TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Scheduler optimization in real-time distributed databases

Citation for published version (APA):
Bodlaender, M. P. (1999). *Scheduler optimization in real-time distributed databases*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Mathematics and Computer Science]. Technische Universiteit Eindhoven. https://doi.org/10.6100/IR520890

DOI:
10.6100/IR520890

Document status and date:
Published: 01/01/1999

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 04. Oct. 2023

Scheduler Optimization in Real-Time Distributed Databases

M.P. Bodlaender

Scheduler Optimization in Real-Time Distributed Databases

Copyright ©1999 by Maarten Peter Bodlaender, Eindhoven, The Netherlands.

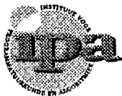
All rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced, in any form or by any means, including but not limited to photocopy, photograph, magnetic or other record, without prior agreement and written permission of the author.

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Bodlaender, Maarten P.

Scheduler optimization in real-time distributed databases /
by Maarten P. Bodlaender. -
Eindhoven : Eindhoven University of Technology, 1999.
Proefschrift. - ISBN 90-386-0681-8

NUGI 855
Subject headings: distributed algorithms / scheduling / databases /
CR Subject classification (1998) : C.2.4, C.3, D.4.1



This thesis has been carried out under the auspices of the Institute for Programming Research and Algorithmics (IPA).

Printed by University Press Facilities, Eindhoven
Cover design by Ben Mobach and drawing by M.K. Bodlaender-Groote

SCHEDULER OPTIMIZATION IN REAL-TIME DISTRIBUTED DATABASES

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR AAN DE TECHNISCHE UNIVERSITEIT EINDHOVEN, OP GEZAG VAN DE RECTOR MAGNIFICUS, PROF.DR. M. REM, VOOR EEN COMMISSIE AANGEWEZEN DOOR HET COLLEGE VOOR PROMOTIES IN HET OPENBAAR TE VERDEDIGEN OP DINSDAG 11 MEI 1999 OM 16.00 UUR

DOOR

MAARTEN PETER BODLAENDER

GEBOREN TE EDE

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.dipl.ing. D.K. Hammer
en
prof.dr. E.H.L. Aarts

Copromotor:
dr. P.D.V. van der Stok

Voorwoord

Ongeveer vier jaar geleden nodigde Peter van der Stok mij uit voor een oriënterend gesprek. Dat was het begin van een heel prettige en vruchtbare samenwerking. Het eindresultaat van deze samenwerking houdt u in uw handen. Discussies met Peter hebben mij altijd enorm geïnspireerd. Hij had altijd tijd voor me, en bijna altijd geduld met me. Daar komt nog bij dat het STW project waarin mijn promotie-onderzoek plaatsvond perfect geregeld was. Mijn dank aan Jean Pierre Veen van de Stichting Technische Wetenschappen voor een soepele begeleiding van ons project. Samen met Jan van der Wal, Ad Aerts en mede-promovendus Simone Sassen hadden we regelmatig projectbesprekingen. Die hielden de druk op de ketel en zorgden ervoor dat onze ideeën kritisch bekeken werden. Simone en ik vulden elkaar goed aan, en hoofdstuk zes is daar een mooi voorbeeld van. Bedankt Simone.

De leden van het gebruikerscommittee hebben veel invloed gehad op de richting van mijn onderzoek. Met name wil ik Jan van der Meer bedanken. Hij bracht me in contact met de database mensen binnen Ericsson. Ik ben veel verschuldigd aan Claes Wikström, een van de ontwerpers van de Mnesia database. Een bezoek aan Ericsson Ellemtel en lange discussies op het strand van Kauaii hebben veel bijgedragen aan de ideeën in dit proefschrift. Bart Nieuwenhuis en Wim Jonker van KPN Research gaven een beschrijving van de telecomsystemen van de toekomst die goed aansloot bij de ideeën die ik bij Ericsson opdeed. Hoofdstuk vier was nooit tot stand gekomen zonder hun hulp. Ian Willers van het CERN was altijd kritisch tijdens de gebruikersbijeenkomsten, wat op dat moment even moeilijk is, maar later erg nuttig blijkt. Toen het schrijven van mijn proefschrift moeilijk ging, bracht een bezoek aan het CERN nieuwe inspiratie. Helaas namen Pascale Minet van INRIA, en A. Vreven van Rabofacet alleen in het begin deel aan ons gebruikerscommittee. In die eerste fase hebben ze bijdragen geleverd die terug te vinden zijn in dit proefschrift. Mijn dank aan professor Sang Son van de universiteit van Virginia, wiens heldere kijk op temporele consistentie mij inspireerde tot het schrijven van hoofdstuk acht.

