rotation the drive can read the data of a full track. Therefore we can compute the transfer speed if we know the rotation speed and vice-versa, as indicated in the first predicate of the $\mathcal{Z}$ specification.

We compute the transfer time given the offset o and size s of data to transfer in the following way:

$$t_{transfer}(o, s) = \frac{2\,s}{s_{transfer}(t_o) + s_{transfer}(t_d)} \tag{4.10}$$

where

$$t_o = D.track(D.block(o))$$
$$t_d = D.track(D.block(o + s))$$

DriveBehaviour

$D : SingleLayerDisk$

$technology : TechnologyType$

$ConstantVelocity : Speed$

$\omega_u : Acceleration$

$\omega_d : Acceleration$

$a_{move} : Acceleration$

$t_{settle} : Time$

$T_{spinup} : Time$

$T_{spindown} : Time$

$T_{inactive} : Time$

$T_{load} : Time$

$T_{recognition} : Time$

$T_{eject} : Time$

$s_{transfer} : TrackNumber \rightarrow Speed$

$s_{rotate} : TrackNumber \rightarrow Speed$

$t_{rotate} : TrackNumber \rightarrow Time$

$t_{changespin} : TrackNumber \times TrackNumber \rightarrow Time$

$t_{move} : TrackNumber \times TrackNumber \rightarrow Time$

$t_{seek} : TrackNumber \times TrackNumber \rightarrow Time$

$t_{transfer} : Offset \times Size \rightarrow Time$

$t_{access} : Offset \times Offset \rightarrow Time$

$t_{open} : Content \rightarrow Time$

$t_{close} : Content \rightarrow Time$

$(\forall\, track : TrackNumber \bullet$
$\quad s_{transfer}(track) = s_{rotate}(track) * D.size(track) \wedge$
$\quad technology = CLV \Rightarrow s_{transfer}(track) = ConstantVelocity \wedge$
$\quad technology = CAV \Rightarrow s_{rotate}(track) = ConstantVelocity)$

$(\forall\, source : Offset;\ dest : Offset \bullet$
$\quad t_{access}(source, dest) =$
$\quad\quad t_{seek}(D.track(D.block(source)), D.track(D.block(dest)))+$
$\quad\quad t_{rotate}(D.track(D.block(dest))))$

$(\forall\, t_0 : TrackNumber;\ t_1 : TrackNumber \bullet$
$\quad t_{seek}(t_0, t_1) = max\{t_{move}(t_0, t_1), t_{changespin}(t_0, t_1)\})$

Equation 4.10 uses the track where the transfer must begin, given by the offset, and the track where the transfer must end, given by the offset plus the number of bytes to read. This equation can be used both for CLV and CAV drives. We deduced this equation from the formulae to compute the time it takes to a uniformly accelerating vehicle to travel a given distance shown in Equation 4.13. We model the number of bytes to read (size) as the distance the head has to travel while reading data.

$$Acceleration = \frac{FinalSpeed - InitialSpeed}{Time} \tag{4.11}$$

$$Distance = InitialSpeed \times Time + \frac{Acceleration \times Time^2}{2} \tag{4.12}$$

$$\Rightarrow time = \frac{2 \times Distance}{InitialSpeed + FinalSpeed} \tag{4.13}$$

Although we do not show the model corresponding to a P-CAV and Z-CLV drive, we shortly explain how they compute the transfer time. A P-CAV drive uses CAV technology to read the first k tracks and CLV technology read the other tracks. To represent a P-CAV drive we use the constant angular velocity corresponding to a CAV drive and provide an extra parameter indicating the track number where the drive changes from CAV to CLV technology. We compute the transfer time as the sum of the transfer time of the bytes falling in the tracks read using CAV technology and the bytes falling in the tracks in which CLV is used. The function to compute the transfer time for a Z-CLV drive uses different cases for the ranges where each speed holds. This is an extension of Equation 4.10.

The *access time* (t_{access}) is the time that elapses between issuing a random read command and starting to read the data from the disk. The access time is computed as the sum of the *seek time* (t_{seek}) to go from the origin to the destination, plus the *rotational latency* (t_{rotate}) at the destination track. This equation is shown in the second predicate of the $\mathcal{Z}$ specification. To compute the access time we need to know the position of the head at the time—origin track—and the destination track. Given that the function to compute the access time takes as parameters the offset of the last byte read and the offset of the next byte to read, we compute the tracks corresponding to these offsets.

The rotational latency (t_{rotate}) is the time spent waiting for the correct byte to rotate to the position under the head. The rotational latency is directly related to the disk rotation speed (s_{rotate}), as expressed in Equation 4.14.

$$t_{rotate}(track) = s^{-1}_{rotate}(track) \tag{4.14}$$

The seek time (t_{seek}) is the time needed to position the reading head at the appropriate track and settle once the track has been reached. The seek time involves the

head movement and the time needed to spin up or down the rotation speed of the motor in the case of CLV drives. In CLV drives different motor speeds are used to read data from the disk. Therefore, the seek time of CLV drives in general higher than that of CAV drives.

When using a CLV drive, the rotation speed on the inner tracks is higher than on the outer tracks. Moving between tracks implies accelerating or decelerating the rotation of the disk. The acceleration is called *spin-up* and occurs at an angular acceleration of ω_u. The deceleration is called *spin-down* and occurs at an angular acceleration of ω_d [25]. We compute the time it takes to change the spin speed ($t_{changespin}$) using the following function:

$$t_{changespin}(t_0, t_1) = \begin{cases} \frac{s_{rotate}(t_0) - s_{rotate}(t_1)}{\omega_d} & \text{if } t_0 < t_1, \\ \frac{s_{rotate}(t_1) - s_{rotate}(t_0)}{\omega_u} & \textit{otherwise}. \end{cases} \tag{4.15}$$

The acceleration and deceleration values are in general not provided in the specification of a drive. However, the manufacturers generally report the *spin-down time* ($T_{spindown}$) and *spin-up time* (T_{spinup}). The spin-down time is the time it takes to stop the spindle motor. The spin-up time is the time it takes the drive to start reading data again after it has spun down.

We can compute the values of ω_d and ω_u acceleration needed to go from rotation speed 0 to full rotation speed or vice-versa. The maximum speed in a CLV drive is needed for reading data from the inner tracks. We use the formula to compute the acceleration presented in Equation 4.11.

$$\omega_d = \frac{0 - s_{rotate}(0)}{T_{spindown}} \tag{4.16}$$

$$\omega_u = \frac{s_{rotate}(0) - 0}{T_{spinup}} \tag{4.17}$$

To prolong the lifetime of the mechanical parts in the drive, most high-speed drives lower or stop the rotation of the disk after a period of inactivity. Some drives maintain a low stationary spinning speed to prevent high spin-up times. In most cases the system administrator can set time after the last read when the drive spins down. We view this value as another parameter of the model called *maximum inactive time* ($T_{inactive}$).

The other component of the seek time is the time to move the head (t_{move}). Moving the head consists of an acceleration phase, a linear coasting phase and a deceleration phase. In a short seek the acceleration and deceleration phase dominate [43], so we consider that the head accelerates until it reaches half the distance to move and then decelerates. The speed of the acceleration and deceleration is a_{move}. The distance

over which the head accelerates is the same as the distance over which it decelerates $\frac{|t_0-t_1|}{2}$. Replacing appropriately in Equation 4.12 we derive the function $t_{move}(t_0, t_1)$ that computes the time needed to move the head between two tracks. Once the destination track is reached the head needs to settle on the track [94]. Settling on the track requires t_{settle} time. We could also use the model proposed by Ruemmler et al. [94] for modelling long seeks on a hard disk and incorporate the coast phase where the arm moves at maximum velocity.

$$t_{move}(t_0, t_1) = 2\sqrt{\frac{2\frac{|t_0-t_1|}{2}}{a_{move}}} + t_{settle} = 2\sqrt{\frac{|t_0 - t_1|}{a_{move}}} + t_{settle} \tag{4.18}$$

The head movement and the change of spin can be done simultaneously. Therefore, the seek time is the maximum time required to complete both tasks. When the distance between the tracks is small, the time needed to move the head weights more and when the distance is big, the time needed to change spin does. The $\mathcal{Z}$ specification shows how the seek time (t_{seek}) is computed.

In order to compute the time to load and unload a drive, we need to compute the time needed to open and close the drive, both when the drive is loaded and when it is empty.

The *close time* (t_{close}) is the time needed to insert the disk in the drive, either by pulling in the tray or pulling in the disk. Once the disk has been loaded, the drive starts spinning up the disk to find out the type of disk loaded in the drive.

The *recognition time* ($T_{recognition}$) is the time it takes to recognize the type of disk loaded. It depends on the type of disk loaded. Even within the same type of disk, the recognition time varies considerably. We could not determine the reason for these differences. Therefore, we use the highest recognition time we could measure for the type of disk. In most drives the load time is low and the recognition time is high, while in some drive the time relation is inverted.

Another way of computing the recognition time is to make it a function of the disk to load. We do not use this computation in the model, because then the recognition time should depend on the contents in the jukebox and not of the type of media or technology used. To compute the recognition time in this way we should have to provide a mapping from each disk in the jukebox to the recognition time, and use the unique identifier assigned to each disk in the jukebox as the parameter to the function.

We compute the time needed to close the drive as:

$$t_{close}(content) = \begin{cases} T_{load} + T_{recognition} & \text{if } content \neq \varnothing, \\ T_{load} & otherwise. \end{cases} \tag{4.19}$$

The *eject time* (T_{eject}) is the time it takes to eject the disk from the drive. If the drive has a tray on which the disk is placed, then it is the time needed to open the tray. If the drive has a slot instead of a tray and the drive is empty, the eject time is 0.

The drive first needs to spin down the disk before it can unload it. To accurately compute the time needed to open the drive we should need as parameters the last track that was read and if the disk is spinning. However, we consider that this is too detailed for the model. Therefore, we compute the worst-case time. The worst case assumes that the disk is spinning at the maximum speed, which means that the last track read was 0, which is precisely the definition of the spin-down time.

$$t_{open}(content) = \begin{cases} T_{spindown} + T_{eject} & \text{if } content \neq \varnothing, \\ T_{eject} & otherwise. \end{cases} \tag{4.20}$$

4.2.3 Jukebox and Robot Model

In most jukeboxes there is only one *shared robot*, which is able to load and unload any RSM in the jukebox to any drive. In the presence of multiple robots, there are different usage scenarios for the robots. We classify the robots according to their functionality and scope. A robot can be specific for loading or unloading, or be able to perform both loads and unloads.

$RobotFunctionality ::= Loader \mid Unloader$

Robot

$t_{position} : Position \times Position \rightarrow Time$

$t_{grab} : Position \rightarrow Time$

$t_{place} : Position \rightarrow Time$

$capabilities : \mathbb{F}\, RobotFunctionality$

The robot model can compute the time needed to move an RSM between any two locations in the jukebox ($t_{position}$), the time needed to grab an RSM from a shelf or drive (t_{grab}), and the time needed to place an RSM in a shelf or drive (t_{place}). The device where the RSM must be grabbed from or placed is given by a specific physical location that is specific for each jukebox type. The model of the robot in the smartDAX describes the functionality in more detail.

In our model the jukebox has m drives, r robots and s shelves. We model each drive and robot separately. The model has l robots capable of loading RSM into drives and u robots capable of unloading them. The l robots for loading may be the same as the u robots for unloading, as is the case in most jukebox architectures. The

total number of robots is r. The functions *getDrive*, *getRobot* and *getShelf* give the model of the corresponding device given the device number. As in the rest of the functions in the model, we assume that the functions are suitably restricted to the actual configuration of the jukebox.

JukeboxResources

$$
\begin{array}{l}
Robots : \mathbb{F}\, Robot \\
Drives : \mathbb{F}\, Drive \\
Shelves : \mathbb{F}\, Shelf \\
getDrive : DriveNumber \rightarrow Drive \\
getRobot : RobotNumber \rightarrow Robot \\
getShelf : ShelfNumber \rightarrow Shelf \\
m, r, s, l, u : \mathbb{N} \\
\hline
\#Robots = r;\ \mathrm{ran}\, getRobot = Robots;\ \#(\mathrm{dom}\, getRobot) = r \\
\#Drives = m;\ \mathrm{ran}\, getDrive = Drives;\ \#(\mathrm{dom}\, getDrive) = m \\
\#Shelves = s;\ \mathrm{ran}\, getShelf = Shelves;\ \#(\mathrm{dom}\, getShelf) = s \\
l = \#\{robot : Robot \mid Loader \in robot.capabilities\} \\
u = \#\{robot : Robot \mid Unloader \in robot.capabilities\} \\
r \leq l + u
\end{array}
$$

The scope of a robot (*scope*) is given by the set of drives and shelves it can serve. A *dedicated robot* is a robot whose scope contains only one drive in the set of drives. A *shared robot* has more than one drive in the scope. In a jukebox that only has dedicated robots, there must be as many robots as there are drives.

We consider only jukeboxes where all the shelves and drives are reachable and each RSM can go back to its shelf. Therefore, each RSM in the jukebox can be loaded and unloaded from at least one drive, and each drive can be loaded and unloaded by at least one robot. Additionally, the model guarantees that every RSM that can be loaded from a shelf to a drive can also be unloaded back into the shelf. These conditions are expressed formally by the predicates in the $\mathcal{Z}$ specification of the robot functionality. The functions *getRobotsThatLoadDrive* and *getRobotsThatUnloadDrive* returns the robots that can load and unload a given drive, respectively. The functions *getRobotsThatLoadShelf* and *getRobotsThatUnloadShelf* provide the same information for a shelf. Finally, the function *getDrivesReachableFromShelf* computes the drives that are reachable from a shelf, i.e., the drives into which the RSM in the shelf can be loaded, and then unloaded and brought back to the shelf. The hardware model does not guarantee that the drives can read the RSM, because the type of the RSM in the shelf could be incompatible with the functionality of the drives. These guarantees are provided by the jukebox controller (see Section 4.3).

The method to describe the scope and functionality is flexible and easy to parameterize. An interesting example is modelling a Memorex Telex 5400 tape-jukebox, which has two robots. The two robots in principle can serve all the shelves and drives, although, in practice they are generally configured to serve a half of the jukebox shelves and drives each [89]. Both configurations (and every other intermediate configuration) can easily be modelled and modified using the jukebox-functionality model. We can also easily model a jukebox with separate aisles, resembling a miniload AS/RS as the Odetics jukebox described in Section 2.3.

However, we do not model the possible interference between multiple robots, because we consider that the interference patterns are very specific to each jukebox. In the presence of multiple robots with overlapping scopes, the model should provide at least the worst-case interference time between each pair of robots. The jukebox scheduler needs these times to take into account the worst-case contention time for the use of 'jukebox space'. Our scheduling-problem models do not take into account the robot interference either.

JukeboxFunctionality

$$
\begin{array}{l}
JukeboxResources \\
scope : Robot \rightarrow \mathbb{F}\, Drive \times \mathbb{F}\, Shelf \\
getRobotsThatLoadDrive : Drive \rightarrow \mathbb{F}\, Robot \\
getRobotsThatUnloadDrive : Drive \rightarrow \mathbb{F}\, Robot \\
getRobotsThatLoadShelf : Shelf \rightarrow \mathbb{F}\, Robot \\
getRobotsThatUnloadShelf : Shelf \rightarrow \mathbb{F}\, Robot \\
getDrivesReachableFromShelf : Shelf \rightarrow \mathbb{F}\, Drive \\
\hline
\forall\, drive : Drive \bullet \\
\quad getRobotsThatLoadDrive(drive) = \\
\quad\quad \{r : Robot \mid (\textbf{let}\ sc == scope(r) \bullet \\
\quad\quad\quad drive \in first(sc) \land Loader \in r.capabilities)\} \land \\
\quad getRobotsThatUnloadDrive(drive) = \\
\quad\quad \{r : Robot \mid (\textbf{let}\ sc == scope(r) \bullet \\
\quad\quad\quad drive \in first(sc) \land Unloader \in r.capabilities)\} \\
\forall\, shelf : Shelf \bullet \\
\quad getRobotsThatLoadShelf(shelf) = \\
\quad\quad \{r : Robot \mid (\textbf{let}\ sc == scope(r) \bullet \\
\quad\quad\quad shelf \in second(sc) \land Unloader \in r.capabilities)\} \land \\
\quad getRobotsThatUnloadShelf(shelf) = \\
\quad\quad \{r : Robot \mid (\textbf{let}\ sc == scope(r) \bullet \\
\quad\quad\quad shelf \in second(sc) \land Loader \in r.capabilities)\} \land
\end{array}
$$

$$
\begin{array}{l}
\quad getDrivesReachableFromShelf(shelf) = \\
\qquad \{drive : Drive \mid (\exists\, r_1, r_2 : Robot \mid r_1 \in Robots \land r_2 \in Robots \bullet \\
\qquad\quad r_1 \in getRobotsThatLoadDrive(drive) \land \\
\qquad\quad r_2 \in getRobotsThatUnloadDrive(drive) \land \\
\qquad\quad r_1 \in getRobotsThatUnloadShelf(shelf) \land \\
\qquad\quad r_2 \in getRobotsThatLoadShelf(shelf))\} \\
\forall\, shelf : Shelf;\ drive : Drive \mid shelf \in Shelves \land drive \in Drives \bullet \\
\quad (\exists\, r_1 : Robot \mid r_1 \in Robots \bullet \\
\quad (\textbf{let}\ sc_1 == scope(r_1) \bullet \\
\quad drive \in first(sc_1) \land shelf \in second(sc_1) \land Loader \in r_1.capabilities)) \\
\quad \Rightarrow (\exists\, r_2 : Robot \mid r_2 \in Robots \bullet (\textbf{let}\ sc_2 == scope(r_2) \bullet \\
\qquad drive \in first(sc_2) \land shelf \in second(sc_2) \land Unloader \in r_2.capabilities))
\end{array}
$$

The main goal of the jukebox model is to be able to predict the time needed to load an RSM into a drive, read data from an RSM once it is loaded in a drive, and unload an RSM.

The drives and shelves have fixed positions in the jukebox. The model uses these positions to compute the time needed to move an RSM and to access a shelf and drive in order to place or grab an RSM. The functions *getDrivePosition* and *getShelfPosition* provide the location of the devices. The robots generally have an idle position where they are parked when idle that is used in the computations as a starting or finishing point of the movements involving the robot.

The jukebox model uses the drive model presented in the previous subsection. As we have already discussed, the performance of the drives depend on the type of media handled. The functions $t_{transfer}$ and t_{access} compute the transfer and access time for a drive and media combination.

Jukebox

$$
\begin{array}{l}
JukeboxFunctionality \\
getDrivePosition : DriveNumber \rightarrow Position \\
getShelfPosition : ShelfNumber \rightarrow Position \\
t_{transfer} : DriveNumber \times MediaType \times Offset \times Size \rightarrow Time \\
t_{access} : DriveNumber \times MediaType \times Offset \times Offset \rightarrow Time \\
t_{unload} : RobotNumber \times DriveNumber \times ShelfNumber \times MediaType \rightarrow Time \\
t_{load} : RobotNumber \times DriveNumber \times ShelfNumber \times MediaType \rightarrow Time \\
t^{max}_{load} : DriveNumber \rightarrow Time \\
t^{max}_{unload} : DriveNumber \rightarrow Time
\end{array}
$$

$$
\begin{aligned}
&\forall\, dn : DriveNumber;\ media : MediaType \bullet \\
&\quad (\textbf{let}\ drive == getDrive(dn) \bullet \\
&\quad (\textbf{let}\ db == drive.behaviour(media) \bullet \\
&\quad (\forall\, offset : Offset;\ size : Size \bullet \\
&\qquad t_{transfer}(dn, media, offset, size) = db.t_{transfer}(offset, size)) \wedge \\
&\quad (\forall\, source : Offset;\ dest : Offset \bullet \\
&\qquad t_{access}(dn, media, source, dest) = db.t_{access}(source, dest))))
\end{aligned}
$$

In general the file systems used for tertiary storage media store the data sequentially (e.g., ISO-9660 [77]). Therefore, reading a file implies reading a given number of bytes starting at a given offset on the medium. However, if the file system uses a more complex way of storing the files (e.g., i-nodes), we can use the information of the file system to transform a request unit corresponding to a file or part of a file to a set of request units as required by this model.

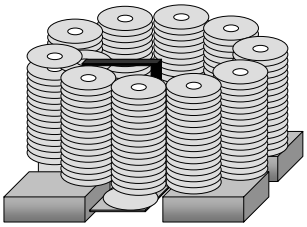
The functions used to compute the time needed to load and unload a drive are t_{load} and t_{unload}, respectively. To compute these functions we use the model of the drive and the robot, which is specific to each jukebox (specified in *DAXRobot* for the DAX jukebox). The model also provides functions to determine the maximum and minimum load time for each drive and the maximum and minimum over all the drives (t_{load}^{max} and t_{unload}^{max}).

The way in which the robot is used depends strongly on the jukebox architecture. Therefore, we are not able to present a general robot model as we do for the drives. Instead, we present the model of the smartDAX700 that we were able to validate.

Model of smartDAX700

In a smartDAX700 the disks are stored as shown in Figure 4.6. It has 18 columns, which interleave in odd and even rows, and 40 rows. The robot is in the centre of the jukebox. At the bottom are four drives, one in each corner of the jukebox.

The position of each device—shelf, drive, mailbox, and robot—is given by a rotational coordinate *x*, a vertical coordinate *y*, and the radius *rad*. The radius is relative to the centre of the jukebox. The vertical coordinate is in the range [3000,155000], the rotational (or horizontal) coordinate is in the range [0,18000] and the radius is in the range [0,2100]. The firmware can provide the position of each device in the jukebox and the idle position of the robot. We have noticed, however, that after resetting the jukebox, the value of the reported positions varies slightly.



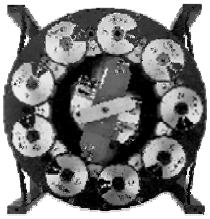


Figure 4.6: Architecture of the smartDAX700 jukebox. In the picture on the left, the boxes at the bottom represent the drives. The front left drive is open and waiting to be loaded, while the robot is grabbing a disk from a shelf. The right picture shows a top view of the jukebox.

Position

$x : \mathbb{N}$
$y : \mathbb{N}$
$rad : \mathbb{N}$

DAXRobot

Robot
$IdlePosition : Position$
$t_{move} : Position \times Position \rightarrow Time$
$t_{calibrate} : Position \times Position \rightarrow Time$
$t_{settle} : Time$
$elevator : Motor$
$rotator : Motor$
$picker : Picker$

$Loader \in capabilities \land Unloader \in capabilities$
$\forall source : Position;\ dest : Position \bullet$
$t_{move}(source, dest) =$
$max\{elevator.t_{move}(abs(dest.y - source.y)),$
$rotator.t_{move}(abs(dest.x - source.x))\} +$
$t_{calibrate}(source, dest) \land$
$t_{position}(source, dest) = t_{move}(source, dest) + t_{settle} \land$
$t_{grab}(source) = picker.t_{grab}(source) \land$
$t_{place}(dest) = picker.t_{place}(dest)$

The firmware is able to control only one picker, although in the right picture in Figure 4.6 we can see two pickers one at each end of the board. Therefore, in *DAXRobot* we model the robot with one picker. The picker is attached to a board that moves vertically along the arm and also rotates along the axis. Two independent motors

perform these movements: the *elevator* and the *rotator*. After performing a load or an unload the robot returns to a predefined idle position (*IdlePosition*). The idle position is around (51160, 6050, 0). The robot has the capacity to load and unload the drives, as expressed in the first predicate of the $\mathcal{Z}$ specification.

The jukebox can be operated in two modes: *user mode* and *operator mode*. In user mode the jukebox performs transactions as moving a disk from a slot to a drive, or moving a disk between two slots. In operator mode, any kind of robot movement can be performed using the coordinates in the jukebox.

When using the operator mode, the model can compute the time to move between two positions (t_{move}) without taking into account the time to settle. We compute the time to move as the maximum time it takes to any of the two motors to move the picker along its corresponding axis, plus the time to calibrate the robot ($t_{calibrate}$). This computation is shown in the second predicate of the $\mathcal{Z}$ specification. We also use t_{move} to compute the time it takes to move the picker back to the idle position, as shown in the *DAXJukebox* model. When using the operator mode, we use the function $t_{position}$ that takes into account the time to settle (t_{settle}). The computation of $t_{position}$ is shown in the third predicate of the $\mathcal{Z}$ specification. The last two predicates show that the time to grab and place an RSM are computed by the picker.

The time to calibrate the robot depends on the source and destination of the movement. Performing multiple measurements we have determined that there are two main calibration areas in the jukebox. One calibration area is at the bottom of the jukebox (when y is in the range $[3000, 6700]$) and another calibration area is at the top of the jukebox (when y is in the range $[146800, 155000]$. In these calibration areas the robot takes longer to calibrate. We compute the time to calibrate using a function we derived from performing curve fits. Basically, the time to calibrate is longer when the robot is close to the limits of the vertical axis.

The motors are defined by the maximum velocity (v_{max}) and the acceleration (a). A motor accelerates until it reaches maximum velocity, moves at maximum velocity, and then decelerates to stop at its destination.

Motor

$v_{max} : Speed$
$a : Acceleration$
$t_{move} : Distance \rightarrow Time$

To compute the time needed to move ($t_{move}(x)$) we first compute the time needed to reach maximum velocity (t_{max}). Using this time we compute the distance covered during acceleration and deceleration (d_{acc}).

$$t_{max} = \frac{v_{max}}{a} \tag{4.21}$$

$$d_{acc} = \frac{a\ t_{max}^2}{2} \tag{4.22}$$

$$t_{move}(x) = \begin{cases} 2\ \frac{x}{a} & \text{if } x \leq 2\ d_{acc}, \\ 2\ t_{max} + \frac{x - 2\ d_{acc}}{v_{max}} & \textit{otherwise.} \end{cases} \tag{4.23}$$

In order to pick-up/place a disk, the picker moves outward toward the corresponding shelf or drive. The picker then grabs/places the disk using a robotic hand. Finally, the picker retracts again. The functions t_{grab} and t_{place} compute the time needed to grab and place a disk from/to a position in the jukebox, respectively. The picker has a *retractor* motor that moves along the radius—extending or retracting. The times to pick up and drop a disk on a shelf or a drive are t_{pickup} and t_{drop}, respectively.

Picker

$t_{grab} : Position \rightarrow Time$

$t_{place} : Position \rightarrow Time$

$retractor : Motor$

$t_{pickup} : Time$

$t_{drop} : Time$

$\forall\ pos : Position \bullet ($
$\quad t_{grab}(pos) = retractor.t_{move}(pos.rad) + t_{pickup} + retractor.t_{move}(pos.rad) \land$
$\quad t_{place}(pos) = retractor.t_{move}(pos.rad) + t_{drop} + retractor.t_{move}(pos.rad))$

The jukebox enforces to perform the activities needed to load and unload a drive sequentially. Furthermore, after moving a disk it always returns the robot to the idle position.

The execution of a load (unload) is done in the following sequence:

1. Open drive.
2. Move robot from idle position to shelf (drive).
3. Pick up disk from shelf (drive).
4. Move robot from shelf (drive) to drive (shelf).
5. Place disk on drive tray (shelf).
6. Close drive.
7. Move robot from drive (shelf) to idle position.

The functions t_{load} and t_{unload} in the *DAXJukebox* specifications compute the load and unload times, respectively.

DAXJukebox

Jukebox
robot : *DAXRobot*

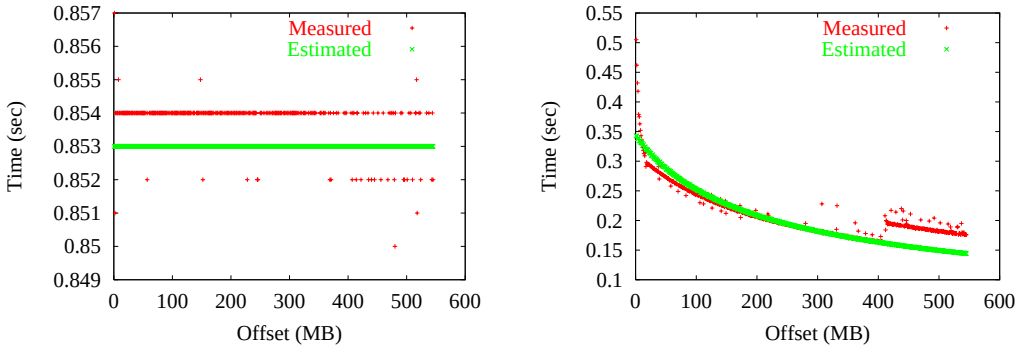
$Robots = \{robot\}$
$scope(robot) = (Drives, Shelves)$
$\forall\ rn : RobotNumber;\ dn : DriveNumber;$
$sn : ShelfNumber;\ media : MediaType \bullet$
(let $drive == getDrive(dn) \bullet$
(let $db == drive.behaviour(media) \bullet$
(let $shelfPosition == getShelfPosition(sn) \bullet$
(let $drivePosition == getDrivePosition(dn) \bullet$
$t_{unload}(rn, dn, sn, media) =$
$db.t_{open}(RSM)+$
$robot.t_{position}(robot.IdlePosition, drivePosition)+$
$robot.t_{grab}(drivePosition)+$
$robot.t_{position}(drivePosition, shelfPosition)+$
$robot.t_{place}(shelfPosition)+$
$db.t_{close}(\varnothing)+$
$robot.t_{move}(shelfPosition, robot.IdlePosition)$
$\wedge$
$t_{load}(rn, dn, sn, media) =$
$db.t_{open}(\varnothing)+$
$robot.t_{position}(robot.IdlePosition, shelfPosition)+$
$robot.t_{grab}(shelfPosition)+$
$robot.t_{position}(shelfPosition, drivePosition)+$
$robot.t_{place}(drivePosition)+$
$db.t_{close}(RSM)+$
$robot.t_{move}(drivePosition, robot.IdlePosition)))))$

4.2.4 Model Validation

This section shows that the model can be used effectively to model the hardware. It is neither the goal of this dissertation to build a system specific for one jukebox, nor to obtain the exact parameters of the hardware. Therefore, we have estimated the parameters using only simple methods. The parameters can be further refined to provide a better fit to the hardware performance.

	CAV	CLV
$s_{rotate}(rpm)$	10500	
$s_{transfer}(KBps)$		1200
$\omega_u(rot/s^2)$	0.0505	0.115
$\omega_d(rot/s^2)$	-0.0462	-0.13
$a_{move}(\mu m/s^2)$	4250	1600
$t_{settle}(s)$	0.01	0.06

Table 4.2: Parameters of the drives for validating the drive model.



(a) CLV drive (b) CAV drive

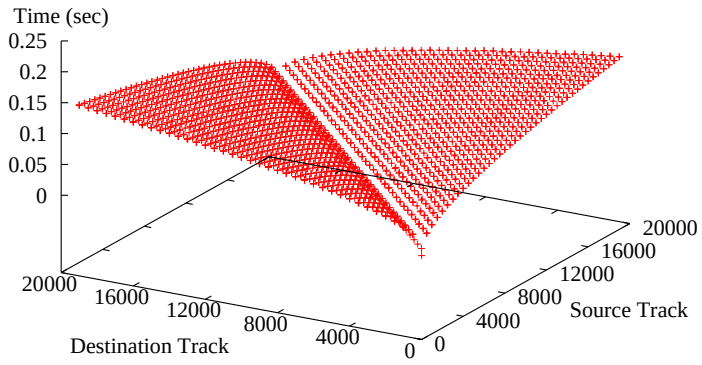
Figure 4.7: Comparison between the estimated and measured read time. The graphics show the time needed to read one-megabyte blocks.

Drives

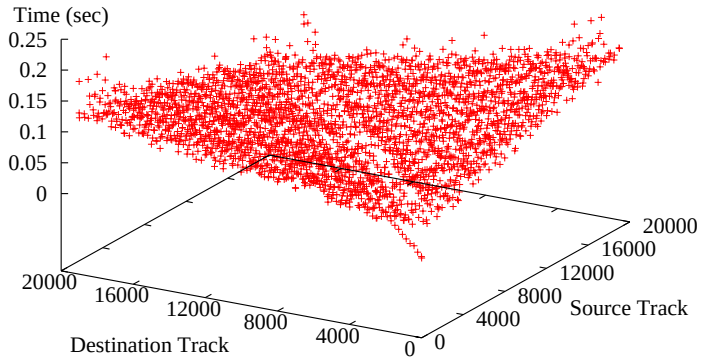
Given that the model presented can be used as it is for CAV and CLV drives, we show the estimates and measurements of one CAV and one CLV drive. Table 4.2 shows the parameters used. We obtained these parameters by performing curve fitting and using the manufacturers' specifications of the drives. The CAV drive is a 48X CD-ROM drive model 'LG CRD-8484B' from NEC and the CLV drive is an 8X CD-ROM drive 'CW-7520' from Matsushita.

Figure 4.7 shows the read time of both drives when reading one-megabyte blocks. The time estimated for the CLV fits the measured time with a difference of only one millisecond. In the case of the CAV drive, the difference is clearly bigger, especially at the outer edge of the disk. All measurements that we performed show a point were the performance of the CAV drives degrades. The point at which the degradation occurs differs, but it is always in the outer tracks of the disk.

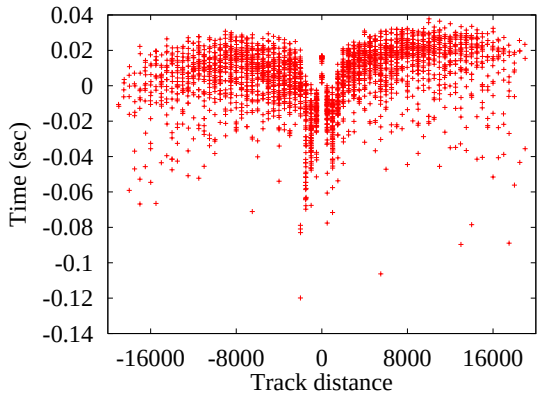
Figures 4.8 and 4.9 show the access time when using a CAV and CLV drive, respectively. The parameters needed to compute the access time are not reported



(a) Estimated

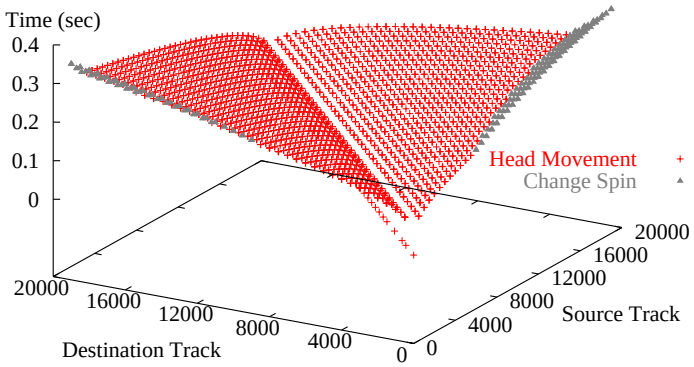


(b) Measured

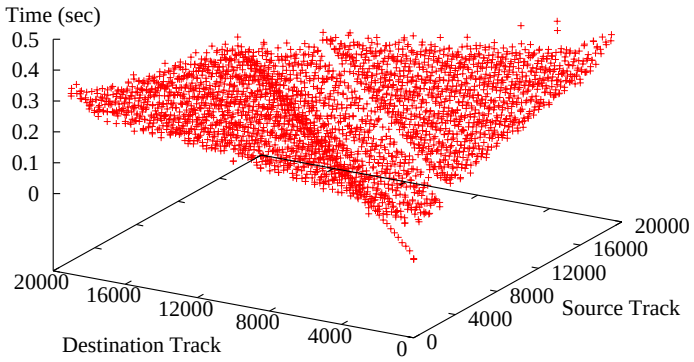


(c) Error estimate

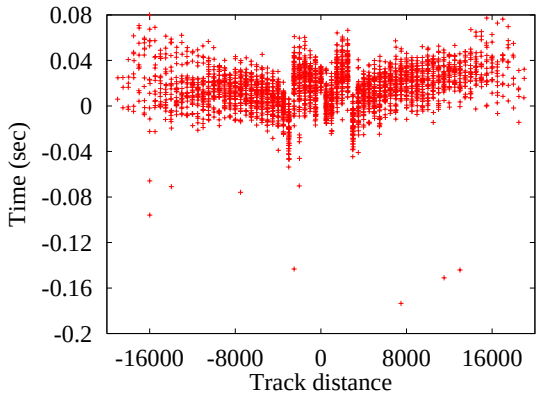
Figure 4.8: Comparison between estimated and measured access time for a CAV CD-ROM drive.



(a) Estimated



(b) Measured



(c) Error estimate

Figure 4.9: Comparison between estimated and measured access time for a CLV CD-ROM drive.

Parameter	Elevator	Rotator	Retractor
v_{max} $(units/s)$	70000	20000	10000
a $(units/s^2)$	300000	35000	15000

Table 4.3: Parameters of the elevator, rotator and retractor motors.

in the manufacturers' specifications, so we derived from the estimated spin-up and spin-down times. We derived the spin-up/-down acceleration and adjusted the head acceleration to fit the measurements.

Jukebox

Table 4.3 shows the parameters used for the elevator, rotator and retractor motors. The maximum velocity and acceleration are provided by the jukebox firmware. All the interactions with the jukebox involve a constant time of approximately $0.16\,s$.

Figure 4.10 shows how the model of the motors fits the measurements obtained from the jukebox in operator mode. In this case we move the picker on only one axis to measure the performance of each motor separately. The right-hand side of the figure shows the error of the estimate.

Figure 4.11 shows the measurements and the error of the estimate when computing the functions t_{move} and $t_{position}$ for random positions in the jukebox. The estimate of the time to move presents larger errors when the robot is moved into the calibration area. The function to compute the time to calibrate ($t_{calibrate}$) is more complex when moving the robot on the two axes. However, when computing $t_{position}$, this error is masqueraded by the time to settle (t_{settle}). Our estimate of t_{settle} is $0.5\,s$.

Figure 4.12 shows how the model of the time to grab and place a disk from/in a slot (t_{grab} and t_{place}, respectively) fit the measurements obtained from the jukebox in operator mode. The estimate for the time to pick-up a disk is $3.0\,s$ (t_{pickup}), and the estimate for the time to drop a disk is $1.9\,s$ (t_{drop}).

Finally, Figure 4.13(a) shows the measured and estimated time to load and unload a drive. The estimate is in most cases far above the measured time. The reason for this is that the time to open and close a drive varies considerably and unpredictably, therefore, we have to use the worst-case times for building the schedules. Figure 4.13(b) shows the time to open the drive when the drive is empty and loaded. The variation in the time to open an empty drive is big ($0.8\,s$). Figure 4.13(c) shows the measured time to close the drive. When the drive is loaded the variation is nearly $1.5\,s$.

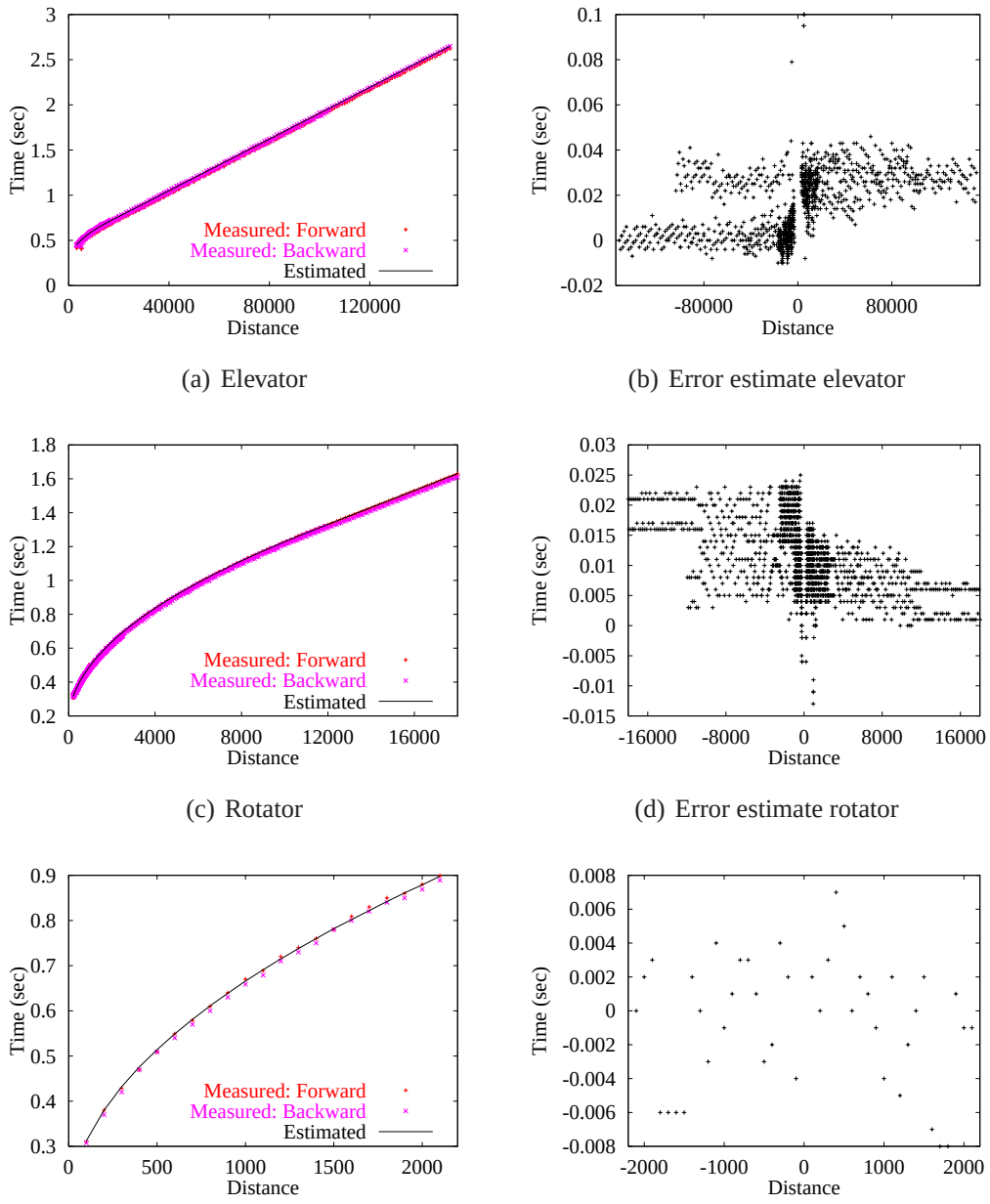


Figure 4.10: Comparison between the estimated and measured performance of the motors for the function $motor.t_{move}$.

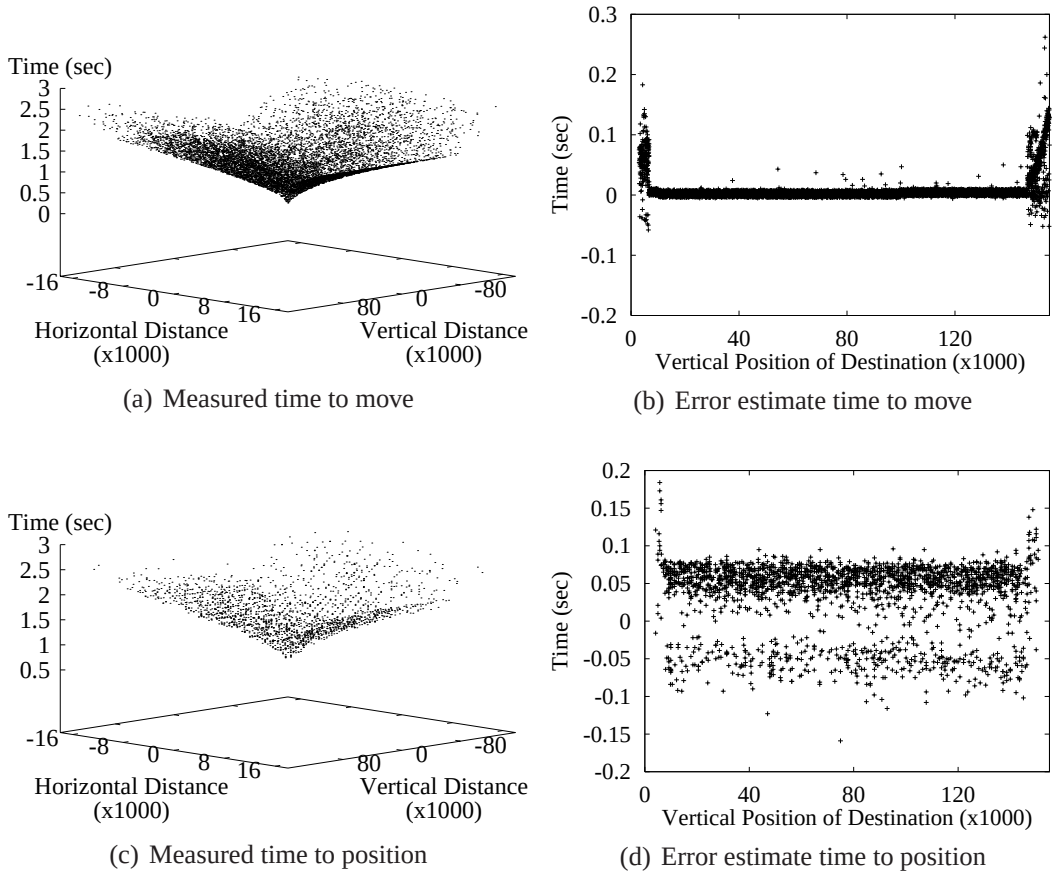


Figure 4.11: Comparison between the estimated and measured performance of the robot for the functions $robot.t_{move}$ and $robot.t_{position}$.