Ook bedank ik mijn promotoren Dieter Hammer en Emile Aarts, en de leden van mijn leescommissie voor de tijd en moeite die ze hebben genomen. Zonder hun commentaar was de uiteindelijke versie heel wat minder mooi geworden. Verder wil ik graag Paul van Gorp, Isabelle Reymen, Huub van de Wetering en Wieger Wesselink noemen. Naast vriendschap gaven ze mij nuchter commentaar en advies. Ook wil ik alle deelnemers aan de wekelijkse DEDOS bijeenkomsten bedanken voor zeer leerzame, maar vaak zware middagen. Ik ben heel, heel veel verschuldigd aan Marijke Bodlaender-Groote en Klaus Bodlaender. Soms besef ik dat onvoldoende. Ten slotte bedank ik al mijn collega's op de Technische Universiteit Eindhoven voor de fijne tijd, en in bijzonder 雪梅. Als zij glimlacht zingt mijn hart.

Contents

Voorwoord **i**

1 Introduction **1**
- 1.1 What are real-time, distributed databases? 1
- 1.2 Database architecture 3
- 1.3 Typical real-time systems that need databases 5
- 1.4 Research objective 7
- 1.5 Related work 8
- 1.6 Organization of this thesis 12

2 Specification of scheduling problems **15**
- 2.1 Specification of design problems 15
- 2.2 Real-time database scheduling problems 18
  - 2.2.1 Schedulers 18
  - 2.2.2 Platform description 21
  - 2.2.3 Data characteristics 22
  - 2.2.4 Transaction characteristics 24
  - 2.2.5 Objective functions 30
- 2.3 Conclusions 31

3 Design issues **33**
- 3.1 Targeted problem specification 33
  - 3.1.1 Candidates for detailed specifications 34
  - 3.1.2 Example: statistical information used for optimization 35
- 3.2 Concurrency 39
- 3.3 Scheduler overhead 41
  - 3.3.1 Overhead reduction techniques 42
- 3.4 The overhead/concurrency tradeoff 43
- 3.5 Available information versus time & quality 44
  - 3.5.1 On-line scheduling decisions 45
  - 3.5.2 Off-line scheduling decisions 45

4 Scheduling in low-conflict environments **49**
- 4.1 Environment study 49
- 4.2 Specification 50
- 4.3 Design 51
- 4.4 Algorithm 53

4.5 Correctness . . . 56
4.6 Performance analysis of OCC-light . . . 58
4.7 Performance analysis of Mnesia's two phase locking scheduler . . . 59
4.8 Test results . . . 59
4.9 Conclusions . . . 63

5 Scheduling in high-conflict environments **65**
5.1 Environment study . . . 65
5.2 Specification . . . 66
5.3 Design . . . 66
5.4 The DOCC-BF algorithm . . . 70
5.4.1 System architecture . . . 71
5.4.2 Scheduler description . . . 72
5.5 Correctness . . . 77
5.5.1 Deadlock freedom . . . 80
5.5.2 Life lock freedom . . . 81
5.6 Optimizing the algorithm . . . 81
5.7 Performance analysis . . . 84
5.8 Test results . . . 84
5.8.1 Experiment 1: local transactions . . . 85
5.8.2 Experiment 2: remote data access, large conflict probability . . . 86
5.9 Conclusions . . . 87

6 Predictable scheduling **89**
6.1 Specification . . . 89
6.2 A family of scheduling algorithms . . . 90
6.2.1 SQSL-soft . . . 91
6.2.2 SQSL-firm . . . 93
6.2.3 SQSL-MLF . . . 94
6.2.4 SQSL-MLF with queue-skipping . . . 95
6.2.5 Scheduler discussion . . . 96
6.3 Performance analysis . . . 97
6.4 Comparison with simulation results . . . 105
6.4.1 Simulation compared to fitting . . . 105
6.4.2 Comparison of the schedulers . . . 106
6.5 Conclusions . . . 108

7 Off-line database scheduling **109**
7.1 Environment study . . . 109
7.2 Specification . . . 114
7.3 Scheduler design and algorithms . . . 115
7.3.1 Conflict detector . . . 115
7.3.2 TPC' Constructor . . . 120
7.4 Conclusions . . . 120

8 Temporal consistency in hard real-time databases **121**
8.1 Environment study 121
8.2 Specification 127
8.3 Scheduler design 129
8.4 Algorithm 132
8.5 Complete example: air traffic control 134
8.6 Conclusions 137

9 Conclusions **139**
9.1 Performance 139
9.2 Targeted design 140
9.3 Analysis and design 141
9.4 Off-line database scheduling 141
9.5 Temporal consistency 142
9.6 A final word 142

A List of symbols **143**

B Pseudo-code conventions **145**
B.1 Expressions 145
B.2 Statements 146
B.3 Concurrent programs 147

C Basic algorithms **149**
C.1 Locking 149
C.2 Wait-die deadlock breaking 150
C.3 Two phase locking 151
C.4 Two phase commit 151

Index **161**

Summary **165**

Samenvatting **167**

Curriculum Vitæ **169**

Chapter 1

Introduction

Computers play a strong supporting role in our society. Governments and large organizations depend on them for administration. They are the core of our banking system. Communication networks are fully digital, and controlled by dedicated computers. In factories, computers are used extensively for process control, resource allocation and routing.

One of the most important tasks of computer systems is to collect, store and provide information. Large systems can contain several terabytes of data, which are accessed and modified by multiple parties. These data are often vital to the organization that owns them. Data should never be lost, should be consistent, and they should be quickly accessible by authorized parties. Databases provide this functionality. They offer a structural way to solve problems that are related to data. In general, all databases allow storage and retrieval of data. Specific databases can solve more data-related problems.

Systems that interact directly with the outside world often have strict timing requirements. A response to an outside world event should be given within a certain time-span. Such systems are called real-time systems. When a real-time system accesses data from a database, the database response should also be given within a certain time-span. Databases that can satisfy timing requirements are called real-time databases.

This thesis deals with the optimization of real-time, distributed database schedulers, to improve performance and predictability.

1.1 What are real-time, distributed databases?

Databases are building blocks for bigger systems. Different types of databases exist. What services they should offer depends entirely on the application(s) that use them. All databases offer the following basic functionality.

Structural storage of data. All data stored in the database is handled in a uniform way, and data can only be accessed through a well-defined interface: applications issue transactions to the database. A transaction consists of a number of read and write operations on data, and some computation. The actual organization of data within the database is hidden from the applications that access the database.

Consistency. The data stored in the database should be consistent. Consistency requirements can be arbitrary predicates over the entire data set. For example, in financial administrations it is required that the balance-sheet of a company always totals to zero. Execution of a transaction should not

destroy this consistency.

The execution of a transaction can be temporarily stalled. For example, the transaction waits for a user response, or waits until a data access is completed. During this time, the processor is available to other transactions. Allowing transactions to execute in an interleaved fashion utilizes the processor more efficiently. In distributed systems interleaving transactions is even more important, since more than one processor is available on which transactions can be executed. To utilize all processors efficiently, an interleaved execution of transactions is required. To maintain consistency when transactions can execute in an interleaved fashion, the database should support the following functionality.

Concurrency control. If applications access the database concurrently and update overlapping sets of data, interferences between different transactions can lead to unexpected and unwanted results. Data is no longer consistent, even if the individual transactions preserve consistency. Concurrency control ensures that transactions only access the database at the same time, if they do not interfere. For example, if transactions access different parts of the database, their executions do not interfere with each other.