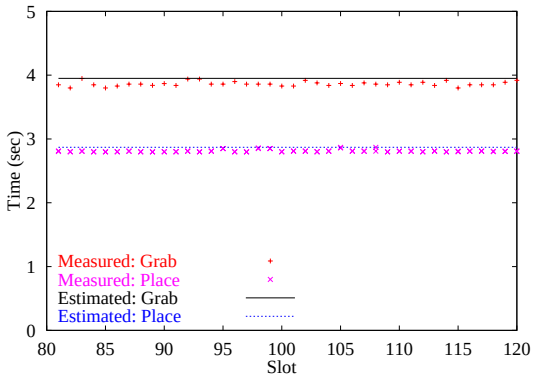
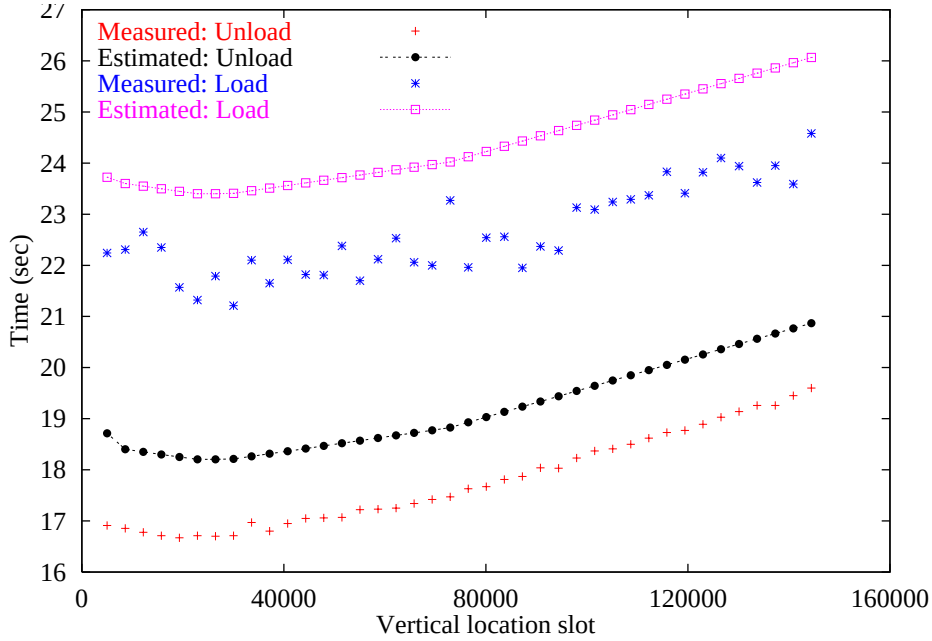
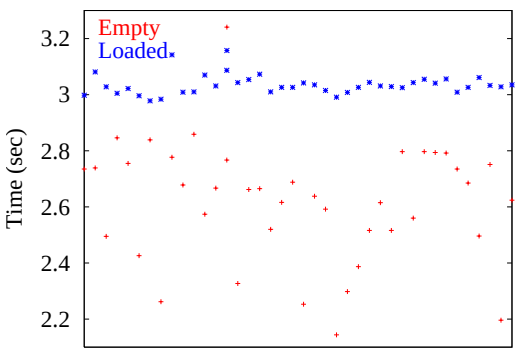


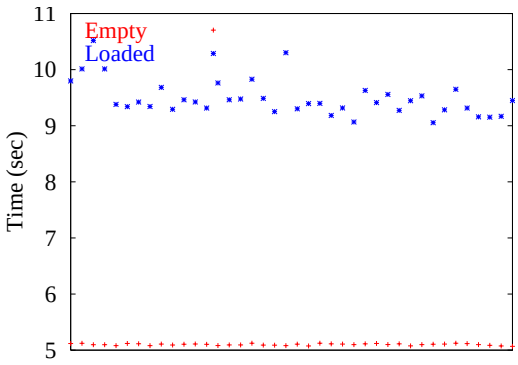
Figure 4.12: Comparison between the estimated and measured performance of the picker for the functions t_{grab} and t_{place}.



(a) Time to load and unload



(b) Measured time to open the drive



(c) Measured time to close the drive

Figure 4.13: Comparison between the estimated and measured Load-/Unload-Time. Details of the time to open and close the drive.

4.3 Jukebox Controller

The jukebox controller is the interface to the jukebox hardware that is used by the jukebox scheduler. It hides the details of the interaction with the hardware and provides a simple set of functions that are hardware independent. The functions offered are: read, load and unload.

The controller assigns a unique identifier to each RSM in the jukebox. Each RSM has an associated shelf where it is stored when not being used. The controller keeps information about the type of media of each RSM and guarantees that each RSM is stored in a shelf so that it can be read with at least one drive. This implies that each RSM can be loaded and unloaded from at least one drive that can handle that type of RSM. In turn, the jukebox model guarantees that at least one robot can load the RSM in the drive and one robot can unload it.

ResourceMapping

$$
\begin{array}{l}
getShelfNumber : IdentifierRSM \rightarrow ShelfNumber \\
getTypeMedium : IdentifierRSM \rightarrow MediaType \\
jukebox : Jukebox \\
drives_{read} : IdentifierRSM \rightarrow \mathbb{F}\, Drive \\
resources_{load} : IdentifierRSM \rightarrow \mathbb{F}(Drive \times Robot) \\
resources_{unload} : IdentifierRSM \rightarrow \mathbb{F}(Drive \times Robot)
\end{array}
$$

$$
\begin{array}{l}
\forall\, id : IdentifierRSM \bullet \\
\quad (\textbf{let}\ shelf == jukebox.getShelf(getShelfNumber(id)) \bullet \\
\quad (\textbf{let}\ media == getTypeMedium(id) \bullet \\
\quad drives_{read}(id) = \\
\qquad \{drive : Drive \mid \\
\qquad drive \in jukebox.getDrivesReachableFromShelf(shelf) \land \\
\qquad (\textbf{let}\ beh == drive.Behaviors \bullet \\
\qquad \exists\, dh : DriveBehaviour \bullet (media, dh) \in beh)\})) \land \\
\quad resources_{load}(id) = \\
\qquad \{drive : Drive;\ robot : Robot \mid \\
\qquad drive \in drives_{read}(id) \land \\
\qquad robot \in jukebox.getRobotsThatLoadDrive(drive)\} \land \\
\quad resources_{unload}(id) = \\
\qquad \{drive : Drive;\ robot : Robot \mid \\
\qquad drive \in drives_{read}(id) \land \\
\qquad robot \in jukebox.getRobotsThatUnloadDrive(drive)\} \\
\forall\, id : IdentifierRSM \bullet \exists\, d : Drive \bullet d \in drives_{read}(id)
\end{array}
$$

An important functionality of the hardware controller is to keep a consistent view of the hardware state. The following specification provides the state of each drive. Using this as a starting point we can also conclude the state of the robots. The possible states for a drive are: empty, loading, reading, loaded, and unloading. The loaded state indicates that the drive is loaded, but it is not reading data from the RSM. If a drive is busy reading, loading or unloading the specification also indicates the time at which the drive will become idle again.

$DriveStatus ::= Empty \mid Loading \mid Reading \mid Loaded \mid Unloading$

DriveState

$$
\begin{array}{l}
status : DriveStatus \\
content : IdentifierRSM \\
t^{est}_{finish} : Time \\
offsetLastRead : Offset \\
robotInvolved : RobotNumber \\
\hline
(status \neq Loading \land status \neq Unloading) \Rightarrow robotInvolved = -1 \\
(status \neq Loaded \land status \neq Reading) \Rightarrow offsetLastRead = -1
\end{array}
$$

JBState

$$
\begin{array}{l}
getDriveForRSM : IdentifierRSM \rightarrow DriveNumber \\
getState : DriveNumber \rightarrow DriveState \\
\hline
\forall\, id : IdentifierRSM \bullet \\
\quad (\exists\, dn : DriveNumber \bullet \\
\quad (\textbf{let}\ state == getState(dn) \bullet state.content = id)) \\
\qquad \Rightarrow getDriveForRSM(id) = dn \land \\
\quad \neg\, (\exists\, dn : DriveNumber \bullet \\
\quad (\textbf{let}\ state == getState(dn) \bullet state.content = id)) \\
\qquad \Rightarrow getDriveForRSM(id) = -1
\end{array}
$$

4.4 Summary

This chapter presented a model of tertiary-storage devices, with special emphasis on optical and magneto-optical jukeboxes. The model allows us to describe any type

of optical and magneto-optical disk and different drive technologies. We limit our analysis to this type of jukebox and leave out tape jukeboxes, because tapes are not capable to cope with random access in an efficient manner.

This model is used by the jukebox scheduler to compute the duration of the tasks that need to be scheduled. It is also used by the jukebox simulator to simulate the behaviour of the jukebox.

This chapter also discussed the functionality of the jukebox controller, which guarantees that the view of the jukebox is consistent and all the data in the jukebox can be read by at least one drive.

Chapter 5

Formalization of the Scheduling Problem

This chapter presents a formalization of the scheduling problem. We present different ways of modelling the scheduling problem and analyze their advantages and disadvantages. We should like to model the original scheduling problem described in Section 3.5 using the scheduling theory presented in Section 2.1. However, we could not find a model that captures all the problem variables, because there is a conflict when deciding when to switch RSM given that the robots may be shared resources. Under the restriction of shared robots, there is no way to represent the option to either interrupt reading from an RSM and unload it or continue reading.

Therefore, we simplify the original scheduling problem in order to make it describable by scheduling theory. We present several alternatives that either restrict the way the request units are clustered, or simplify the hardware model.

The most flexible of the models is the *fixed switching model*, which only requires the way in which the request units are clustered to be predefined. The *optimal model* is obtained by clustering the request units in all possible ways. A solution of the optimal model is the optimal solution to the scheduling problem, however this solution can not be computed in polynomial time.

We show that there is one simple way of clustering the request units that makes good use of the jukebox resources and results in a simpler scheduling problem. If we cluster all request units for an RSM in one job, the number of switches is minimized, which, in the long run results in good performance. We call this model the *minimum switching model.*

We prove in this chapter that the scheduling problem is $\mathcal{NP}$-hard and that all our simplifications of the original scheduling problem are $\mathcal{NP}$-hard as well.

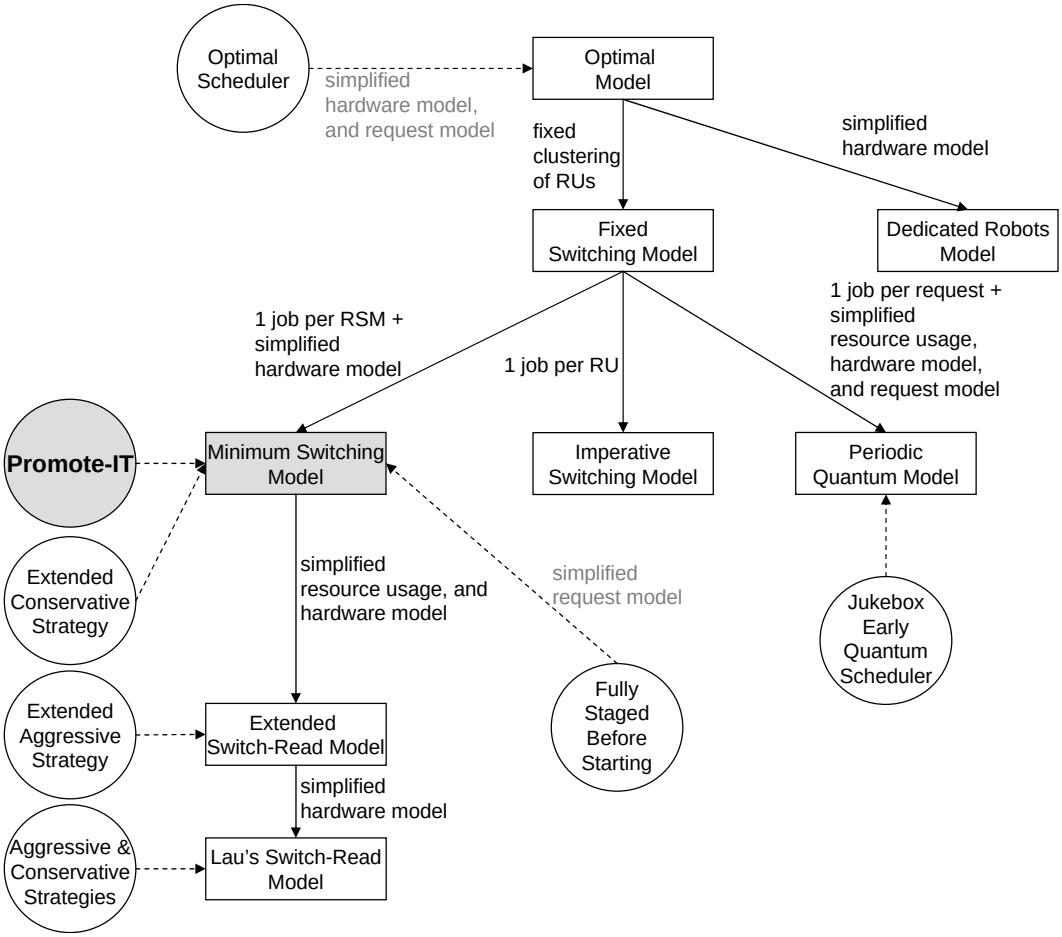


Figure 5.1: Scheduling-problem models and the schedulers that use them. The full arrows represent simplifications performed to the scheduling problem. The circles represent schedulers. The schedulers are attached to the model they are based on.

5.1 Model Hierarchy

When determining the parameters of the jobs to schedule, we need to know in advance which type of shared resources are needed to execute the job. We have three types of shared resources in the jukebox: drives, robots, and RSM. The scheduler must deal with three types of operations: read, load, and unload. Every operation involves a drive and an RSM, but only the load and unload operations involve a robot.

The conflict when trying to represent the scheduling problem using scheduling theory is that the optimal scheduler should be able to decide after scheduling the read of each request unit, whether more data must be read from the same RSM.

However, leaving the RSM in the drive does not require scheduling the use of the robots, while unloading the RSM does. Therefore, if we do not know in advance the sequence in which the request units will be read, we cannot determine in advance the parameters needed to represent the jobs. We call this conflict the *switch/no-switch conflict*.

We propose two solutions to solve the switch/no-switch conflict (first branching in Figure 5.1). One solution is to determine in advance how to group the request units corresponding to the same RSM. We divide the request units into subsets and for each subset we create a job that involves a load, a read and an unload. In this way we determine resource needs for the jobs in advance. Therefore, we call this model the *fixed switching model*. Section 5.2 presents this model in detail. As shown in Figure 5.1, we have refined this model into other models that define how to cluster the request units.

Another solution, which we call the *dedicated robots model* is to assume that the robots are dedicated resources, i.e., there is one robot to serve each drive. A scheduler using this model does not need to schedule the robots separately from the drives, because there are no possible conflicts in the use of the robots. Therefore, the scheduler can decide to go on reading from the same RSM or to switch RSM while building the schedule. Section 5.7 presents this model in detail.

The assumption made for the dedicated robot model is too restrictive to be used for scheduling a jukebox because jukeboxes in general do not have dedicated robots. Thus, we did not implement a jukebox scheduler based on this model. However, this model may be useful in a manufacturing environment, where it is more plausible that there is a dedicated robot serving each machine.

The fixed switching model is the model that imposes fewer restrictions on the original scheduling problem. Using the fixed switching model we build the optimal model. The *optimal model* represents all the possible ways of grouping the request units into jobs that can be handled by the fixed switching model. By refining the fixed switching model we obtain different models for which efficient heuristic schedulers can be built.

The *minimum switching model* is a special case of the fixed switching model in which there is one job per RSM. This model requires that all the requested data from an RSM must be read before the RSM is unloaded from a drive. The name of the model derives from the fact that a feasible schedule built using this model will have the minimum possible numbers of RSM switches. This model eliminates the possibility of using the robot to unload an RSM before all the requested data from that RSM has been read. Section 5.3 presents this model in detail.

We believe that the minimum switching model is the most flexible model that permits to build efficient polynomial heuristic algorithms to solve the scheduling problem. Promote-IT is based on this model. The extended conservative strategy

also uses this model, because as we explained in Section 2.2.1 the strategy cannot use a flexible flow shop with two stages and at the same time handle non-constant load and unload times. As shown in Figure 5.1, the Fully-Staged-Before-Starting (FSBS) scheduler also uses this model, but it assumes that all the request units of a request have the same delta deadline.

The *switch-read model* used by Lau et al. for the conservative and aggressive strategies (see Section 2.2.1) is a special case of the minimum switching model with strong restrictions on the jukebox architecture and the use of the resources. The switch-read model couples the unload and the load of a drive into one task. Additionally, the model assumes that all drives are identical and that the load and unload time is constant—independently of the shelf, drive, and RSM involved. Section 5.4 presents a formalization of this model. Although this formalization was not provided by Lau et al., we consider it important to present, because it shows that the scheduling model used by Lau et al. is different and much more restricted than the minimum switching model.

The *extended switch-read model* relaxes some restrictions on the hardware model to be able to handle non-identical drives, variable load and unload time and multiple robots. The extended aggressive strategy is based on the extended switch-read model.

The *imperative switching model* is another special case of the fixed switching model for which there is one job per request unit in $\mathcal{U}'$. The model requires that the RSM should be loaded, read and unloaded for every request unit. The name of the model is due to the fact the model requires the scheduler to switch RSM for each request unit. Each request unit is processed independently from the others. So, for each request unit, the RSM is loaded, the data is read and the RSM is unloaded. This model does not permit to keep an RSM loaded in a drive even if there are multiple request units for it. Thus, it prevents building schedules that can make an efficient use of the drives and the robots, because the drives cannot read as much data as possible from an RSM before it is unloaded. Section 5.5 briefly presents this model. We did not implement any scheduler for this model, because the utilization of the resources is clearly very poor.

The *periodic quantum model* is another special case of the fixed switching model. The jobs are represented as periodic tasks. There is one such periodic task for each request r_k arriving at the scheduler. In this model the resources are used in a cyclic way. The robot switches the RSM in the drives at fixed intervals. The drives read data from an RSM during a fraction of the cycle, while the robot is serving the other drives. This model can only be used when all requests are for continuous-media, which is stored contiguously in one RSM. Section 5.6 presents this model. The jukebox early quantum scheduler (JEQS) is based on this model.

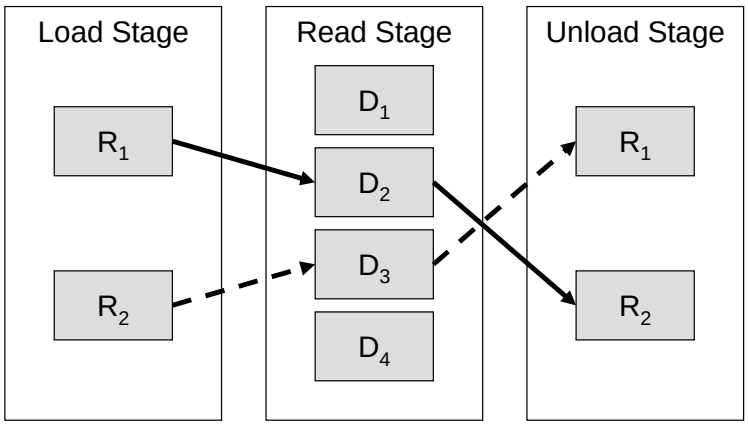


Figure 5.2: Model of a Jukebox as Flexible Flow Shop with Three Stages. The jukebox has 4 drives and 2 robots, which are capable of loading and unloading all the drives.

As presented in Section 3.5 the scheduling problem model is constructed for each candidate starting time. Therefore, we build a scheduling problem for the set $\mathcal{U}'$ under the following conditions:

$$\begin{aligned} \mathcal{U}' &= \mathcal{U} \cup \{u'_{k1}, u'_{k2}, \ldots, u'_{kl_k}\} \wedge \\ u'_{kj} &= (\tilde{d}_{kj}, m_{kj}, o_{kj}, s_{kj}, b_{kj}) \wedge \\ \tilde{d}_{kj} &= st^x_k + \Delta\tilde{d}_{kj} \end{aligned}$$

The schedule is computed at time t_0.

5.2 Fixed Switching Model

The processor environment of the *fixed switching model* is a *flexible flow shop* with three stages (FF_3). The first stage is to load an RSM to a drive, the second stage is to read the data from an RSM and the third is to unload the RSM. The jobs to be processed are of the form $J_j = \{T_{1j}, T_{2j}, T_{3j}\}$, with one task for each stage.

Figure 5.2 shows an example of a jukebox with 4 drives and 2 robots. Both robots can serve all the drives. The full line represents a possible assignment to a job, using R_1 to load, D_2 to read and R_2 to unload. At the same time another job, represented by the dotted line, is executed using R_2, D_3 and R_1.

The processing time of a reading task T_{2j} is determined by computing a separate scheduler for all request units that are grouped into J_j. We call this schedule for an RSM a *Medium Schedule (MS)*. An MS determines in which order the data must be read once the RSM is in the drive. As the drives may be non-identical, we compute

a separate MS for each drive. The optimization criterion for an MS is to maximize the time at which the RSM has to be loaded in a drive to start reading data from it, in such a way that the deadlines of the request units are met. In other words we want to determine the latest possible starting time of the read. If the RSM is already loaded in a drive, the goal is to read the requested data before the RSM must be unloaded. We provide the details of the model to build the MS in Subsection 5.2.4.

Any algorithm based on the fixed switching model must compute the schedule in two stages. In the first stage, the algorithm computes the medium schedules corresponding to each job J_j. As a result, the algorithm establishes the values of the processing times and deadlines of the reading tasks T_{2j}. In the second stage, it assigns resources—robots and drives—to each job.

This model does not impose restrictions on the hardware model. It can even handle double-sided disks that need to be turned. We model double-sided disks as two different RSM that have a shared resource in common. This shared resource corresponds to the disk. Therefore, the request units for each side will always be on different jobs. By using the disks as shared resources, we indicate that mutual exclusion is needed on jobs involving the same disk.

As we discussed in Section 4.2.3, both the drives and robots may all have different characteristics. Therefore, the processors at each stage are modelled as *unrelated*. In the first stage there are l processors representing to the l loader robots. In the second stage there are m processors representing the drives. In the third stage there are u processors representing the unloader robots. The robots in the first and third stage may have some elements in common and in the worst case all the elements will be the same: when all robots are able to load and unload.

We assume that the drives are involved in the whole activity of being loaded or unloaded in such a way that a drive cannot read data from an RSM during that time.[1] The load and unload of a drive require a lock on the drive, which we will represent as additional resources. Because the robots can be shared, we need to specify that a robot cannot be used for a load and an unload simultaneously. We do this, as well, by using additional resources.

A schedule built for this model consists of m *drive schedules* $D_1, \ldots, D_m$ and r *robot schedules* $R_1, \ldots, R_r$, where r is the number of different robots in the jukebox. Thus, $r \leq l + u$.

In the model there are no buffers between the processors and this may lead to blocking (see Section 2.1.1), given that an RSM can be only in a shelf, in a robot or in a drive. Although there are no additional buffers where the RSM can be placed once the processing on a stage finishes and the processing on the next stage be-

1 Subsection 5.2.5 presents an extension to the model for the case when the drives are not involved in the whole load and unload operations.

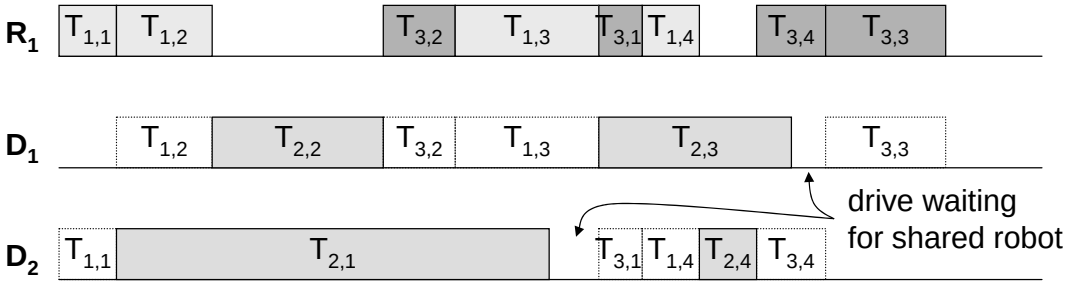


Figure 5.3: Example showing how the jobs can interleave in the use of the robot, but not in the use of the drives.

gins, an RSM can stay loaded in a drive even if the drive is not reading data from it. So, there is no restriction indicating that immediately after a drive is loaded the reading must begin or immediately after the reading finishes the RSM must be unloaded. Another consequence of the absence of buffers is that the tasks of different jobs cannot interleave in the use of a drive. However, tasks from different jobs can interleave in the use of the robots.

Figure 5.3 shows how tasks interleave in the robot schedule R_1, while the drives are blocked waiting to be unloaded. While D_2 is reading the data of $T_{2,1}$, R_1 executes $T_{1,2}$, $T_{3,2}$ and $T_{1,3}$, all tasks involving D_1. When D_2 finishes executing $T_{2,1}$ it has to wait until the robot becomes idle in order to be unloaded.

The model uses the jukebox state information provided by the jukebox controller (see Section 4.3) to determine the state of the drives and robots. When an RSM is being unloaded from a drive and there are request units in $\mathcal{U}'$ for that RSM, we cannot stop the active unload. Thus, we must first wait until the RSM is completely unloaded to be able to load the RSM again in a drive and read the requested data from it.

5.2.1 Problem Formalization

We model the problem as

$$FF_3 \mid \tilde{d}_j, r_j, M_j, res \,.\, 1\ 1 \mid -$$

The machine environment is a flexible flow shop with three stages (FF_3). The jobs have deadlines ($\tilde{d}_j$), release times (r_j), machine eligibility restrictions (M_j) and resource constraints ($res \,.\, 1\ 1$). The optimization criterion is to find a feasible schedule ($-$).

At each stage of the FF_3 there are unrelated parallel processors. The set of processors at the first stage is the set $\mathcal{R}_l$ of loading robots. The set of processors at the

second stage is the set $\mathcal{D}$ of drives. The set of processors at the third stage is the set $\mathcal{R}_u$ of unloading robots. If there are robots capable of loading and unloading, as is in general the case in jukeboxes, then the intersection between the set of processors of the first and third stage is not empty ($\mathcal{R}_l \cap \mathcal{R}_u \neq \varnothing$).

Having a non-empty intersection between the processor sets on the different stages is a special feature of our flexible flow shop problem. We need to guarantee mutual exclusion in the used of the robots belonging to the first and third stage. We use resource constraints to indicate the presence of shared resources. We use a shared object to indicate each drive, robot, RSM, and disk in the case of double-sided disks. The first parameter of the resource constraints indicates that there are many resource types. The second parameter of the resource constraints indicates that there is one resource of each type and the third parameter indicates that a task may need at most one resource of each type.

Because the robots may be limited to serve only a subset of drives and shelves, there are job that can be executed only in a subset of resources. In the model we indicate this by using machine eligibility restrictions.

The optimization criterion is feasibility, because all the request units in $\mathcal{U}'$ have fixed deadlines.

5.2.2 Job Parameters

We now define the parameters of each of the tasks of a job. We assume that J_j is a job corresponding to the RSM that is stored in shelf x. The type of the RSM is $type_x$. We obtain these parameters from the hardware controller using the functions *getShelfNumber* and *getTypeMedium*, respectively.

The **load task T_{1j}** has the following parameters:

- The **matrix of processing times (p_{1j})** indicates the time it takes to load the RSM in each drive with each robot. Thus, p_{ki1j} denotes the time it takes to load the RSM with robot k into drive i. We obtain these values from the jukebox model.

$$p_{1j} = \begin{bmatrix} t_{load}(1,1,x,type_x) & \dots & t_{load}(1,m,x,type_x) \\ & \vdots & \\ t_{load}(l,1,x,type_x) & \dots & t_{load}(l,m,x,type_x) \end{bmatrix}$$

- The **deadline ($\tilde{d}_{1j}$)** indicates the time by which the load must finish.

- The **release time (r_{1j})** indicates the earliest time at which the load can begin.

- The **machine eligibility restrictions (M_{1j})** indicate that the load can only be executed by a given set of robots and drives. This parameter is obtained from the hardware controller using the function $resources_{load}$.
- The **resource constraints (RC_{1j})** indicate the resources involved in the operation. It includes the robot and drive to use, which must be one of the tuples of M_{1j}. It also includes the RSM involved (x) and the shared resource representing the disk when handling double-sided disks.

The **read task T_{2j}** has the following parameters:

- The **vector of processing times (p_{2j})** indicates the time it takes to read the data from the RSM with each drive. This parameter derives from the computation of the medium schedules corresponding to the RSM. The time needed to read the data with drive i is p_{i2j}.
- The **vector of deadlines ($\tilde{d}_{2j}$)** indicates the time by which the data must be in secondary storage when reading the data with the different drives. This parameter also derives from the computation of the medium schedules corresponding to the RSM. The deadline for drive i is $\tilde{d}_{i2j}$.
- The **release time (r_{2j})** indicates the earliest time at which the read can begin.
- The **machine eligibility restrictions (M_{2j})** indicate that the load can only be executed by a given set of drives. The set of drives is obtained from the hardware controller using the function $drives_{read}$.
- The **resource constraints (RC_{2j})** indicate the drive to use. The drive must be in M_{2j}. It also includes the RSM involved (x) and the shared resource representing the disk when handling double-sided disks.

The **unload task T_{3j}** has the following parameters:

- The **matrix of processing times (p_{3j})** indicates the time it takes to unload the RSM from each drive with each robot. We obtain these values from the jukebox model. The time needed to unload the RSM from drive i with robot k is p_{ki3j}.

$$p_{3j} = \begin{bmatrix} t_{unload}(1,1,x,type_x) & \dots & t_{unload}(1,m,x,type_x) \\ & \vdots & \\ t_{unload}(u,1,x,type_x) & \dots & t_{unload}(u,m,x,type_x) \end{bmatrix}$$

- The **deadline ($\tilde{d}_{3j}$)** indicates the time by which the unload has to finish.
- The **machine eligibility restrictions (M_{3j})** indicate that the unload can be executed only by a given set of robots and drives. This parameter is obtained from the hardware controller using the function $resources_{unload}$.
- The **resource constraints (RC_{3j})** indicate the resources involved in the operation. It includes the robot and drive to use, which must be one of the tuples of M_{3j}. It also includes the RSM involved (x) and the disk object corresponding to the RSM when handling double-sided disks.

5.2.3 Complexity Analysis

The fixed switching scheduling problem is $\mathcal{NP}$-hard. We prove this by performing simplifications to the problem and showing that even the simpler versions of the problem are $\mathcal{NP}$-hard.

A special case of this scheduling problem is obtained by removing the unload stage. Now the deadlines of the read tasks coincide with the deadlines of the job. The problem can be simplified even more by assuming that drives and robots are identical processors instead of unrelated processors. We can also define the load-time to be the same for every drive. These simplifications reduce the vector of processing times, the vector of deadlines and the matrix p_{1j} of T_{2j} to single values. We can also assume that all the robots can serve all the drives, thus removing the M_j parameter. We can safely do this using the fact that $\varnothing \propto M_j$[2] [83, page 20]. We can further model the deadlines as due-date times and obtain a due-date problem $FF_3 \mid res\ .\ 1\ 1 \mid L_{max}$ with $L_{max} \leq 0$.

From the complexity hierarchy for optimality criteria shown in Figure 5.4 [83] derives that

$$FF_3 \mid res\ .\ 1\ 1 \mid L_{max} \quad \propto \quad FF_3 \mid \tilde{d}_j, r_j, M_j, res\ .\ 1\ 1 \mid -$$

We can further simplify the problem by assuming that there is only one drive and one arm and that we permit to start loading a drive while it is reading. In this way we eliminate the resource constraints and we can model the processor environment as a simple flow shop with two machines.

$$F_2 \mid\mid L_{max} \quad \propto \quad FF_3 \mid res\ .\ 1\ 1 \mid L_{max}$$

And $F_2 \mid\mid L_{max} \in \mathcal{NP}$-hard [65] so $FF_3 \mid \tilde{d}_j, r_j, M_j, res\ .\ 1\ 1 \mid - \in \mathcal{NP}$-hard.

2 The symbol $\propto$ represents a *polynomial transformation* or *reduction*. $P_1 \propto P_2$ means that P_1 can be reduced to P_2. Thus, if P_1 is $\mathcal{NP}$-hard, then P_2 is also $\mathcal{NP}$-hard.

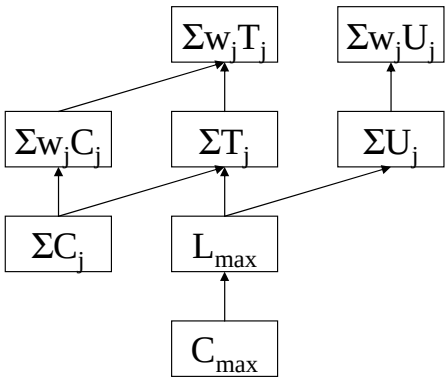


Figure 5.4: Complexity hierarchy of the optimality criteria.

5.2.4 Medium Schedule

The parameters of the read tasks T_{2j} are obtained from computing the medium schedules. We compute a medium schedule for each job in $\mathcal{J}$ and each drive in the jukebox. We, thus, obtain the parameters of the vector of processor times and deadlines of the reading tasks.

A *medium schedule (MS)* determines how the data requested from an RSM must be read. The tasks to include in the MS derive from the request units in $\mathcal{U}'$ corresponding to the RSM. If there is an intersection in the data requested by different request units, we first subdivide them into request units without intersections and compute the deadline for each of the new request units.

Each task needs time to transfer the data, which we represent as the duration of the task, and time to go to the next task, which we represent as sequence-dependent setup times. Additionally, each task has a deadline indicating the time by which the data of the request unit should be fully staged and a release time indicating the time when the transfer of data may begin.

By assigning a constant value R to the release time, we can compute the schedule for different release times. The goal of the scheduler is to find the biggest value of R that makes the set of tasks feasible. We call this value of R the *latest starting time (LST) of the MS.* The bigger the value of the LST, the later the corresponding read task T_{2j} needs to start. We compute the deadline of T_{2j} using drive i ($\tilde{d}_{i2j}$) as the LST of the MS plus the total time needed to read the requested data. Thus, the larger the value of the LST is, the more flexibility for including the job J_j in the schedule.

The model of the scheduling problem to solve is

$$1 \mid r_j = R, \tilde{d}_j, s_{ij} \mid -$$

This model is a special case of the *asymmetric travelling salesman problem with time windows (TSPTW)* [8, 81] for which all $r_j = R$. It is $\mathcal{NP}$-hard because the *travelling salesman problem* is a special case of it when $r_j = 0$, $p_j = 0$ and $\tilde{d}_j = 0$.

When the RSM is loaded, we do not need to maximize the value of R, but just find a feasible schedule for $R = r_{2j}$. This value of R is the earliest time at which the reading of the data can begin. Even in this case the problem is $\mathcal{NP}$-hard.

The set of tasks to schedule is $\mathcal{T}$. Let D be the drive for which the MS is computed. The generic task T_j—for data in RSM x—has the following parameters:

- The **duration (p_j)** indicates the time it takes to transfer the data of the request unit with the drive D. We use the function defined in the jukebox model to compute the transfer time ($p_j = t_{transfer}(D, type_x, o_j, s_j)$).
- The **deadline ($\tilde{d}_j$)** indicates time by which the data of the request unit must be staged.
- The **release time (r_j)** indicates the earliest time at which the data of the request unit can begin to be staged.
- The **sequence-dependent setup times (s_{jk})** indicate the time to go from the last byte of T_j to the first byte of T_k. We use the function defined in the jukebox model to compute the access time ($s_{jk} = t_{access}(D, type_x, o_j + s_j, o_k)$).

  We define an extra task T_0 to define the setup time of the first task in the schedule and the time is computed as $s_{0j} = t_{access}(D, type_x, o_{initial}, o_j)$, where $o_{initial}$ represents the initial position of the reading head on the RSM when the RSM is loaded into a drive.

5.2.5 Model Extension for Partially Blocking Loads and Unloads

For completeness, we present an extension of the problem model for the case when the drives are not involved in the whole load and unload operations, but only in part of them. This can be the case when the robots perform most of the RSM movement independently of the drive. We do not further work with this extension of the problem, because we do not take into account this type of jukeboxes in the hardware model.

In this extended model the jobs consist of five tasks $T_{1j}, \ldots, T_{5j}$. T_{1j} represents the part of the load that does not involve the drive, T_{2j} represents the part of the load that involves the drive, T_{3j} is the read, T_{4j} is the part of the unload that involves the drive and T_{5j} is the part of the unload that does not involve the drive.

T_{1j} and T_{2j} need the same robot R_l and, thus, include it as a resource constraint. Additionally, T_{2j} also needs the drive D_r and includes it as well as a resource constraint. T_{4j} needs both the drive D_r and a robot for the unload R_u. Finally T_{5j} needs the same robot as T_{4j} and, thus, includes it as a resource constraint.

Such a model allows some superposition between loads and reads, and reads and unloads. A drive can start reading while the last part of the unload is still being executed. If there are multiple robots or the robots have a dual-picker, a drive can go on reading from the RSM loaded, while the robot is executing the first part of the next load corresponding to the drive.

5.3 Minimum Switching Model

The *minimum switching model* is a special case of the fixed switching model, which has one job for each RSM with request units that need scheduling. This model imposes the restriction that all the data requested from an RSM must be read before the RSM is unloaded. However, given the high switching time, it makes good sense to read all data of an RSM without unnecessary switches. In Section 2.2.4 we have shown that this is a good way to use the jukebox resources efficiently. Promote-IT is based on this model and in Chapter 9 we show that Promote-IT is an efficient heuristic scheduler.

This model determines exactly what jobs need to be in the set of jobs to schedule. There is one job in $\mathcal{J}$ for each RSM that satisfies one of the following conditions:

1. The RSM has request units in $\mathcal{U}'$.
2. The RSM is in a drive—loading, loaded or reading—and there are no request units in $\mathcal{U}'$ for the RSM.
3. The RSM is being unloaded from a drive.

If there are request units in $\mathcal{U}'$ for an RSM that is loaded or being loaded in a drive, we read the data corresponding to those request units before the RSM is unloaded. This makes good use of the fact that the RSM is already loaded in a drive. However, it may lead to an unfeasible schedule if the RSM cannot be kept longer in the drive without making other RSM miss their deadlines.

When an RSM is being unloaded from a drive and there are request units in $\mathcal{U}'$ for that RSM, we cannot stop the active unload. Thus, we must first wait until the RSM is completely unloaded to be able to load the RSM again in a drive and read the requested data from it.

As a result of these rules, in the general case, there is at most one job for every RSM in the jukebox. The exception is given by the RSM that are being unloaded

Parameter	Load Task T_{1j}	Read Task T_{2j}	Unload Task T_{3j}
Processing time	$\boldsymbol{p_{ki1j}}$ (*)	$\boldsymbol{p_{i2j}}$ (*)	$\boldsymbol{p_{ki3j}}$ (*)
Deadline	$\tilde{d}_{1j}$	$\boldsymbol{\tilde{d}_{i2j}}$	$\tilde{d}_{3j}$
Release time	r_{1j}	r_{2j}	
Machine eligibility restrictions	$\boldsymbol{M_{1j}}$ (*)	$\boldsymbol{M_{2j}}$ (*)	$\boldsymbol{M_{3j}}$ (*)
Resource constraints	RC_{1j}	RC_{2j}	RC_{3j}

Table 5.1: Parameters of the job tasks of the minimum switching model. A sub-index *i* indicates a drive and a sub-index *k* indicates a robot. The parameters marked with (*) are always initialized.

by the robots at the time of building the schedule and have request units for them in $\mathcal{U}'$. In this case there are two jobs for the RSM, one representing the request units that need reading (condition 1) and another for the unload under way (condition 3).

The goal of having at most one job per RSM is to eliminate the need to check for mutual exclusion on the use of the RSM. To achieve this goal, we need to impose the restriction that the RSM are not double-sided disks. When using double-sided disks, we still should have to check for mutual exclusion on the use of the disks. We believe, that this restriction on the hardware model is not really important, because in order to make double-sided disks succeed in the market, the drives will have to be able to read double-sided disks without turning them over. We have already seen these two technological steps in the early days with floppy disks.

The minimum switching model—like the fixed switching model from whom it derives—has the peculiarity that the deadlines are specified for the reading tasks and not for the jobs. The unloads can be delayed as long as the drive is not needed for another task. Table 5.1 shows all the parameters of the jobs and indicate which parameters are always initialized at the beginning of the computation. Because the RSM are not double-sided, so we do not need to include the RSM and disk in the resource constraint parameters.

Parameter initialization

We now define how the parameters are initialized. We want to guarantee that the tasks that are already active are assigned the correct resources, i.e., the same resources on which they are executing. We set the resource constraints in such a way that no other device can be used for the tasks already executing and that the deadline of the active task coincides with the time at which the task should finish executing.

We show the initialization of the job parameters for the three special cases that derive from a busy drive. We obtain the values from the hardware controller.

If J_j corresponds to RSM x that is being loaded into drive i by robot k, and t_l is the time remaining to finish the load, do the following:

1. Add the drive and the robot to the resource constraints of the load tasks, because the load should be assigned to the same drive and robot in which it is currently active. $RC_{1j} = (D_i, R_k)$.

2. Set the processing time of the load task to the remaining time to finish the load t_l. $(p_{ki1j} = t_l) \wedge (\forall x, m \mid 1 \leq x \leq l \wedge x \neq k \wedge i \neq y \wedge 1 \leq y \leq m : p_{xy1j} = \infty)$.

3. Set the deadline of the load task to the time when the load should finish. $\tilde{d}_{1j} = t_0 + t_l$.

4. Set the resource constraints of the read task to D_i to make sure that the read task is assigned to D_i. $RC_{2j} = (D_i, \_)$.

5. Add D_i to the resource constraints of the unload task to make sure that the unload is done from D_i. $RC_{3j} = (D_i, \_)$.

If J_j corresponds to RSM x, which is loaded in drive i—either reading or idle—and t_r is the time at which the read will finish, do the following:

1. Set the release time of the read task to the time at which the read that the drive is performing should finish. If the drive is idle, then $t_r = t_0$. $r_{2j} = t_r$.

2. Add D_i to the resource constraints of the three tasks. $RC_{1j} = RC_{2j} = RC_{3j} = (D_i, \_)$.

3. Indicate that the load task does not need to be performed by setting the processing time to 0.

If RSM x is being unloaded from drive i by robot k, there is a job J_u to represent the unload, and t_u is the time remaining to finish the unload, set the parameters of J_u to the following values:

1. Add the drive and the robot to the resource constraints of the unload tasks, because the unload should be assigned to the same drive and robot in which it is currently active. $RC_{3j} = (D_i, R_k)$.

2. Set the processing time of the unload task to the time needed to finish unloading the RSM. $(p_{ki3u} = t_u) \wedge (\forall x, y \mid 1 \leq x \leq u \wedge x \neq k \wedge y \neq i \wedge 1 \leq y \leq m : p_{xy3j} = \infty)$.

3. Set the deadline of the unload task to the time by which the unload should finish. $\tilde{d}_{3j} = t_u$.

RU	Deadline (seconds)	RSM Identifier	Offset (KB)	Size (KB)	Bandwidth (Kbps)
$u_{1,1}$	50	10	60	204800	1024
$u_{1,2}$	210	10	204860	409600	1024
$u_{2,1}$	150	300	200	102400	0
$u_{2,2}$	150	300	204800	51200	0
$u_{5,5}$	90	300	409600	204800	512
$u_{8,1}$	180	720	60	409600	2048
$u_{8,2}$	180	3	614400	15360	128

Table 5.2: Request units in $\mathcal{U}'$ for the example of the minimum switching model.