Traditionally, databases are implemented on single processor computers. As there are limits to the power of single processor systems, databases are implemented on other hardware architectures with more processors. This enlarges the capacity of the database, allowing for more than one database access at the same time, but it introduces coordination problems between the different processors. We discern the following two multiple-processor hardware architectures: a shared-memory architecture, and a distributed architecture. In shared-memory architectures, processors communicate with each other through a memory that is directly accessible by all processors. In distributed architectures, processors communicate through a network. Databases that use the specific features of these architectures are called shared-memory databases and distributed databases. Burdening transaction programmers with the precise details of the hardware architecture is unwanted, therefore the following functionality is supported by databases.

Distribution transparency. Applications that access the database need to have no knowledge about the underlying hardware. Assignment of transactions to processors and the fetching of accessed data is handled by the database.

In real-time applications the computer interacts with an outside world that is constantly changing. Real-time applications often store data that describe the state of the outside world. Since the outside world changes, these data are no longer accurate after some time, and lose their value to the application. Usually, whether certain data is useful or not depends on its age: the older it gets, the less useful it becomes for the applications that access it. When an application reacts to the outside world, it has to do so within a bounded time-span. The reaction should not come too early, and should not be too late. If the application issues a transaction to the database as part of this reaction, the transaction needs to be completed in bounded time. Real-time databases offer the following functionality.

Transaction timeliness. Applications that issue transactions to the database can determine in which interval a transaction should be completed. Three types of requirements are recognized: hard, firm, and soft real-time requirements. If the database does not satisfy a hard real-time requirement, the

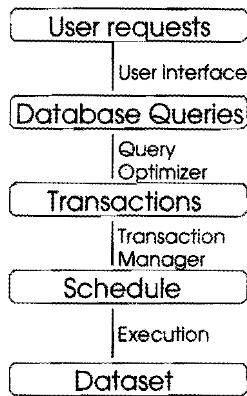


Figure 1.1: DBMS

entire system fails. If the database does not satisfy a firm real-time requirement, the corresponding transaction is worthless and can be discarded. If a soft real-time requirement is not satisfied by the database, the value of the corresponding transaction decreases, but does not instantly become zero.

Data accuracy. Applications that access the database can rely on the fact that data stored in the database accurately describes the outside world.

Furthermore, databases can offer reliability guarantees, backup mechanisms in case of failures, security mechanisms, and active response to certain database states. Many other extensions to databases exist, for example adding structures to the database that facilitate queries and searches on the data set.

The thesis focuses on real-time scheduling of transactions on multi-processor databases. The databases that are considered support structural storage of data, consistency and concurrency control and distribution transparency. Special attention is given to transaction timeliness and data accuracy. Since we are interested in real-time systems with strong timing requirements, we limit ourselves to main-memory databases. This means that all data is stored in primary memory, and system crashes lead to loss of all information. Disk-based databases have not been investigated.

1.2 Database architecture

A database consists of several subsystems, and a set of stored data, see figure 1.1. All subsystems together form the *database management system* (DBMS). The stored data is called the *dataset*. This is often called the database, but to avoid confusion with the complete system, we define a more precise term.

Database management systems consist of a *user interface* that converts *user requests* into database queries. The user interface can be a powerful command language like SQL. Each user request is a statement in this language. Alternatively, the interface can be less flexible, like a form that must be filled by a user. Even simple pressure sensors can be the interface to the database. In this case, a

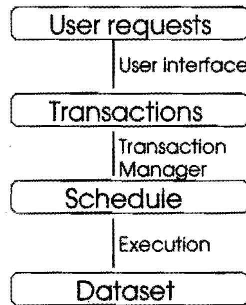


Figure 1.2: FLAT DBMS

pressure change represents a user request.

Database queries are generated by the user interface, and are usually specified in a relational or algebraic language. The *query optimizer* manipulates the evaluation-order of database queries to minimize computation cost. Query optimization is a research field in itself, that has drawn considerable attention over the years [5, 45, 33]. The query optimizer produces executable *transactions*. A transaction is composed of read and write actions on the dataset, together with some internal computation.

An example of the importance of query optimization is the following query. It is executed in a database where the names and ages of several persons are stored. Its task is to collect the names of all persons that are older than three years, and whose name starts with 'X'. This consists of two algebraic operations: selection of persons, using predicate 'age > 3' ($\sigma_{\text{age}>3}$), and selection of persons, using predicate name $= X...$ ($\sigma_{\text{name}=X...}$). Suppose thousand persons in the database satisfy age > 3, and ten persons satisfy name $= X....$ Then $\sigma_{\text{age}>3}$ followed by $\sigma_{\text{name}=X...}$ has an intermediate result of thousand persons, while $\sigma_{\text{name}=X...}$ followed by $\sigma_{\text{age}>3}$ has an intermediate result of only ten persons. The second order is more efficient.

The *transaction manager* determines when and how transactions are executed. For concurrency control, it produces a *schedule* that consists of a partial order on the (interleaved) execution of transactions. To guarantee transaction timeliness, the schedule is enhanced with start-times for each transaction execution. Data accuracy is guaranteed by adding a set of data-refresh transactions to the total set of transactions that is executed. Furthermore, the execution of transactions can be enhanced with mechanisms that implement various other database services, like reliability or security. Finally, the transactions are *executed* on the dataset according to the schedule.

Flat database management systems

Real-time systems mostly work with pre-defined responses to external events. There is no need to generate new transactions, each time the database is accessed. Instead of generating new transactions using database queries, transactions are taken from a set of pre-defined transactions. These transactions can be defined and optimized beforehand. This eliminates the need for a generic query language and query optimizer.

Replacement of generic database queries and the query optimizer by a set of pre-defined transactions leads to the *flat database management system*, as shown in figure 1.2. Besides the fact that

it eliminates the overhead of the query optimizer, it provides other benefits as well. When the set of pre-defined transactions is limited, the transaction manager can be optimized to handle those transactions more efficiently. By decreasing the generality of the transaction manager, overhead caused by superfluous mechanisms is avoided.

In this thesis we investigate flat DBMSes, and focus on the scheduling aspects of the transaction manager. To accomplish this, we define the required functionality of the transaction manager, and then propose solutions that maximize the overall performance.

1.3 Typical real-time systems that need databases

Real-time databases range from full-blown systems that offer all the features described above, to lightweight databases that offer little more than speed and structural storage. We studied the characteristics of six application domains, and we investigated what kind of real-time databases are used in those environments. Below, we briefly discuss some of the results of our investigations. Information from private communications with people that work in these areas, and information from papers that describe these applications [43, 23, 25, 97] is used in these descriptions.

Telecommunication. A private automatic branch exchange (PABX) contains a database in which amongst others the signatures of PABX users are stored. A signature contains information about the different networks accessible to the user and the correspondence between a short user defined number and the physical telephone number within a network. This is typically represented by one kilobyte of information. Most of the operations are look-up only, so execution times are short.

Telecom operators have millions of customers, and real-time access to the database is required whenever a customer makes a call. The arrival pattern of customer calls is chaotic, and is well-described by a Poisson process. The nature of these arrival processes is such that occasionally more customers arrive with a request than can be handled by the system at that time. Therefore, a 100% service guarantee can never be given.

As telecom systems are expanding rapidly, for example by the growth of on-line computing, arrival rates of as much at ten-thousand transactions per second are expected in the future. Right now, one database handles about fifty transactions each second. Less frequent and less time-critical are updates that are performed for system maintenance.

The large amount of information stored, and the high rate of database accesses suggest that a single processor architecture does not suffice. A distributed architecture is preferable, due to the reduced costs and the ability to scale the system by adding more machines. Another major benefit of distributed systems is the increased availability and reliability of the system, aspects that are very important in the telecom environment.

High energy physics. High energy physics (HEP) experiments generate vast amounts of data in a very short time-span. The interaction of two particles in a magnetic field generates a set of particles following tracks with a certain curvature. The passage of the particles is measured by a number of detectors. The type of the interaction can be determined from the spatial reconstruction of the particle tracks from the detector data.

The data received from the sensors is large, several megabytes for one measurement. Reconstruction transactions extract the relevant information from measured data. This reduces the necessary memory requirements, such that the extracted data can be stored in a database. The database should ensure that the reconstruction transactions can handle the arrival rate of measurements. Main-memory

acts as a buffer, in which new measurements are stored until a processor becomes available to execute a reconstruction transaction. Hard deadlines on the reconstruction transactions are determined by the interaction rate, the number of processors, and the amount of storage space available for intermediate results.