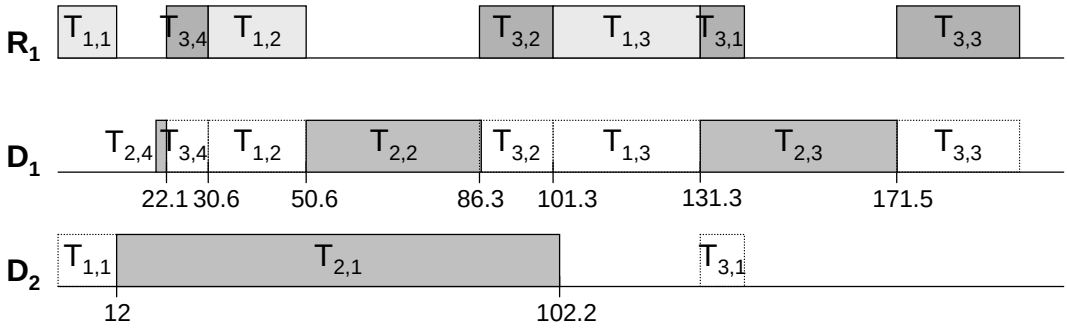


Figure 5.5: A feasible schedule for the example of the minimum switching model.

4. Indicate that the load and read tasks do not need to be executed by setting the processing time to 0.

Additionally if there are request units for RSM x in $\mathcal{U}'$, then there is another job J_j corresponding to the data that needs to be read from the RSM. If such a job J_j exists, then the model must guarantee that J_j is only scheduled after J_u finishes. Thus, the precedence constraint $J_u \rightarrow J_j$ must hold in any feasible schedule that is built. To guarantee this, we set the release time of the load task of J_j to the time at which the unload finishes ($r_{1j} = t_u$).

5.3.1 Example

We illustrate now with an example how a set of request units $\mathcal{U}'$ (shown in Table 5.2) is transformed into an instance I of the minimum switching model (shown in Tables 5.4 and 5.5). Figure 5.5 shows a feasible schedule built for the resulting I.

The jukebox has two drives and a shared robot. The drives have different characteristics. The functions shown in Table 5.3 determine the hardware behaviour. The functions are very simple; we assume, for example, that the access time does not depend on the offset of the data.

Function	$d = 1$	$d = 2$
$t_{transfer}(d, CD, o, size)$	$size/(10 * 1024)$	$size/(6.66 * 1024)$
$t_{access}(d, CD, o_s, o_d)$	$0.2 + \mid o_d - o_s \mid /1024 * 0.002$	$0.1 + \mid o_d - o_s \mid /1024 * 0.001$
$t_{load}(r, d, sh, CD)$	$10 + 0.5 * (sh \bmod 40)$	$7 + 0.5 * (sh \bmod 40)$
$t_{unload}(r, d, sh, CD)$	$5 + 0.5 * (sh \bmod 40)$	$4 + 0.5 * (sh \bmod 40)$

Table 5.3: Functions for the hardware model of the examples in this chapter.

Job	RSM	Release	Tasks	Deadline	Drive 1		Drive 2	
					p_j	s_{jk}	p_j	s_{jk}
J_1	10		T_1	50	20	[0, 0.2]	30	[0, 0.1]
			T_2	210	40	[0.6, 0]	60	[0.3, 0]
J_2	300		T_1	150	10	[0, 0.4, 0.8]	15	[0, 0.2, 0.4]
			T_2	150	5	[0.7, 0, 0.5]	7.5	[0.35, 0, 0.25]
			T_3	90	20	[1.4, 1, 0]	30	[0.7, 0.5, 0]
J_3	720		T_1	180	40	[0]	60	[0]
J_4	3	20	T_1	180	1.5	[0]	2.25	[0]

Table 5.4: Data to build the medium schedules for the example of the minimum switching model.

The state of the jukebox is the following. Drive 1 is loaded with RSM 3 and the drive is busy reading data from it for another 20 seconds. Drive 2 is empty. We assume that Drive 1 is reading the first 200 MB of data from RSM 3, so the time needed to go from the position in the RSM to the one required by J_4 is 0.6*s*.

There are four jobs that need scheduling, corresponding to the four RSM with requests. The RSM of the last job (J_4) coincides with the RSM loaded in Drive 1. Table 5.4 shows the mapping of the RSM to the jobs and the data used to build the medium schedules for each job. Figure 5.6 shows the medium schedules built for jobs J_1 and J_2 using both drives. The medium schedules of J_3 and J_4 are trivial.

Table 5.5 shows the data of the jobs to schedule. Only job J_4 has a drive constraint indicating that the read must be processed in D_1, because the RSM is already loaded in that drive. Figure 5.5 shows a feasible schedule for the job set.

5.4 Lau's Switch-Read Model

Using the description of the aggressive and conservative strategies provided by Lau et al. [61, 63] we formalize the scheduling-problem model on which their scheduler is based. The goal of presenting this formalization is two-folded. On the one hand, we show that their scheduling-problem model is different from the minimum

Job	Load (T_{1j})	Read (T_{2j})	Unload (T_{3j})
J_1	$p_{1,1} = [15, 12]$	$p_{2,1} = [60.4, 90.2]$ $\tilde{d}_{2,1} = [90.2, 110, 1]$	$p_{3,1} = [10, 9]$
J_2	$p_{1,2} = [20, 17]$	$p_{2,2} = [35.7, 53.35]$ $\tilde{d}_{2,2} = [105.5, 113.25]$	$p_{3,2} = [15, 14]$
J_3	$p_{1,3} = [30, 27]$	$p_{2,3} = [40.2, 60.1]$ $\tilde{d}_{2,3} = [180, 180]$	$p_{3,3} = [25, 24]$
J_4	$p_{1,4} = [0, \infty]$ $\tilde{d}_{1,4} = [0, 0]$ $RC_{1,4} = (D_1, \_)$	$p_{2,4} = [2.1, \infty]$ $\tilde{d}_{2,4} = [180, 180]$ $r_{2,4} = 20$ $RC_{2,4} = (D_1, \_)$	$p_{3,4} = [8.5, 5.5]$ $RC_{2,4} = (D_1, \_)$

Table 5.5: Jobs to schedule in the example of the minimum switching model.

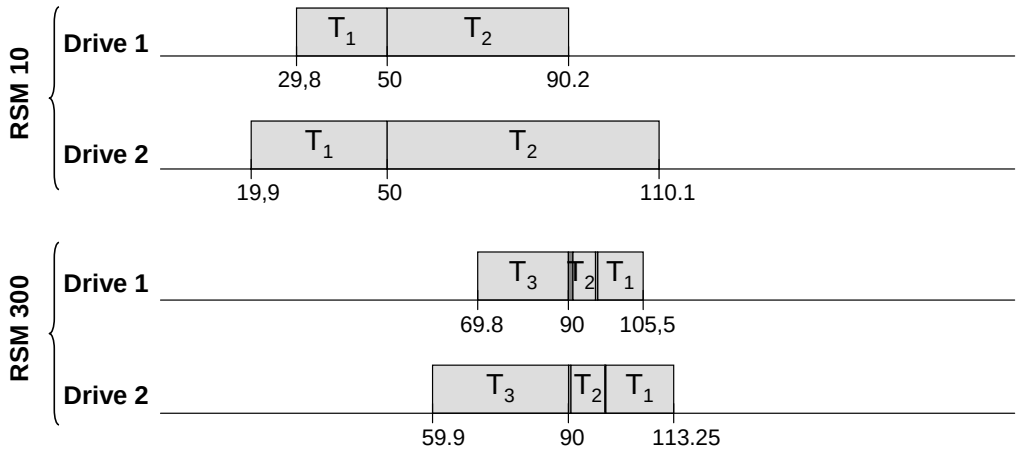


Figure 5.6: Medium schedules for J_1 and J_2 for the example of the minimum switching model. For each job the medium schedule built with Drive 1 and Drive 2 are different.

switching model. On the other hand, we show that the switch-read model puts unnecessary restrictions on the way that the resources are used by coupling the unload and load operations into a single *switch* operation. Using the switch operation has as consequence that the drives stay loaded until they are needed again. Therefore, before reading data from a new RSM, the robot must always unload the drive first and then load it with a new RSM, even if the robot and the drives had been idle before the request arrived. In Chapter 9 we show that using a switch operation has a negative influence on the performance of the jukebox scheduler.

The processor environment of the switch-read model is a *flexible flow shop* with two stages (FF_2). The first stage is to switch the RSM loaded in a drive and the second stage is to read the data from the RSM. The switch stage comprises unloading and immediately loading a drive. An RSM stays loaded in a drive until the drive is

needed for another RSM. The jobs to be processed are of the form $J_j = \{T_{1j}, T_{2j}\}$, with one task for each stage.

Like the minimum switching model, this model imposes the restriction that all data requested from an RSM must be read before the RSM is unloaded. This model also uses the concept of the *medium schedule* as presented in Subsection 5.2.4 for the minimum switching model.

Additionally, this model imposes these restrictions on the hardware model:

- Identical drives
- Constant switch time, independently of the drive, robot, shelf and type of RSM involved
- One robot

The problem is formalized as

$$FF_2 \mid \tilde{d}_j, r_j, res \,.\, 1\ 1 \mid -$$

The machine environment is a two stage flexible flow shop. The first stage has only one processor that represents the robot. The second stage has m identical processors representing the drives. The jobs have deadlines, release times, and resource constraints.

As in the case of the minimum switching model, the problem can be simplified to a simple flow shop with two machines. Therefore, this scheduling problem is $\mathcal{NP}$-hard, as well.

5.4.1 Job parameters

We now define the parameters of each of the tasks of a job J_j.

The **switch task T_{1j}** has the following parameters:

- The **processing time (p_{1j})** is a constant time t_{switch} indicating the duration of the switch.
- The **deadline ($\tilde{d}_{1j}$)** indicates time by which the switch must finish.
- The **release time (r_{1j})** indicates the earliest time at which the switch can begin.
- The **resource constraints (RC_{1j})** indicate the drive to use.

The **read task T_{2j}** has the following parameters:

- The **processing time (p_{2j})** indicates the time to read the data. This parameter derives from the computation of the medium schedule corresponding to the RSM. Note that it is a scalar and not a vector as in the minimum switching model.
- The **deadline ($\tilde{d}_{2j}$)** indicates the time by which the data must be in secondary storage. Note that it is a scalar and not a vector as in the minimum switching model.
- The **release time (r_{2j})** indicates the earliest time at which the read can begin.
- The **resource constraints (RC_{2j})** indicate the drive to use.

5.4.2 Extended Switch-Read Model

We have extended the switch-read model to a model that is more flexible in the type of hardware it can handle. The extended model can deal with non-identical drives, variable switch times, and multiple robots. However, it cannot deal with robots that have a limited scope or functionality. The switch operation must be performed by the same robot. Therefore, the robot must be able to unload the drive, moving the RSM to its corresponding shelf, and load the drive with the new RSM.

5.5 Imperative Switching Model

The *imperative switching model* is another special case of the fixed switching model where each request unit in $\mathcal{U}'$ is a separate job in $\mathcal{J}$. Consequently no medium schedule needs to be computed, because each request unit is read separately.

This model makes inefficient use of the resources, especially if the request units are small and there are multiple request units for the same RSM. In jukeboxes where the drives are much faster than the robot, the minimum switching model has even a bigger advantage over the imperative switching model. It is easy to see that the example shown for the minimum switching model is not schedulable with this model.

5.6 Periodic Quantum Model

In the periodic quantum model the robots and drives are used according to a predetermined cycle. The robot first unloads Drive 1 and loads it with another RSM. Then it does the same for Drive 2, and so on until all drives have been served and

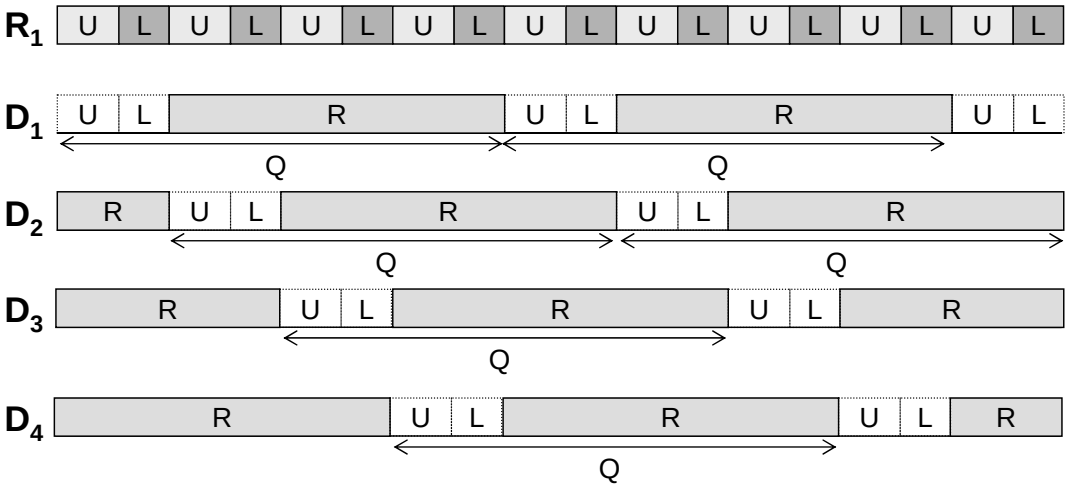


Figure 5.7: Cyclic Use of the Resources.

the cycle starts again at Drive 1. Figure 5.7 shows the cycle for a jukebox with four drives and one robot.

As the robot is used in a predetermined way, we do not need to compute a schedule for it. This way of using the robot guarantees that the use of the robot causes no resource-contention problems, because we know exactly when a robot can be used to serve a drive.

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

$$Q = m\,(t_{unload}^{max} + t_{load}^{max}) \tag{5.1}$$

We use *quantum tasks* to represent the user requests. Using quantum tasks [10] 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 the 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:

$$TT = \frac{m-1}{m}Q - t_{access}^{max} \tag{5.2}$$

Any periodic model either needs to take into account the worst-case time caused by robot contention in the execution time of the tasks or has to impose restrictions on the way the robot is used. The first approach is used by Lau et al. [63] in the time-slice algorithm (see Section 2.2.3). The worst-case time caused by robot contention is $\frac{m-1}{m}Q$, which is the time needed to perform a switch in all the other drives. The second approach is used by Golubchik et al. [36] in their algorithm Rounds (see Section 2.2.3) and in the periodic quantum model we present here.

Furthermore, any periodic model needs to couple the unload and the load. In the case of computing using the worst-case time for robot contention, 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. In the case of cyclic robot utilization, the coupling is natural to the cycle.

A shortcoming of the periodic quantum model (and any other periodic model) is that it needs to reserve the worst-case execution time of all 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, therefore, the amount of data that can be read during TT is not constant.

We define B to be the minimum amount of data that can be transferred in time TT. We compute B using a function $s_{transfer}^{min}$ that computes the amount of data that can be transferred in time TT in the worst-case scenario. When using optical drives, the worst case is determined by reading data from the inner tracks starting at track 0.

$$B = s_{transfer}^{min}(TT) \tag{5.3}$$

Another restriction of a periodic model is that it can handle only requests with one request unit for continuous media. A periodic model requires that the data of a request is stored sequentially in the medium. The scheduler needs to know a priori that a certain amount of data can be read during an instance of a task and the need to

access different parts of the RSM does not allow this predictability. It also needs the bandwidth to be constant. Thus, if the request should have multiple request units, all the request units should have the same bandwidth. Furthermore, all the data of a request needs to be stored in one RSM, because during each instance of the periodic task only one RSM can be accessed. If the data should be stored in more than one RSM, it could happen that executing an instance of a task could require more than one RSM. Thus, that instance should require an additional switch and the time of the switch should have to be taken into account into all instances, because the duration of the instance to use in the computation is the worst-case execution time.

Therefore, without loss of generality, when using the periodic quantum model, we assume that any incoming request r_k has only one request unit u_{k1} and $b_{i1} > 0$. Given that the resulting request has only one request unit, we will simplify the notation by using o_i, s_i, m_i, and b_i for the offset, size, medium, and bandwidth, respectively, instead of o_{i1}, s_{i1} m_{i1}, and b_{i1}.

A request is treated as a periodic task τ_i.[3] 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.

In the periodic quantum model the processing time of the tasks 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. 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. [5] and Bosch [14] show that a variable bit-rate stream can be treated as a constant bit-rate stream when the buffer size is large enough.

Clearly the bandwidth required by the request cannot be higher than the bandwidth offered by an individual drive. This is a restriction that does not exist in the original problem and in the other models presented in this chapter. Another disadvantage of this model is that during the last instance of the task the drive may be idle most of the time after having read the data.

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 $\lceil t_{transfer}(o_i, s_i)/TT \rceil$. Because 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. But periodic scheduling theory provides equations to easily verify the schedulability of task sets without having to build the schedules. Therefore, we use the periodic scheduler JEQS.

3 This model uses the notation corresponding to periodic scheduling theory (see Section 2.1.2).

Task Parameters

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

- The **execution time (C_i)** is always Q.

- The **period (T_i)** of the task is always a multiple of Q. We compute the period of the task as $T_i = \lfloor \frac{B}{b_i Q} \rfloor Q$, where B the amount of data read by a drive during a cycle and is computed as shown in Equation 5.3, and b_i is the requested bandwidth. We take the floor of the division, because the period needs to be a multiple of Q to be able to use quantum theory.

- The **shared resources (ρ_i)** indicate the RSM on which the data of the request is stored. The RSM is used as a shared resource, so that an RSM is not assigned to different drives during overlapping time periods. When using double-sided disks ρ_i also includes a shared resource representing the disk, because reading data from one side of the disk prevents other task from reading data from the disk, even if the RSM is different.[4]

  The current state of the jukebox is represented as shared resources. If τ_i is executing in drive j at the moment of computing the schedule (t_0), then D_j is also included in ρ_i. Therefore, the new schedule built has to assign τ_i to D_j.

  A scheduler may also decide to execute all instances of a task in a single drive and indicate the drive as a shared resource. In this case, the new task is the only task that does not include in ρ_i a shared resource representing a drive, and can so be easily distinguished when building the new schedule.

- The **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 time when the schedule is computed, 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. This parameter is used by the dispatcher to go on operating normally, even if the active schedule changes.

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

4 We use the same model for double-sided disks as discussed in Section 5.2, where each side of the disk is modelled as a separate RSM.

If there are multiple robots, we define one cycle for each robot. We also define which drives and shelves will be served by one robot.

Chapter 7 presents a scheduler that uses this model—JEQS.

5.7 Dedicated Robots Model

The goal of the dedicated robots model is to build schedules without predefining how to cluster the request units into jobs. However, this model imposes a strong restriction on the hardware model. It assumes a dedicated robot for each drive. Because dedicated robots are not frequent in tertiary-storage jukeboxes, the usability of this model is restricted to manufacturing environments.

The dedicated robots model derives from the problem model for a jukebox with only one drive. In such a jukebox, the robot is dedicated, because it can only serve one drive. For a one-drive jukebox, the problem can be easily represented as an *asymmetric travelling salesman problem with time windows*, which is a problem of the well studied family of travelling salesman problems (see Section 2.4). In the case of multiple drives, each with a dedicated robot, the problem can be represented as an extension of the *asymmetric multi-travelling salesmen problems with time windows (m-TSPTW)*.

We first model the problem for a jukebox with only one drive as an asymmetric travelling salesman problem with time windows. We then extend this model for the case of multiple drives and show the need of having dedicated robots to be able to use this model.

Basically, the model for one drive represents each request unit as a *node* with a deadline and a processing time. The processing time is the time it takes to transfer the data of the request unit using the drive in the jukebox. The *edges* in the graph represent the time it takes to jump from one node to the next. If both request units are for the same RSM, the length of the edge is the time needed to go from the offset of the last byte of the first request unit to the first byte of the second. If the request units are on different RSM, then the length of the edge is the time needed to unload the RSM of the first request unit, load the RSM of the second request unit and go to the first byte of the second request unit. There is one directed edge between each node representing the request units, because, in principle, every combination is possible.

The goal is to find a *Hamiltonian path* in this graph in such a way that the time window restrictions of each node are respected. The time restrictions of the nodes are given by the deadline of the request units. We add an initial and final node to the graph. The initial node has an outgoing edge to every other node representing a request unit and the edge weight is the time to load the RSM and go to the first

byte of the request unit. The final node has an incoming edge from every node representing a request unit with a weight representing the time to unload the RSM. The path must start at the initial node, pass once through each node representing a request unit and finish at the final node.

If the drive is busy at the time of computing the schedule, either reading or seeking, then we also add another node to the graph. The node has as deadline, the time at which the reading of the data will finish, or if it has already finished and the drive is switching, the time at which the schedule is computed. If the drive is performing a 'jump', the node has only one edge to the node representing the destination of the jump, and the value of the edge is the time needed to finish the jump.

We can thus model the scheduling problem for a jukebox with one drive with a known and well studied problem. Although this problem is $\mathcal{NP}$-hard, there are known heuristic algorithms to solve it [98, 99].

We can extend this model to the case of multiple drives by building such a graph for each drive. The goal is now to find Hamiltonian partial paths on each graph. Together, the Hamiltonian partial paths on each graph must cover all the nodes. This problem, although relatively easy to model, is far more complex than the problem for one drive: the algorithm must now decide which nodes to include in the partial path for each drive and has to guarantee that two nodes involving the same RSM are not scheduled during the same time period on different drives.

The problem for multiple drives is equivalent to the *asymmetric multi-travelling salesmen problem with time windows (m-TSPTW)* with the additional restriction that some nodes and paths are mutually exclusive and the salesmen are not identical.

There is, however, a more serious problem in the model for multiple drives than assuring the mutual exclusion on the RSM. The dedicated robots model assumes that there is a dedicated robot for each drive, because the paths on the different graphs can be computed independently. If there are shared robots, then choosing an edge in one graph and assigning it a time in the schedule, prevents choosing other edges in other graphs and the computation of the different graphs can no longer be made independently.

5.7.1 Problem Formalization

We model the scheduling problem as

$$R_m \mid \tilde{d}_j, s_{ijk}, res \,.\, 1\ 1 \mid -$$

The drives are represented as m unrelated processors, because they can have different characteristics and the processing time depends on the drive and on the RSM

being read. There are m robots, one robot dedicated to each drive. We do not need to represent the robots as processors, because operations of robot i can never occur in parallel to operations of drive i. If robot i is busy performing an operation then drive i is also busy performing that same operation. It is, thus, enough to model the processor environment by representing only the drives.

The tasks to schedule are the set $\mathcal{U}'$ of request units with added setup times. We map each $u'_{xy} \in \mathcal{U}'$ to a task $T_j \in \mathcal{T}$. A task has as deadline, which is the deadline of the request unit, and a vector of processing times indicating the time it takes to transfer the data with each drive in the jukebox. The setup time is defined for each pair of tasks using each drive. For each pair of tasks T_j and T_k, it represents the time needed to start reading the data of T_k after the data of T_j has been read. If the data of T_j is stored in a different RSM as the data of T_k, the setup time is the time to unload the RSM of T_j and load the RSM of T_k. Otherwise, the setup time is the access time to go from the last byte of T_j to the first byte of T_k.

The set of tasks $\mathcal{T}$ also includes one initial and final task for each drive. The initial task T_i^0 indicate for each drive i the setup time needed to start executing the first task. The graph representation has an edge from each of these initial tasks to every other task that is not initial. The final tasks T_i^f are used to represent the time needed to unload the RSM of the last task in the schedules. The processing time of the initial and final tasks is 0.

The optimization criterion is to find a feasible schedule for the task set. In this model, this is equivalent to finding Hamiltonian partial paths in the m graphs representing each drive, which together include all the nodes in $\mathcal{T}$. Each path corresponds to one drive and has the basic structure $T_i^0, \ldots, T_i^f$.

5.7.2 Example

We illustrate now with a simple example how a set of request units $\mathcal{U}'$ is transformed into a task set $\mathcal{T}$ of the dedicated robots model. We then show a feasible schedule for $\mathcal{T}$. Table 5.6 shows the request units that need to be scheduled. With one exception, the request units are the same as in the example for the minimum switching model (see Section 5.3.1): the deadline of $u_{8,1}$ is more restrictive and forces every feasible schedule to unload RSM 10 after the data of $u_{1,1}$ has been read.

The jukebox has two non-identical drives and two identical dedicated robots. The functions to determine the load, unload, read and access time are the same as in the example of the minimum switching model (see Table 5.3 on page 99). As in the previous example, Drive 1 is loaded with RSM 3 and the drive is busy reading data from it for another 20 seconds, and Drive 2 is empty. We assume that Drive 1 is reading the first 200 MB of data from RSM 3.

RU	Deadline (seconds)	RSM Identifier	Offset (KB)	Size (KB)	Bandwidth (Kbps)
$u_{1,1}$	50	10	60	204800	1024
$u_{1,2}$	210	10	204860	409600	1024
$u_{2,1}$	150	300	200	102400	0
$u_{2,2}$	150	300	204800	51200	0
$u_{5,5}$	90	300	409600	204800	512
$u_{8,1}$	140	720	60	409600	2048
$u_{8,2}$	180	3	614400	15360	128

Table 5.6: Request units in $\mathcal{U}'$ for the example of the dedicated robots model.

Task	Deadline	Processing times	RSM
T_1	50	$[20, 30]$	10
T_2	210	$[40, 60]$	10
T_3	150	$[10, 15]$	300
T_4	150	$[5.0, 7.5]$	300
T_5	90	$[20, 30]$	300
T_6	140	$[40, 60]$	720
T_7	180	$[1.5, 2.25]$	3
T'_1	20	$[20, \infty]$	3
T^0_1	0	$[0, \infty]$	
T^0_2	0	$[\infty, 0]$	
T^f_1	∞	$[0, \infty]$	
T^f_2	∞	$[\infty, 0]$	

Table 5.7: Tasks to schedule for the example of the dedicated robots model. The first column identifies the task, the second column indicates the deadline of the task, the third column indicates the processing time of the task using each drive, and the last column indicates in which RSM the data of the task is stored. The setup times are shown in Tables 5.8 and 5.9.

	T'_1	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T^f_1
T^0_1	0	15.2	15.6	20.2	20.6	21	30.2	11.7	0
T'_1	∞	21.7	22.1	26.7	27.1	27.5	36.7	0.6	6.5
T_1	∞	∞	0.2	30.2	30.6	31	40.2	21.7	10
T_2	∞	0.6	∞	30.2	30.6	31	40.2	21.7	10
T_3	∞	30.2	30.6	∞	0.4	0.8	40.2	26.7	15
T_4	∞	30.2	30.6	0.7	∞	0.5	40.2	26.7	15
T_5	∞	30.2	30.6	1.4	1	∞	40.2	26.7	15
T_6	∞	40.2	40.6	40.2	40.6	41	∞	36.7	25
T_7	∞	21.7	22.1	26.7	27.1	27.5	36.7	∞	6.5

Table 5.8: Setup times for Drive 1 for the example of the dedicated robots model.

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T^f_2
T^0_2	12.1	12.3	17.1	17.3	17.5	27.1	8.6	0
T_1	∞	0.1	26.1	26.3	26.5	36.1	17.6	9
T_2	0.3	∞	26.1	26.3	26.5	36.1	17.6	9
T_3	26.1	26.3	∞	0.2	0.4	38.1	22.6	14
T_4	26.1	26.3	0.35	∞	0.25	38.1	22.6	14
T_5	26.1	26.3	0.7	0.5	∞	38.1	22.6	14
T_6	36.1	36.3	38.1	38.3	38.5	∞	35.6	24
T_7	17.6	17.8	22.6	22.8	23	32.6	∞	5.5

Table 5.9: Setup times for Drive 2 for the example of the dedicated robots model.

The request units are transformed into the tasks that need to be scheduled. These tasks are shown in Table 5.7. T'_1 represents the task currently active in Drive 1. The duration of T'_1 is set to 20, which is the time needed to finish executing the read. T^0_1 and T^0_2 represent the initial tasks of the graphs corresponding to Drive 1 and Drive 2, respectively, and T^f_1 and T^f_2 represent the final tasks. Tables 5.8 and 5.9 show the setup times when using Drive 1 and Drive 2, respectively.

Figure 5.8 shows a feasible schedule for this example. All the tasks are scheduled before their deadline and there are no conflicts in the use of the resources. Note that RSM 10 is loaded twice, each time to read the data of one of the tasks corresponding to it. Any feasible schedule for this task set needs to unload RSM 10 after reading the data of T_1, because otherwise the deadline of T_6 cannot be met.

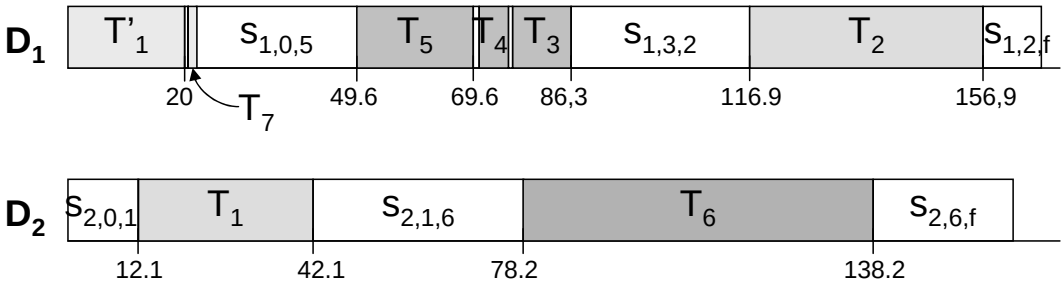


Figure 5.8: A feasible schedule for the example of the dedicated robots model.

5.8 Optimal Model

The optimal model can represent all possible ways of grouping the request units in $\mathcal{U}'$ into jobs to solve the *switch/no-switch conflict*. The jobs have the structure described for the fixed switching model in Section 5.2.

Each job represents request units for one RSM. In order to find the optimal schedule, all the possible ways of clustering the request units into jobs must be analyzed. Each clustering represents a different job set $\mathcal{J}$ of the fixed switching model.

Finding a solution for the optimal model requires finding a solution for at least one of the possible job sets. In the worst case, when no job set is feasible, all job sets need to be analyzed. Clearly, analyzing all the sets results in an exponential algorithm. The optimal scheduler presented in Chapter 7 prunes the tree of possible sets in an efficient manner by using constraint logic programming. However, the search tree is so big, that it is not possible to use the optimal scheduler for $\mathcal{U}'$ with more than a few request units per RSM.

To reduce the computational complexity the optimal scheduler imposes some restrictions on the hardware model. It assumes that the drives are identical, that the access time is constant, and that the RSM are single-sided disks. The optimal scheduler also restricts the requests to ASAP requests, because it uses the minimization of the response time as a parameter to prune the tree of possible schedules.

5.9 Summary

In this chapter we presented several alternative models that capture the environment of the scheduling problem with some restrictions. Table 5.10 presents a summary of the restrictions of each model.

The original scheduling problem is so complicated that we could not find a representation of the full problem using scheduling theory. The model corresponding

to the original problem, the *optimal model*, is equivalent to finding a solution to any of the possible ways of clustering the request units for one RSM into jobs that need a load, read and unload operation.

The most general of the models is the *fixed switching model*, which can deal with any type of hardware and request. This model only imposes a fixed clustering of the request units for an RSM into a job. However, it can handle any possible clustering. We use the fixed switching model to define the optimal model.

The *minimum switching model* restricts the unloading of an RSM to the moment when all the data requested from it has been read. We use the medium schedules to determine the sequence to read the data from each individual RSM. Using this model we cannot build a schedule in which the data of an RSM is read in multiple steps with intermediate switches, although such a schedule may be the optimal schedule. However, imposing this restriction on the resource utilization results in good system performance. During a switch the drive cannot be used to read data, therefore, bandwidth is lost during each switch. This model minimizes the number of switches and, thus, strives to minimize the lost bandwidth. This model imposes one restriction on the hardware model: the RSM must not be double-sided disks that need turning. We believe that the restriction on this type of disk will soon become irrelevant, because in order to make double-sided disks a commercial success the drives must be able to read double-sided disks without turning them.

The *switch-read model* is the model underlying the aggressive and conservative strategies proposed by Lau et al. Apart from the restrictions imposed by the minimum switching model, this model imposes additional restrictions on the robot usage, because the unload and load operations are coupled into a single switch operation. Thus, the drives stay loaded until they are needed again. As a consequence a scheduler based on this model has longer response times than a scheduler based on the minimum switching model (see Chapter 9). Additionally, the switch-read model imposes strong restrictions to the hardware model. We have relaxed some of these restrictions on the *extended switch-read model*, which can deal with non-uniform drives, variable load and unload times and multiple robots. However, this extended model still imposes restrictions on the scope and functionality of the robots.

The *imperative switching model* forces the unloading of the RSM after reading the data of a request unit. The schedules built to fit this model make unnecessary switches, resulting in a poor use of the resources.

The *periodic quantum model* uses the robot in a cyclic way, switching the RSM in the drives at periodic times. Every time an RSM is loaded into a drive only a limited amount of data can be read from it. The model handles each request as a periodic task. A strong restriction of this model is that it can only handle requests with one request unit and the data must be continuous (e.g., video). Additionally,

the bandwidth that can be requested is limited by the bandwidth offered by a drive. This model also puts restrictions on the hardware model, as it can only deal with identical drives and every robot must be able to load and unload.

Finally, The *dedicated robots model* captures only one subset of the hardware model. It assumes that the robots are dedicated resources, i.e., that there are as many robots as drives and each drive uses only one robot to load and unload the RSM. Although, this type of hardware is not realistic in a jukebox and, therefore, the model is not useful to build a jukebox scheduler, the model could be used in the manufacturing scheduling application discussed in Section 2.3.

In this chapter we have shown also that the scheduling problem is $\mathcal{NP}$-hard. We have done this by showing that our simplifications of the problem are already $\mathcal{NP}$-hard.

Model Name	**Restrictions**		
	HW Model	**Resource Usage**	**Requests**
Fixed Switching		• Fixed clustering of request units	
Minimum Switching	• Single-sided RSM	• Read all data from an RSM before unloading it	
Extended Switch-Read	• Single-sided RSM • Restriction on robot scope and functionality	• Coupled unload and load • Read all data from an RSM before unloading it	
Switch-Read	• Single-sided RSM • Identical drives • Constant switch time • One robot	• Coupled unload and load • Read all data from an RSM before unloading it	
Imperative Switching		• Execute load and unload for each request unit	
Periodic Quantum	• Identical drives • Restriction on robot functionality	• Coupled unload and load • Cyclic use of robot • Forced unload	• One request unit • Only continuous media • Restricted bandwidth
Dedicated Robots	• Dedicated robots		

Table 5.10: Summary of the restrictions of the scheduling-problem models.

Chapter 6

Promote-IT

Promote-IT (Promote In Time) is a heuristic jukebox scheduler based on the *minimum switching model* (see Section 5.3). Promote-IT owes its name to the fact that it guarantees that the data is promoted into secondary storage in time. As we have shown in the previous chapter, the scheduling problem is $\mathcal{NP}$-hard. Therefore, there is no optimal polynomial algorithm to solve the problem and the only way to build an on-line system is to use a heuristic algorithm.

We define different 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)*. The strategies define how the jobs must be sorted and incorporated into the schedule. Because there is an exponential number of possible assignments of resources to the jobs, we define a branch-and-bound algorithm to prune the tree of possible assignments that uses the *best-drives heuristic*.

Promote-IT uses early dispatching to improve the utilization of the jukebox resources. Early dispatching makes the Back-to-Front strategies—LDL and LSTL—competitive.

6.1 Scheduling Algorithm

Promote-IT refines the generic scheduling algorithm presented in Section 3.5. It represents the scheduling problem as an instance of the minimum switching model, and tries to build a feasible scheduler for it.

The structure of the scheduling algorithm of Promote-IT 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. Model $\mathcal{U}'$ as an instance of the minimum switching model. We represent the instance of the problem by the set $\mathcal{J}$ of jobs to schedule.

3. Compute the *medium schedules*. For each RSM, compute m medium schedules—one MS for each drive. Set the parameters of the duration and deadline of the read tasks T_{2j} to the corresponding values of the computed MS (as explained in Section 5.2)

4. Compute the resource assignment. The algorithm must incorporate each job $J_j \in \mathcal{J}$ 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 $S^x = \varnothing$. 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 best solution is the earliest candidate starting time for which step 4 could compute a feasible schedule ($min\{st_k^x \mid S^x \neq \varnothing\}$). 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.

Steps 2 and 3 are structured differently than in the generic algorithm presented in Section 3.5. We now divide the modelling of $\mathcal{U}'$ into an instance of the scheduling problem into two steps. In step 2 the set of jobs is determined, following the principles of the minimum switching model discussed in Section 5.3. However, the parameters of the read tasks T_{2j} are still undetermined. Only once the medium schedules are computed in step 3, all the parameters of the instance of the scheduling problem are defined. In step 4, we refine the meaning of finding a feasible schedule to finding a resource assignment for the jobs in $\mathcal{J}$. In the next paragraphs we describe the heuristic used to determine the starting time.

We determine the candidate starting time for an ASAP request using a variation the bisection method. The bisection method is a simple iterative method to find roots of mathematical functions. In our case, the goal is to find the smallest value for which the algorithm succeeds. We use an interval, which has as lower bound the highest value for which the algorithm failed, and as upper bound the lowest value for which the algorithm succeeded. We reduce the size of the interval until it reaches a lower limit ε in the size of the interval.

However, if we view the success of the algorithm as a function $success(st)$ on the starting time, this function is not monotonous. The scheduling algorithm cannot guarantee that whenever it can build a feasible schedule for the candidate starting time x it can also build a feasible schedule for all candidate starting times greater

than x. Therefore, in any interval, there may be multiple zeros for the function *success*(*st*) and the bisection method will converge to one of them. However, the situations in which the function *success*(*st*) is not monotonous are rare, thus, we can use this method with a high degree of confidence.

The scheduling problem is monotonous and an optimal algorithm should be able to find a feasible schedule for every $y \geq x^*$, where x^* is the minimum possible starting time for the request. But there is no optimal polynomial algorithm for the scheduling problem, because the problem is $\mathcal{NP}$-hard.

The first candidate starting time is the deadline of the request. If the deadline is infinite, then it uses a predefined 'big' candidate starting time. If the algorithm fails to build a feasible schedule for that candidate starting time, then the algorithm stops. However, if it succeeds, there may be a smaller starting time for which a feasible schedule can be built. As lower bound for the bisection interval, it tries the time at which the schedule is computed. If it is possible to build a feasible schedule for this time, then the request can start immediately and the algorithm stops. Otherwise it suggest an intermediate time between this last time and the first time. The algorithm continues to reduce the size of the interval until the size reaches ε.

The maximum number of candidate starting times n to try is derived from the formula that determines the convergence speed of the bisection algorithm: $(\tilde{d}_i - t_0)/2^n < \varepsilon$. If the request had an infinite deadline, then $\tilde{d}_i$ must be replaced in the formula by the first candidate time tried.

In the implementation we use a variation of this method which converges to a solution faster and provides slightly better solutions. This variation first tries successively bigger starting times, starting with the minimum response time, until it succeeds. Only then it uses the bisection method between the last unsuccessful candidate starting time and the successful starting time.

6.2 Scheduling Strategies

We define four strategies to incorporate the jobs to the schedule, which are shown in Figure 6.1. The strategies are defined by two axes indicating the way in which the jobs are incorporated to the schedule: *Front-to-Back (F2B)* and *Back-to-Front (B2F)*, and the parameter of the tasks to use for sorting the jobs: *deadline* or *LST* .

When using a 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. The tasks are incorporated F2B as well, 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

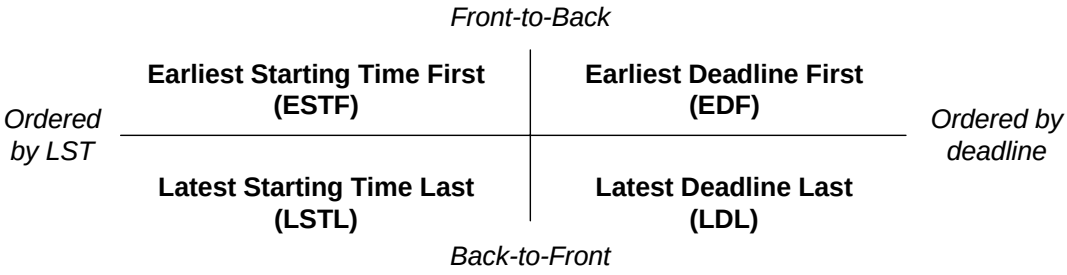


Figure 6.1: Classification of the scheduling strategies.

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 to place the most 'restrictive' tasks at the front of the schedule and the less restrictive tasks at the back of the schedule.

The 'restrictiveness' of a job can be determined in different ways, in our algorithm we use either the *deadline* or the *latest starting time* of the read tasks. We define the *latest starting time (LST)* of a task as the latest time at which the task must start executing in order not to miss its deadline ($lst_j = \tilde{d}_j - p_j$).

The principle of Front-to-Back derives from Jackson's *earliest due date (EDD)* algorithm [50] defined in the uniprocessor environment for tasks with due dates ($1 \parallel L_{max}$). The tasks are ordered by increasing due date (alternatively deadline) and incorporated into the schedule in that order, each task as early as possible. EDD is optimal in the uniprocessor environment. It is optimal as well when the tasks have deadlines and the goal is to find a feasible schedule ($1 \mid \tilde{d} \mid -$), in which case it is generally refereed as the *earliest deadline first (EDF)* algorithm.[1]

The EDF algorithm is a simple polynomial algorithm. It keeps a lower bound *lb* that indicates the time that the next task incorporated to the schedule will be assigned. If this time is later than the latest starting time of the task, then the task cannot be added and there is no feasible schedule. Thus, the first task in the schedule is scheduled to start at time t_0, where t_0 is the time at which the schedule is computed, and the last task is scheduled to finish at the time given by the sum of the processing times of each task $t_0 + \sum_{j=1}^{n} p_j$.

We define the *latest deadline last (LDL)*, which sorts the tasks in decreasing order of deadline and incorporates the tasks to the back of the schedule. The algorithm keeps an upper bound *ub* that indicates the latest time at which a task incorporated to the schedule can finish. Each task is scheduled at the latest time it can be incorporated without missing its deadline and respecting the value of the upper bound of the schedule. Thus, each task T_j is assigned the starting time $min(\tilde{d}_j, ub) - p_j$. If this

1 Strictly speaking the EDF algorithm [48] was defined for tasks with arbitrary arrival times and the possibility to preempt the active task, so using this name for the $1 \mid \tilde{d} \mid -$ is not completely correct.

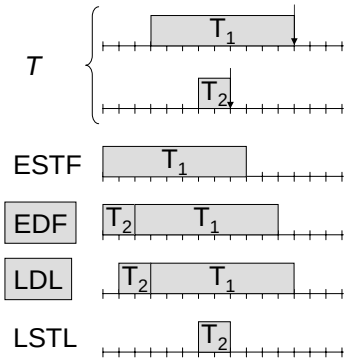


Figure 6.2: Scheduling strategies in a uniprocessor environment. ESTF and LSTL are not successful.

time is lower than t_0, then the schedule is not feasible. The starting time assigned to the first task is greater than or equal to t_0.

In the uniprocessor environment LDL is equivalent to EDF, because it produces the same sequences. Consequently, the LDL algorithm has the same complexity as the EDF algorithm and is optimal. Additionally, LDL has the advantage that it provides the latest starting time at which the schedule must begin executing so that no task misses its deadline. The schedule has 'holes' in which the processor is idle. If desired, these 'holes' can be eliminated easily, by dispatching the next task to the processor whenever the processor becomes idle, or by re-computing the assigned times of the tasks in a front-to-back manner, assigning each task the finishing time of the previous task in the schedule.

In the parallel processor environment neither EDF nor LDL are optimal. Because the scheduling problem in the multiprocessor environment is $\mathcal{NP}$-hard, there are no optimal polynomial algorithms that can solve it. Therefore, EDF and LDL cannot be optimal. Clearly, EDF and LDL are not optimal for the more complex flexible flow shop environment problem we are trying to solve.

However, although EDF and LDL are not optimal in our problem environment, we believe that they are a good basis for building simple polynomial algorithms to solve the scheduling problem we are concerned with. In Chapter 9 we show through experiments that this claim is valid.

The other two strategies we define—*earliest starting time first (ESTF)* and *latest starting time last (LSTL)*—sort the tasks using the latest starting times instead of the deadlines. Although sorting the tasks by LST is not optimal in the uniprocessor environment this way of ordering the tasks can be effective in the multiprocessor environment. Figure 6.2 shows that ESTF and LSTL are not optimal in the uniprocessor environment: the task set can be scheduled using EDF and LDL, but not ESTF and LSTL.