Execution times of reconstruction transactions are large, because they access a large amount of data, and complex physical equations have to be solved. Since all relevant data is combined into one large data item, reconstruction transactions access only a few data items. Sensor data arrives at the system at regular intervals, so the maximum load of the system can be computed before hand. 100% service guarantees can be given, if the system is built correctly. Periodic transactions with soft deadlines display statistics on the accuracy of the measurements over the last few hours or visualize spatial reconstructions of particularly interesting interactions.

Container port. Ships loaded with containers are scheduled for arrival in time slots during which the quay is at their disposal. Containers, stored at predetermined locations in the ship's hold, are transported to specified storage locations on the quay. Automatic guided vehicles (AGV) transport the containers over a predetermined route from the crane to the specified storage location or vice-versa. Transactions access the database at three different time scales: 1) planning of the ships' arrivals (days), 2) planning of the storage of containers once the ship arrived (hours), and 3) the almost continuous routing and collision avoidance of the AGVs (minutes, seconds). Especially transactions of type 3) have strict real-time requirements.

The transactions that are needed to route AGVs have firm deadlines. If a deadline is missed, it means that the AGV has to stop to ensure that no collisions take place. Since these AGVs weigh over ten-thousand kilo, stopping and starting is slow. Therefore, it is important that deadlines are almost never missed. Data items accessed by transactions mostly consists of positional information, which can be described in a few kilobytes. The executing times of routing transactions are short, as most routing decisions are pre-computed.

Routing transactions are essentially periodic in nature, since AGV tracks are divided into sections. However, when multiple AGVs request routing information from the database at the same time, there are peak-moments. Still, the maximum number of concurrent database accesses is bounded.

Automatic teller machines. Automatic teller machines (ATMs) are linked to a number of central computers connected to a database with information on clients and their accounts. Requests for information about accounts are sent with a high rate from the ATMs to the central site. Bounds on the response times of these requests are firm. On the other hand, updates of the account are handled with soft deadlines. The data stored for one account is limited to a few kilobytes, and no complex operations have to be carried out by the transactions. Therefore, the execution time of a transaction is short. The arrival pattern of customer requests is essentially a chaotic process, so no 100% service guarantees can be given.

Actual developments indicate that there is a growing market for accounts with a continuous real-time access, especially in connection with stock transfer. Consistency and reliability are important requirements for these databases.

Financial trading applications. Financial trading applications (FTA) follow market developments and try to recognize trends, such that profitable trading becomes possible. An *instrument* is an entity with a price that can be traded. Typical examples of instruments are equities, bonds, options, and

commodities like pork bellies. Real-time data about instruments are received from various stock exchanges and brokerage houses, and are stored in temporal databases. Historical data about instruments are used by traders to identify trends in the market, such that future prices can be extrapolated.

Updates to the database arrive at a high rate, typically more than five-hundred transactions arrive each second from a ticker. A single update transaction affects only a few data items. The data stored in the database are read by trend-recognition transactions. Trend-recognition transactions are complex and access often more than hundred instances of the same data item. For example, the price of one instrument at different points in time can be used for prediction.

The system load of financial trading applications will increase in the future. Markets grow and traders want to combine information from multiple markets in their information systems. Trend recognition will become increasingly complex, as the financial theories are further developed.

Command and control systems. Command and control systems (C& C) monitor the environment with a set of sensors. The data received from the sensors are correlated to get an integrated view of the real-world. A decision support system uses this view to decide on actions that have to be taken by the system.

Databases are useful to store and update the sensor data and the correlation data that are derived from the sensor data. The database contains information about a set of monitored objects. Sensor data for one object arrives periodically, with different periods for different sensors. The arrival process of new objects that have to be monitored is chaotic, due to the open nature of the environment. The total arrival rate of sensor data is very high, depending on the area that is monitored by the C&C system. As much as thousand updates every second have to be expected and transient overloads cannot be avoided. The size of sensor information on one monitored object is about half a kilobyte. Correlation of sensor data is complex, requiring advanced correlation techniques. To predict future states, multiple instances of the same data item can be used for extrapolation.

All of the systems described above require databases that can handle a very high rate of database accesses. Furthermore, each separate database access should be handled in real-time, with minimal delay. The systems differ fundamentally in a number of important characteristics. Real-time requirements in the HEP system are hard, while deadlines in the AGV routing system are firm. PABX, ATM, FTA, and C&C systems can still use the results of late transactions, so their real-time requirements are soft. The transaction arrival patterns of ATM, PABX, FTA, and C&C systems are very chaotic in nature, so transient overloads are unavoidable. The HEP system and the AGV routing system have predictable, bounded transaction arrival patterns. Transactions in ATM, PABX, and the AGV routing system have low data requirements, while HEP sensor data is large, and the financial applications deal with large sets of data. C&C systems consist of sensor updates that have low data requirements, and correlations that deal with larger sets of data. Correctness requirements differ; ATM, HEP, FTA, and the AGV routing system place high demands on consistency. The PABX application allows that transactions occasionally read inconsistent data, but never write inconsistent values to the database. C&C systems relax functional consistency in favor of temporal consistency: a global view of the environment at one point in time has to be constructed.

1.4 Research objective

We focus on the scheduling aspect of the transaction manager in a flat database management system architecture. The scheduler within the transaction manager actually has two tasks: concurrency con-

trol and real-time scheduling. Both tasks are difficult by themselves, as is exemplified by the large volume of research in these areas (see section 1.5). Combining them increases the complexity of the resulting scheduling problem. Generated schedules have to satisfy real-time requirements that restrict start times and finish times of transactions, and consistency requirements that restrict the functional behavior of the system. The fact that the scheduler is often executed on the same platform as the transactions further complicates the scheduling problem.

This thesis does not discuss allocation of transactions to processors nor allocation of data items to sites. These subjects have a strong impact on the performance of the system, but are beyond the scope of this thesis. Data allocation, and the accompanying transaction allocation is usually treated in combination with query optimization [5]. In this thesis we focus on the scheduling aspects, and assume that the allocation is fixed and that data items are not replicated. Furthermore, we abstract from communication scheduling in distributed systems by assuming a fixed communication delay.

1.5 Related work

The field of real-time (distributed) databases stems from the combination of real-time scheduling and database concurrency control. We discuss results from both areas that directly relate to real-time distributed databases. Readers that are interested in real-time systems can start with the excellent survey that appeared in [87] or look up the books [76, 49]. Databases are treated in depth in [51, 73]. In the discussion below, we treat the specification of scheduling problems separately from the algorithms that have been designed to solve them.

Problem specification

Lawler [56] introduced the $\alpha \mid \beta \mid \gamma$ notation to describe real-time machine scheduling problems. The meaning of this notation is as follows. A set of tasks with certain characteristics β is executed on a platform with characteristics α. The quality of solutions is measured with performance criterion γ. A large class of scheduling problems, not limited to machine scheduling, can be described by varying the contents of $\alpha,\ \beta,\ \gamma$.

Lawler's notation clearly identifies problem characteristics. The notation is open, new descriptions can be added to the α, β, and γ fields. Over the years, a large vocabulary has been created, in which most real-time scheduling problems can be expressed. This categorization brings some order in a very diverse research area. No comparable categorization exists for database concurrency control problems.

Database scheduling problems are quite similar to real-time scheduling problems, in most aspects. A set of transactions (comparable to tasks) is executed on a platform. Traditionally, solutions should optimize the transaction throughput of the database. Contrary to the problems that Lawler's notation was intended for, database scheduling problems are necessarily dynamic and on-line, because insufficient scheduling information is available off-line. Furthermore, the requirements that are placed on the execution of transactions are quite different from the requirements that are placed on tasks. Esrawan, Gray, Lorie, and Traiger [30] introduced the *transaction concept* in 1976: a task (transaction) that reads values from a consistent database, writes consistent values back to the database.