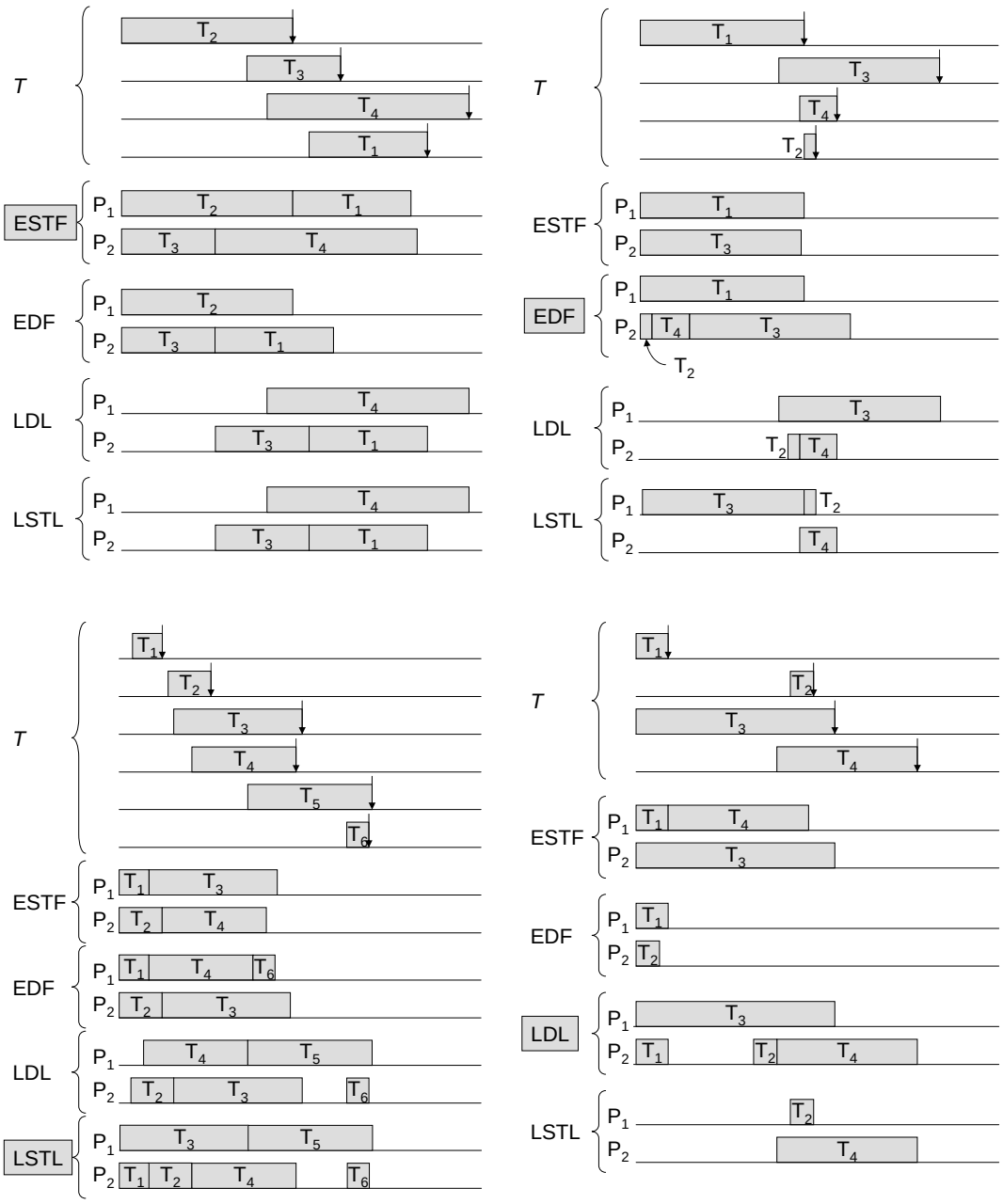


Figure 6.3: Scheduling strategies in a multiprocessor environment (P_2). In each example one strategy is successful (boxed).

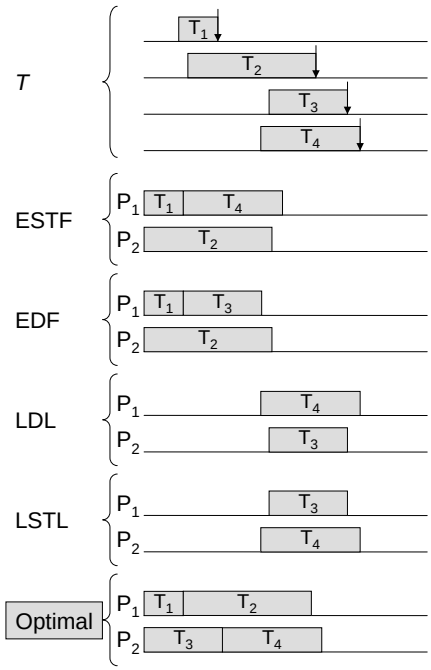


Figure 6.4: Example where no strategy is successful, although a feasible schedule exists.

In the multiprocessor environment none of the strategies is absolutely better than the others. Each strategy can find schedules that cannot be found by the others. Figure 6.3 shows examples where only one strategy is successful in finding a feasible schedule. Figure 6.4 shows an example where no strategy successfully finds a feasible schedule, although the task set is feasible.

In our scheduling problem we sort the jobs using the parameters of the read tasks T_{2j}, because initially these tasks are the only tasks that have defined deadlines. However, the deadline and LST of the read tasks are not unique values, because they depend on the drive to use. Both the deadline and the processing time of T_{2j} are vectors. In all cases we use the worst-case value of the corresponding parameter of T_{2j}.

6.3 Drive and Robot Schedules

As described in the model, a schedule consists of *m drive schedules* $D_1, \ldots, D_m$ and *r robot schedules* $R_1, \ldots, R_r$. We define for each drive schedule D_i a time window in which new tasks can be incorporated to the schedule. This window is determined by a lower and upper bound, lb_i and ub_i, respectively. These two parameters derive directly from the definition of the F2B and B2F algorithms.

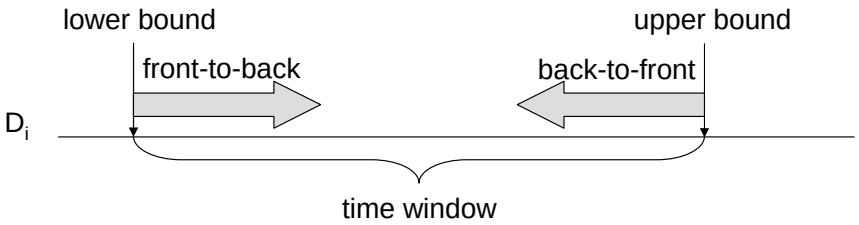


Figure 6.5: Time window of the drive schedule. The time window shrinks as jobs are added to the drive schedule.

Figure 6.5 shows the time window of a drive schedule and how the window shrinks when jobs are added to the drive schedule. When using F2B the lower bound increases its value as jobs are added, while the upper bound does not change. When using B2F the upper bound decreases its value as jobs are added, while the lower bound does not change. The initial value of the upper bound is infinite and the initial value of the lower bound is t_0, the time at which the schedule is being computed.

Restricting the scheduling of the new jobs to the time window makes the algorithm to add a new job to a drive schedule very simple. The algorithm must only try to include the job at the corresponding end of the window and if it fails then the job is not schedulable using that drive.

The jobs are sorted according to the strategy used: by increasing values when using F2B and by decreasing values when using B2F. If we ignore the shared robots and analyze what happens when using F2B we see that each job is scheduled as early as possible. If there should be a job that could be scheduled earlier than the lower bound, then the previous job could also have been scheduled earlier and the lower bound should be lower. However, the presence of shared robots and non-constant load and unload times, reduces this rule to a simple heuristic. The shared robots create situations in which the window has to shrink more than needed, in order to be able to use the robots.

The robot schedules R_i have no windows restricting when the jobs can be scheduled. The loading and unloading tasks of the different drives can be interleaved on the robots. The scheduler keeps an ordered list of tasks that have been scheduled on R_i and these tasks must not overlap. Thus, adding a new task on R_i requires finding an appropriate hole in the list of tasks.

6.4 Model Extension

We extend the minimum switching model with some parameters, which simplify the formalization of the scheduling algorithm. These new parameters are not strictly necessary to define the scheduling problem, which is why they were not presented

Parameter	Load Task T_{1j}	Read Task T_{2j}	Unload Task T_{3j}
Processing time	p_{ki1j}	p_{i2j}	p_{ki3j}
Deadline	$\tilde{d}_{1j}$	$\tilde{d}_{i2j}$	$\tilde{d}_{3j}$
Release time	r_{1j}	r_{2j}	$\boldsymbol{r_{3j}}$ (*)
Resource constraints	RC_{1j}	RC_{2j}	RC_{3j}
Machine eligibility restrictions	M_{1j}	M_{2j}	M_{3j}
Assigned time	$\boldsymbol{a_{1j}}$ (*)	$\boldsymbol{a_{2j}}$ (*)	$\boldsymbol{a_{3j}}$ (*)

Table 6.1: Parameters of the job tasks. A sub-index *i* indicates a drive and a sub-index *k* indicates a robot. The parameters marked with (*) are new.

in the model, but make the computation and verification of the algorithm easier. Table 6.1 shows all the parameters of the jobs and indicates the new parameters.

We add to each task the *assigned time* (a_{xj}) that indicates the time at which the task must start execution. The value of the assigned time is set when the task is assigned the resource where it will execute.

We also add a release time parameter to the unload task (r_{3j}) to indicate the earliest time at which the task may be scheduled. This parameter is also set as the algorithm assigns the jobs to the drives.

Sometimes we will abuse the notation and refer to $\tilde{d}_{2j}$ and p_{2j} as a single value instead of a vector meaning really $\tilde{d}_{i2j}$ and p_{i2j} respectively, where i is the drive assigned to T_{2j}. We will do the same with the processing time of the load and the unload, referring to them as p_{1j} and p_{3j} when we really mean p_{ki1j} andp_{ki3j}, where i is the drive and k is the robot assigned to the task.

The following relations must hold for every feasible schedule:

$$r_{xj} \leq a_{xj} \leq \tilde{d}_{xj} - p_{xj} \tag{6.1}$$

$$a_{xj} + p_{xj} \leq a_{x+1,j} \tag{6.2}$$

$$r_{1j} \geq lb_i \tag{6.3}$$

$$\tilde{d}_{3j} \leq ub_i \tag{6.4}$$

The first relation indicates that every task must begin execution at a time later than its release time and finish execution before its deadline. The second relation establishes that each task of a job must finish executing before the next task of the job begins. The third relation establishes that a load task on drive i cannot begin earlier than the lower bound of D_i. And the last relation establishes than an unload task on drive i cannot finish later than the upper bound of D_i.

Job	Load (T_{1j})	Read (T_{2j})	Unload (T_{3j})
J_1	$p_{1,1} = [15, 12]$	$p_{2,1} = [60.4, 90.2]$ $\tilde{d}_{2,1} = [90.1, 110, 2]$	$p_{3,1} = [10, 9]$
J_2	$p_{1,2} = [20, 17]$	$p_{2,2} = [35.7, 53.35]$ $\tilde{d}_{2,2} = [105.2, 112.9]$	$p_{3,2} = [15, 14]$
J_3	$p_{1,3} = [30, 27]$	$p_{2,3} = [40.2, 60.1]$ $\tilde{d}_{2,3} = [180, 180]$	$p_{3,3} = [25, 24]$
J_4	$p_{1,4} = [0, \infty]$ $\tilde{d}_{1,4} = [0, 0]$ $RC_{1,4} = (D_1, R_1)$ $a_{1,4} = 0$	$p_{2,4} = [2.1, \infty]$ $\tilde{d}_{2,4} = [180, 180]$ $r_{2,4} = 20$ $RC_{2,4} = (D_1, \_)$ $a_{2,4} = 20$	$p_{3,4} = [8.5, 5.5]$ $r_{3,4} = 21.60$ $RC_{2,4} = (D_1, \_)$

Table 6.2: Job set of the example after including the inflexible tasks ($T_{1,4}$ and $T_{2,4}$).

6.5 Resource Assignment

We now explain how the resources are assigned to the tasks. We divide the tasks to schedule into *inflexible* and *flexible* tasks. The algorithm first incorporates the inflexible tasks to the schedule and then the flexible tasks.

The *inflexible tasks* represent the RSM that are active in a drive. We call them inflexible, because they are restricted in the resources and time slots they may be assigned. Scheduling these tasks is straightforward, because there is no real decision to make. They must simply be assigned to the same resources in which they are actually executing.

We add the inflexible tasks to the front of the schedule, thus, modifying the lower bound of the drive schedules. Table 6.2 shows the job set used in the previous chapter (see Table 5.5 on page 100) after the assignment of the inflexible tasks $T_{1,4}$ and $T_{2,4}$.

The *flexible tasks* are all the tasks that are not inflexible. Scheduling these tasks is the interesting problem because there is an exponential number of possible assignments and, thus, the algorithm must prune the tree in an effective way.

The principle of the algorithm is that for each job it first chooses a drive and then tries to find robots that will allow loading and unloading the drive in time. It starts by choosing the drive, because it is the resource that is involved in all the tasks of the job and, thus, must be reserved for the whole execution of the task. When using F2B, the algorithm first schedules the load, then the read and finally the unload. It incorporates each task to the schedule as early as possible. When using B2F, instead, the algorithm first schedules the unload, then the read and finally unload; each task as late as possible.

The algorithm begins by sorting the jobs from which no task has yet been scheduled, using one of the strategies defined in Section 6.2. We call this subset of jobs $\mathcal{J}'$. It then incorporates the jobs in $\mathcal{J}'$ to the schedule in the appropriate order.

6.5.1 Branch-and-Bound Algorithm

The full tree of possible assignments of resources to the jobs in $\mathcal{J}'$ has m^n nodes, where m is the number of drives, and $n = |\mathcal{J}'|$ is the number of jobs. Level j in the tree represents the incorporation of job J_j into the schedule. Each level has m nodes, one for each drive in the jukebox. In principle, each job can be assigned to any of the drives. Thus, traversing the full tree needs exponential time.

Our algorithm uses a branch-and-bound algorithm to search for a solution in the tree and prune paths that are not promising. The algorithm uses the *best-drives heuristic*. At each level, the algorithm assigns a heuristic value to each of the drives. This value depends on the bounds of the corresponding drive-schedule window. All drives that have the maximum heuristic value are considered 'best'.

The algorithm traverses the tree in a depth-first manner. When it succeeds in going down in the tree to node *i* on level *j*, it prunes all the siblings of node *i* that are not 'best'.

When using B2F, the algorithm uses the upper bound ub_i of the drive schedules as parameter for the heuristic. The heuristic function is $h(ub_i) = ub_i$, because the drive that may allow the job to finish later should be analyzed first.

When using F2B, the algorithm uses the lower bound lb_i of the drive schedules as parameter for the heuristic. The heuristic function is $h(lb_i) = -lb_i$, because the drive that may allow the job to be scheduled earlier should be analyzed first.

The intuition behind the best-drives heuristic is that at each point with multiple equally best drives, the algorithm is choosing one. This choice may turn out to be a bad choice when other jobs are incorporated into the schedule. The algorithm is then able to try another best drive and see if this yields a feasible schedule. This strategy is especially effective when incorporating the jobs B2F, because the algorithm first incorporates the jobs with less restrictive deadlines and when it tries to incorporate the jobs with more restrictive deadlines it may be restricted to drives with a high lower-bound.

Figure 6.6 shows a task set that is schedulable using the best-drives heuristic and the LDL strategy. The jukebox has three identical drives and one shared robot. Drive 2 and Drive 3 are busy reading data corresponding to $T_{2,6}$ and $T_{2,5}$, respectively. Four more jobs need to be incorporated into the schedule in the order J_1, J_2, J_3, and J_4.

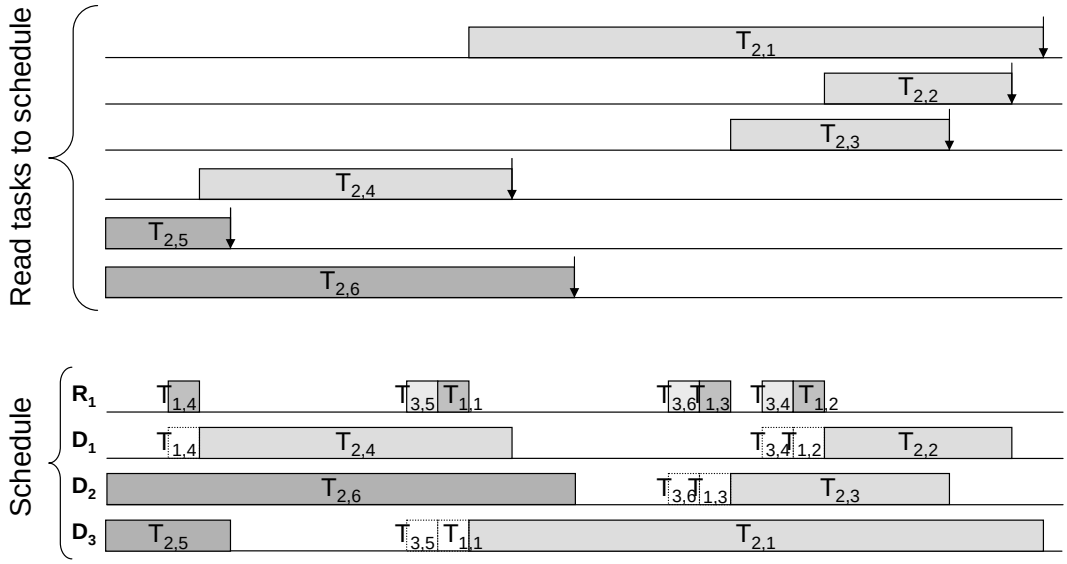


Figure 6.6: Example of a feasible schedule built using the best-drives heuristic.

Figure 6.7 shows the search tree that the algorithm traverses. At each step it shows which parts of the tree are pruned using the best-drives heuristic. When the algorithm starts, all drives have the same heuristic value, which is infinite, because no job has yet been added to the end of the schedule. At this step all drives are 'best'. The algorithm can choose any of the drives to schedule J_1. It chooses D_1 and it succeeds. It tries to prune all the 'non-best' siblings, but there are none. It then incorporates J_2 into the schedule. The computation of the heuristic values marks D_2 and D_3 as 'best', so the algorithm first chooses one of them. It succeeds in assigning J_2 to D_2 and, therefore, it prunes all the 'non-best' siblings, which in this case is the branch of D_1. It succeeds in assigning J_3 to D_3, which is the only 'best' drive at level 3, and prunes the branches of D_1 and D_2. However, the algorithm does not succeed in assigning J_4 to any of the drives and has to backtrack to a place in the tree with open branches. It tries then the assignment of J_2 to D_3 and it succeeds. This branch does not lead to a good solution either, because after J_3 is assigned to D_2, the algorithm does not succeed in scheduling J_4.

At this point, the branch corresponding to the assignment of J_1 to D_1 has been exhausted and can be discarded. The algorithm fails to assign J_1 to D_2 but succeeds in assigning it to D_3. This leads to a feasible schedule, assigning J_2 to D_1, J_3 to J_2 and J_4 to D_1.

An important property of the best-drives heuristic is that with a single shared robot, the possible number of points with multiple best candidate drives is restricted. In the first round of the algorithm the number of drives is at most m, at the second

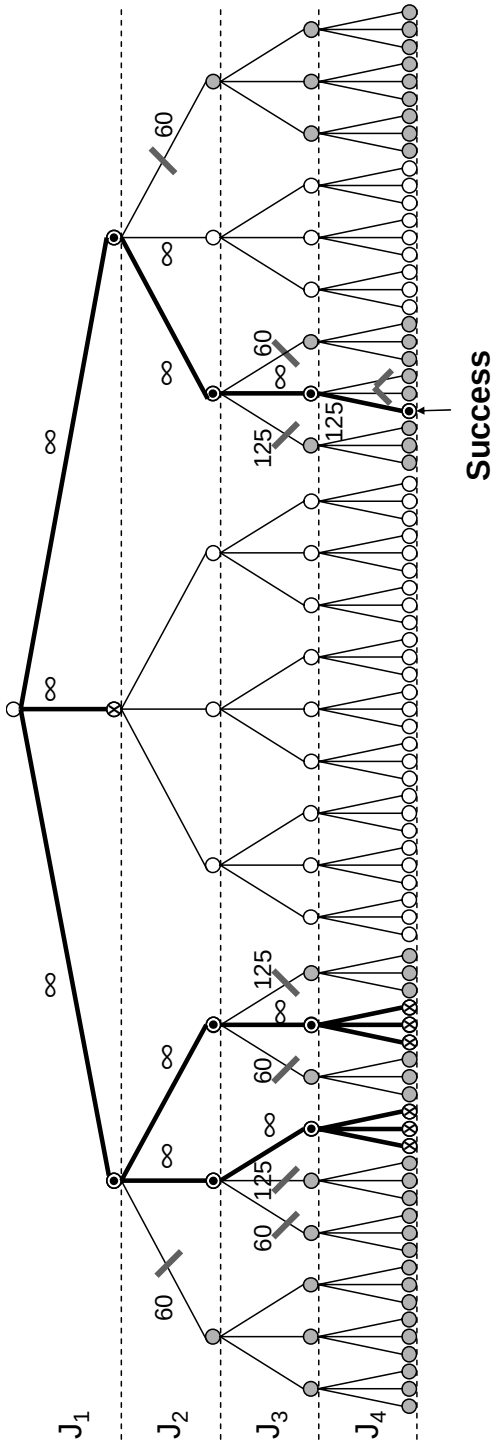


Figure 6.7: Search tree of the example of a feasible schedule using the best-drives heuristic for the tasks shown in Figure 6.6.

step $m-1$, and so forth until after the first $m-1$ rounds there is always only one best candidate drive. The load and unload influence the value of the bounds of the drive: when using F2B the value of the lower bound is established after the unload has been scheduled, while when using B2F the value of the upper bound is established after the load has been scheduled. Therefore, if there is a single shared robot, two drive schedules with a job assignment to each, must have different values for the bound defining the ordering.

The maximum number of nodes in the tree that are analyzed is $m!\,m\,n$, where n is the number of jobs and m is the number of drives. So, the complexity of the algorithm is polynomial in the number of jobs, because m is constant.

Without shared robots we cannot guarantee that the complexity of the algorithm is polynomial. Let us assume that there are two identical drives with identical dedicated robots and all the jobs have the same characteristics. Then after scheduling the first two jobs, there is again a situation with two best drives. This situation will present itself every two steps and thus the maximum number of schedules the algorithm will try is $2^{\frac{n}{2}}$. But the probabilities of having such a job set are low—especially because the load and unload times depend on the shelf where the RSM is stored. Thus, in practice the maximum number of schedules to build is still $m!$.

6.5.2 Job Incorporation

We now explain how we incorporate one job into the schedule using the Front-to-Back and Back-to-Front strategies. When using F2B, the algorithm schedules first the load, then the read and finally the unload. It schedules each task as early as possible. When using B2F it first schedules the unload, then the read and finally the load—each task as late as possible.

With drive D_i assigned by the branch-and-bound algorithm and the successful incorporation of the job, the time window of D_i is adjusted. With F2B the lower bound of D_i increases its value, while the upper bound does not change. With B2F the upper bound decreases its value and the lower bound does not change.

The robot schedules do not have a time window, but a list of scheduled tasks. The new tasks can be added in the holes in the list. The tasks are ordered by assigned time. Each task can be viewed as an interval during which the robot is reserved. The lower bound of the interval is the assigned time of the task and the upper bound is the assigned time plus the duration.

The algorithm to add a task to the robot schedules uses two different strategies called *earliest fit* and *latest fit*. The *earliest fit* strategy looks for the first hole in the list in which the task fits, while the *latest fit* looks for the last hole in which it fits. A task fits if the assigned time is later than the release time and earlier than the

deadline. This algorithm is based on the family of *fit* algorithms used for memory management in operating systems [115]. The complexity of the algorithm is $O(n)$.

Front-to-Back Strategy

1. Schedule the load task T_{1j}:
   a) Add D_i to the resource constraints of T_{1j}.
   b) Set the release time of T_{1j} to the lower bound of D_i ($r_{1j} = ub_i$).
      If r_{1j} has already a value assigned to it, because it corresponds to a job for an RSM that is currently being unloaded, then set the release time to the maximum of both values to guarantee that the job does not conflict with the active unload of the RSM.
   c) Set the deadline of T_{1j} to the latest starting time of the read task ($\tilde{d}_{1j} = \tilde{d}_{i2j} - p_{i2j}$).
   d) Find the robot R_k that can include T_{1j} earliest. For this purpose, the algorithm uses the earliest-fit strategy on each of the robots that can load drive i with the RSM and picks up the robot that provides the earliest fit. If there is an R_k that can include T_{1j}, assign this time to T_{1j} and add R_k to the set of resource constraints of T_{1j}. Otherwise, undo the assignments and stop.
2. Schedule the read task T_{2j}:
   a) Add D_i to the resource constraints of T_{2j}.
   b) Set the release time and assigned time of T_{2j} to the finishing time of the load task ($r_{2j} = a_{1j} + p_{ki1j}$ and $a_{2j} = a_{1j} + p_{ki1j}$). The read task is thus scheduled immediately after the load finishes. This starting time is valid, because the load task was scheduled with the LST of the read task as deadline.
3. Schedule the unload task T_{3j}:
   a) Add D_i to the resource constraints of T_{3j}.
   b) Set the release time of T_{3j} to the time at which the read task T_{2j} finishes ($r_{3j} = a_{2j} + p_{i2j}$).
   c) Set the deadline of T_{3j} to infinite ($\tilde{d}_{3j} = \infty$).
   d) Find the robot R_k that can include T_{3j} earliest. If there is an R_k that can include T_{3j}, assign this time to T_{3j} and add R_k to the set of resource constraints of T_{3j}. Otherwise, undo the assignments and stop.

4. Update the value of the lower bound of D_i to the time when the unload task finishes ($lb_i = r_{3j} + p_{3j}$).

Before executing step 1 the algorithm may have to schedule the unload of the drives. If the job is the first job assigned to the D_i and drive *i* is loaded, then before scheduling the load the algorithm must schedule the unload of the RSM loaded in the drive. This task already has a release time which was assigned when the read task was scheduled at the beginning of the algorithm. Once this task is scheduled, update the lower bound of D_i and proceed to step 1.

If after executing step 4 there are jobs $J_u \in \mathcal{J}$ that have unload tasks that have not been scheduled yet, then these tasks need to be scheduled. These situations appear only when some loaded drives have not been assigned to any job $J_j \in \mathcal{J}$.

The algorithm sorts these unload tasks using the release times of the tasks and schedules them in a robot schedule as early as possible. The deadline of these tasks is infinite, because this situation can only be reached when the drive is not needed for any other job than the one already active in the drive.

Back-to-Front Strategy

1. Schedule the unload task T_{3j}:

   a) Add D_i to the resource constraints of T_{3j}.

   b) Set the release time of T_{3j} to the earliest time that should provide enough time to schedule the read and load task in D_i ($r_{3j} = lb_i + p_{i2j} + min_k\{p_{ki1j}\}$). If the drive is loaded or being loaded, then add as well the time needed to unload task of the corresponding job J_u ($r_{3j} = lb_i + p_{i2j} + min_k\{p_{ki1j}\} + min_k\{p_{ki3u}\}$).

   The algorithm follows an optimistic approach and uses the minimum time needed for the load task and for the unload of the loaded RSM. If it is not possible to incorporate T_{3j} into the schedules using these values, then it will not be possible using other values either. However, succeeding to incorporate T_{3j} does not guarantee that the algorithm will be able to schedule T_{1j} using the robot that provides the minimum processing time.

   c) Set the deadline of T_{3j} to the upper bound of D_i, because the operation must also fit in the window of D_i ($\tilde{d}_{3j} = ub_i$).

   d) Find the robot R_k that can include T_{3j} latest. For this purpose, the algorithm uses the latest-fit strategy on each of the robots that can load drive *i* with the RSM and picks up the robot that provides the latest fit. If there

is an R_k that can include T_{3j}, assign this time to T_{3j} and add R_k to the set of resource constraints of T_{3j}. Otherwise, undo the assignments and stop.

2. Schedule the read task T_{2j}:

   a) Add D_i to the resource constraints of T_{2j}.

   b) Set the release time of T_{2j} to the earliest time that gives enough time to schedule the load task in D_i. $r_{2j} = lb_i + min_k\{p_{ki1j}\}$. If the drive is loaded or being loaded, then add as well the time needed to schedule the unload task of the corresponding job J_u ($r_{2j} = lb_i + min_k\{p_{ki1j}\} + min_k\{p_{ki3u}\}$).

   c) Assign T_{2j} the latest time at which it may start so that it is ready before the unload begins ($a_{2j} = \tilde{d}_{2j} - p_{i2j}$). If $a_{2j} < r_{2j}$, it is not possible to include the read task in D_i, so undo the assignments and stop.

3. Schedule the load task T_{1j}:

   a) Add D_i to the resource constraints of T_{1j}.

   b) Set the release time to lb_i ($r_{1j} = lb_i$) if the drive is not loaded or being loaded, and to $r_{1j} = lb_i + min_k\{p_{ki3u}\}$, otherwise.

   If r_{1j} already has a value assigned to it, because J_j is a job for an RSM that is currently being unloaded, then set the release time to the maximum of both values to guarantee that the job does not conflict with the active unload of the RSM.

   c) Set the deadline of T_{1j} to the latest time that will allow the read to start in time ($\tilde{d}_{1j} = a_{2j}$).

   d) Find the robot R_k that can include T_{1j} latest. If there is an R_k that can include T_{1j}, assign this time to T_{1j} and add R_k to the set of resource constraints of T_{1j}. Otherwise, undo the assignments and stop.

4. Update the value of the upper bound of D_i to the time at when the load task starts ($ub_i = a_{1j}$).

After all the jobs in $\mathcal{J}$ have been scheduled, the algorithm still has to schedule the unload tasks of the drives that are being loaded or already loaded during the computation of the schedule. There is at most one of such tasks per drive, so the algorithm simply assigns to the deadline of each task the upper bound of D_i. It then sorts the tasks by decreasing order of deadline and tries to schedule them as late as possible.

Example

Figure 6.8 shows an example of schedules constructed with the four scheduling strategies. The jukebox has two identical drives and one robot. The drives and robot are not busy at the time of building the schedule. There are four jobs to schedule, which are identified in the example by the read tasks that need scheduling. To simplify the example we assume that the load and unload tasks need constant time independently from the RSM, robot and drive involved.

In all cases it is possible to build a feasible schedule, although the schedule built in each case is different. The main differences can be seen between the schedules built Front-to-Back and Back-to-Front. The B2F schedules have large idle times when the resources are not assigned to any job. As we will show in Section 6.8, these holes can be effectively used by the dispatcher.

When using EDF the job order is J_1, J_2, J_3, and J_4. When using ESTF it is J_1, J_2, J_4, and J_3. When using LDL the job order is J_4, J_3, J_2, and J_1 and when using LSTL it is J_3, J_4, J_2, and J_1.

6.6 Medium Schedule

The medium schedule (MS) is the schedule that indicates in which order the data of an RSM must be read, once the RSM is loaded into a drive. The tasks to schedule are requests for data blocks corresponding to the request units for the RSM. If the same data block is requested more than once, then the deadline of the data block is the one of the request unit with the earliest deadline. The data blocks do not overlap, even if the request units originally were for overlapping data blocks.

The MS indicates the order in which the tasks will be executed and the latest time at which each task must start executing. The time assigned to the first task in the schedule indicates the latest time at which the RSM must be loaded in a drive and the data must begin to be staged. We call this time the latest starting time (LST) of the MS. Once the drive finishes reading the data of the first task, it continues immediately with the rest in the order given by the schedule, because it does not make sense to have the RSM loaded in a drive with pending tasks for it and not read the data. Thus, we eliminate all the idle times in the original schedule. The resulting processing time of the schedule is the sum of the transfer times for each data block plus the sum of the access times needed to go from the data of one task to the next.

As explained in the description of the formal model (see Section 5.2.4), the goal of our algorithm is to build the MS with the highest LST, in order to provide flexibility to incorporate the job corresponding to the RSM in the schedule.

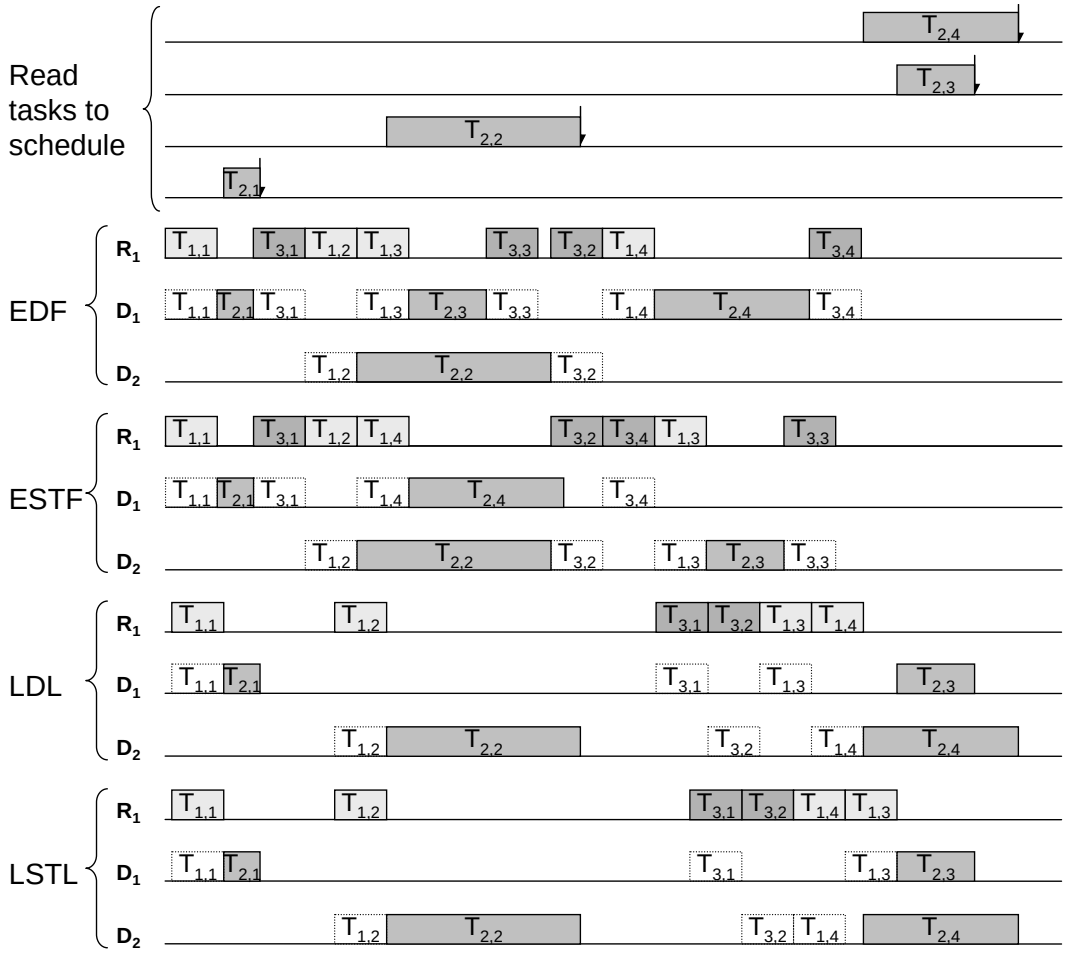


Figure 6.8: Example of schedule construction with the different strategies.

The algorithm uses the LDL strategy to build the schedule. The tasks are sorted by deadline and when the deadline of two tasks is the same, they are further sorted by offset. The algorithm incorporates the tasks to the schedule back-to-front based on deadline and when the deadline is the same, it tries to minimize the access time by using a SCAN algorithm (see Section 2.2.5). The algorithm we propose for building the MS is somewhat similar to the SCAN-EDF algorithm of Reddy et al. that we discussed in Section 2.2.5.

Figure 6.9 shows an example of the medium schedules built for an RSM using two different drives. Drive 2 is twice as fast as Drive 1. The access function is very simple and the same for both drives. The top of the graphic shows the tasks to schedule. The deadline of the tasks is indicated by a down arrow and the offset and size are given inside the task bars. Each task shows the time needed to transfer the

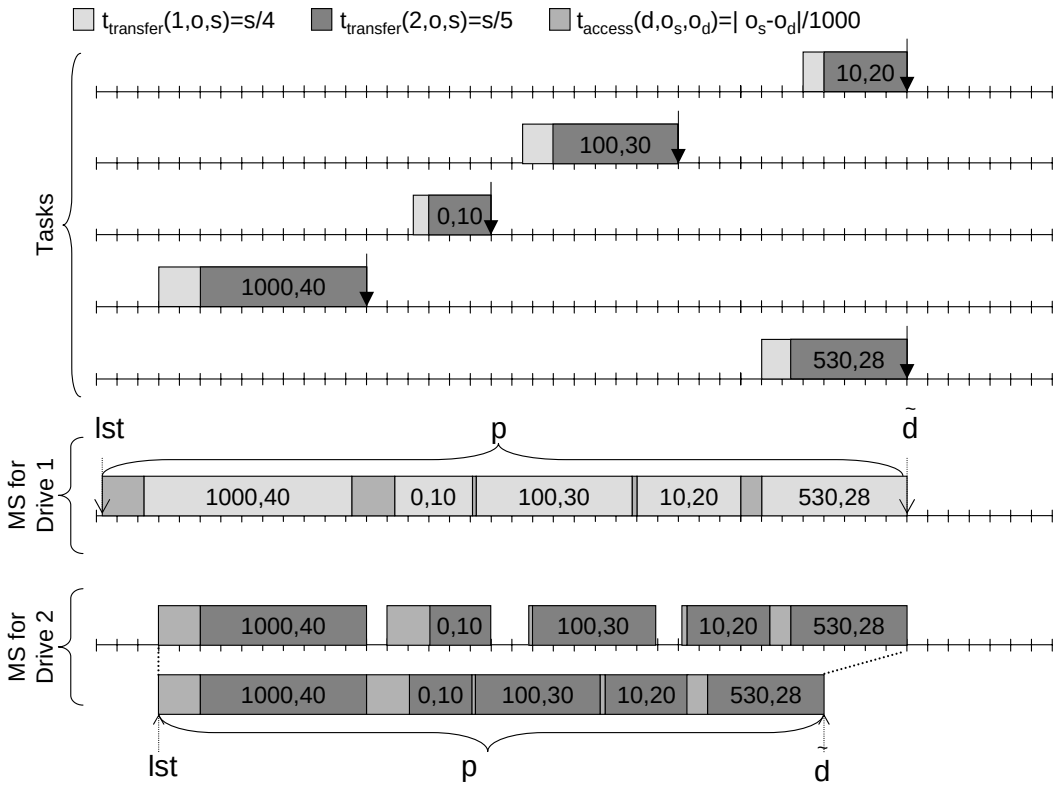


Figure 6.9: Example of the medium schedules built for two drives. Drive 2 is twice as fast as Drive 1.

data using both drives—the lighter blocks show the time needed when using D_2 and the darker blocks the time needed when using D_1.

In the MS for Drive 1 some tasks are 'pushed to the front', and the LST is earlier than it is with Drive 2. Instead in the MS of Drive 2, the LST is simply the one that should have resulted from scheduling a set only with the first task. As shown under the MS for Drive 2, the total processing time p of the MS is computed without the idle times, because once the schedule is dispatched, each task is executed as early as possible. The deadline $\tilde{d}$ of the MS is the time by which all the data of the RSM must be staged. The values of p and $\tilde{d}$ are used to determine the processing time and deadline of the read task T_{2j}.

If the access time is constant, this algorithm computes an optimal schedule. However, the access times are not constant. Therefore, the problem is $\mathcal{NP}$-hard and, so, it is not possible to compute an optimal solution in polynomial time. In this case the solutions our algorithm provides are pseudo-optimal. The difference between the

optimal solution and our solution is given by the difference in reading the tasks in the perfect order, and in the order given by the algorithm.

The perfect order is given by using a scan algorithm that reads the data as it advances in its sweep through the RSM. The perfect order has got as lower bound performing n times the lowest step possible between one data-block and the next, while the schedule produced by the algorithm may have the tasks in such a way that it performs every time the maximum possible step. The solution is, thus, not worse than $n(t_{access}^{max} - t_{access}^{min})$. This bound is much higher than the real difference, because not all access times will influence the value of the LST. The LST is influenced only in the cases in which the tasks are being pushed to the front of the schedule by tasks that have later deadlines (i.e., when $min(lst, \tilde{d}_i) = lst$). This is mainly the case when several tasks have the same deadline, but the tasks with the same deadline are incorporated using the scan strategy that in most cases minimizes the access time. Furthermore, when using optical disks, as is our main study case, the access time does not vary much, causing only a small difference from the optimum.

A further optimization to our algorithm is to run it for a second time with re-defined values of the deadline of some tasks. We assign a new deadline $\tilde{d}$ to the tasks whose deadline is later than the deadline of the MS. This new deadline does not violate the restrictions of those tasks, because their deadlines are after $\tilde{d}$. In this way there are more tasks with the same deadline at the first step of the computation. Thus, there are more possibilities to minimize the time spent in jumping between tasks, because the set of tasks on which the SCAN algorithm is performed is bigger.

6.7 Complexity Analysis

We analyse here the total complexity of the scheduling algorithm. We present a complexity analysis of each step and show finally that the total complexity is low.

We will use the following notation:

$n = \|\mathcal{U}'\|$	Number of request units to schedule
m	Number of drives in the jukebox
s	Number of RSM in the jukebox, which provides an upper bound to $\|\mathcal{J}\|$, the number of jobs to schedule
u	Number of tasks in a medium schedule

The complexity of computing the job set is $O(n + m)$, because each request unit in $\mathcal{U}$ must be included in the appropriate job. Additionally, there is the possibility

of having an extra job per busy drive, but given that m is a constant, the resulting complexity of this step is $O(n)$.

The computation of a medium schedule has complexity $O(u \cdot \log u)$. This complexity comes from sorting the tasks to incorporate into the MS. An upper-bound for u is the number of request units in $\mathcal{U}'$, so the complexity of this step is $O(n \cdot \log n)$.

The scheduler must compute $m \cdot n$ medium schedules. One MS for each job in $\mathcal{J}$ using each drive. The upper bound of $|\mathcal{J}|$ is s. So, the complexity of computing all the medium schedules is $O(m \cdot s \cdot n \cdot \log n)$. The values of m and s are constant, but we will not ignore them, because they reflect the influence of the size of the jukebox in the performance. In the implementation of the algorithm, we do not compute an MS for each drive in the jukebox, but only for each different drive model, so m can be replaced with $m' \leq m$.

Incorporating one job to the schedule once the drive has been determined has complexity $O(2 \cdot 2 \cdot s)$, because the fit algorithm on the robot schedule is executed twice, once for the load task and once for the unload task. The maximum number of tasks is the robot schedule is $2 \cdot s$.

The maximum number of nodes visited when traversing the tree is $s \cdot m! \cdot m$. So the complexity of building the schedule is $O(s \cdot m! \cdot m \cdot 2 \cdot 2 \cdot s)$, which can be simplified to $O(s^2 \cdot m! \cdot m)$. It is important to note that for a given system, all the elements in this formula are constants.

Finally, we have shown in Section 6.1 that the heuristic to determine the candidate time will execute at most $\log_2(\tilde{d}_i - t_0)/\varepsilon$ steps.

The complexity of the algorithm is:

$$O(\log(\frac{\tilde{d}_i - t_0}{\varepsilon})\,(n + n \cdot \log n + s^2 \cdot m! \cdot m)) \tag{6.5}$$

If we remove the constants, the resulting complexity is

$$O(\log(\frac{\tilde{d}_i - t_0}{\varepsilon})\,(n \cdot \log n)) \tag{6.6}$$

6.8 Dispatcher

The dispatcher guarantees that the tasks are sent to the jukebox controller in time. Furthermore, it also tries to dispatch each task as early as possible (ASAP), so that the resources are not idle. The dispatcher can modify the active schedule as long as no task in the schedule is delayed and the sequence dependencies are respected.

The dispatcher uses the 'holes' in the schedules to dispatch early tasks. The bigger the holes, the more chances it has to dispatch tasks early. Therefore, the dispatcher works better in combination with the Back-to-Front strategies, because they

create big holes in the schedules. The dispatcher also profits from the fact that the schedules are built using the worst-case times for the different tasks and, thus, some resources become idle earlier than planned and can start processing the tasks scheduled for later moments.

If the scheduler is idle and there are idle resources, the dispatcher tries to dispatch the next task scheduled for the idle resources as soon as possible. It can only dispatch an early task to a resource if the task can be executed immediately.

There are two conditions that must hold in the dispatched schedule. The first condition is that no task must be dispatched at a later time than its assigned time. The dispatcher does not compute the schedule again, and does not know about the deadlines of the tasks. It simply knows the times that the tasks were assigned by the scheduler and uses those assigned times as deadlines for dispatching. Thus, if no task is dispatched later that its assigned time, no task misses its deadline. The second condition is that the tasks of different jobs must not interleave in the use of the drives.

Every time a task is dispatched to the jukebox, the dispatcher eliminates it from the schedule. The rest of the tasks are not modified. So, at each moment, the tasks in the schedule represent only the tasks that still need to be dispatched. If t_{now} is the present time, the assigned time of the first task in each of the resources schedules must be later than t_{now}.

The dispatcher decides which task to dispatch next by analyzing the drive schedules. The dispatcher is always able to dispatch the first task of a drive early when this task corresponds to a read, because the RSM needed is already loaded and the only resource needed is the drive. If the first task of a drive is not a read, it can only dispatch the task if the robot needed for the task R_k is also idle and either the corresponding robot task is also the first task in R_k, or there is enough slack to execute the task before the assigned time of the first task of R_k.

In order to be fair with the early dispatching of tasks scheduled for different drives, the dispatcher sorts the drive schedules corresponding to idle drives by increasing time assigned to the first task of each schedule.

The sequence of the drive schedules does not change after dispatching, but the sequence of the robot schedules may be different than the sequence defined by the scheduler.

Figure 6.10 shows the effect of the dispatcher. At the top we show the schedule built using LDL for the example in Figure 6.8. No requests arrive during the execution of the schedule, so the active schedule does not change. The dispatcher starts by dispatching $T_{1,1}$ early, because both R_1 and D_1 are idle and the assigned time of $T_{1,1}$ is earlier than the assigned time of $T_{1,2}$. The dispatcher must now wait until the robot becomes idle again. At that moment it dispatches the read task $T_{2,1}$ to D_1 and

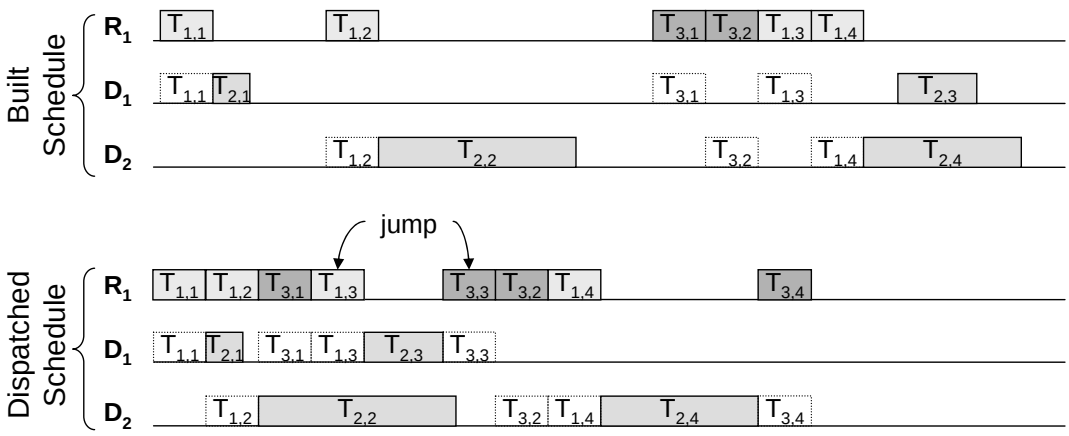


Figure 6.10: Example of early dispatching.

the load task $T_{1,2}$ corresponding to the robot and D_2. When the execution of $T_{1,2}$ finishes, it dispatches the read task $T_{2,2}$ to D_2 and the next task corresponding to D_1, $T_{3,1}$, because D_1 is again idle. When the execution of $T_{3,1}$ finishes, only D_1 is idle, so the dispatcher dispatches the next task for D_1, $T_{1,3}$. When $T_{1,3}$ finishes, D_2 is still busy, so the next task dispatched, $T_{3,3}$[2], again corresponds to D_1. When the robot is idle again after executing $T_{3,3}$, D_2 is idle as well, so the dispatcher can dispatch the next task for D_2, $T_{3,2}$. Finally it dispatches one by one the tasks corresponding to J_4 scheduled on D_2.

There are two types of events that trigger the dispatcher to dispatch tasks to the jukebox controller. The first type of event indicates that the deadline to dispatch a task in the schedule has been reached. The second type of event indicates that a resource is idle and, thus, may execute a task immediately. The dispatcher interrupts the computation of a new schedule when the first type of events occurs.

6.9 Implementation Notes

For efficiency reasons the implementation of Promote-IT includes the following features that are not part of the algorithm described in this chapter:

- The scheduler computes a medium schedule per drive model in the jukebox instead of per drive. In this way the number of MS computed for each job is smaller.

2 This task is not shown in the original schedule, because it is assigned a time near infinite.

- To simplify the implementation of the scheduling algorithm we have removed the possibility of having two different jobs for the same RSM. Thus, when a request arrives involving an RSM that is being unloaded from a drive, the scheduler puts the request in the queue of unscheduled requests until the unload of the RSM finishes. To provide first-come-first-serve fairness to the request that is waiting for the unload to finish, the scheduler puts also all the other incoming requests on-hold during the duration of the unload.

- The dispatcher dispatches all the commands to read the data from an RSM at once instead of one by one as described in Section 6.8. The drive controllers have a queue where they store the commands that need to be executed. The scheduler considers the time by which the drive will be available to be the time by which all the commands in the queue of the drive will have been executed.

  An advantage of dispatching all the read commands at once is that the dispatcher does not need to be informed when the reading of each request unit finishes. When multiple small files are read, this optimization makes a big difference in the number of events generated in the system.

  A disadvantage of dispatching all the commands at once is that when a new request comes for an RSM loaded in a drive, the scheduler cannot reschedule the other tasks for the RSM that have not yet been executed. Therefore, the only request units in $\mathcal{U}'$ for the RSM will be those of the new request.

6.10 Comparison of the Strategies

In Section 6.2 we showed that no strategy is absolutely superior to the others. In this section we show that in general ESTF is the best strategy. However, when the system load is high and the robot is a strong bottleneck, LDL seems to perform better. When the system load is low, all strategies show a similar performance. The performance of ESTF and LDL is more stable and predictable than that of EDF and LSTL. We believe that the reason could be that both in ESTF and LDL we are using the more natural parameter to sort the tasks: the starting time when incorporating Front-to-Back and the deadline when incorporating Back-to-Front.

In Chapter 9 we show that the performance of the two best strategies of Promote-IT, ESTF and LDL, is better than the performance of the other heuristic schedulers and near the performance of the optimal scheduler.

We will analyze two examples that vary in the degree in which the robot is the bottleneck of the system. For each case we will show first the performance of the strategies under a low and medium load. We will then show the performance under

high load. In these first two cases the system does not reject requests. Finally we will show the performance of the system when requests can be rejected during overload situations. When rejecting requests the performance of all the strategies is similar.

Regarding the main parameter for evaluating the strategies, the response time, the graphics show the mean response time and the maximum response time for 90% of the requests. The standard deviation is big, therefore, the mean response time cannot be used as only parameter to determine the quality of the strategies. The maximum response time for 90% of the requests indicates the maximum time that the users have to wait in 90% of the cases. Thus, it provides an idea of worst-case behaviour pruning the exceptionally bad cases. The really bad cases are pruned when the system can reject requests. In general, the curve of the mean response time and the maximum for 90% of the requests are very similar for low and medium loads, but when under high loads the maximum for 90% of the requests indicates the performance of the schedulers better, because it is more resilient to a few bad cases.

When rejecting requests we show the rejection ratio, which is measured as the percentage of rejected requests over the total number of requests. In order to be able to reject requests, the requests also specify a deadline and maximum confirmation time. In the two cases shown in this section the deadline is 5 minutes and the maximum confirmation time is 30 seconds.

The graphics for the mean robot and drive utilization show the percentage of time that the robot and drives are used in a productive way during the run. For the robot, it indicates the percentage of time it is loading or unloading disks, while for the drives it means the time spent on reading data.

Independently of the jukebox architecture and test set, the mean computing time of the F2B strategies is shorter than that of the B2F strategies. B2F requires more computation than F2B because it needs to backtrack more often and is deeper into building the schedule when it discovers resource conflicts. First, the number of possible schedules is greater for B2F because it always starts with m equally eligible drives for the first job. F2B only needs to decide between the drives that are actually idle at the moment of computing the schedule. Second, B2F generally discovers resource conflicts later when it attempts to incorporate the most restrictive jobs. F2B starts with these most restrictive jobs and discovers the conflicts earlier. However, sometimes when the system load is high and the system cannot reject requests, the B2F strategies can find schedules than the F2B strategies cannot and the mean confirmation time of B2F is shorter than that of the F2B strategies.

In general, the resource utilization of the strategies is nearly the same (in many plots it even appears completely overlapping). Furthermore, the resource utilization grows linearly with the system load until the system approaches an overload situation and the resource utilization curves flatten a little. When the scheduler is able

to reject requests, the resource utilization of the strategies differs more. In this case, however, the utilization is influenced by the particular requests rejected.

Case 1: Shared Robot is a Strong Bottleneck

We first show a case where the robot is a strong bottleneck is the system. While the robot utilization reaches 95%, the drive utilization is less than 10%.

The request set consists of 1000 ASAP requests that follow a Zipf distribution. The requests are generated following the proportions: 10% long videos, 40% short videos, 30% music, and 20% discrete data. This is the same proportion of data that is stored in the jukebox. Section 8.6 explains how the jukebox contents and requests are generated. The data is stored on single-layered DVD-ROM. The jukebox has four identical drives with transfer speed in the range [7.9, 20.5] MBps. The load time is in the range [21.8, 24.9] seconds and the unload time is in the range [15.4, 18.5] seconds.

The size of the cache is 10% of the jukebox capacity. The average cache-hit rate is 62%. The cache-hit rate is nearly constant, independently of the scheduler used or the system load.

Figure 6.11 shows that the performance of the two Front-to-Back strategies is better than that of the two Back-to-Front strategies when the load is medium or low. Under these load circumstances, ESTF performs better than EDF.

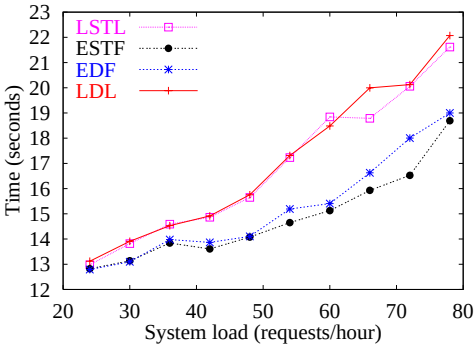
Figure 6.12 shows that when the system load passes a certain threshold, the two Back-to-Front strategies perform better. The robot utilization degrades steeply after that threshold, because the system is in an overload situation.

Finally, Figure 6.13 shows that, when rejecting requests during overload situations, all the strategies perform much better and there is a small difference in favour of the F2B strategies. The rejection ratio that allows this improvement is very small: less than 3% (shown in Figure 6.13(a)).

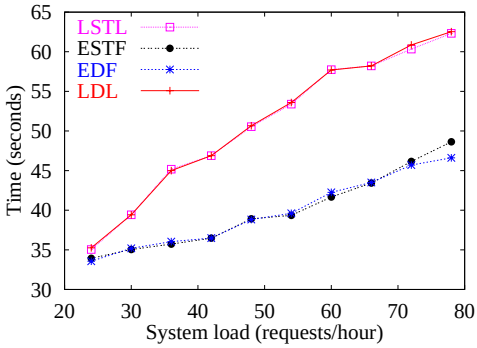
Case 2: Shared Robot is not a Bottleneck

In this case there is not such an imbalance between the robot utilization and the drive utilization, as in the previous example. The robot utilization reaches 75%, while the drive utilization reaches 50%.

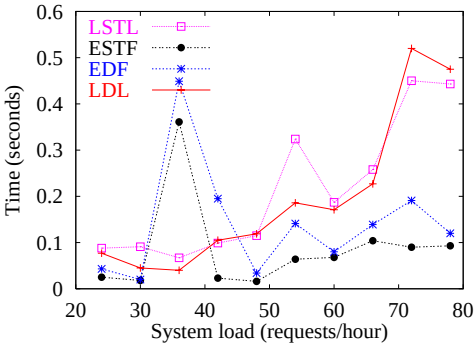
The request set consists of 1000 ASAP requests that follow a Zipf distribution. The requests were generated following the proportions: 30% long videos, 30% short videos, 30% music, and 10% discrete data. This is the same proportion of data that is stored in the jukebox. The data is stored in double-layered DVD-ROM. The jukebox has four identical drives with transfer speed of 5.11 MBps. The load time is in the range [21.8, 24.9] seconds and the unload time is in the range [14.3, 17.4] seconds.



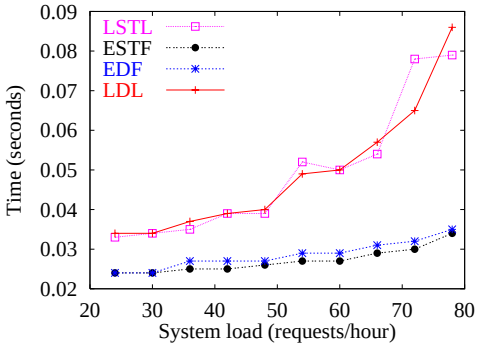
(a) Mean response time.



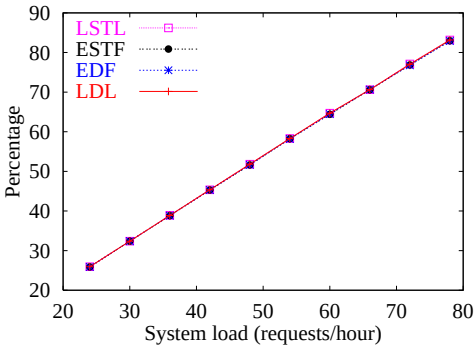
(b) Maximum response time for 90% of the requests.



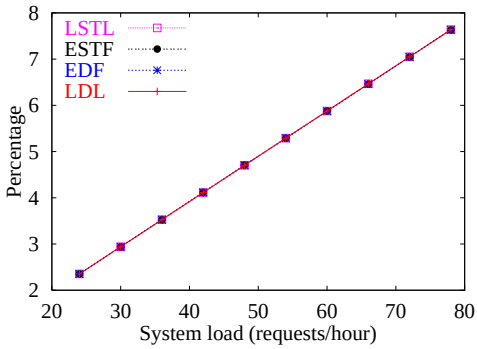
(c) Mean confirmation time.



(d) Mean computing time.

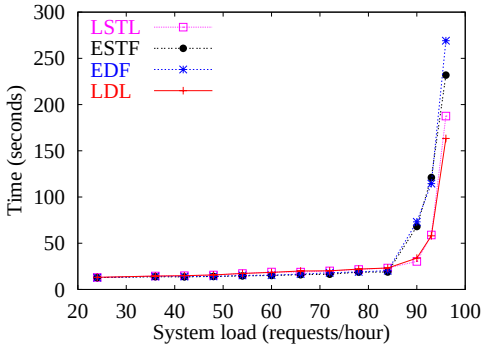


(e) Mean robot utilization.

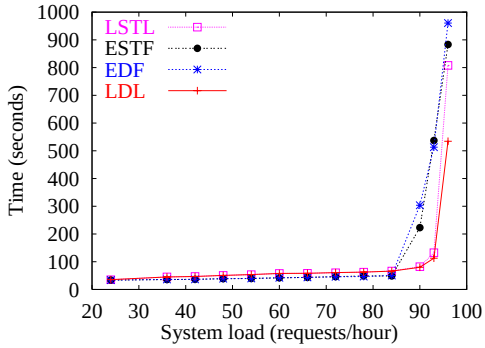


(f) Mean drive utilization.

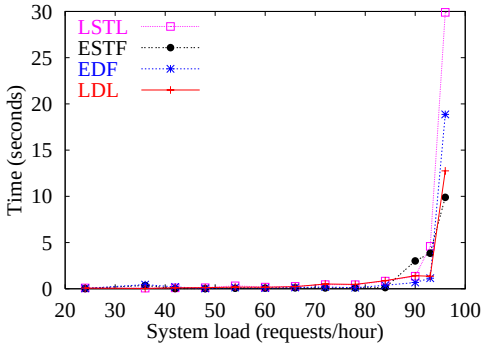
Figure 6.11: Performance of the strategies under low and medium load when the shared robot is a strong bottleneck (case 1).



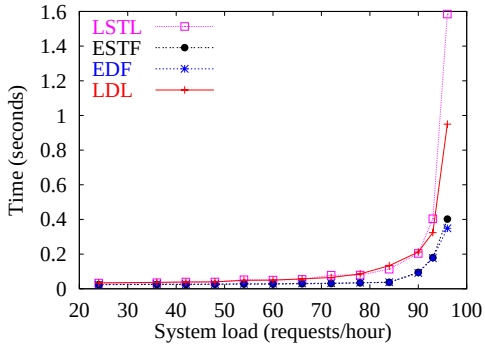
(a) Mean response time.



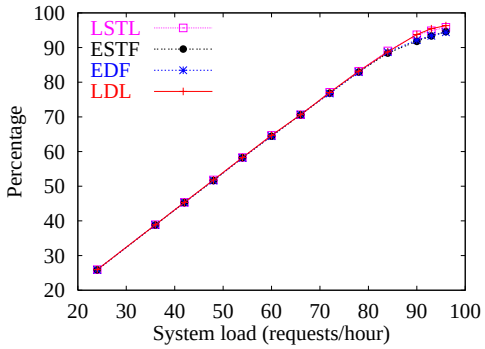
(b) Maximum response time for 90% of the requests.



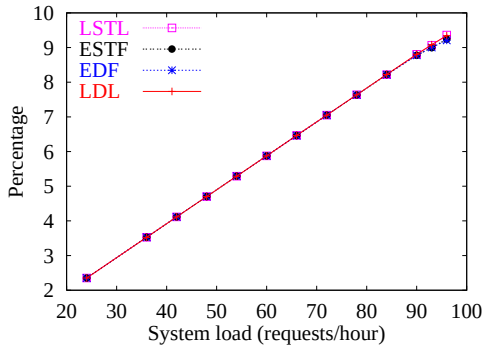
(c) Mean confirmation time.



(d) Mean computing time.

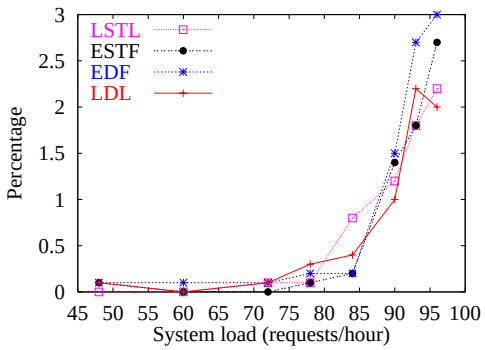


(e) Mean robot utilization.

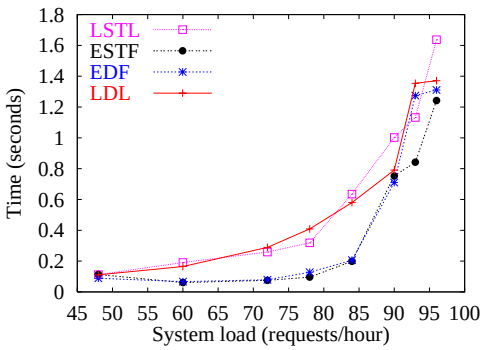


(f) Mean drive utilization.

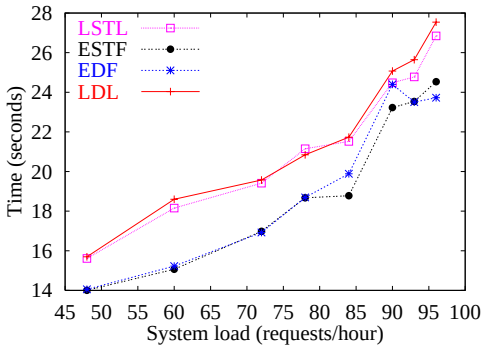
Figure 6.12: Performance of the strategies as the load increases when the shared robot is a strong bottleneck (case 1).



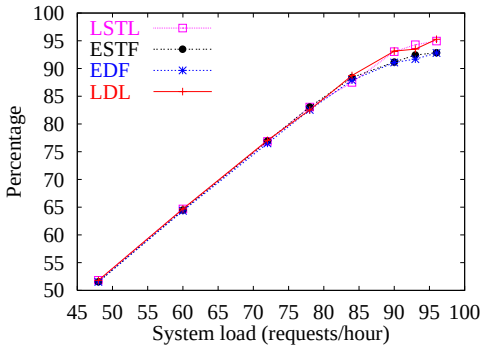
(a) Rejection ratio.



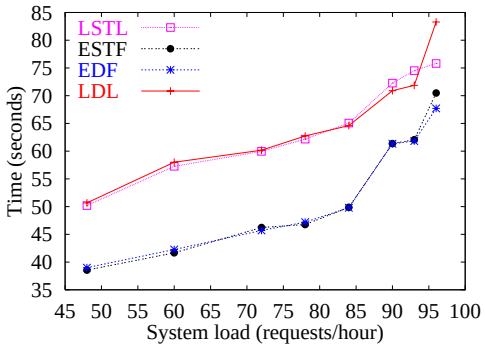
(b) Mean confirmation time.



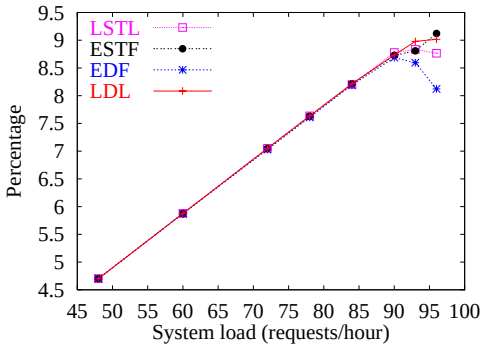
(c) Mean response time.



(d) Mean robot utilization.

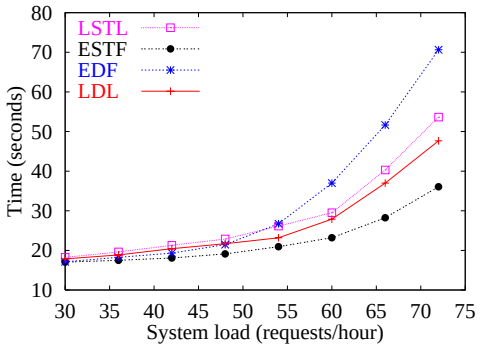


(e) Maximum response time for 90% of the requests.

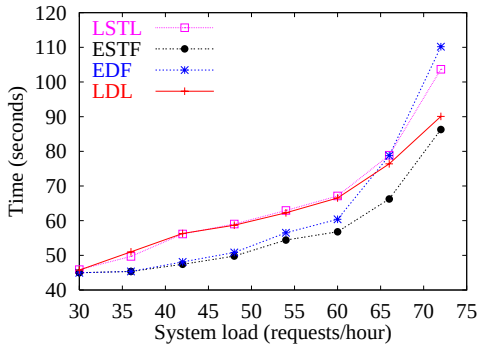


(f) Mean drive utilization.

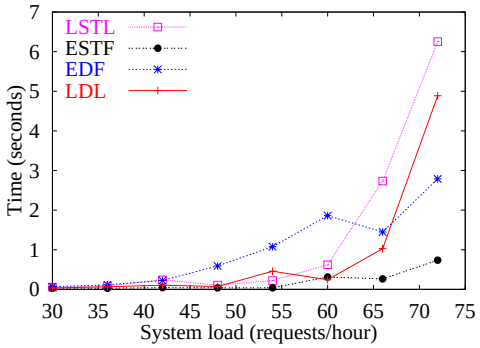
Figure 6.13: Performance of the strategies when rejecting requests when the shared robot is a strong bottleneck (case 1).



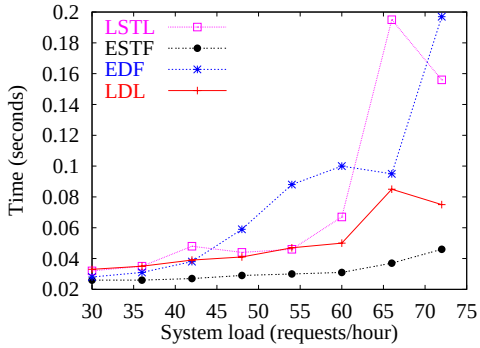
(a) Mean response time.



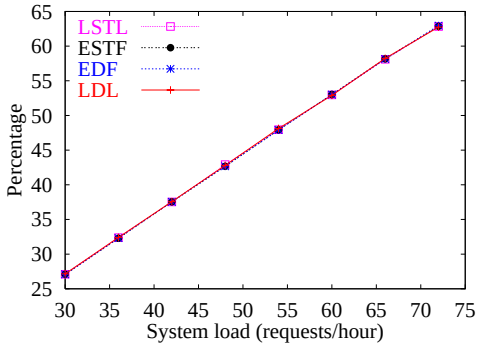
(b) Maximum response time for 90% of the requests.



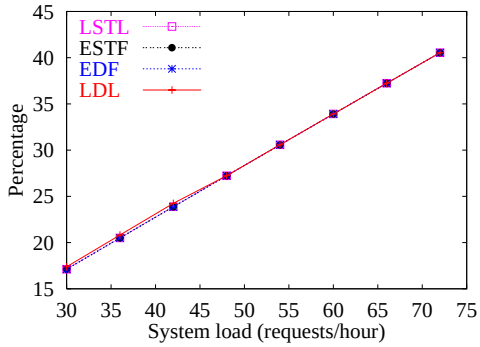
(c) Mean confirmation time.



(d) Mean computing time.

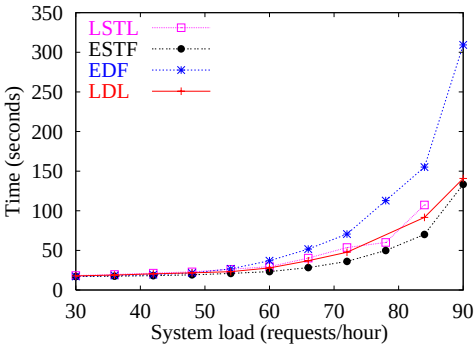


(e) Mean robot utilization.

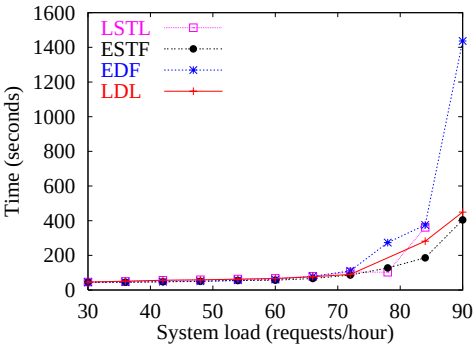


(f) Mean drive utilization.

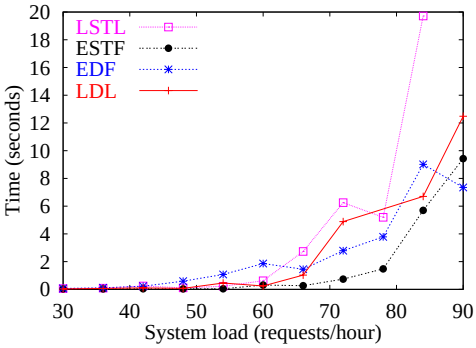
Figure 6.14: Performance of the strategies under low and medium load when the shared robot is not a bottleneck (case 2).



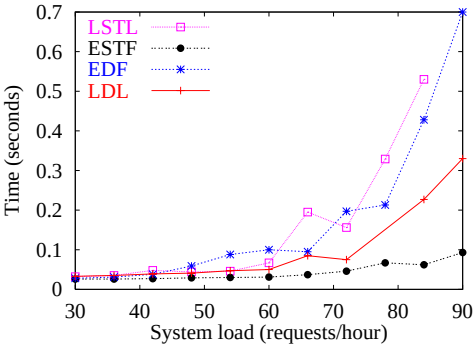
(a) Mean response time.



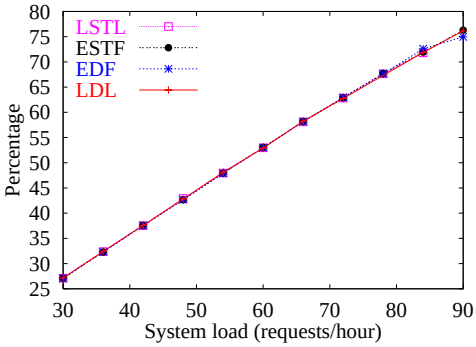
(b) Maximum response time for 90% of the requests.



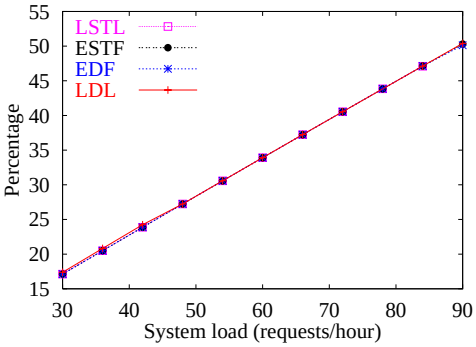
(c) Mean confirmation time.



(d) Mean computing time.

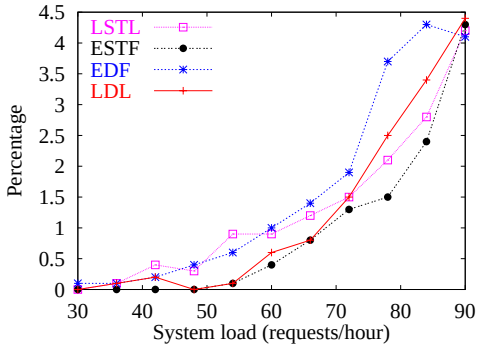


(e) Mean robot utilization.

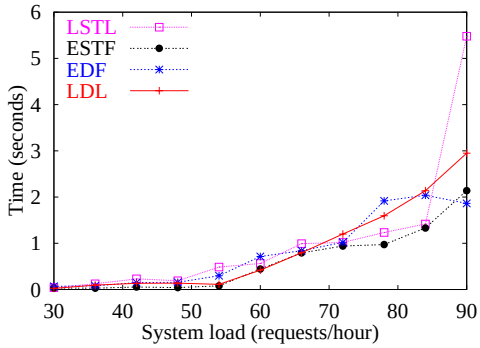


(f) Mean drive utilization.

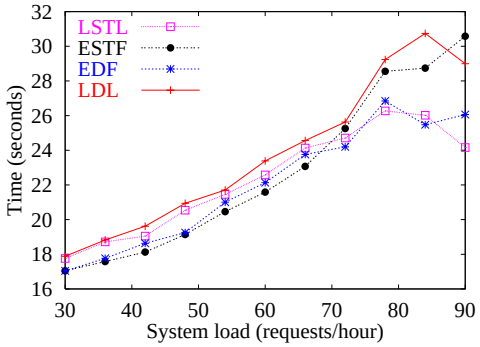
Figure 6.15: Performance of the strategies as the load increases when he shared robot is a not a bottleneck (case 2).



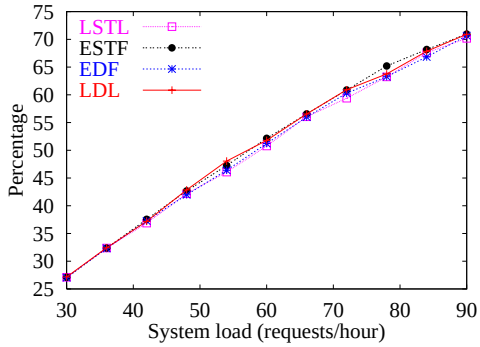
(a) Rejection ratio.



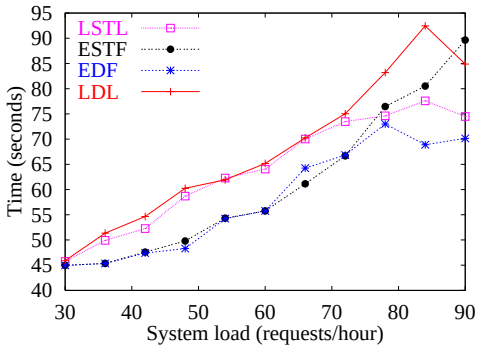
(b) Mean confirmation time.



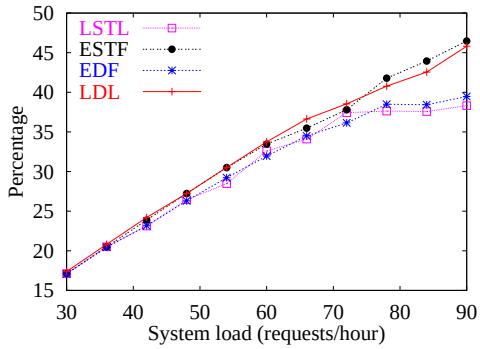
(c) Mean response time.



(d) Mean robot utilization.



(e) Maximum response time for 90% of the requests.



(f) Mean drive utilization.

Figure 6.16: Performance of the strategies when rejecting requests when the shared robot is not a bottleneck (case 2).

The size of the cache is 10% of the jukebox capacity. The average cache-hit rate is 60%. The cache-hit rate is nearly constant, independently of the scheduler used or the system load.

Figures 6.14 and 6.15 show that ESTF always performs best, followed by LDL. Figure 6.16 shows that the performance of all strategies is similar when rejecting some requests. The rejection ratio is less than 4.5% as shown in Figure 6.16(a), however, not all the strategies have the same rejection ratio. EDF and LSTL have a higher rejection ratio than ESTF and LDL, which explains the fact that the response time of the two former strategies is slightly lower.

6.11 Summary

This chapter explained in detail how Promote-IT builds the schedules. It discussed the four strategies that Promote-IT can use to incorporate the tasks to the schedule: earliest starting time first (ESTF), earliest deadline first (EDF), latest deadline last (LDL), and latest starting time last (LSTL). The former two strategies incorporate the tasks to the schedule Front-to-Back, scheduling each task as early as possible. The latter two incorporate the tasks Back-to-Front, scheduling each task as late as possible. When incorporating the tasks Back-to-Front, the resulting schedule has idle times that can be effectively used by the early dispatcher. The use of the early dispatcher makes Back-to-Front strategies competitive, in opposition to the results showed by Lau et al. for the conservative strategy (see Section 2.2.1) where no early dispatcher was used.

The chapter shows that no strategy is absolutely better than the others. However, ESTF performs best in most cases, and when the system load is very high it is convenient to use LDL. The difference in performance between the different strategies is small when compared with other schedulers. In Chapter 9 we show the superiority of Promote-IT over the other heuristic schedulers.

Chapter 7

Alternative Schedulers

This chapter presents two jukebox schedulers that we developed to evaluate the performance of Promote-IT: the *jukebox early quantum scheduler* and the *optimal scheduler*. The significance of these two schedulers is that they are based on different paradigms than Promote-IT.

The *jukebox early quantum scheduler (JEQS)* is a periodic scheduler that uses the scheduling theory about *early quantum tasks* presented in [51]. To our best knowledge, JEQS is the only correct periodic jukebox scheduler. As we discussed in Chapter 2, the other periodic jukebox schedulers presented in the literature do not deal correctly with the resource-contention problem. Thus, they cannot guarantee that the deadlines are always met.

Although JEQS performs worse than Promote-IT, it has the advantage that it uses a very simple scheduling algorithm. In Section 9.1 we show that the comparatively bad performance of JEQS is intrinsic to any periodic jukebox scheduler.

The *optimal scheduler* is a scheduler that computes a schedule with the minimum response time for each request. This scheduler solves the *switch/no-switch conflict* presented in Chapter 5 and always finds an optimal schedule. The problem with this scheduler is that it may need several days to compute a schedule. Therefore, this scheduler is only useful for assessing the quality of the heuristic schedulers presented in this dissertation.

The optimal scheduler was developed mainly by Sandro Etalle. Ferdy Hanssen implemented a first version of the jukebox early quantum scheduler that assumed that secondary storage was used only as a buffer for the jukebox and not as cache. His work on administering these buffers is reported in [51].

This chapter also presents the extensions to some existing jukebox schedulers: the *extended aggressive strategy*, the *extended conservative strategy* and *Fully-Staged-Before-Starting*. These three jukebox schedulers are aperiodic schedulers that are used for comparison in Chapter 9 to highlight the advantages of different design elements of Promote-IT. Compared to Promote-IT, the extended aggressive strategy couples the unload and load operations in a single switch operation, the extended

conservative strategy does not use an early dispatcher, and Fully-Staged-Before-Starting does not use pipelining.

7.1 Jukebox Early Quantum Scheduler

The *jukebox early quantum scheduler (JEQS)* is a periodic scheduler based on the periodic quantum model presented in Section 5.6. In this model, the robot and drives are used in a cyclic way and the time is divided in constant units called *quanta*. 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 ($\frac{m-1}{m}Q$). During this time the drive can read data from the RSM.

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.

JEQS uses the scheduling theory on *early quantum tasks (EQT)* presented in [51]. 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 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.

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 and increase 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.

7.1.1 Scheduler

The scheduler builds a separate schedule for each drive. When a new request r_k arrives the scheduler tries to include it in the schedule of one of the drives. The scheduler first tries to include the corresponding task τ_k 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 τ_k 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 using double-sided disks, all the tasks for the same disk are assigned the same drive. When a task is scheduled for the first time it is assigned a drive and all the instances of the task will be executed on that drive. The scheduler distinguishes a new task from the others, because in the former ρ_k will not indicate the drive to which the task is assigned.

As discussed in Section 5.6, using quantum tasks has the advantage that, although the tasks are non-preemptable, they can be treated as preemptable during the feasibility analysis. Therefore, we can use a simple variation of EDF [68], called *quantum earliest deadline first (QEDF)*, to determine if the task set is schedulable. EDF 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 benefit 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.

The only condition to treat the tasks as preemptable 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.

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 simplification has the advantage that it is extremely simple to determine if a request is schedulable. 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. In many cases, this last restriction prevents JEQS from providing an immediate confirmation to the user about the schedulability of the request.

The scheduler will try at most $2\,m$ possible starting times to schedule an incoming request. In the case of an ASAP 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:

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

When transforming the request units 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 us to represent that the first instance of the task has an 'almost immediate' deadline and the next instances behave normally.

When scheduling a non-ASAP request, the scheduler tries to schedule the request on the drive that will permit to meet the deadline of the request at the latest time. If possible, the scheduler will not introduce the request as an EQT, thus, increasing the chances of accepting the next ASAP request as EQT.

Let us illustrate with an example how an incoming request r_k is transformed into the periodic task τ_k. The request r_k represents a two hour long video with a bandwidth of $6Mbps$.

Table 7.1 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 7.1 shows the candidate starting times for r_k. The candidate starting times, shown as $st_k^1, \ldots, st_k^8$, are tried in that order. Figure 7.2 shows the task τ_k when using

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

Table 7.1: Jukebox specification and problem parameters of the example for JEQS.

st_k^1 and st_k^5. In both cases the drive to use is D_1. τ_k^1 is an early quantum task and as such it requires that the drive executes $\tau_{k,1}$ 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 the task set of the drive corresponding to st_k^j. If the feasibility analysis succeeds, then 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 [68]. The task set Γ_i is schedulable if:

$$U = \sum_{j=0}^{n} \frac{C_j}{T_j} \leq 1 \tag{7.1}$$

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 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 τ_k.

EQT Admission Test 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

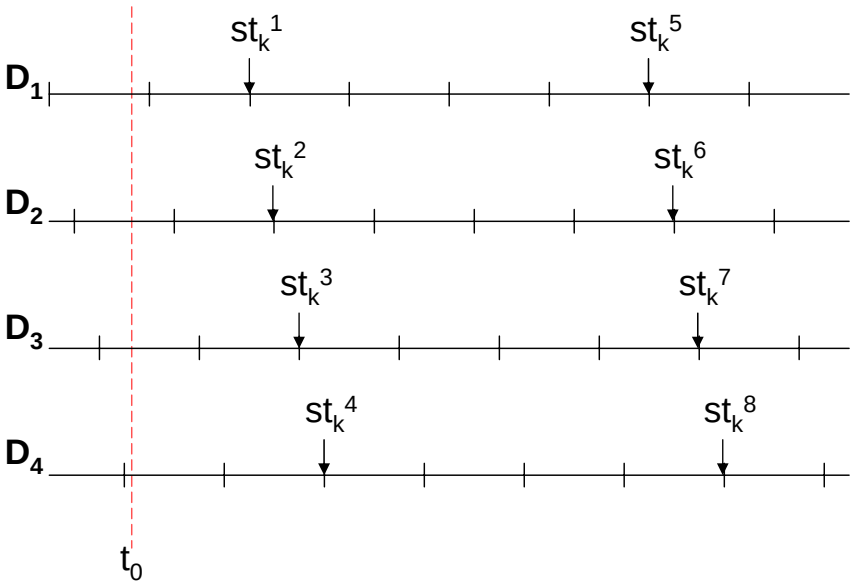


Figure 7.1: Candidate starting times for the Request in the example for JEQS.

incorporated in Γ as an early quantum task, τ_k may be incorporated in Γ as an early quantum task if:

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

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. For a complete proof see [51].

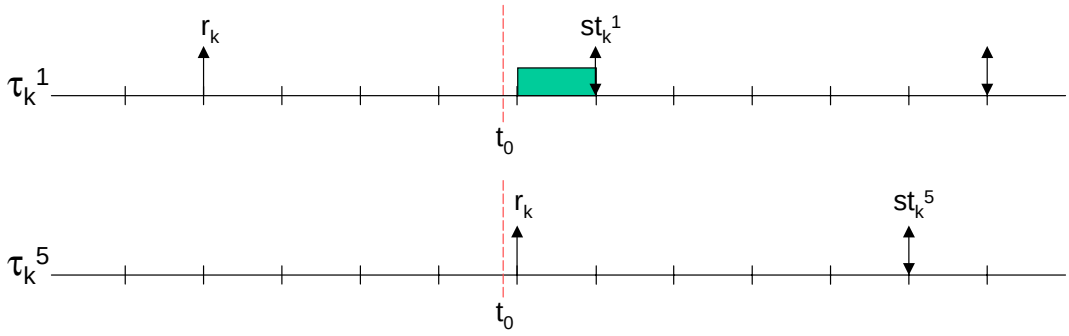


Figure 7.2: Details of the resulting tasks in the example for JEQS when using the first and fifth candidate starting time. Top: Resulting task τ_k^1 when using st_k^1. Bottom: Resulting task τ_k^5 when using st_k^5.

Variations of the Scheduling Algorithm

There is a trade-off between optimizing the response time and the confirmation time. The scheduler may accept a request as a 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 EQT. 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 EQT.

A task τ_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 ASAP requests as EQTs. If the request cannot be incorporated as an EQT, 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 1 on average 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 Chapter 9 we compare the performance of both approaches and show the point at which using normal quantum tasks is more beneficial than using only EQTs.

7.1.2 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 buffered already. 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 the RSM, it loads the RSM of another task. Only if no task has 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 in the drive constantly until all the data has been read and only then is 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 with idle slots it can advance in the execution of a task. Once the deadline of the last instance of the task is reached, the task is removed from the schedule.

7.1.3 Example

We illustrate with an example how the scheduler works. We use the jukebox specification shown in Table 7.1. Table 7.2 shows the requests that arrived at the system since time 0. All the requests are ASAP and the data of each request is stored on different RSM ($\forall\, i, j \mid m_i \neq m_j$). The table shows the period of the corresponding task. We assume that before these requests arrived, the system was idle.

Table 7.3 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 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 $d_i^{last} = (RI_i - 1)\, T_i\, Q + st_i$. The table provides an upper bound for both values,

Arrival (seconds)	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 7.2: Requests that arrived at the system in the example for JEQS.

because the dispatcher does not know in advance the exact number of instances that need to be used for each task.

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 EQTs. 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 7.3 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 EQT begins. The blocks without tags represent work that has been dispatched early.

7.2 Optimal Scheduler

The objective of the optimal scheduler is to serve as a quality reference for Promote-IT. Given that Promote-IT uses a heuristic scheduler we want to determine the average difference between the response time achieved with Promote-IT and the optimal response time.

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 7.3: State of the scheduler at the arrival times and resulting starting time and response time of the example for JEQS. 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.

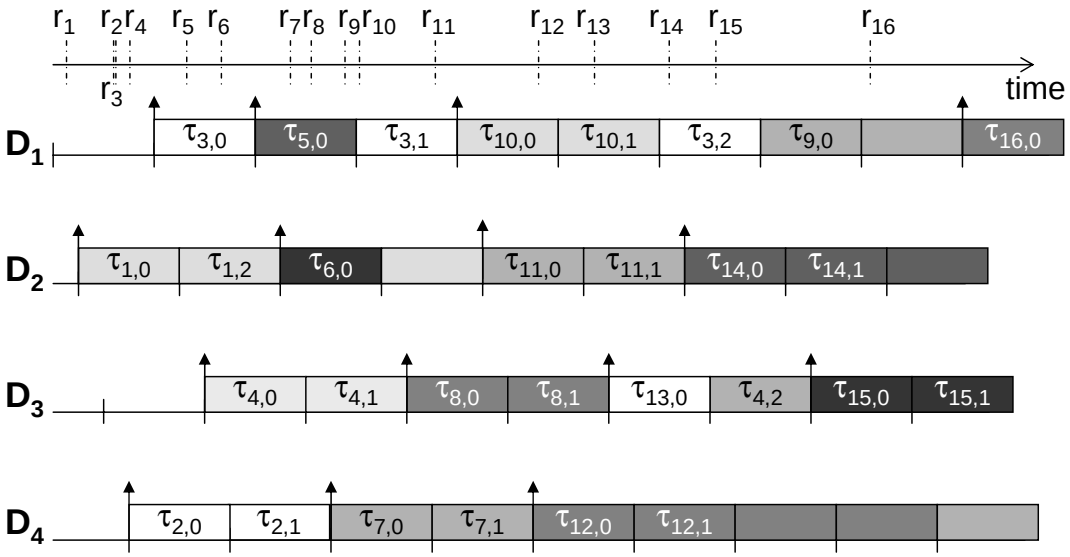


Figure 7.3: Resulting schedule of the example for JEQS. The blocks without text are idle slots used for filling the buffers early. The top line indicates the arrival times of the requests. The up arrows indicate when an EQT begins.

The optimality criterion for the scheduler is to find the minimum response time for each incoming request. The requests arrive at the scheduler on-line and the scheduler must provide a confirmation to each request. When a request arrives the scheduler builds an optimal schedule for the set $\mathcal{U}'$ of request units that need to be scheduled. The schedules are only locally optimal, because the scheduler is *non-clairvoyant* (i.e., it does not know what requests will arrive in the future).

We have made some simplifications to the original scheduling problem to reduce the already extremely high complexity. On the one hand, we assume that all requests are ASAP requests. Therefore, we can use the response time as a minimization parameter to prune the tree of possible schedules. On the other hand, we assume that the jukebox has one shared robot and m identical drives, and that the access time is constant. By using constant access times, computing the medium schedule becomes trivial, because we do not need to optimize the order in which the data must be read from an RSM. Using constant access time also allows us to compute in advance the time needed to read the data of a job, without taking into account the previous job that was incorporated to the schedule.

Because the scheduler is optimal it always finds a feasible schedule for an incoming request. However, it may take arbitrary long (weeks) to find a feasible schedule. Therefore, the scheduler does not take into account the passing of time while building a schedule. It behaves as if the requests should be scheduled instantly. This is needed, because if the passing of time should be taken into account the computa-

tions should be invalid by the time they are finished. Thus, the only goal of this scheduler is to build optimal schedules. It is not concerned about minimizing the computing time, and the confirmation time is always 0, because time does not pass.

The scheduler is based on the *optimal model* presented in Section 5.8. For efficiency reasons the optimal scheduler does not determine the set of jobs to schedule in advance, but prunes the possible set of jobs as the computation advances.

There is an important difference between the method to schedule a request used by the optimal scheduler and the method presented in Section 3.5, which is used by the non-optimal schedulers. In the latter, we first guess a candidate starting time for a request and then try to build a schedule that includes the request using that time. In the optimal scheduler, determining the candidate starting time and building the schedule are coupled. The optimal scheduler uses the starting time as the branch-and-bound variable to find an optimal schedule.

The optimality criterion for the optimal scheduler is to find the minimum st_k that permits to build a feasible schedule for the set of jobs $\mathcal{J}$ representing the request units in $\mathcal{U}'$. Contrary to heuristic schedulers, the optimal scheduler always succeeds in finding a feasible schedule for all the candidate starting times that are greater or equal to the optimal starting time st_k. However, the time needed to compute the schedule is prohibitive, because the algorithm used has exponential complexity.

The scheduler incorporates the new request r_k into $\mathcal{U}'$ without first determining the starting time of the request units in r_k. The deadlines of the new request units is $st_k + \Delta\tilde{d}_{kx}$, where st_k is a value that has yet to be determined.

The optimal scheduler uses *constraint logic programming (CLP)* [110]. CLP profits from having a minimization factor in the optimality criterion. The scheduler is implemented in *Eclipse* [114].

The optimal scheduler builds a schedule for each drive. The jobs are incorporated to the drive schedules as early as possible. When the scheduler adds job J_j to drive schedule D_i, it determines the starting and finishing time of the load and read tasks. The read starts immediately after the load finishes. The time to start the unload is not set until the next job is added to D_i. If the next job is on the same RSM, then the unload task is not used. Otherwise the scheduler first schedules the unload of the previous job added to D_i for the earliest possible time and then adds the new job to D_i. Not scheduling the unload until it is really needed has three advantages. First, the scheduler is flexible about keeping an RSM loaded in a drive. Second, the chances to be able to start loading another drive as early as possible increase, because the scheduler does the unloads as late as possible. Third, the scheduler does not create unnecessary holes in the robot schedule.

The scheduler explores the whole tree of feasible schedules. It carries out the exploration by using a branch-and-bound algorithm whose ‘bind-criterion’ is the

minimization of the starting time of the new request, st_k. The schedule is built in an incremental way using the following algorithm:

1. Choose a job J_j of the set of jobs $\mathcal{J}'$ that still need to be incorporated into the schedule.

2. Choose a drive D_i to use for J_j.

3. Determine the set of jobs S_j in $\mathcal{J}'$ corresponding to the same RSM as J_j.

4. Choose non-deterministically a subset $S_{jx} \subseteq S_j$.

5. Build an optimal medium schedule for $S_{jx} \cup J_j$. The optimal medium schedule is obtained by ordering the jobs by increasing deadline of the read task, because the access time is constant.

6. Schedule the unload and load tasks if needed. If the RSM was loaded in the drive, no unload and load tasks are used and the starting time of the read of the first job in the medium schedule is the maximum between t_0, the time at which the schedule is being computed, and the time at which the read task from the last job in the schedule will finish.

There are two conditions for which the algorithm for building a schedule stops and backtracks. The first one is that the read task of the job being incorporated to the schedule cannot meet its deadline. The second condition is that incorporating a job to the schedule should make the starting time st_k receive a bigger value than the best starting time obtained so far for a feasible schedule.

Although all the task and drive combinations may be tried, the scheduler begins by trying the combinations that have better chances of obtaining a low upper-bound for the starting time st_k. The tasks are initially ordered by increasing deadline. As the request units corresponding to the new request r_k do not have deadlines yet, the value used for sorting these jobs is the arrival time t_0 plus the relative deadline of the request unit. The scheduler first tries to schedule all the jobs on the same RSM together, because this makes the best use of the jukebox resources and, thus, is expected to provide the best starting time. Therefore, the first subset of jobs on the same RSM to try for each job J_j is the full set $S_{jx} = S_j$.

There are two different strategies for choosing the drives. One strategy chooses first the drive that will become available earlier, while the second strategy analyzes the drives by index order (D_1, D_2, ..., D_m). Both strategies allow building optimal schedules, although the schedules built may be different. As the optimality is measured on a local basis, every time a request arrives at the system, using one or the

other strategy may lead to an overall different performance. The reason for the potential difference in performance is that the two strategies build different schedules, thus, when a new request arrives the state of the system is different according to the strategy used. The state of the system, in turn, determines what the minimum starting time of the new request is.

The scheduler guarantees that two jobs on the same RSM are not assigned to different resources in the same period of time, by using a *build-and-test approach*. After assigning a set of jobs $J_j \cup S_{jx}$ to drive D_i, the scheduler checks that there are no conflict with the other jobs in the schedule. If there are conflicts it considers the combination of the set of jobs $J_j \cup S_{jx}$ and drive D_i invalid and discards it. However, it could be possible to assign $J_j \cup S_{jx}$ to D_i at a slightly later time, once the other jobs on the same RSM have finished executing. The choice of discarding the combination is a small sub-optimization which does not seem to change the optimality of the solution.

In Chapter 9 we compare the performance of Promote-IT with that of the optimal scheduler. Unfortunately, because of the exponential complexity of the optimal scheduler, we could only obtain results for simple request sets, where each request has only few request units.

7.3 Extensions to Existing Jukebox Schedulers

We present now the extensions to existing jukebox schedulers that we made in order to compare them with Promote-IT. We discuss briefly the algorithms used—in the extent that they differ from the algorithms originally proposed—and the implementation. The implementation of the three schedulers is based on the implementation of Promote-IT.

7.3.1 Extended Aggressive Strategy

The *extended aggressive strategy* is based on the *extended switch-read model* presented in Section 5.4.2. It is an extension of Lau's aggressive strategy that allows us to deal with non-identical drives, variable load and unload times, and multiple robots.

As we discussed in Sections 2.2.1, Lau et al. try assigning a job only to one drive. If that fails, then the job cannot be assigned and the algorithm fails for the attempted starting time. This approach is correct for identical drives and constant switching times. In the presence of non-identical drives and variable switching times, this approach is too limited. The extended aggressive strategy uses the tree pruning of Promote-IT to assign resources to the jobs (see Section 6.5.1).

Lau et al. do not specify in their articles what to do when data corresponding to a new request must be read from an RSM that is already loaded in a drive. We assume that because the RSM are left loaded in the drives until the drives are needed again, the algorithm first reads the data from the loaded RSM before unloading them.

We use the implementation of Promote-IT as a basis for implementing the extended aggressive strategy. The implementation optimizations of Promote-IT presented in Section 6.9 also hold for the extended aggressive strategy.

7.3.2 Extended Conservative Strategy

The *extended conservative strategy* is based on the *minimum switching model* as shown in Figure 5.1 on page 84. Thus, it models the unload and load operations separately. As we discussed in Section 2.2.1, the only way to keep the unload and load operations together in the conservative strategy is to use worst-case unload times or to assume constant switching times.

The extended conservative strategy uses the LDL strategy of Promote-IT, which can build correct Back-to-Front schedules without using the worst-case time.

In Section 9.4 we show that decoupling the load and unload improves the performance of the scheduler. Therefore, decoupling the load and unload in the extended conservative strategy provides a better performance than using a single switch operation and computing with the worst-case switching time.

The implementation of the extended conservative strategy is the same as the implementation of Promote-IT with the exception that it does not use the early dispatcher. Therefore, each task is dispatched at precisely the time that the scheduler indicates.

7.3.3 Fully-Staged-Before-Starting

To implement the FSBS scheduler we use the earliest starting time first (ESTF) scheduling policy of Promote-IT and restrict the requests to having all request units with the same delta deadline ($\forall\, i,j \,:\, \Delta\tilde{d}_{ij} = 0$). Therefore, the user will only be able to start consuming the data once the data of all request units has been staged. If the request consists of request units on multiple RSM, the scheduler will try to use multiple drives to read the data in parallel. If there are multiple requests for the same RSM, the scheduler will try to read all the requested data from the RSM at once. Thus, our implementation meets the goals proposed by Federighi et al. [29] and the other work proposing full staging before starting discussed in Section 2.2.2. Additionally, our version of FSBS provides a confirmation ASAP and guarantees that the starting time confirmed to the user is respected.

7.4 Summary

This chapter presented the five schedulers we use to evaluate the performance of Promote-IT. The *jukebox early quantum scheduler (JEQS)* and the *optimal scheduler* are new schedulers we propose in this dissertation, while the *extended aggressive strategy*, the *extended conservative strategy* and *Fully-Staged-Before-Starting* are extensions of existing schedulers.

JEQS is a periodic scheduler that uses the robot in a cyclic way. It is based on the use of *early quantum tasks (EQTs)*. The goal of JEQS is to incorporate the requests into the schedule so that they can start in the next cycle of a drive as EQTs. Although, JEQS can incorporate requests in the schedule early, in Chapter 9 we show that it performs much worse than Promote-IT. This bad performance is intrinsic to any periodic jukebox scheduler.

The optimal scheduler computes an optimal schedule for each incoming request. This scheduler finds the minimum response time for each request. As we have shown in Chapter 5 the problem is $\mathcal{NP}$-hard, therefore, the optimal scheduler cannot compute the schedules in polynomial time. The computational complexity of the scheduler increases rapidly with the complexity of the requests and the system load. In Chapter 9 we show that this scheduler can only be used for small test sets and can only cope with low system loads. Therefore, we use this scheduler only as a performance parameter for the other schedulers.

Chapter 8

Implementation and Simulation Environment

To support the implementation and evaluation of jukebox schedulers, both in real and simulated environments, we developed a toolbox called *JukeTools*, which we describe in this chapter.

8.1 JukeTools

JukeTools is a toolbox that provides an environment for implementing and evaluating jukebox schedulers. The schedulers can be tested in numerous real and simulated environments. They can be tested with different types of requests, caching policies and hardware. The verification functionality of the toolbox checks the validity of the schedules and helps the developer to detect errors. JukeTools is especially useful to detect resource-contention problems. Analyzer tools create detailed reports on the behaviour and performance of a scheduler, and provide comparisons between schedulers.

The toolbox supports simulated and real users to use simulated and real hardware. It hides the difference between ‘simulated’ and ‘real’ from the HMA components. Therefore, the developer can use the same code for the simulations and the operational system. In this way, the HMA components can be thoroughly tested in a controlled environment before using them in operational systems. Bosch et al. used a similar approach to develop a file system for secondary storage [15]. They argue that using the same code for real and simulated helps to: (1) be more confident that simulated off-line performance numbers show real and representative on-line performance numbers, (2) detect performance bottlenecks of the algorithms easily, (3) analyze new algorithms off-line before they are integrated into a production system, (4) be more confident that no side effects are introduced when a simulated algorithm is moved into a real system, (5) construct a reference system into which

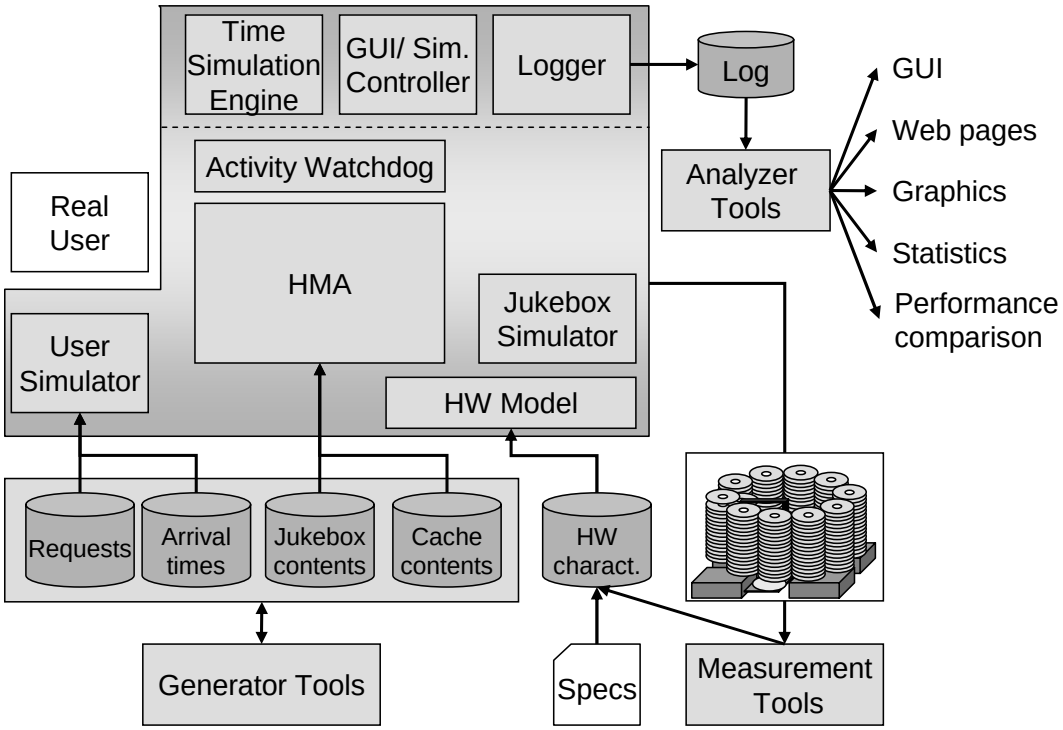


Figure 8.1: Architecture of JukeTools.

other algorithms can be easily integrated and compared, and (6) easily migrate algorithms form off-line simulators into real systems.

Figure 8.1 gives an overview of the toolbox. In the following sections we discuss the different components of the toolbox. The jukebox scheduler, which is part of the HMA, is at the core of JukeTools. The main goal of JukeTools is to provide a flexible environment to develop and evaluate jukebox schedulers. We classify the functionality provided by JukeTools into:

Time simulation. The toolbox component involved is the *time simulation engine.* The time simulation engine provides the time to all the components that run simultaneously with the HMA (i.e., the components shown in Figure 8.1 sharing the same outer box as the time simulation engine).

Output control and analysis. The toolbox components involved are the *GUI and simulator controller*, the *logger*, and the *analyzer tools*. Like the time simulation, this functionality is available to all active components.

Interface to hardware. The toolbox components involved are the *jukebox simulator*, the *hardware model*, and the *measurement tools.*

Framework for pluggable jukebox scheduler. The toolbox components involved are the components of the HMA—specially the *jukebox controller*—and the *activity watchdog*.

Workload & content generation. The toolbox components involved are the *generator tools*, and the *user simulator*.

The toolbox is implemented in Java, except for some functionality which is operating system dependent. The drive controllers use the Java Native Interface (JNI) to call C functions on Linux in order to open and close the drives and get drive specific information. Additionally, the analyzer tools use gnuplot to generate the graphics.

Although the operating system and the programming language do not provide real-time guarantees, the timeliness of the system is enough for the time requirements of scheduling a jukebox. In cases where strong real-time guarantees are required the implementation can be ported to real-time Java [90]. In [66] we present an early implementation of the jukebox scheduler that works under the real-time operating-system Nemesis [92].

Using JukeTools we have implemented and compared Promote-IT, JEQS, the extended aggressive, the extended conservative strategy and Fully-Staged-Before-Starting (FSBS). The off-line optimal scheduler presented in the previous chapter uses some of the tools provided by JukeTools, but runs outside the framework of JukeTools.

8.2 Time Simulation

The *time simulation engine* is an event simulator with the capacity to eliminate waiting time from the simulation. Waiting times in a simulation are the periods where a real system would wait for hardware operations or new user requests, and the system is not busy performing computations. The engine reduces the execution time of a simulation considerably. We have executed simulations corresponding to 24 hours of system use in 15 minutes.

The engine manages the virtual time of the system, delivers timed wake-up events and provides thread synchronization through semaphores. Threads register themselves with the engine at creation time. By tracking the state of the threads through the use of the semaphores the engine can detect and eliminate waiting times.

A thread can be in one of three states: *active*, *blocked* or *sleeping*. An active thread performs computations and needs to run in real time. A blocked thread is waiting for an event from an active thread, e.g., waiting on a semaphore. A sleeping

thread is waiting for its wake-up event, e.g., to simulate waiting for the robot to finish moving.

As long as at least one thread is active, the virtual time advances in real time. When all threads are blocked or sleeping, the time simulation engine advances the virtual time to the time of the next event.

The time simulation engine is not a thread scheduler—thread scheduling is performed by the Java Virtual Machine (JVM)[67]. The engine runs in a thread with the highest priority so that it always gets the right to execute when ready. Bosch et al. [15] use real and virtual time in a similar manner.

8.3 Interface to Hardware

JukeTools provides a uniform interface to the jukebox hardware through the jukebox simulator. Figure 8.2 shows that the difference between using real and simulated hardware is transparent to the scheduler and the dispatcher. The jukebox simulator uses the time simulation engine to simulate the execution of the operations.

The jukebox simulator uses the hardware model presented in Chapter 4. The hardware model is strongly data-driven. It is built using the specifications provided in the *hardware-characteristics file*. The information of the file may correspond to vendor specifications, the output of the *measurement tools* or the imagination of the toolbox user. We developed several measurements tools to measure the performance of our jukebox and CD-ROM drives.

A goal of the jukebox simulator is to provide the same execution pattern during each execution of a simulation, so that the results are reproducible: if we run a simulation with the same input we want to obtain the same performance results. Therefore, the jukebox simulator assumes that the execution of the operations take exactly the time indicated by the model.

The simulator uses the 'simulation model' proposed by Ruemmler et al. [94] when relevant. The drive simulator, for example, keeps track of the last time that the drive performed a read to decide if a spin-up is needed. However, we do not consider it relevant to know the exact rotation time in an access to data on a disk, because it is very small compared to the other components of the access. Additionally, the exact rotation time varies if a task is dispatched with a small time difference (e.g., one millisecond), therefore, the resulting execution will not be the same.

We performed many comparisons between using real and simulated hardware using our smartDAX jukebox. We concluded that the performance of the HMA is not affected in an important way by the type of hardware used (real or simulated). When using real hardware an early dispatcher has more opportunities of dispatching

tasks earlier. However, in most cases, the difference in time is so small, that it does not affect the overall system performance.

At present the implementation of the hardware model can handle any type of optical and magneto-optical jukebox. We are mainly concerned with this type of jukebox technology, because disks are better suited for random access than tapes, and can be loaded and unloaded faster. The implementation can easily be extended to include other type of storage media, drives and jukebox hardware. A good starting point to include magnetic tapes is to implement the model to estimate the locate-time on serpentine tapes provided by Hillyer et al. [44] and the benchmark methodology presented by Johnson et al. [53].

8.4 Output Control and Analysis

JukeTools provides different ways to evaluate and monitor the operation of the system. The *logger* provides a log service to all the components in the toolbox. The log is a set of XML-files [113]. We use XML, because it permits to have logs that are human readable and at the same time can be easily processed automatically by *analyzer tools*—using XSLT [112] or XML-parsers. Another advantage of using XML is that each component of the toolbox can generate specific log entries without affecting existing analyzers, because unknown entries are ignored.

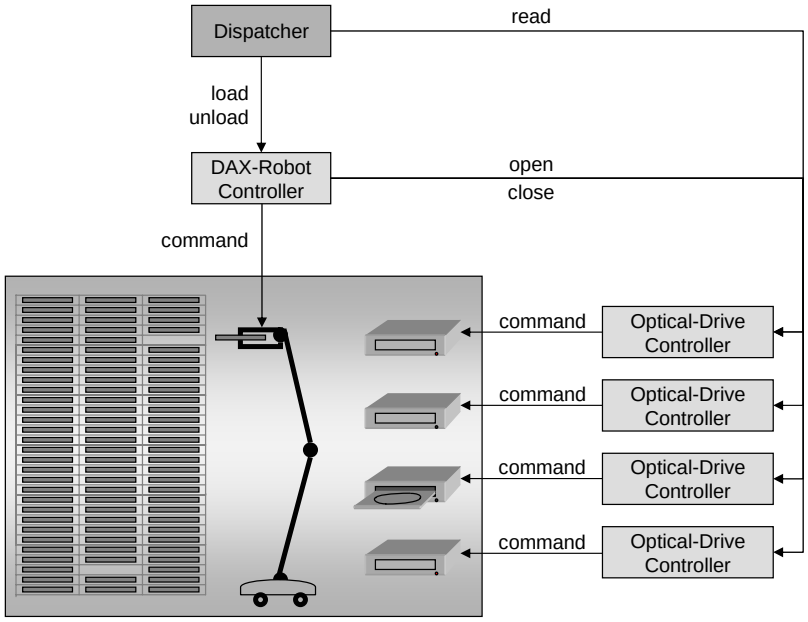
The toolbox user can easily indicate what type of messages should be stored in the log and the logger filters out all the messages that do not correspond to that specification. Thus, it is easy to control the size of the log files.

The toolbox components can request specific log files and indicate to which log file each log entry must be written. The logger adds timestamps to every log entry and writes the entries in the log when the other jukebox components are idle.

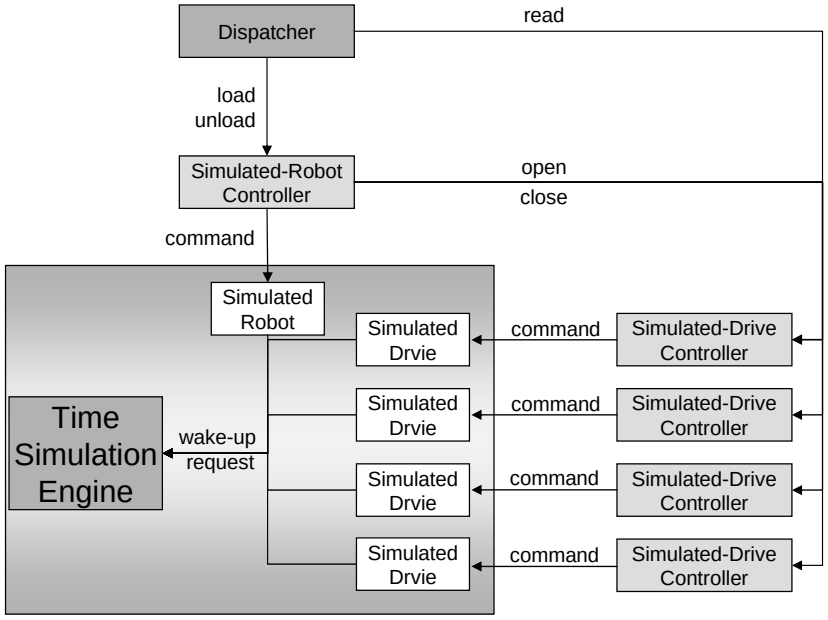
The log is processed by *analyzer tools* to produce reports in the form of web pages, graphics, statistics and performance comparisons. Other analyzers combine the statistical data of multiple runs to compare the performance of schedulers under different load conditions. The graphics shown in the next chapter and in Section 6.10 were generated by the analyzer tools.

More complex analyzers provide graphical representations of the schedules created during the execution of the system and details about the utilization of the jukebox resources.

The log and the analyzer tools are also very useful to implement and debug a jukebox scheduler. The *jukebox controller* and multiple *activity watchdogs* report incorrect behaviour of the jukebox scheduler components.



(a) Dispatching tasks to a smartDAX700 jukebox.



(b) Dispatching tasks to a simulated jukebox.

Figure 8.2: Dispatching tasks to real and simulated jukeboxes.

Apart from the logger, the *GUI* offers different views of the running system and an attached simulation controller allows the user to pause, resume and stop the simulation. New views can be easily added to the GUI.

8.5 Framework for Pluggable Jukebox Scheduler

The implementation of the HMA is highly modular and the interfaces between the components are small and simple. This makes plugging in different jukebox schedulers and combining different schedule builders and dispatchers easy. The schedule builder, the dispatcher and the jukebox controller run as independent threads.

The jukebox controller operates as a schedule verifier, because it only performs valid commands. It determines if a command is valid using information about the location of each RSM in the jukebox and the state of each device (see Section 4.3). This information cannot be modified directly by the scheduler. The state of the devices only changes as a result of executing commands, which are handled by the dispatcher.

The jukebox controller can detect illegal commands that request to: unload an empty drive, load a loaded drive, read data from an empty drive, unload an RSM different from the one loaded or load the same RSM in two different drives. The controller also reports if the commands are being executed late or if the execution time is different from the time that was estimated by the scheduler. The latter is especially important when using real hardware.

Different components observe the execution of the HMA using the publisher-subscriber pattern [31]. The *activity watchdog* uses the events generated by the HMA to detect possible deadlocks in the use of shared resources. The watchdog can detect deadlocks in the HMA components, if the components publish the beginning and end of an operation that may lead to a deadlock, and indicate the maximum time to perform the operation.

8.6 Workload and Content Generation

The requests for the HMA originate either from real users outside the toolbox or from the user simulator. The *user simulator* generates the workload for the HMA based on data from a *requests file* and an *arrival-times file*. These files are created by the *generator tools*. The generator tools also create synthetic *jukebox contents* and *cache contents*. The toolbox works with real jukebox and cache contents as well.

The generator tools store the data in XML-files, which can be used for multiple simulations. The parameters specified by the toolbox user are also defined in XML.

8.6.1 Jukebox-Contents Generator

The toolbox user can specify which type of contents should be generated. We have defined four basic types—long videos, short videos, music and discrete data—for which the user may further define parameters to determine bandwidth, duration and size. The user also defines the proportion of the four types in the jukebox. The contents are generated using a uniform distribution based on the parameters given by the users. The contents are generated independently of the type of RSM in which they are stored. So we can perform simulations using the same jukebox contents with different jukeboxes and RSM types.

The contents are organized in albums and files. In the case of a long video, an album is the video and the files are the parts in which the video must be chopped to fit in the RSM when the capacity of an RSM is less than the length of a video. For the other content types, an album is simply a directory.

Examples of long videos are movies, documentaries and sport competitions. The idea of a short video is to represent cartoons, commercials, video-clips and short news items. The short videos are grouped together into albums by affinity, e.g., multiple cartoons of Bugs-Bunny.

The contents of the jukebox are assigned a popularity value at the time of creation. The popularity distribution is a parameterized Zipf distribution [117]. Zipf distributions have been detected for most data access systems (see Section 3.4). By parameterizing a Zipf distribution [85] we can generate different types of distributions, including the uniform distribution. The files and albums are assigned independent popularity values, except for the case of long videos. Assigning independent popularity values to the files and the albums seems to fit correctly the world of (pop-)music where albums may have just one popular song that is in the charts and requested very often, while the rest of the songs are hardly ever heard. People also listen to full albums, because they like all the music on the album, even though the album may not have any song in the charts.

The contents are further classified in clusters. Cluster should represent genres, e.g., soul, disco, etc. This information is used by the request generator when deciding what data to request together.

8.6.2 Request Generator

We have implemented a request generator that generates requests for the synthetic jukebox contents described in the previous subsection. The output provided by the

tool is a request set containing as many requests as indicated by the toolbox user. The user specifies which type of data should be requested giving probabilities for each type of data. The probability determines the proportion of each type of data in the request set.

All request units in a request generated by this tool are for the same type of data. The tool assumes that continuous-media (i.e., audio and video) is streamed and the data is consumed in a sequential way. Therefore, the relative deadline of each request unit is at the time at which the consumption of the data of the previous request unit should finish and the relative deadline of the first request unit is 0 (formulated in Equation 8.1). The tool can split the files of continuous-media in multiple request units to obtain better response times. If the data of the request is discrete, then all request units have the same relative deadline of 0.

$$\begin{aligned} \Delta\tilde{d}_{k,0} &= 0 \\ \Delta\tilde{d}_{k,j} &= \Delta\tilde{d}_{k,j-1} + b_{k,j-1}\, s_{k,j-1} \end{aligned} \tag{8.1}$$

All the data in a request belongs to the same cluster. The request generator first selects a cluster randomly using a uniform distribution and then uses the popularity of the files and the albums to include data in a request. We have defined the following basic types of requests:

One file The request is only for one file. Therefore, the number of RSM is one.

One full album The request contains a request unit for each file in the album in the order given by the album. In the case of long video, the request units may be on different RSM, otherwise they will always be on the same RSM.

Multiple full albums The request contains a request unit for each file in each of the albums. The request units are ordered per album. They are with a high probability in different RSM.

Parts of one album The request contains request units corresponding to some files of one album.

Parts of multiple albums The request contains request units corresponding to just parts of albums. The request units are with a high probability in different RSM.

8.6.3 Cache-Contents Generator

The cache-contents generator generates synthetic cache contents. The *cache contents* keep information about the data currently in the cache and map the data to

Seed 1			Seed 2		
6 req/hour	**18 req/hour**	**60 req/hour**	**6 req/hour**	**18 req/hour**	**60 req/hour**
4.17	1.39	0.41	2.50	0.83	0.25
18.03	6.01	1.80	14.40	4.80	1.44
18.72	6.24	1.87	15.07	5.02	1.50
22.97	7.65	2.29	35.95	11.98	3.39
39.73	13.24	3.97	36.83	12.27	3.68
53.15	17.71	5.31	36.84	12.28	3.68
70.51	23.50	7.05	57.35	19.11	5.75

Table 8.1: Example of arrival times generated by the arrival-times generator using two seeds and three average arrival times.

file-system identifiers in secondary storage. The cache contents are managed by the *cache manager*, which is a component of the HMA (see Section 3.4). For the cache manager there is no difference between synthetic data and real data stored by itself during the operation of the system. This is another point that supports the transparent use of the HMA in a simulated or real environment.

The cache-contents generator operates in the following way. It first generates a request set—using the request generator. It then generates a report indicating the last time each file (or part of a file was requested) and how often it was requested. This data can be interpreted as the request arrival history of a previous run. The tool can also generate the cache contents from the log of a previous run.

When using synthetic cache contents, the cache manager decides which data should be in the cache at the beginning of the simulation. It does so by using either the last time the data was requested or the frequency with which the data was requested, or a combination of both.

8.6.4 Arrival-Times Generator

The arrival-times generator generates the times at which requests arrive at the system. The tool can use different known distributions, e.g., Poisson, Uniform, etc. We generate different load factors by varying the average inter-arrival time and using the same specific distribution. The tool can also use as input the request-arrival times of a real system.

To evaluate the performance of a scheduler, we feed the system the same request set and different inter-arrival rates. Table 8.1 shows the arrival times of the requests when at an average of 6, 18 and 60 requests per hour. The times are generated using Poisson and two different initial seeds (Seed 1 and Seed 2).

8.7 Summary

This chapter presented JukeTools: a toolbox to develop, evaluate and implement jukebox schedulers. The toolbox allows the developer to concentrate on the topics relevant for scheduling and abstract from secondary issues. JukeTools helps to detect inconsistencies in the use the jukebox resources and missed deadlines. It also provides detailed reports on the performance of the scheduler, which can be used to detect bottlenecks and inefficiencies. The toolbox is very flexible and can be configured easily to simulate numerous hardware architectures, scheduling policies and user behaviour. Therefore, it provides an ideal framework to evaluate and compare different schedulers. JukeTools provides an environment that makes simulation transparent to the developer to the extent that the same code can be used for simulations and operational systems.

The toolbox can also be used to evaluate different hardware architectures, caching policies and services offered by the storage system. Furthermore, making some modifications to the semantics of the components and the interfaces, the toolbox can also be used to implement real-time schedulers for the production application presented in Section 2.3.

Chapter 9

Performance Evaluation

This chapter evaluates the performance of different jukebox schedulers: Promote-IT, JEQS, optimal, FSBS, extended aggressive strategy, and extended conservative strategy. The first three are the new schedulers that we present in this dissertation, while the others are extensions of existing schedulers.

The extended schedulers have better properties than the original ones, while still keeping the features of the original schedulers that we consider most important to evaluate. The chapter is organized in a way that shows the importance of each of the characteristic features of Promote-IT. Table 9.1 provides a summary and comparison of the features of Promote-IT and of the other jukebox schedulers.

Section 9.1 shows the need to schedule the jukebox in an aperiodic way by comparing Promote-IT and FSBS against JEQS. It shows that Promote-IT is clearly better than JEQS and that many times even FSBS is better than JEQS.

Section 9.2 shows the benefits of using pipelining instead of full staging before providing access to the user by comparing the performance of Promote-IT against that of FSBS. The comparison shows that the response time of Promote-IT is much shorter than that of FSBS.

Section 9.3 shows the need to use an early dispatcher, especially when using a Back-to-Front strategy. It shows that the LDL strategy of Promote-IT is clearly superior to the extended conservative strategy, even if they both use the same scheduling algorithm. It shows also that JEQS and even the Front-to-Back strategies benefit from early dispatching. As anticipated in Chapter 6, the benefits for Front-to-Back strategies are small.

Section 9.4 shows the benefits of decoupling the load and unload operation by comparing Promote-IT against the extended aggressive strategy. The comparison shows that Promote-IT performs better than the extended aggressive strategy. However, the difference in performance is much smaller than with the other schedulers.

Section 9.5 shows that the response time of Promote-IT is near to the optimal response time by comparing Promote-IT against the optimal scheduler. Due to the

		Extensions			New		
		FSBS	Extended Aggressive	Extended Conservative	Promote-IT	JEQS	Optimal
Requests	Multiple request units	+	+	+	+	-	+
	Unrestricted location data	+	+	+	+	-	+
	Continuous data	+	+	+	+	+	+
	Discrete data	+	+	+	+	-	+
Hardware	Multiple drives	+	+	+	+	+	+
	Non-identical drives	+	+	+	+	-	-
	Multiple robots	+	+	+	+	+	-
	Flexible robot functionality	+	-	+	+	-	-
	Flexible robot scope	+	-	+	+	+	-
Scheduler	Real-time guarantees	+	+	+	+	+	+
	Variable load/unload times	+	+	+	+	+	+
	Aperiodic scheduling	+	+	+	+	-	+
	Pipelining	-	+	+	+	+	+
	Decoupled Load/Unload	+	-	+	+	-	+
	Early dispatching	+	-	-	+	+	-
	Polynomial algorithm	+	+	+	+	+	-

Table 9.1: Characteristic of the evaluated schedulers.

limitations of the optimal scheduler regarding computational complexity, we could only verify this for small test sets.

Finally, section 9.6 compares all the schedulers and evaluates them regarding performance and flexibility. It shows that Promote-IT is the overall-best scheduler. Other schedulers are better than Promote-IT regarding particular performance criteria (e.g., the optimal scheduler provides shorter response times and JEQS has excellent computing times), but they perform very poorly regarding other criteria (e.g., the optimal scheduler requires exponential computing time and JEQS provides very long response times).

As shown in Table 9.1, not all schedulers can deal with flexible requests and any type of jukebox hardware. Therefore, in each section we restrict the type of requests and the jukebox architecture to the limitations of the most restrictive scheduler being evaluated.

In this chapter we show only a few representative test cases that highlight the difference between the schedulers being compared. Furthermore, for Promote-IT we present only the performance of ESTF and LDL and leave out EDF and LSTL. ESTF and LDL are good representatives of Front-to-Back and Back-to-Front, respectively (see Section 6.10).

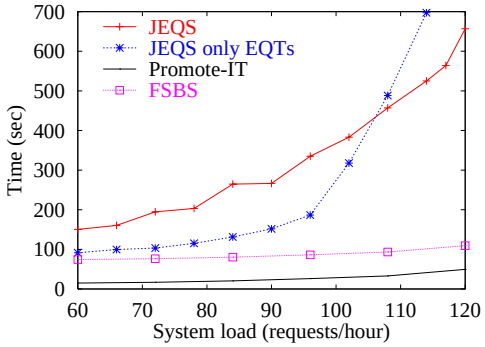
9.1 Aperiodic vs. Periodic Scheduling

This section presents a comparison between aperiodic and periodic scheduling. Aperiodic scheduling is represented by Promote-IT[1] and FSBS; periodic scheduling is represented by JEQS. For JEQS we consider the two variations proposed in Section 7.1.1: scheduling normal quantum tasks (shown in the plots as 'JEQS') and scheduling only EQTs (shown in the plots as 'JEQS only EQTs').

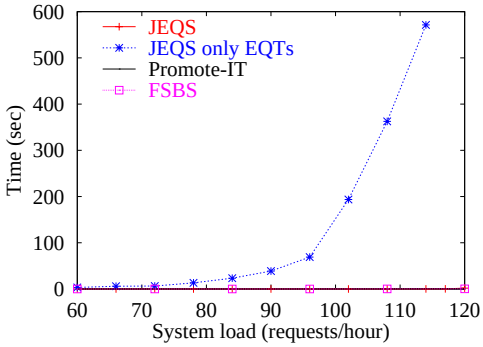
We use FSBS in this comparison, because even though FSBS is very simple in many cases it performs better than JEQS. FSBS has a similar behaviour to a First-Come-First-Serve scheduler, which virtually means that no serious scheduling is done. It first serves a request completely and only then it provides access to the data of the request.

We show some performance results for simulations with two jukebox architectures: the *fast jukebox* and the *slow jukebox*. The architectures differ in the speed of the drives, fast and slow, respectively. The test setup is described at the end of this section. Figure 9.1 shows performance results for the fast jukebox and Figure 9.2

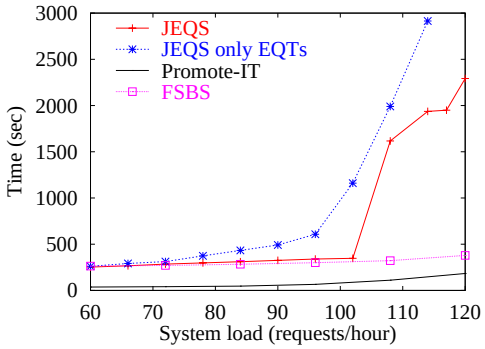
1 In this comparison the performance of Promote-IT is representative for the performance of the extended aggressive strategy and extended conservative strategy, because the difference in performance among these schedulers is negligible when compared with the difference among Promote-IT, FSBS and JEQS.



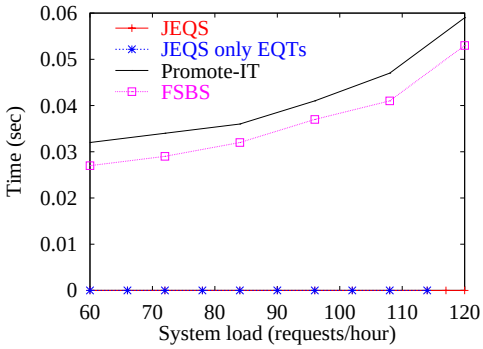
(a) Mean response time.



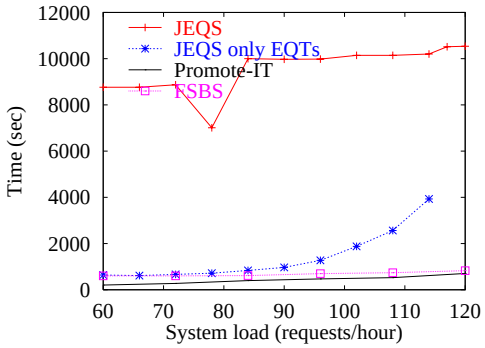
(b) Mean confirmation time.



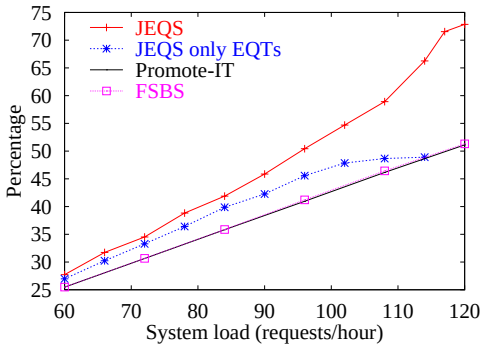
(c) Maximum response time for 90% of the requests.



(d) Mean computing time.

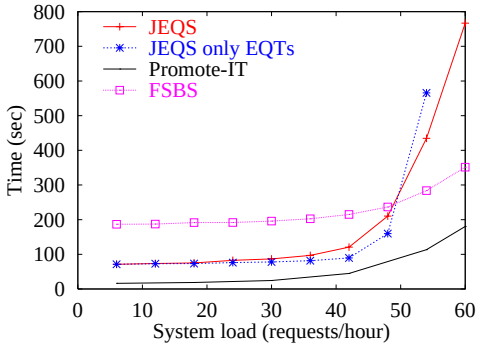


(e) Maximum response time.

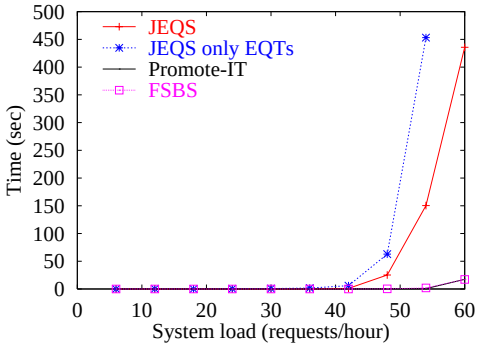


(f) Mean robot utilization.

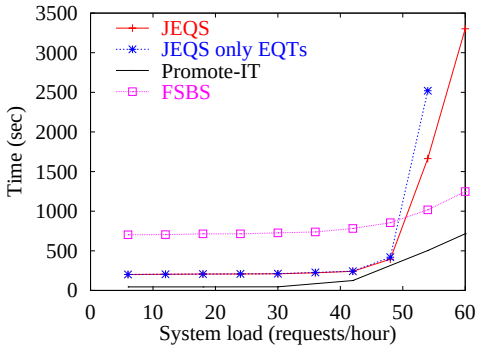
Figure 9.1: Aperiodic vs. periodic scheduling for the *fast jukebox*.



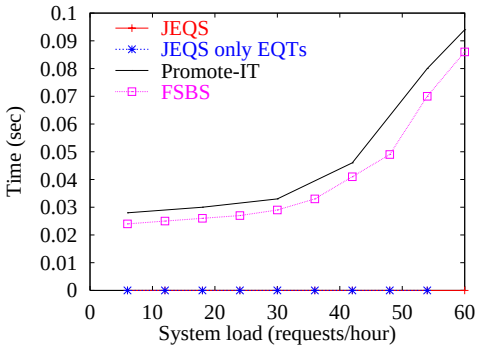
(a) Mean response time.



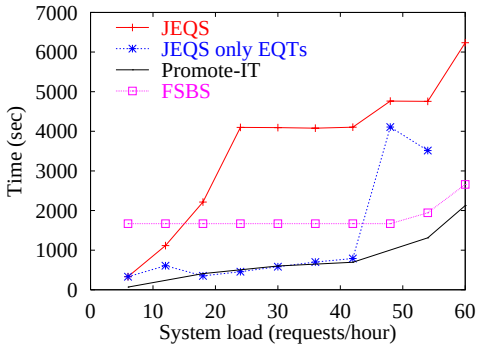
(b) Mean confirmation time.



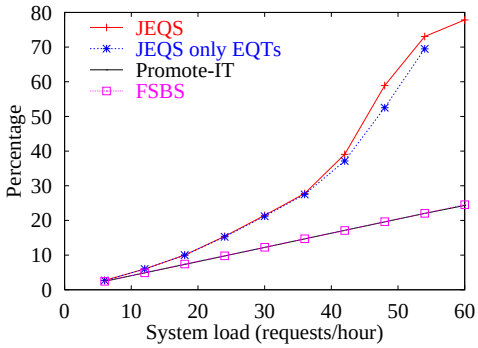
(c) Maximum response time for 90% of the requests.



(d) Mean computing time.



(e) Maximum response time.



(f) Mean robot utilization.

Figure 9.2: Aperiodic vs. periodic scheduling for the *slow jukebox*.

show the corresponding results for the slow jukebox. The plots in each figure show: (a) the mean response time, (b) the mean confirmation time, (c) the maximum response time for 90% of the requests, (d) the mean computing time, (e) the maximum response time, and (f) the mean robot utilization. The average cache-hit rate is 63%. The cache-hit rate is almost independent of the scheduler used or the system load.

The lines corresponding to 'JEQS only EQTs' do not show the values at the highest load levels in the plots, because the response time and confirmation times are unreasonably high. When the system load passes a certain limit, the length of the waiting queues grows so much that no new requests can be scheduled. When the system load is high, the performance of 'JEQS only EQTs' degrades very fast.

The response time of Promote-IT is much shorter than the response time of JEQS. As the system load increases, the performance of FSBS is also better than that of JEQS. When using the fast jukebox, the performance of JEQS is proportionally worse than when using the slow jukebox, because with the former more drive bandwidth is wasted with each switch.

The response time of Promote-IT is also shorter than that of FSBS. FSBS stages the whole file before giving access to the user. Therefore, the response of FSBS has as lower limit the time to buffer the whole file, while the lower limit for Promote-IT is the time to buffer the first request unit. In Section 9.2 we show that the difference in performance between Promote-IT and FSBS is even bigger when the data of a request is stored in multiple RSM.

JEQS uses the resources poorly, because it performs multiple switches for reading data from an RSM. In contrast, Promote-IT and FSBS use the resources efficiently by performing the minimum amount of switches required to read the data. Therefore, the robot utilization of JEQS is higher than that of Promote-IT and FSBS (see Figures 9.1(f) and 9.2(f)). However, the fact that the robot utilization of 'JEQS only EQTs' is proportionally lower than that of JEQS is not due to a better robot utilization of the former, but to the fact that 'JEQS only EQTs' accepts new requests at a lower rate.

The confirmation time of the aperiodic schedulers is shorter that that of JEQS (see Figures 9.1(b) and 9.2(b)). The main difference can be seen with 'JEQS only EQTs', because this scheduler waits to accept a request until it can schedule it as an EQT. As the system load increases, the possibilities of accepting a request as an EQT diminish drastically (see discussion in Section 7.1.1).

Periodic schedulers have a clear advantage over aperiodic schedulers in the computing time (see Figures 9.1(d) and 9.2(d)). However, this advantage is not visible to the end user, who notices only the response time and the confirmation time.[2]

2 When evaluating the performance of the optimal scheduler, we will show that the computing time becomes an important parameter when it influences the confirmation time.

To compare the performance of the two versions of JEQS, it is useful to analyze the maximum response time. Figure 9.1(e) shows that the maximum response time of JEQS is very high, which is a result from incorporating into the schedule a normal quantum task with a long period. As we explained in Chapter 7, scheduling normal quantum tasks is a trade-off to make the scheduler able to cope with higher system loads. Figure 9.1(c) shows that in 90% of the cases, scheduling normal quantum tasks results in a better response time than scheduling only EQTs. Figure 9.1(b) clearly shows that scheduling only EQTs also results in long confirmation times. When the schedulers are able to reject requests, the difference in performance between the two variations of JEQS nearly disappears, because the conflicting requests (i.e., those that cannot be accepted as EQTs) are rejected.

We conclude that periodic scheduling is a bad technique for scheduling a jukebox, because even the FSBS scheduler—which is extremely simple—performs better than JEQS in many cases. The bad performance of JEQS is not a characteristic of this particular scheduler, but is intrinsic to any periodic jukebox scheduler. As discussed in Section 5.6, a periodic scheduler either needs to use the robot in a cyclic way, or take into account the worst-case time for robot contention 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. For FSBS it is the time needed to stage all the data of the request.

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 when the system load increases, because the periodic scheduler wastes drive bandwidth with unnecessary switches.

Test Setup

The request set consists of 1000 ASAP requests that follow a Zipf distribution. Each request corresponds to one long-video file, because of the restrictions imposed by JEQS. To be able to use JEQS the request must be only for data stored in one RSM in a contiguous fashion. Additionally, JEQS needs the data to be continuous-media. When using Promote-IT the request is split in request units of 100 MB in size. The requests cannot be rejected, i.e., deadline and maximum confirmation time are infinite.

The contents of the jukebox consist only of long videos. The bandwidth of the videos is uniformly distributed in the range from 1 to 8 Mbps. Their duration is

	Fast-drives jukebox	Slow-drives jukebox
Number of Robots	1	
Number of Drives	4 (identical)	
Load Time (seconds)	21.8–24.9	
Unload Time (seconds)	14.3–17.4	
Media Type	Double-layered DVD-ROM	
Drive Technology	CAV	CLV
Transfer Speed (MBps)	7.96–20.53	7.96
Maximum Access time (seconds)	0.17	1.5

Table 9.2: Jukebox specification for the comparison between periodic and aperiodic scheduling.

uniformly distributed in the range from 15 minutes to 2.5 hours. The data in the jukebox is stored in double-layered DVDs and each video is stored completely in one disk. However, one disk may store multiple videos.

We use two jukebox architectures. Both architectures are based on a smartDAX as the one modelled in Chapter 4. The jukeboxes have four identical drives. Table 9.2 shows the most important parameters of the jukebox. We refer to the jukeboxes as *fast-drives jukebox* (or *fast jukebox*) and *slow-drives jukebox* (or *slow jukebox*).

The size of the cache is 10% of the jukebox capacity.

9.2 Pipelining vs. Full Staging

This section shows the benefits of pipelining over full staging. We compare the performance of Promote-IT against that of FSBS. As we explained in Section 7.3.3 FSBS uses the ESTF strategy of Promote-IT and the difference is in the requests fed to the system. The requests for FSBS have all delta deadlines set to 0, therefore, all the data of a request must be staged before the user can start consuming the data.

Figure 9.3 shows some performance results for simulations with the *fast jukebox* described in the previous section (see Table 9.2). The plots in the figure show: (a) the mean response time, (b) the maximum response time for 90% of the requests, (c) the mean confirmation time, (d) the mean computing time, (e) the mean robot utilization, and (f) the mean drive utilization.

The request set consists of 1000 ASAP requests that follow a Zipf distribution. The data in the jukebox consists of 30% long videos, 30% short videos, 30% music and 10% discrete data. The requests follow that pattern as well. The requests cannot be rejected, i.e., deadline and maximum confirmation time are infinite.

The previous section showed that Promote-IT provides shorter response times than FSBS in the case of requests for data stored in one RSM. This section shows that the difference is even greater when the request involves multiple RSM (see Figures 9.3(a) and 9.3(b)). In the latter case, FSBS needs to perform multiple switches before giving access to the data, while in most cases Promote-IT only needs to perform one switch to read the data corresponding to the first request unit and the rest of the switches are performed at a later time, when the scheduler finds time for them.

Additionally, Promote-IT has shorter confirmation times than FSBS, although the difference is not very significant (see Figure 9.3(c)). The reason for the higher confirmation time of FSBS is not the computing time (see Figure 9.3(d)), but the fact that FSBS becomes overloaded earlier and must keep the requests longer in the queue of unscheduled requests. Figures 9.3(e) and 9.3(f) show that the resource utilization of FSBS and Promote-IT is very similar.

9.3 Early vs. Conservative Dispatching

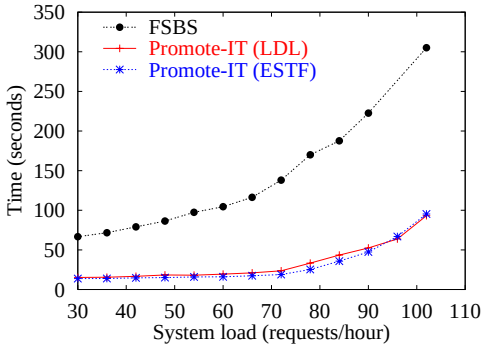
In this section we show that the performance of every heuristic jukebox scheduler benefits from early dispatching. The most important benefits occur when using the LDL policy in Promote-IT. Also JEQS benefits from early dispatching. Even a Front-to-Back strategy like the ESTF strategy of Promote-IT benefits from early dispatching, however, not in a significant way.

9.3.1 Back-to-Front Strategies

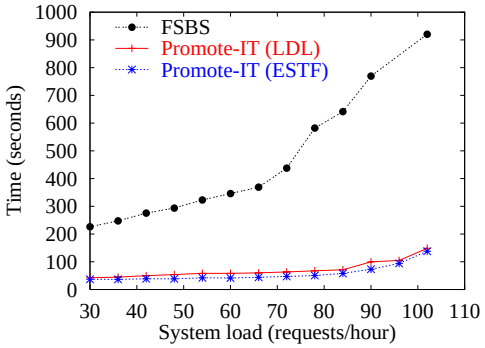
We compare the performance of Promote-IT and the extended conservative strategy. The difference between LDL and the extended conservative strategy is the early dispatching of the tasks, because the extended conservative strategy uses decoupled load and unload operations and the same scheduling algorithm as Promote-IT (see Section 7.3.2). We denote the extended conservative strategy simply as ‘Conservative’.

Figure 9.4 shows some performance results for the same test set used in the previous section. As in the previous section, the plots in the figure show: (a) the mean response time, (b) the maximum response time for 90% of the requests, (c) the mean confirmation time, (d) the mean computing time, (e) the mean robot utilization, and (f) the mean drive utilization.

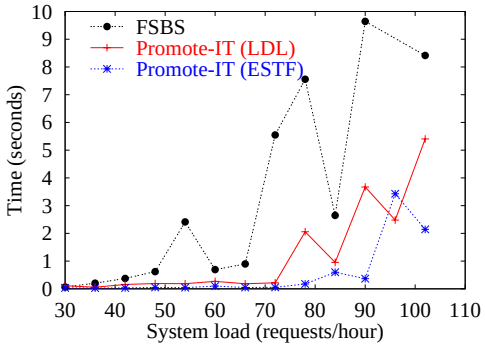
The response time and confirmation time of LDL and ESTF are very similar when compared against the corresponding times of Conservative (see Figures 9.4(a) and 9.4(b)). Furthermore, we showed in Section 6.10 that when the system load is



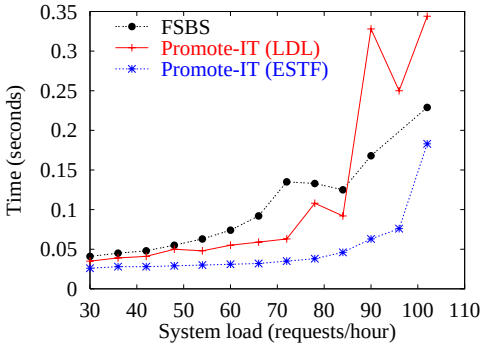
(a) Mean response time.



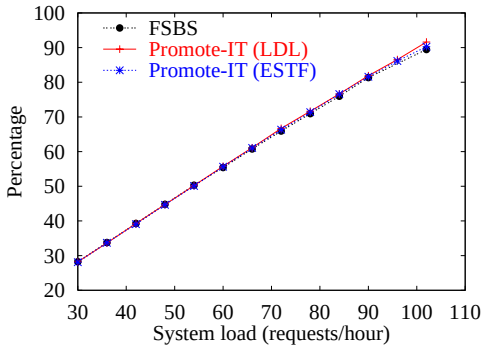
(b) Maximum response time for 90% of the requests.



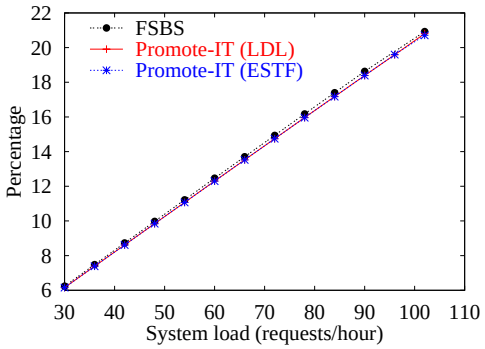
(c) Mean confirmation time.



(d) Mean computing time.

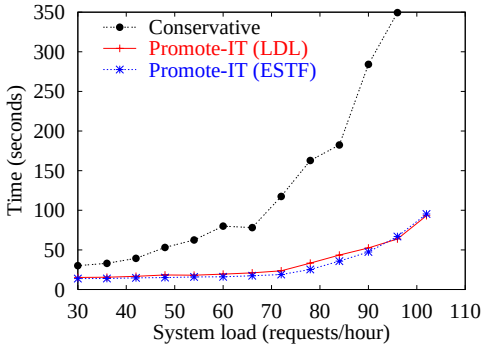


(e) Mean robot utilization.

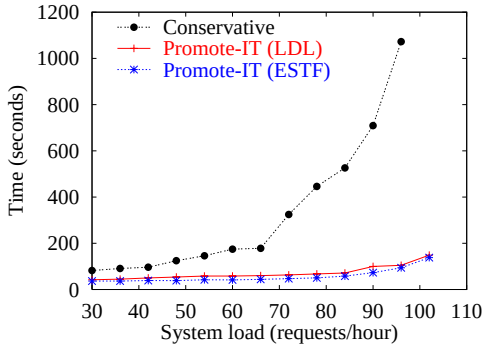


(f) Mean drive utilization.

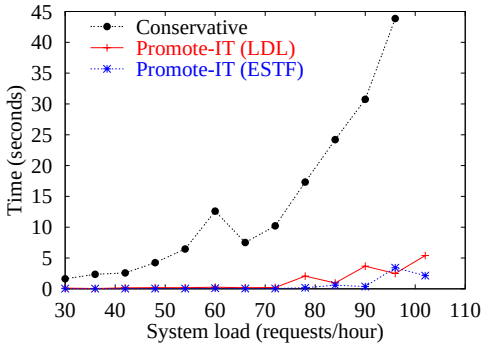
Figure 9.3: Pipelining vs. Full Staging.



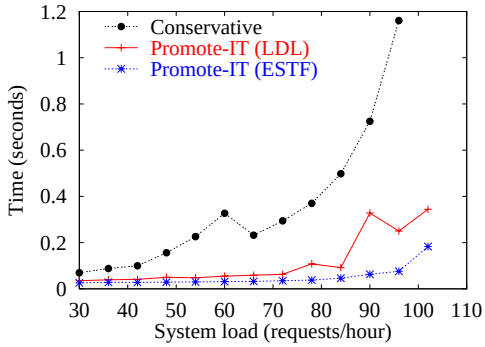
(a) Mean response time.



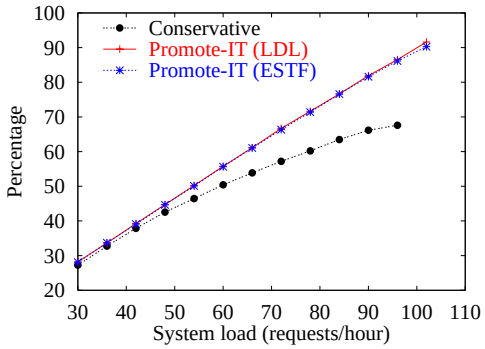
(b) Maximum response time for 90% of the requests.



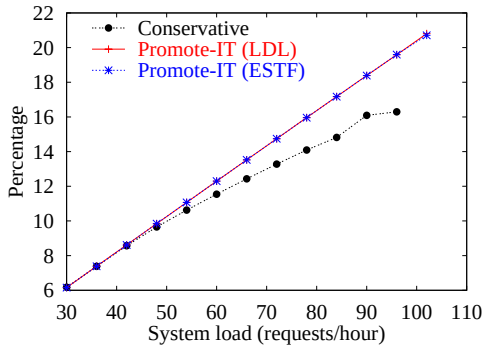
(c) Mean confirmation time.



(d) Mean computing time.

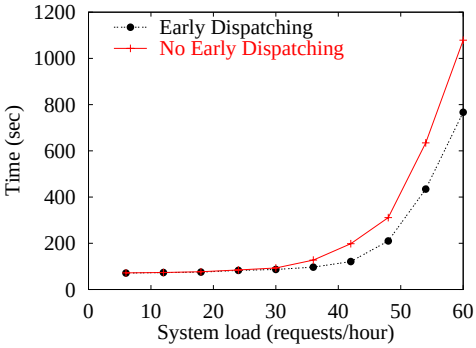


(e) Mean robot utilization.

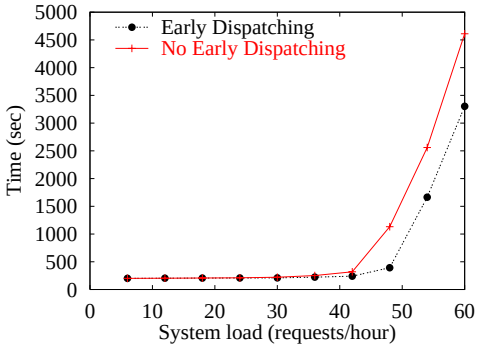


(f) Mean drive utilization.

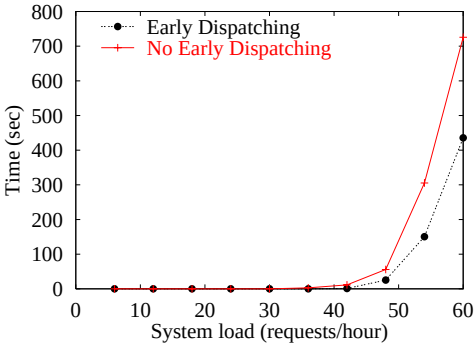
Figure 9.4: Early vs. conservative dispatching using a Back-to-Front scheduling strategy.



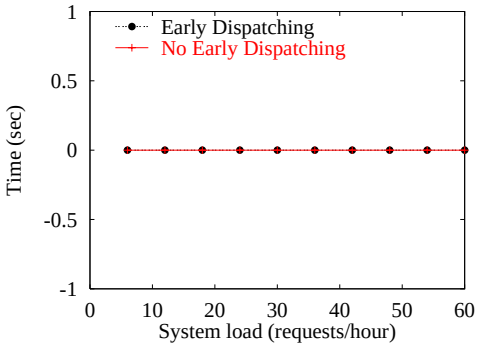
(a) Mean response time.



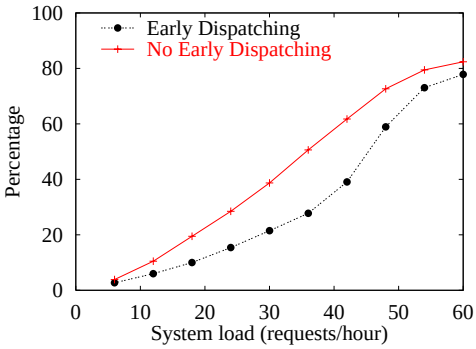
(b) Maximum response time for 90% of the requests.



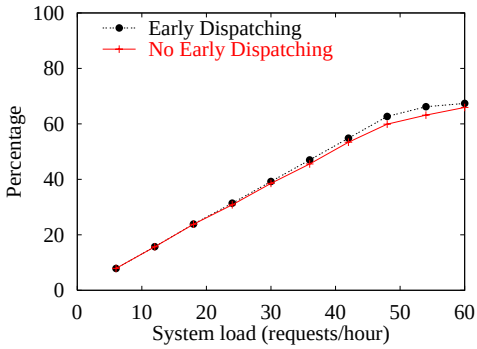
(c) Mean confirmation time.



(d) Mean computing time.



(e) Mean robot utilization.



(f) Mean drive utilization.

Figure 9.5: Early vs. conservative dispatching when using JEQS.

high, LDL performs slightly better than ESTF. This reinforces the idea that Back-to-Front is an interesting scheduling mechanism, when it is combined with early dispatching.

The confirmation time of Promote-IT is also lower than of Conservative (see Figure 9.4(c)). Conservative often fails to schedule incoming requests, because the starting time they should be assigned is too far into the future. Thus, the requests stay in the queue of unscheduled requests until the scheduler can incorporate them to the schedule. The computing time of Promote-IT is also shorter than that of Conservative (see Figure 9.4(d)), because the latter repeatedly attempts to build schedules for the requests in the queue without success.

As the system load increases, the difference in performance between Promote-IT and Conservative grows very fast. At the highest load level plotted, Conservative is unable to handle the load, because the waiting queue is too long.

The robot and drive utilization of Conservative is much less than that of LDL (see Figures 9.4(e) and 9.4(f)). When not using early dispatching, the resources are left idle, even if there are tasks in the schedule. Thus, when new requests arrive, their chances to be scheduled immediately are lower, even when the system load is low, because the scheduler has tasks scheduled for the future. The plots also show that while the resource utilization of LDL grows proportionally to the increase of the system load, the resource utilization of Conservative saturates at around 90 req/hour.

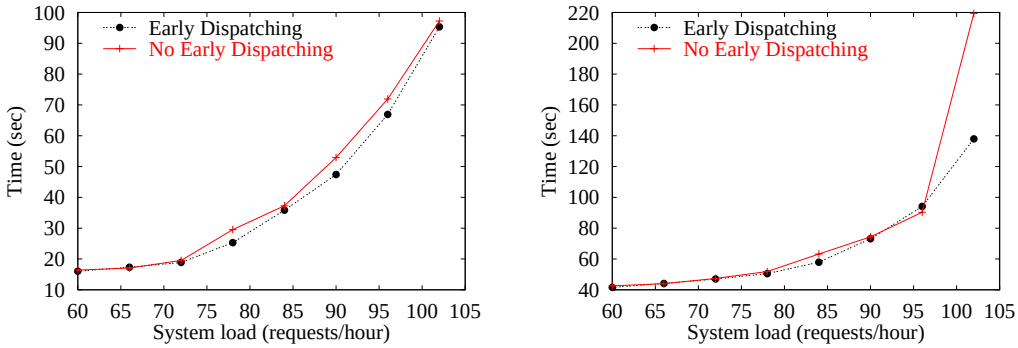
9.3.2 JEQS and Front-to-Back Strategies

Figure 9.5 shows the performance improvements in JEQS that result from using an early dispatcher. We use the request set and the *slow jukebox* presented in Section 9.1.

When dispatching early the system has the opportunity to read data from the RSM loaded in the drive. Thus, early dispatching reduces the number of switches needed to read the data. When there is enough idle time, the system has more opportunities of keeping the RSM loaded in the drive more often, thus, reducing the robot utilization (see Figure 9.5(e)). As a result the drives are used more efficiently—within the intrinsic inefficiency of JEQS—resulting in slightly higher drive utilization (see Figure 9.5(f)).

Using the resources more efficiently results in shorter response times (see Figures 9.5(a) and 9.5(b)) and shorter confirmation times (see Figure 9.5(c)). Given that the computing time of JEQS is nearly negligible, there is no noticeable difference in the computing time (see Figure 9.5(d)).

Figure 9.6 shows that even ESTF, which builds the schedules Front-to-Back, can profit from early dispatching. However, in this case, the benefit is small.



(a) Mean response time.

(b) Maximum response time for 90% of the requests.

Figure 9.6: Early vs. conservative dispatching when using ESTF.

Early dispatching is not beneficial for every request. In some occasions early dispatching may result in the new incoming request finding all drives busy and having to wait until one becomes free again. However, averaged over a large number of requests, early dispatching always results beneficial.

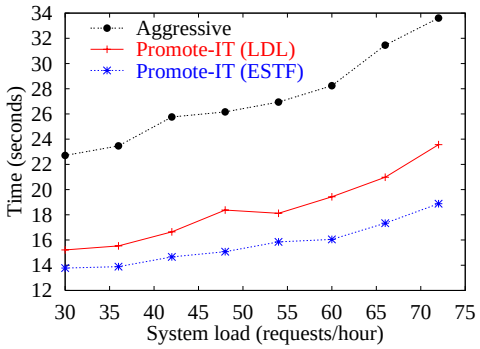
9.4 Decoupled vs. Coupled Load and Unload

In this section we compare the extended aggressive strategy and Promote-IT. We will denote the extended aggressive strategy simply as 'Aggressive'. The difference between the former strategy and Promote-IT is that Aggressive couples the load and unload into a single switch operation.[3] This means that the RSM stay loaded into the drives until the drives are needed again. Therefore, Aggressive needs to perform first an unload before using a drive, even if the drive and the robot are idle before the request arrival.

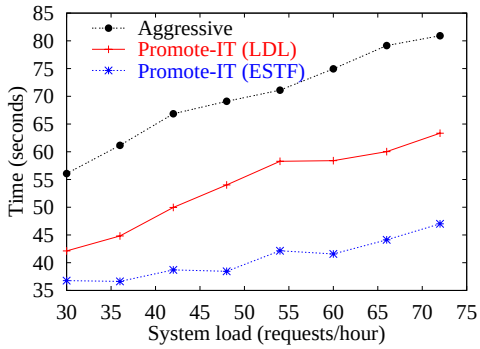
We show some performance results for the *fast jukebox* and the *slow jukebox* described in Table 9.2. The test set is the one described in Section 9.2: 30% long videos, 30% short videos, 30% music and 10% discrete media. Figures 9.7, 9.8, 9.8, and 9.10 show the results when the requests cannot be rejected (infinite deadline and maximum confirmation time). The first two figures show the performance of the system under load and medium load, while the other two show the performance also under high load.[4] Figures 9.11, and 9.12 show the corresponding results when

3 In Sections 5.4.2 and 7.3.1 we discussed the extensions to the original aggressive strategy, which differed much more from Promote-IT.

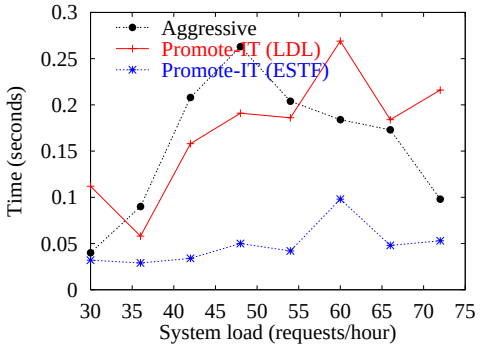
4 This is the same approach we used in Section 6.10 to compare the strategies of Promote-IT.



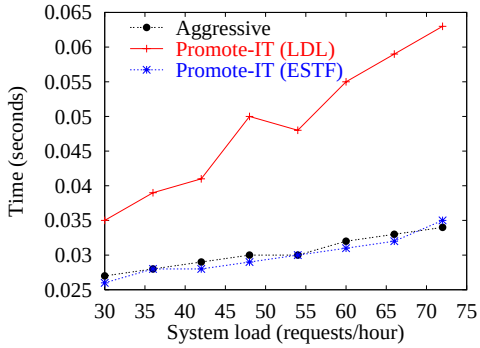
(a) Mean response time.



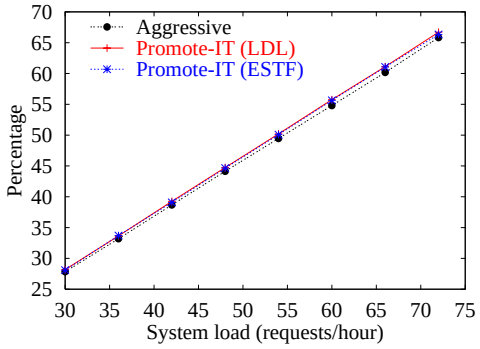
(b) Maximum response time for 90% of the requests.



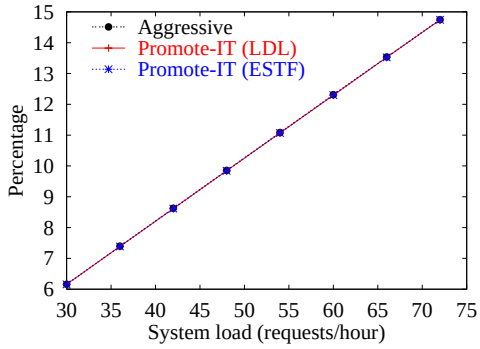
(c) Mean confirmation time.



(d) Mean computing time.

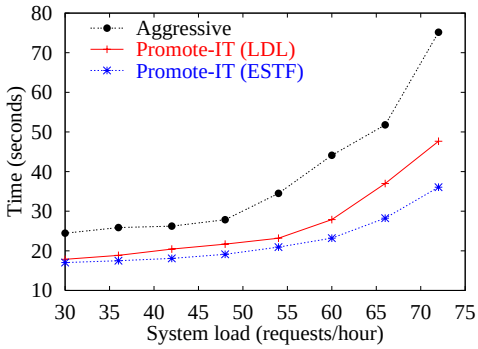


(e) Mean robot utilization.

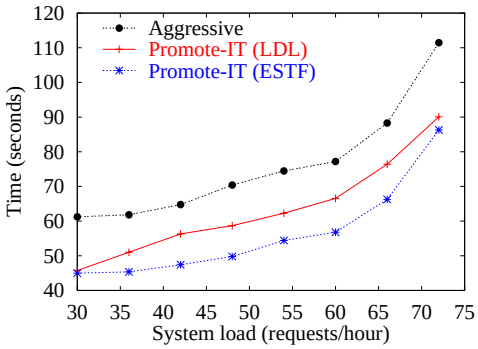


(f) Mean drive utilization.

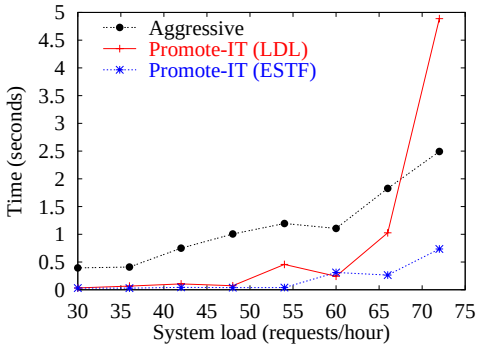
Figure 9.7: Uncoupled vs. coupled load and unload for the *fast jukebox* under low and medium load.



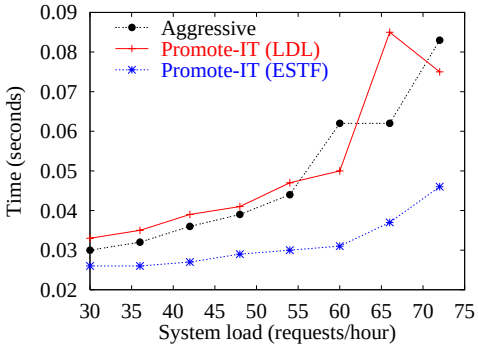
(a) Mean response time.



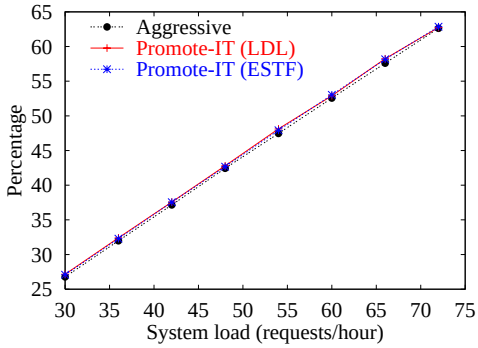
(b) Maximum response time for 90% of the requests.



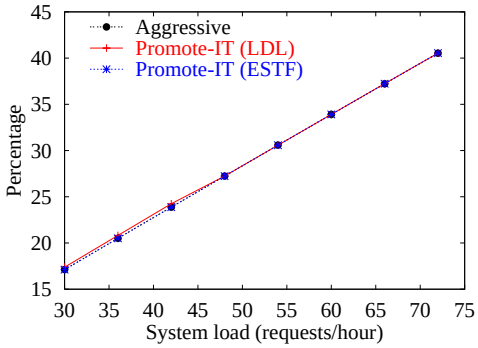
(c) Mean confirmation time.



(d) Mean computing time.

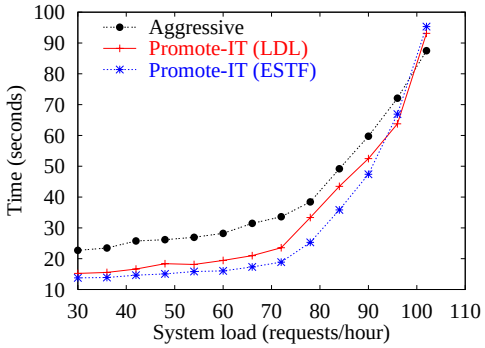


(e) Mean robot utilization.

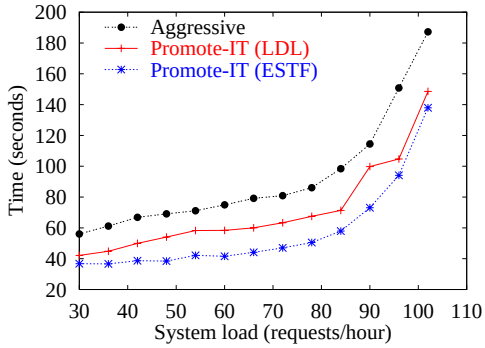


(f) Mean drive utilization.

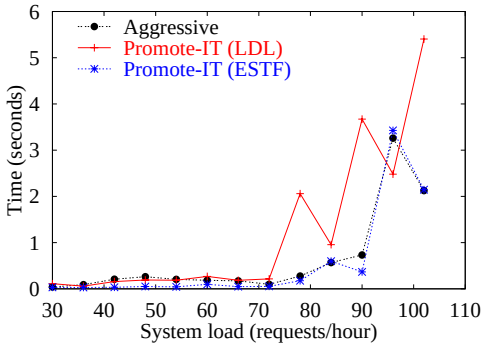
Figure 9.8: Uncoupled vs. coupled load and unload for the *slow jukebox* under low and medium load.



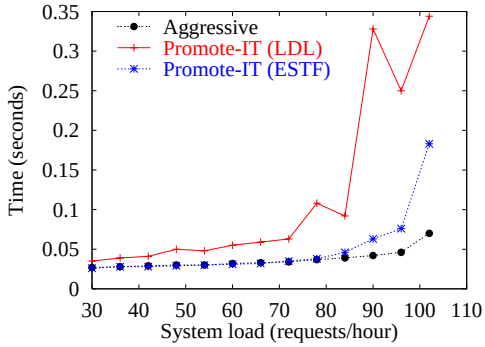
(a) Mean response time.



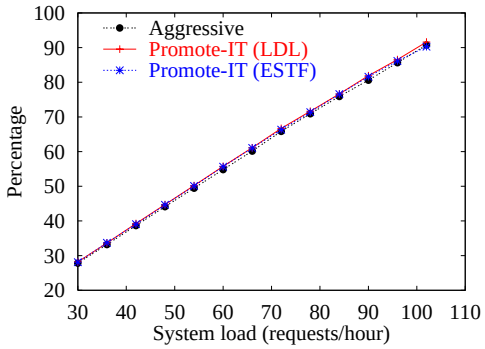
(b) Maximum response time for 90% of the requests.



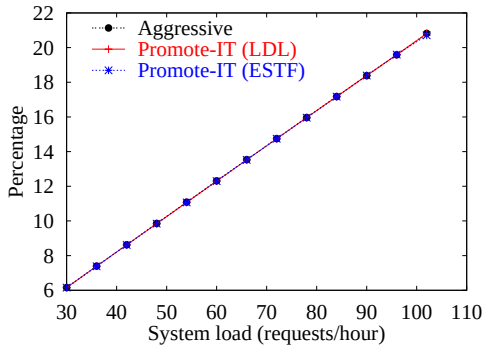
(c) Mean confirmation time.



(d) Mean computing time.

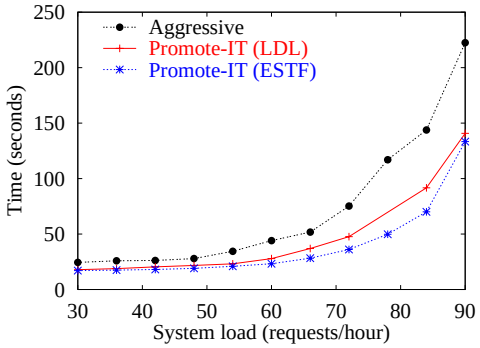


(e) Mean robot utilization.

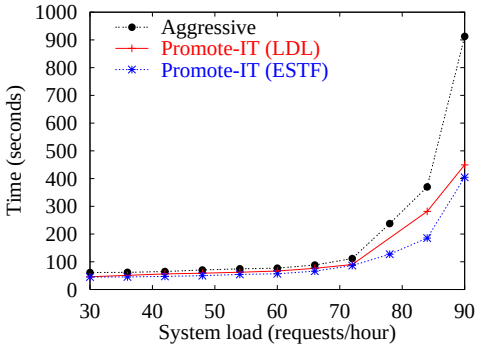


(f) Mean drive utilization.

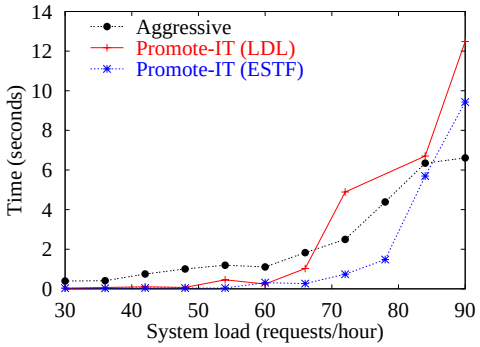
Figure 9.9: Uncoupled vs. coupled load and unload for the *fast jukebox* under high load.



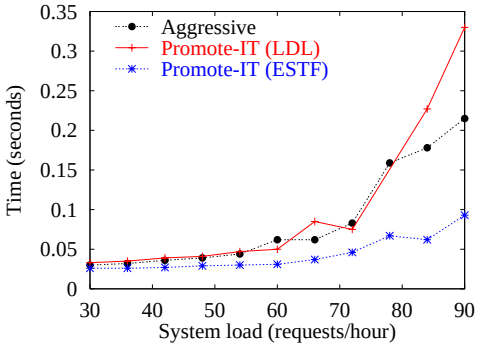
(a) Mean response time.



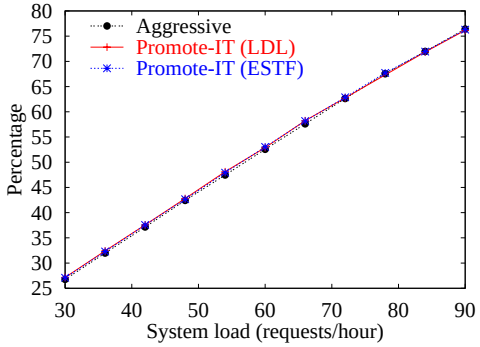
(b) Maximum response time for 90% of the requests.



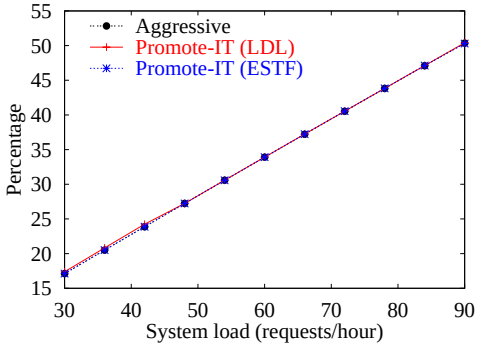
(c) Mean confirmation time.



(d) Mean computing time.

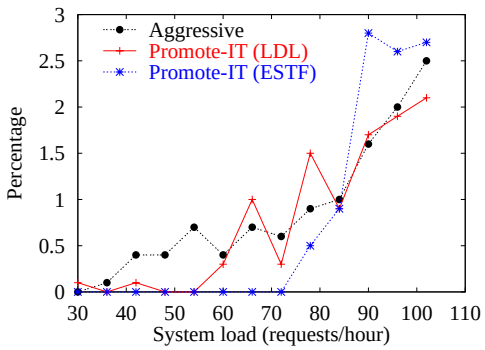


(e) Mean robot utilization.

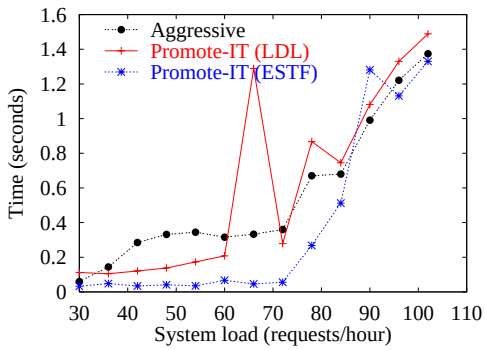


(f) Mean drive utilization.

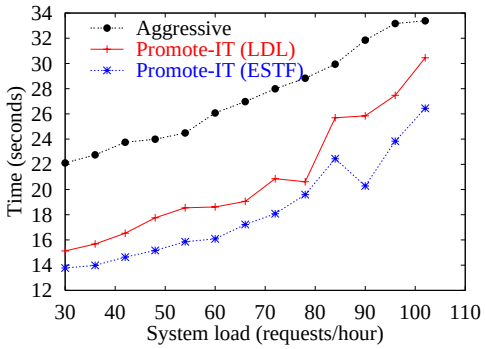
Figure 9.10: Uncoupled vs. coupled load and unload for the *slow jukebox* under high load.



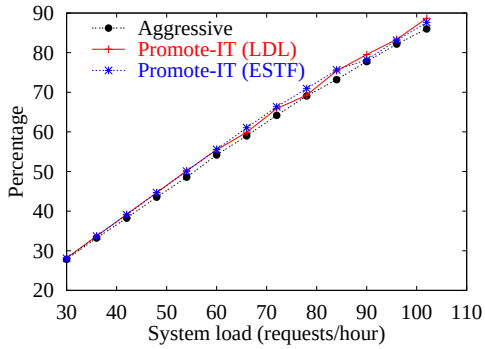
(a) Rejection ratio.



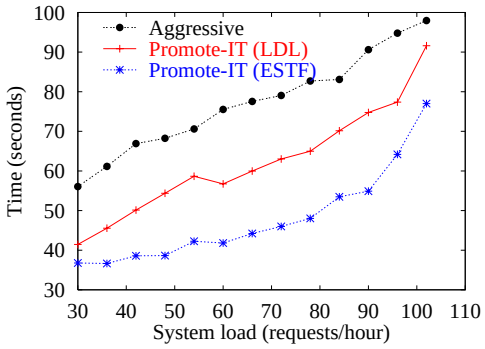
(b) Mean confirmation time.



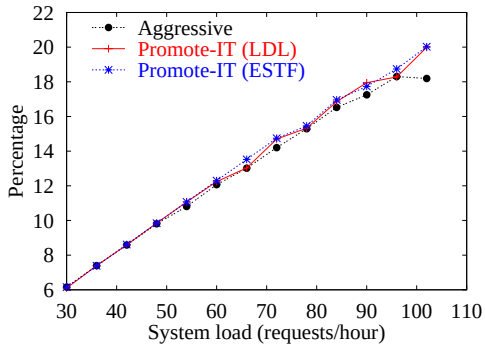
(c) Mean response time.



(d) Mean robot utilization.

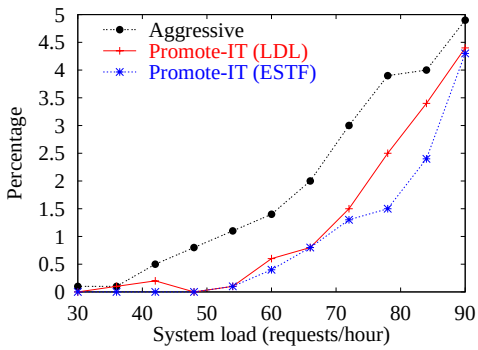


(e) Maximum response time for 90% of the requests.

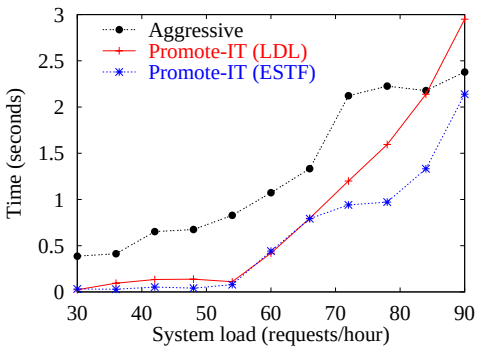


(f) Mean drive utilization.

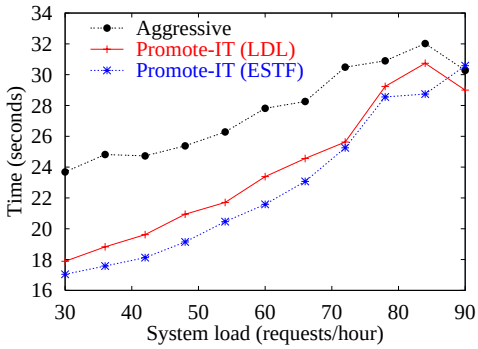
Figure 9.11: Uncoupled vs. coupled load and unload for the *fast jukebox* when rejecting requests.



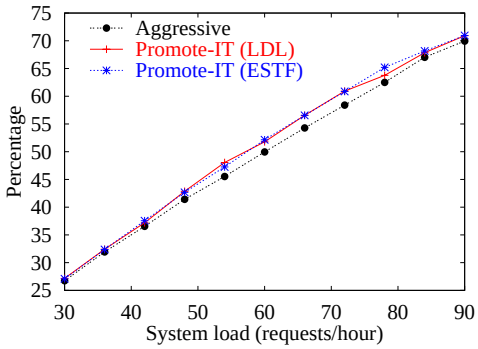
(a) Rejection ratio.



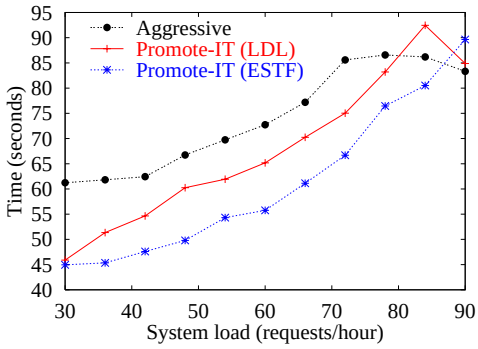
(b) Mean confirmation time.



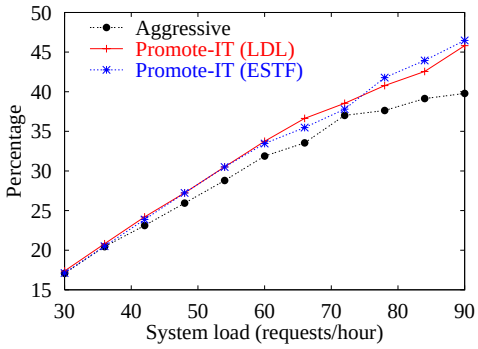
(c) Mean response time.



(d) Mean robot utilization.



(e) Maximum response time for 90% of the requests.



(f) Mean drive utilization.

Figure 9.12: Uncoupled vs. coupled load and unload for the *slow jukebox* when rejecting requests.

requests can be rejected. In this case, the deadline of every request is 5 minutes and the maximum confirmation time is 30 seconds.

When the requests cannot be rejected the plots in the figures show: (a) the mean response time, (b) the maximum response time for 90% of the requests, (c) the mean confirmation time, (d) the mean computing time, (e) the mean robot utilization, and (f) the mean drive utilization. When the requests can be rejected the plots in the figures show: (a) the rejection ratio, (b) the mean confirmation time, (c) the mean response time, (d) the mean robot utilization, (e) the maximum response time for 90% of the requests, and (f) the mean drive utilization.

When the system load is low and medium, Promote-IT provides shorter response times than Aggressive (see Figures 9.7(a), 9.8(a), 9.7(b) and 9.8(b)). Also when the system is able to reject requests during overload, Promote-IT provides shorter response times than Aggressive (see Figures 9.11 and 9.12). The rejection ratio of Promote-IT and Aggressive is similar (see Figures 9.11(a) and 9.12(a)).

However, when the system load is high and the robot is a clear bottleneck in the system, as is the case of *fast jukebox*, Aggressive has a better mean response time than Promote-IT (see Figure 9.9(a)). In this situation, the response time of Aggressive is similar to that of LDL, although Aggressive builds schedules Front-to-Back and LDL builds them Back-to-Front. However, Aggressive delays the last unload of a drive as much as possible, until the drive is needed again, which is the original goal of a Back-to-Front strategy. When the system load is low or medium, it is highly probable that at the time when a new request arrives there are idle resources. Therefore, delaying the unloads as much as Aggressive does affects the performance negatively. When the load is high it does not really matter, because there is no opportunity to unload the drives early anyhow.

When the robot is not a strong bottleneck, Promote-IT provides shorter response times than Aggressive, even under high system loads (see Figure 9.10). In this case unloading late is not beneficial: also ESTF performs better than LDL.

The maximum response time of the Aggressive strategy is lower and more stable than that of Promote-IT under high system loads. However, the maximum response time for 90% of the requests of Promote-IT is better (see Figures 9.9(b) and 9.10(b)).

The mean confirmation time of Promote-IT and Aggressive is similar. Furthermore, when the bottleneck of the system is in the use of the robot, the mean confirmation and computing time of ESTF resembles more that of Aggressive, than the corresponding values of LDL. The reason is that both ESTF and Aggressive build the schedules Front-to-Back, using a similar algorithm.

The mean drive and robot utilization of Aggressive is nearly identical to that of LDL when not rejecting requests. When rejecting requests, the drive and robot utilization depend on the rejection ratio.

9.5 Heuristic vs. Optimal Scheduling

The goal of this section is to evaluate the heuristics used by Promote-IT when compared against an optimal scheduler. We compare the optimal scheduler presented in Section 7.2, Promote-IT and the extended aggressive strategy. Due to the inability of the optimal scheduler to deal with high system loads and complex requests, the results shown in this section are restricted to quite simple request sets.

Figures 9.13 and 9.14 show the performance comparison between the heuristic and optimal schedulers using two jukebox architectures: the *fast jukebox* and the *slow jukebox*. These jukeboxes are different from the ones used in the previous sections. The test setup is described at the end of the section.

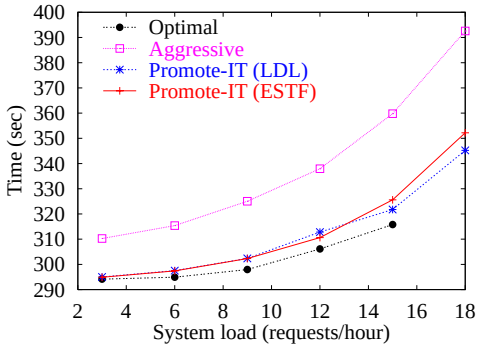
Figures 9.13 and 9.14 show the performance comparison between the heuristic and optimal schedulers. The plot of the optimal scheduler is interrupted early, because we could not obtain results for higher loads. Performing the runs shown already took several weeks for the optimal scheduler. As the system load increases, the number of request units that need to be rescheduled when a new request arrives also increases. Therefore, the optimal scheduler is unable to handle higher system loads, because its complexity is exponential in the number of request units to schedule.

The plots in the figures show: (a) the mean response time, (b) the maximum response time for 90% of the requests, (c) the mean computing time, (d) the mean robot utilization, and (e) the mean drive utilization. We do not show the confirmation time, because the optimal scheduler assumes that time does not pass during computation. As we explained in Section 7.2, without this assumption the optimal scheduler is unable to compute schedules, because by the time the computation to schedule a request has finished—which can easily take several weeks—the schedule computed should be completely irrelevant and invalid.

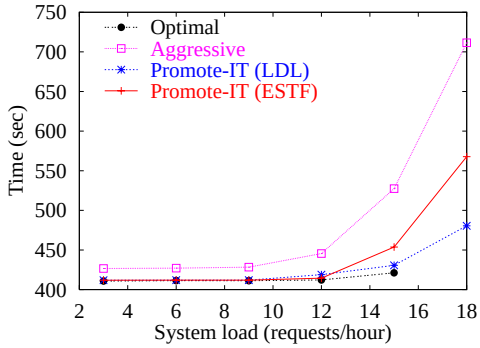
The response time provided by Promote-IT is near the optimal response time (see Figures 9.13(a), 9.14(a), 9.13(b) and 9.14(b)). Moreover, the difference in response time between Promote-IT and the optimal scheduler is smaller than the difference between Aggressive and Promote-IT. The plots indicate that the difference in response time between Promote-IT and the optimal is larger as the system load increases. Therefore, we regret that we cannot run the optimal scheduler with higher loads.

The computing time of the optimal scheduler increases exponentially when the load of the system increases, while the computation time of the heuristic schedulers is nearly constant (see Figures 9.13(c) and 9.14(c)). The computing times of the optimal scheduler are so high that the scheduler cannot be used in an on-line system.

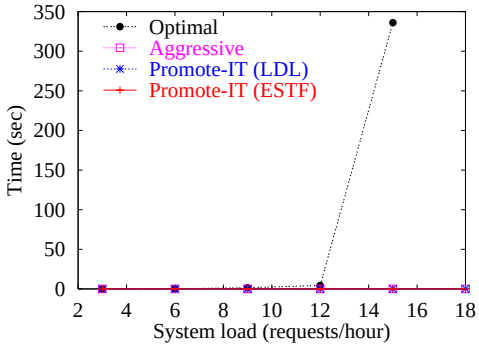
The mean robot and drive utilization of the optimal scheduler, Promote-IT and Aggressive is nearly identical. Moreover, we have observed that the optimal sched-



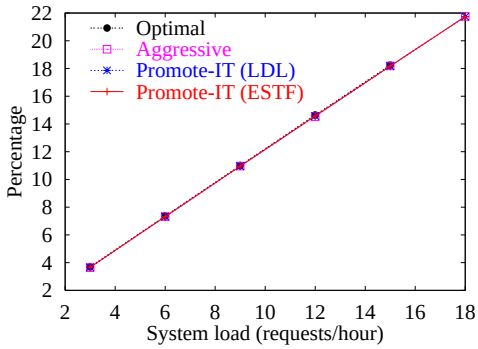
(a) Mean response time.



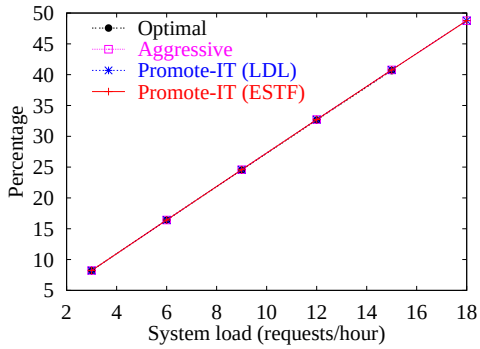
(b) Maximum response time for 90% of the requests.



(c) Mean computing time.

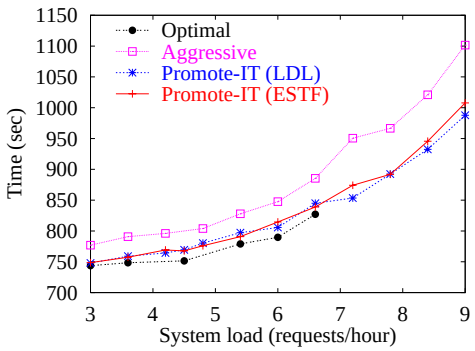


(d) Mean robot utilization.

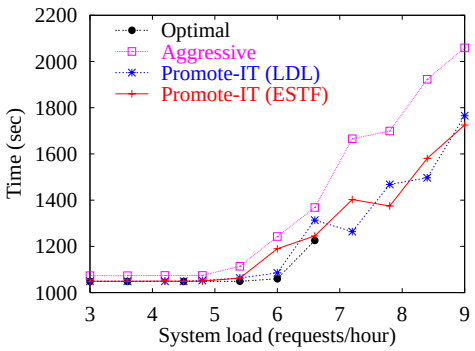


(e) Mean drive utilization.

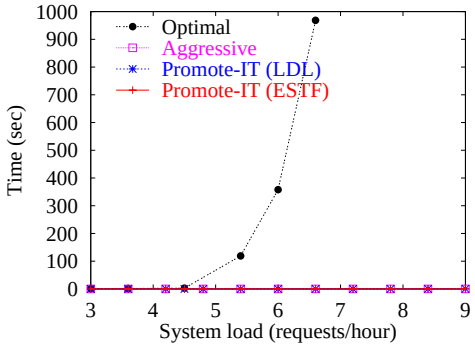
Figure 9.13: Heuristic vs. optimal scheduling for the jukebox architecture with the *fast jukebox*.



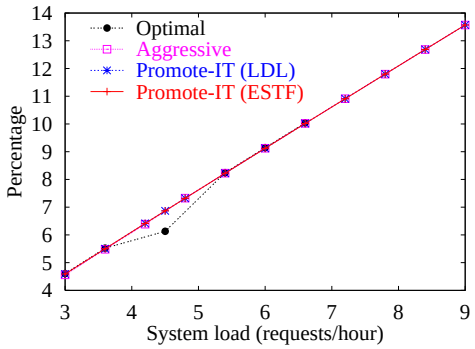
(a) Mean response time.



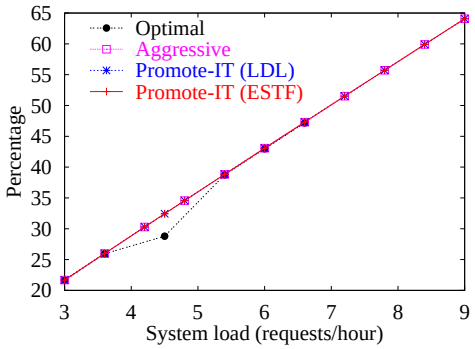
(b) Maximum response time for 90% of the requests.



(c) Mean computing time.



(d) Mean robot utilization.



(e) Mean drive utilization.

Figure 9.14: Heuristic vs. optimal scheduling for the jukebox architecture with the *slow jukebox*.

	Fast-drives jukebox	Slow-drives jukebox
Number of Robots	1	
Number of Drives	4 (identical)	
Load Time (seconds)	21.4–24.7	22.1–25.4
Unload Time (seconds)	14.2–17.6	23.1–26.5
Media Type	Single-layered DVD-ROM	
Drive Technology	CLV	
Transfer Speed (MBps)	6.45	2.44
Access time (seconds)	0.1	0.1

Table 9.3: Jukebox specification for evaluation of heuristic vs. optimal schedulers.

uler does not unload an RSM before all the data has been read in any of these runs and other tests we have performed, which are not showed here. This is an important result in favour of the minimum switching model, on which Promote-IT is based, because even if the optimal scheduler has the possibility of performing intermediate switches, it does not do so.

Test Setup

The request set consists of 200 ASAP requests for long videos. The optimal scheduler does not deal with the cache administration. Therefore, each request corresponds to a different video and the cache is empty at the beginning of the runs. Thus, there are no cache-hits.

The jukebox only contains long videos, which were generated in the same way as those described in Section 9.1. The bandwidth of the videos is uniformly distributed in the range from 1 to 8 Mbps. Their duration is uniformly distributed in the range from 15 minutes to 2.5 hours. However, the data in the jukebox is stored in single-layered DVDs.

When building the request sets we have to make a trade-off between keeping the number of request units per request low and having more than one request unit per RSM. The computational complexity of the optimal scheduler increases exponentially with the number of request units to schedule, so we should only split each file in a small number of request units. On the other hand, we want to give the optimal scheduler the possibility to switch an RSM without reading all the requested data from the RSM. Therefore, it is desirable to have more than one request unit per RSM. In the tests we show here, we chose to chop the files in request units of 2.5 GB. Thus, the number of request units per request is between 1 and 4 and the number of RSM involved is 1 or 2.

	FSBS	Extended Aggressive Strategy	Extended Conservative Strategy	Promote-IT	JEQS	Optimal
Flexibility: requests	++	++	++	++	−−	+
Flexibility: hardware	++	+	++	++	−	−
Response time	−−	+	−	++	−−	+++
Confirmation time	+	++	+	++	−	−−−
Computing time	++	++	++	++	+++	−−−
Deal with high load	+	++	−−	++	−−	−−−

Table 9.4: Summary of the performance comparison. The notation used is: excellent (+++), very good (++), good (+), bad (−), very bad (−−), and unusable (−−−).

We show simulation results for two jukebox architectures: the *fast-drives jukebox* (or *fast jukebox*) and the *slow-drives jukebox* (or slow drives). In both cases the jukebox is a smartDAX, as the one modelled in Chapter 4, with four identical drives. Table 9.3 shows the characteristics of each jukebox. In both cases the drives use CLV technology and the access time is constant. The requirement for constant access time is needed by the optimal scheduler.

9.6 Summary

Throughout this chapter we have shown that Promote-IT performs better than the other schedulers. However, the magnitude of the performance difference varies in each case. In this section we put the differences in context and compare all the schedulers among each other.

We evaluate the capacity of the schedulers to deal with flexible requests and hardware. We also evaluate the schedulers regarding the response time, confirmation time, computing time and the capacity to deal with high load. Table 9.4 summarizes the evaluation. The classification we assigned to the schedulers in the last four categories is the result of observing their performance in multiple test setups. Although, the classification is quite subjective and difficult to quantify, we believe that it reflects correctly the average performance of the schedulers.

We classify the schedulers that can deal with any combination of the request parameters defined for the requests (see Section 3.2) as ‘very good’. We classify the optimal scheduler only as ‘good’, because it cannot handle requests with deadlines

and maximum confirmation times. The optimal scheduler is also restricted in the complexity of the requests it can deal with, because of the computational complexity involved. We classify JEQS as 'very bad', because JEQS puts very strong restrictions on the requests: they must contain only one request unit, the data requested can only be continuous-media, and the bandwidth is limited by the bandwidth of the drives.

Promote-IT, FSBS, and the extended conservative strategy can deal with any type of jukebox hardware, thus, they are classified as 'very good'. The extended aggressive strategy restricts the functionality and scope of the robots. However, these are not very serious restrictions, because in most jukeboxes there is only one robot. Therefore, we classify the extended aggressive strategy as 'good'. JEQS and the optimal scheduler require that all the drives are identical. Additionally the optimal scheduler requires the access time to be constant. These restrictions are strong and, thus, we classify both schedulers as 'bad' regarding flexibility of hardware. We do not classify them as 'very bad', because at least they can deal with multiple drives and shared robots, which are jukebox characteristics that many of the schedulers we discussed in Section 2.2.3 cannot deal with.

The response time of the optimal scheduler is 'excellent', because it gives the best possible response time. The response time of Promote-IT is 'very good', because it responds better than all the other heuristic schedulers and is close to the optimal as far as we could measure. The response time of the extended aggressive strategy is 'good', but considerably worse than that of Promote-IT. JEQS and FSBS have 'very bad' response times, while the extended conservative strategy has 'bad' response times. As we have shown, in some cases FSBS is better that JEQS and in some other JEQS is better.

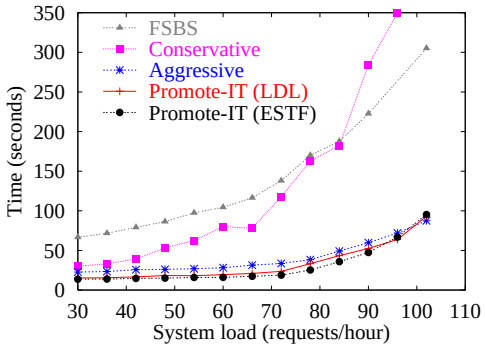
The confirmation time of Promote-IT and the extended aggressive strategy are 'very good' compared to those of the other strategies. The extended conservative strategy and FSBS perform worse than the former two schedulers, but still much better than JEQS. The extended conservative strategy and FSBS provide a confirmation within a reasonable time when being able to reject requests, while JEQS does not. Finally, the optimal scheduler cannot even be used on-line, and it requires computing the schedules assuming that time does not pass during the computation.

JEQS scores best when evaluating the computing time. Basically, it cannot be better because it just needs to evaluate a couple of formulae to decide if a request is schedulable. Thus, we evaluate JEQS as having an 'excellent' computing time. Promote-IT, FSBS, and the two extended strategies have 'very good' computing times. They compute the schedules in just few milliseconds. The optimal scheduler, instead, may require days or weeks to compute a single schedule. Therefore, we evaluate it as 'unusable' on this criterion.

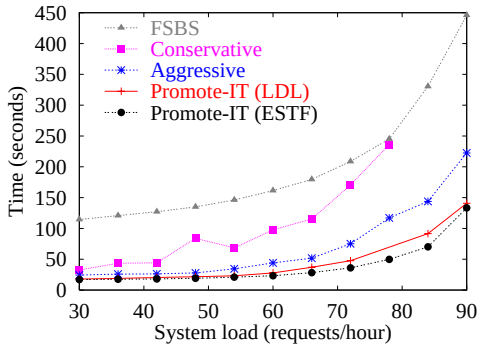
The optimal scheduler cannot deal with high system loads, because the computational complexity increases as a function of the system load and it becomes 'unusable'. JEQS and the extended conservative strategy score as 'very bad', because in both cases the response time increases very fast as the system load increases and the rejection ratio is high when the system is able to reject requests. The performance of the other three schedulers degrades in a gracious manner as the system load increases, but given its initial restrictions, FSBS performs comparatively worse than Promote-IT and the extended aggressive strategy.

Figure 9.15 shows in a graphical way the comparison among the aperiodic heuristic schedulers. It shows some results for the *fast jukebox* and the *slow jukebox* described in Table 9.2. The request set used is the same 30/30/30/10 request set used in Sections 9.2, 9.3.1 and 9.4.

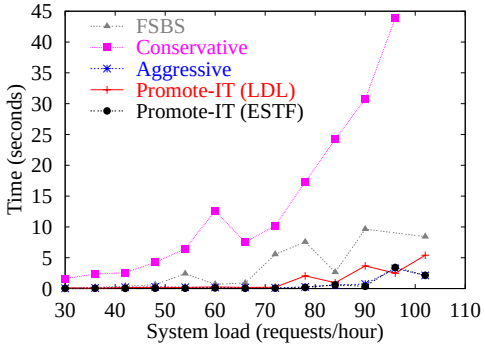
Figures 9.15(a) and 9.15(b) show that FSBS provides the worst response times among all heuristic aperiodic schedulers, but it can cope with higher loads better than the extended conservative strategy.



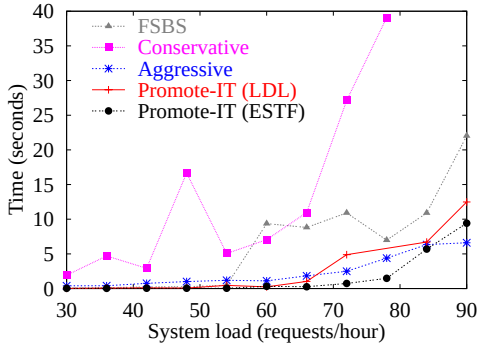
(a) Mean response time for *fast jukebox*.



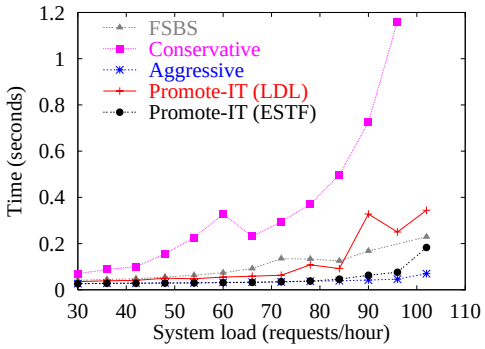
(b) Mean response time for *slow jukebox*.



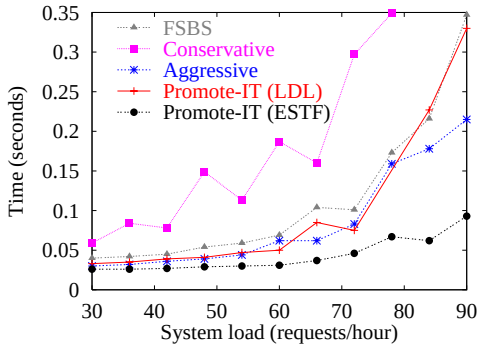
(c) Mean confirmation time for *fast jukebox*.



(d) Mean confirmation time for *slow jukebox*.



(e) Mean computing time for *fast jukebox*.



(f) Mean computing time for *slow jukebox*.

Figure 9.15: Comparison of all aperiodic heuristic schedulers using the *fast jukebox* and the *slow jukebox*.

Chapter 10

Conclusions

This dissertation addresses the problem of scheduling tertiary storage in order to provide real-time guarantees. It shows that tertiary storage can be used effectively in systems with real-time requirements, for instance in a hierarchical multimedia archive. However, careful scheduling is needed in order to provide those guarantees, to use the resources efficiently, and to provide short response times to the users.

Although the scheduling problem we deal with is specific—almost every interesting real-life scheduling problem is—we can derive some general scheduling principles, which can serve as a guideline for anyone dealing with a complex scheduling problem. The principles are:

Separate the scheduling-problem model from the scheduling algorithm.
It is important to formalize the scheduling problem, so that different heuristic and optimal algorithms can be compared. The formalization of the scheduling problem also helps to abstract from the particular application and look for similar scheduling problems in other environments. For example, by modelling our scheduling problem formally it is clearer to see that it has many common features with industrial production and logistics applications.

Model the scheduling problem accurately.
The scheduling problem model should not impose unnecessary coupling between operations, restrictions on the hardware model or restrictions on the request structure. These unnecessary restrictions are only eliminated by providing an accurate formal model of the scheduling problem. We show that coupling the unload and load operations into a single switch operation has a negative influence on the system performance, however, most existing jukebox schedulers enforce that coupling.

Model the hardware accurately.
The hardware model should provide good estimates of the operation times. Moreover, it should clearly identify the variables that influence the behaviour

of the hardware, so that different hardware can be easily modelled and the limitations of the scheduling problem can be defined. Additionally, such a hardware model allows us to identify other environments where the scheduler can also be used.

Identify the restrictions of the scheduling-problem model.
The restrictions on the hardware model and the operations should be clearly visible and appear in the formalization of the scheduling-problem model. Making the restrictions visible allows us to compare the different models and decide which model is more suitable for the envisioned usage scenarios. For example, we show that the dedicated-robots model is not suitable for scheduling a tertiary storage jukebox, but can be used in a manufacturing environment to schedule an automated storage/retrieval system.

Separate schedule building and dispatching.
Separating these two functions allows the schedule builder to concentrate on building feasible schedules, without optimizing the use of the resources. The dispatcher is responsible for utilizing the jukebox resources in an efficient manner. We use an early dispatcher that dispatches the tasks to the jukebox controller as early as possible. 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.

Define the heuristics used in the scheduling algorithm and the dispatcher.
The heuristics used in the algorithms should be defined clearly, so that they can be easily exchanged and compared. Also the implementation of the algorithms should be modular and the interfaces between the different components should be small and well defined. In our scheduling algorithms, we identify the heuristics for guessing the starting time of the requests, for sorting the jobs to schedule and for pruning the tree of resource assignments. We analyze and implement several options, and discuss their advantages and disadvantages.

Use a flexible developing and testing environment.
Use an environment that permits the test of schedulers under different usage scenarios and with different hardware. The environment should also make it possible to easily plug in other schedulers in order to evaluate different schedulers under the same conditions. The HMA permits the easy plug-in of different schedule builders and dispatchers. This has allowed us to evaluate different schedulers and multiple parameter combinations within each scheduler. Additionally, JukeTools has tools to easily simulate different hardware,

generate workloads, evaluate the output, and easily find design and implementation problems in the schedulers.

Hide 'real' and 'simulated' from the scheduler.
Use the same code for real and simulated users and real and simulated hardware. This facilitates a faster and more reliable move from the simulation phase to the production phase. The algorithms can be thoroughly tested offline in a simulation environment, while regularly testing if they do perform as expected in the real environment. JukeTools provides this functionality for jukebox schedulers.

This dissertation demonstrates the benefits of these principles by applying them to the creation of the flexible and efficient jukebox scheduler: Promote-IT. Promote-IT is based on the minimum switching model, which is a scheduling-problem model that only imposes restrictions on the utilization of the resources that are beneficial for the scheduler performance. The most important restriction of the model—reading all the requested data of an RSM (removable storage medium) once the RSM is in a drive—results in good overall system performance, because the number of media switches is minimized. The scheduling algorithm used by Promote-IT can use multiple strategies to sort and incorporate the jobs into a schedule. The heuristic for pruning the tree of resource assignments is efficient, both in performance and complexity. However, it can easily be replaced by another strategy if desired. The early dispatcher of Promote-IT can reorder the tasks for the robots in such a way that the jukebox resources are utilized effectively, while guaranteeing that all the deadlines are met. Last, but not least the implementation of Promote-IT is so flexible that we could implement some of the other schedulers evaluated in the dissertation by slightly modifying or parameterizing the algorithms of Promote-IT.

We are convinced that the hierarchical multimedia archive and Promote-IT will eventually be used in a production system to serve a large user population. Moreover, we also believe that Promote-IT can be used effectively in other environments, such as industrial production and logistics.

10.1 Directions for Future Research

In order to effectively use the HMA in a production environment, further analysis should be done to dimension the secondary storage level and determine the degree of distribution of the cache. Another topic that requires further analysis is the connection and data flow between tertiary and secondary storage. There are interesting open questions regarding the technology (e.g., fibber, SCSI), regarding the access to the devices (e.g., should we have an all-to-all connection or should the hard disks

and the jukebox drives be grouped into clusters?), and regarding the type of file systems to use in secondary storage (e.g., should we use mixed-media file systems, or should we have some file systems for continuous-media and some others for best-effort data?).

Another topic for further research is to make the schedulers fault-tolerant so that they can deal with hardware failures, which cause operations to take longer than estimated. Thanks to the unwanted misbehaviour of *our* jukebox, we noticed that Promote-IT can adapt to operations taking longer than scheduled in quite a graceful manner, especially when using Back-to-Front. Apart from recuperating from operations taking longer than estimated, a fault-tolerant scheduler should keep information about notoriously problematic RSM and estimate the time to operate on these RSM taking into account their defects. A way to implement this with our hardware model is to create a separate model for each problematic RSM and consider it a special type of RSM.

A promising direction for future work is to adapt the request structure and scheduler for a *just-in-time production* environment. However, some important questions need to be answered before embarking into such an adventure. Is it feasible to use the type of requests proposed in this dissertation in a production environment to schedule the retrieval of components from an *automated storage/retrieval system*? How should the buffer be implemented in such an environment? Is it cost effective to implement such a system?

Bibliography

[1] Atlas Copco AB. The case of the exploding CD-ROM record. `http://www.qedata.se/e_js.htm`.

[2] Demet Aksoy, Michael Franklin, and Stan Zdonik. Data staging for on-demand broadcast. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 571–580, September 2001.

[3] Virgilio Almeida, Azer Bestavros, Mark Crovella, and Adriana deOliveira. Characterizing reference locality in the WWW. In *Fourth International Conference on Parallel and Distributed Information Systems (PDIS '96)*, pages 92–107. IEEE Computer Society, December 1996.

[4] Kevin C. Almeroth and Mostafa H. Ammar. An alternative paradigm for scalable on-demand applications: Evaluating and deploying the interactive multimedia jukebox. *Knowledge and Data Engineering*, 11(4):658–672, 1999.

[5] 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.

[6] Martin W. P. Savelsbergh Ann Campbell, Lloyd Clarke. Inventory routing in practice. In D. Viego P. Toth, editor, *The Vehicle Routing Problem*, SIAM monographs on discrete mathematics and applications, pages 309–330. SIAM, 2002.

[7] Asaca. Specification of the asaca jukeboxes. `http://www.asaca.com`.

[8] Norbert Ascheuer, Matteo Fischetti, and Martin Grötschel. Solving the asymmetric travelling salesman problem with time windows by branch-and-cut. *Math. Program.*, 90(3):475–506, 2000.

[9] Mikhail J. Atallah, editor. *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.

[10] 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.

[11] Jeanne Behnke and Alla Lake. Eosdis: Archive and distribution systems in the year 2000. In *Proceedings of the 8th NASA Goddard Conference on Mass Storage Systems and Technologies and 17th IEEE Symposium on Mass Storage Systems*, April 2000.

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

[13] Janek Blazewicz, Jan Karel Lenstra, and Alexander H.G. Rinnooy Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete Appl. Math.*, 5:11–24, 1983.

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

[15] Peter Bosch and Sape J. Mullender. Cut-and-paste file-systems: Integrating simulators and file-systems. In *USENIX Annual Technical Conference*, pages 307–318, 1996.

[16] Jihad Boulos and Kinji Ono. Continuous data management on tape-based tertiary storage systems. Technical report, NACSIS, November 1997.

[17] Jihad Boulos and Kinji Ono. Continuous data management on tape-based tertiary storage systems. In *Proceedings of the 5th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services*, volume 1483 of *Lecture Notes in Computer Science*, pages 290–301. Springer Verlag, Berlin, September 1998.

[18] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM (1)*, pages 126–134, 1999.

[19] David W. Brubeck and Lawrence A. Rowe. Hierarchical storage management in a distributed VoD system. *IEEE Multimedia*, 3(3):37–47, 1996.

[20] 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.

[21] 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.

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

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

[24] John L. Cole and Merrit E. Jones. The IEEE storage system standards working group overview and status. In *Proceedings of the 14th IEEE Symposium on Mass Storage Systems*. IEEE, September 1995.

[25] Simon de Groot. Scheduling real-time streams for heterogeneous storage media. Master's thesis, Department of Computer Science, University of Twente, 2002.

[26] Peter J. Denning. Effects of scheduling on file memory operations. In *AFIPS Spring Joint Computer Conference*, pages 9–21, April 1967.

[27] Disc. Specification of the disc jukeboxes. `http://www.disc-storage.com`.

[28] Standardizing Information and Communication Systems ECMA. Data interchange on read-only 120 mm optical data disks (CD-ROM). Standard ECMA 130, ECMA, June 1996.

[29] Craig Federighi and Lawrence A. Rowe. Distributed hierarchical storage manager for a video-on-demand system. In *Storage and Retrieval for Image and Video Databases (SPIE)*, pages 185–197, February 1994.

[30] Fujitsu. Storage management overview. White Paper, January 1998.

[31] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series. Addison-Wesley, Reading, Mass., 1995.

[32] Robert Geist and Stephen Daniel. A continuum of disk scheduling algorithms. *ACM Transactions on Computer Systems (TOCS)*, 5(1):77–92, 1987.

[33] Jim Gemmell, Harrick M. Vin, Dilip D. Kandlur, P. Venkat Rangan, and Lawrence A. Rowe. Multimedia storage servers: A tutorial and survey. *IEEE Computer*, 28(5):40–49, November 1995.

[34] Costas Georgiadis, Peter Triantafillou, and Christos Faloutsos. Fundamentals of scheduling and performance of video tape libraries. *Multimedia Tools and Applications*, 18(2):137–158, 2001.

[35] Shahram Ghandeharizadeh and Cyrus Shahabi. On multimedia repositories, personal computers, and hierarchical storage systems. In *Proceedings of th ACM Multimedia Conference*, 1994.

[36] 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.

[37] Ron L. Graham, Eugene L. Lawler, Jan Karel Lenstra, and Alexander H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling theory: a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[38] Mikell P. Groover. *Automation, Production Systems and Computer Integrated Manufacturing*. Prentice-Hall, second edition, 2001.

[39] Robert Grossman, Xiao Qin, and W. Xu. An architecture for a scalable, high-performance digital library. In *Proceedings of the 14th IEEE Symposium on Mass Storage Systems*, September 1995.

[40] Karsten Halse. *Modeling and solving complex vehicle routing problems*. PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, 1992.

[41] 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.

[42] Hewlett-Packard. Specification of the HP jukeboxes. `http://www.hewlett-packard.com/go/optical`.

[43] Bruce K. Hillyer, Rajeev Rastogi, and Avi Silberschatz. Storage technology: Status, issues, and opportunities. Unpublished technical report.

[44] Bruce K. Hillyer and Avi Silberschatz. On the modeling and performance characteristics of a serpentine tape drive. In *Proc. of the 1996 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 170–179, May 1996.

[45] 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.

[46] 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.

[47] Hitachi. Large capacity optical disc video recording format ”blu-ray disc” established, February 2002.

[48] W.A. Horn. Some simple scheduling algorithms. *Naval Res. Logist. Quart.*, 21:177–185, 1974.

[49] Petros Ioannou, Anastasios Chassiakos, Hossein Jula, and Ricardo Unlaub. Trucks in metropolitan areas with adjacent ports. Technical Report FWHA/CA/OR-2002/14, University of Southern California, California State University at Long Beach, 2002.

[50] James R. Jackson. Scheduling a production line to minimize maximum tardiness. Management Science Research Project 43, UCLA, January 1955.

[51] 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.

[52] Theodore Johnson. Queuing models of tertiary storage. In *Proc. of the 5th NASA Goddard Mass Storage Systems and Technologies Conference*, September 1996.

[53] Theodore Johnson and Ethan L. Miller. Benchmarking tape system performance. In *Proc. of Joint NASA/IEEE Mass Storage Systems Symposium*, March 1998.

[54] Theodore Johnson and Ethan L. Miller. Performance measurements of tertiary storage devices. In *Proc. of 24th International Conference on Very Large Data Bases*, pages 50–61, August 1998.

[55] JVC. Specification of the JVC jukeboxes. `http://www.jcvpro.co.uk`.

[56] S.-W. Kim, Sung-Jo Kim, Tae Il Jeong, and S. W. Yoo. The optimal retrieval start times of media objects for the multimedia presentation. *Information and Software Technology*, 43(4):219–229, March 2001.

[57] John T. Kohl, Carl Staelin, and Michael Stonebraker. HighLight: Using a log-structured file system for tertiary storage management. In *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 435–447, San Diego, CA, USA, 25–29 1993.

[58] Niklas Kohl. *Exact methods for Time Constrained Routing and Related Scheduling Problems*. PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, 1995.

[59] Kubota. Specification of the kubota jukeboxes. `http://www.kubota.co.jp`.

[60] Jesper Larsen. *Parallelization of the Vehicle Routing Problem with Time Windows*. PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, 1999.

[61] 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.

[62] 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.

[63] Siu-Wah Lau and John C.S. Lui. Scheduling and data layout policies for a near-line multimedia storage architecture. *Multimedia Systems*, 5:310–323, September 1997.

[64] Eugene L. Lawler, Jan Karel Lenstra, Alexander H. G. Rinnooy Kan, and David B. Shmoys, editors. *The traveling salesman problem.* Wiley and Sons, New York, 1985.

[65] Jan Karel Lenstra, Alexander H.G. Rinnooy Kan, and Peter Brucker. Complexity of machine scheduling problems. *Ann. Discrete Math.*, 1:343–362, 1977.

[66] Maria Eva Lijding, Peter Bosch, and Sape J. Mullender. Tertiary storage for nemesis. Technical report, University of Twente, October 1999.

[67] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Sun Microsystems, second edition, 1999.

[68] 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.

[69] Michael T. LoBue. Surveying today's most popular storage interfaces. *Computer*, 35(12):48–55, December 2002.

[70] S. Mahajan, B. Rao, and Brett A. Peters. A retrieval sequencing heuristic for miniload end-of-aisle automated storage/retrieval systems. *International Journal of Production Research*, 36(6):1715–1731, 1998.

[71] Oge Marques and Borke Furht. Issues in designing contemporary video database systems. In *Proceedings of the IASTED International Conference on Internet and Multimedia Systems and Applications*, pages 198–211, October 1999.

[72] Ethan Leo Miller. *Storage Hierarchy Management for Scientific Computing*. PhD thesis, University of California at Berkeley, 1995.

[73] ChanHo Moon and Hyunchul Kang. Heuristic algorithms for I/O scheduling for efficient retrieval of large objects from tertiary storage. In *Proceedings of the Australiasian Database Conference*, pages 145–152. IEEE, February 2001.

[74] Sachin More and Alok Choudhary. Scheduling queries on tape-resident data. In *Proceeding of the European Conference on Parallel Computing*, 2000.

[75] James W. Mott-Smith. The jaquith archive server. Technical Report CSD-92-701, University of California, Berkeley, September 1992.

[76] Internet Surveys NUA. How many online? `http://www.nua.com/surveys/how_many_online`, 2002.

[77] OSI. *ISO-9660:1988 - Information Processing - Volume and file structure on CD-ROM for information interchange*.

[78] HweeHwa Pang. Tertiary storage in multimedia systems: Staging or direct access? *ACM Multimedia Systems Journal*, 5(6):386–399, November 1997.

[79] Byung Chun Park, Edward H. Frazelle, and John A. White. Buffer sizing models for end-of-aisle order picking systems. *IEE Transactions*, 31:31–38, 1999.

[80] PCTechGuide. `http://www.pctechguide.com`.

[81] Gilles Pesant, Michel Gendreau, Jean-Yves Potvin, and Jean-Marc Rousseau. An exact constraint logic programming algorithm for the travelling salesman with time windows. *Transportation science*, 32(1), February 1998.

[82] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[83] Michael Pinedo. *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, 1995.

[84] Pioneer. Specification of the pioneer jukeboxes. `http://www.pioneerelectronics.com`.

[85] Viswanath Poosala. Zipf's law. Technical Report 900 837 0750, Bell Labs.

[86] Sunil Prabhakar, Divyakant Agrawal, and Amr El Abbadi. Impact of media exchanges in robotic storage libraries. Technical Report TRCS97-07, University of California, Santa Barbara, 1997.

[87] Sunil Prabhakar, Divyakant Agrawal, Amr El Abbadi, and Ambuj Singh. Efficient I/O scheduling in tertiary libraries. Technical Report TRCS96-26, University of California, Santa Barbara, 1996.

[88] 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.

[89] Sanjay Ranade. *Mass Storage Technologies*. Meckler, 1991.

[90] The Real-time for Java Expert Group. *The Real-time Specification for Java*, June 2000.

[91] A. L. Narasimha Reddy and Jim Wyllie. Disk scheduling in a multimedia I/O system. In *Proceedings of the 1st ACM international Conference on Multimedia*, pages 225–233. ACM Press, 1993.

[92] Dickon Reed and Robin Fairbairns. Nemesis : The kernel - overview. Technical report, University of Cambridge, May 1997.

[93] Bernd Reiner and Karl Hahn. Tertiary storage support for large-scale multidimensional array database management systems. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, August 2002.

[94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[95] Koichi Sadashige. Optical recording and recordable DVD overview. In *Proceedings of the 8th Nasa Goddard Conference On Mass Storage Systems And Technologies and 17th IEEE Symposium On Mass Storage Systems*, April 2000.

[96] Ingolf Sander. Fluorescent multilayer optical data storage. White Paper.

[97] Sunita Sarawagi. Query processing in tertiary memory databases. In *Very Large Databases (VLDB) Journal*, pages 585–596, 1995.

[98] Martin W. P. Savelsbergh. Local search for routing problems with time windows. *Annals of Operations Research*, 4:285–305, 1986.

[99] Martin W. P. Savelsbergh. An efficient implementation of local search for constrained routing problems. *European Journal on Operations Research*, 47:75–85, 1990.

[100] Sanjeev Setia, Ophir Frieder, and David Grossman. *Data Storage Technology*, chapter 2.

[101] Vijnan Shastri, V. Rajaraman, H.S. Jamadagni, P. Venkat Rangan, and Srihari Sampath-Kumar. Design issues and caching strategies for CD-ROM-based multimedia storage. In *Multimedia Computing and Networking*, volume 2667 of *SPIE Proceedings*, pages 30–47, 1996.

[102] DAX Archiving Solutions. Specification of the DAX jukeboxes. `http://www.smartdax.com`.

[103] J.M. Spivey. *The Z Notation*. C.A.R. Hoare series editor. Prentice Hall, second edition, 1992.

[104] ASM Mass Storage Systems. Specification of the ASM jukeboxes. `http://www.asm-jukebox.de`.

[105] ASM Mass Storage Systems. Switch time of the 1400 series. Personal communications with Cornelia Reinhards.

[106] Danny Teaff, Dick Watson, and Bob Coyne. The architecture of the High Performance Storage System (HPSS). In *Proc. of the Fourth NASA GSFS Conference on Mass Storage Systems and Technologies*, 1995.

[107] Peter Triantafillou and Ioannis 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.

[108] Peter Triantafillou and Thomas Papadakis. On-demand data elevation in hierarchical multimedia storage servers. In *Proc. of 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 226–235, 1997.

[109] Shiao-Li Tsao, Yueh-Min Huang, Chia-Chin Lin, Shiang-Chung Liou, and Chien-Wen Huang. A novel data placement scheme on optical discs for near-vod servers. In *Proceedings of the 4th International Workshop on Interactive Distributed Multimedia System and Telcommunication Services*, volume 1309 of *Lecture Notes in Computer Science*, pages 133–142. Springer, 1997.

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

[111] Rodney van Meter and Minxi Gao. Latency management in storage systems. In *Proceedings of the 4th Symposium on Operating System Design & Implementation*, 2000.

[112] W3C. *XSL Transformations (XSLT) 1.0*, November 1999.

[113] W3C. *Extensible Markup Language (XML) 1.0*, second edition, October 2000.

[114] Mark Wallace, Stefano Novello, and Joachim Schimpf. Eclipse: A platform for constraint logic programming, 1997.

[115] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proccedings of the Inernational Workshop on Memory Management*, September 1995.

[116] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling algorithms for modern disk drives. In *Proceedings of the 1994 conference on Measurement and Modeling of Computer Systems*, pages 241–251. ACM Press, 1994.

[117] George Kingsley Zipf. Relative frequency as a determinant of phonetic change. reprinted from Harvard Studies in Classical Philiology , XL, 1929.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-01

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-04

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-05

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-06

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-07

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-08

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-09

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Schedulere Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Progam Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

Biography

María Eva Lijding was born in Buenos Aires, Argentina, on May 14, 1970. From 1993 to 1996 she held a research grant for students from the University of Buenos Aires. In May 1996 she graduated in Computer Science at the University of Buenos Aires. Her master's thesis entitled 'A New Internal Routing Protocol' won the third prize of the Latin-American contest for master's works in computer science organized by CLEI and UNESCO.

After graduation she worked for the National Institute of Industrial Technology (INTI) and the University of Buenos Aires, both in Argentina. In August 1997 she got a postgraduate grant from the YPF Foundation to research and study abroad. In the context of this grant she stayed nine month at the Computer Architecture department of the Polytechnic University of Catalonia (Spain) and three month at the Distributed and Embedded Systems department of the University of Twente. In the latter she has further pursued a Ph.D.