When transactions execute concurrently, interferences between their executions can destroy the database consistency. The notion of *serializable* schedules that Esrawan, et.al. [30] introduced can be used to guarantee that interferences do not occur. It was further formalized by Papadimitriou [75] and Vidyasankar [100]. Serializability is summarized as follows. The effect of a serializable schedule is

functionally equivalent to a sequential execution of the same set of transactions. The serializability notion is quite intuitive. Sequential executions cannot interfere with each other. Therefore, if the result of a schedule is equivalent to the result of a sequential schedule, no interferences occur. Different notions of serializability exist, depending on the equivalence-relation that is used. Vidyasankar related several notions of serializability [100].

All *view-serializable* schedules have to be view-equivalent to a sequential schedule. Two schedules S and S' are view-equivalent if each transaction t reads the same values in both S and S' (note that in this definition is assumed that each schedule starts with a write-only transaction that writes all data items, and ends with a read-only transaction that reads all data items). Papadimitriou showed that the problem of deciding whether a given schedule is view-serializable is NP-complete [75].

Conflict-serializability is more restrictive. A conflict between two transactions p and q occurs, if both access the same data item, in conflicting modes (at least one of the transactions performs a write). A conflict has a direction: if p performs the access before q, the conflict is ordered $p \rightarrow q$. Two schedules S and S' are conflict-equivalent if they contain the same set of conflicts, and the conflicts are ordered in the same direction. A schedule is conflict-serializable if it is conflict-equivalent to a sequential schedule. Concurrency control mechanisms usually recognize a subset of all conflict-serializable schedules.

Epsilon serializability [78] was introduced by Ramamritham and Pu in an attempt to simplify the scheduling problem by weakening the serializability requirement. Interferences between transactions are allowed, as long as the resulting inconsistencies are bounded to a specified margin. When the margins are wide, the scheduling problem is easy.

Kuo and Mok introduced view-Δ-similarity [54] in another attempt to avoid the NP-complete scheduling problem. A "similarity bound" is specified for each data item. Normal serializability is replaced by a weaker variant. Transactions can concurrently write data items, as long as writes occur within an interval shorter than the similarity bounds of the data items. The rationale for this approach is that instances created at roughly the same time, are "similar" enough to be treated as the same.

Audsley, Burns, Richardson, and Wellings propose to drop serializability altogether [8, 9] for data that describes a continuously changing real-world. Instead, a notion of *temporal consistency* is introduced: a data item is consistent, if it has been measured recently in the real-world. This notion of consistency we call *data-based temporal consistency* [46, 102, 2]. This notion is generalized to *transaction-based temporal consistency* [20, 19] in this thesis. Datta and Viguier present similar ideas [28], but arrive at their result by amalgating temporal [89] and real-time databases.

If we describe the concurrency control scheduling problem in terms of Lawler's notation, serializability would qualify as a task-characteristic. It is a generalization of the shared resources characteristic, which is often used to describe mutual exclusion requirements in real-time scheduling problems. Buckley and Silberschatz [22] describe a database where transactions follow fixed database access patterns, and show that efficient schedulers can use this information. Agrawal, El Abbadi, and Jeffers [3, 4] study the effects of relaxing transaction atomicity. Atomicity is also relaxed in altruistic locking, which has been introduced by Salem, Garcia-Molina, and Shands [81].

Traditionally, database systems were disk-based (for textbooks, see Prad and Adamski [77], or Korth and Silberschatz [51]). Since the advent of large and cheap memories, main-memory databases are a viable alternative [52, 82, 16]. Another important platform characteristic is the basic architecture, i.e. centralized databases [52], or distributed databases [93, 13, 31, 92, 86].

Performance criteria are poorly developed for the concurrency control scheduling problem. Papadimitriou evaluated the scheduler performance by counting the number of different schedules that it can generate. The underlying idea is that a scheduler which can generate a large number of schedules, can generate a schedule that utilizes the available processors effectively. Already in 1979 Papadim-

itriou noted [75] that this performance criterion is not concrete enough for more practical applications. It completely neglects the platform and task characteristics of the scheduling problem! A scheduler that achieves a 100% utilization on one processor is perfect in a centralized system, but can be poor in a distributed system.

The poor performance criteria have led to discussions regarding the relative performance of different concurrency control algorithms [65, 93, 69]. Comparison of schedulers is mostly based on simulations of the concurrency control algorithms [41, 92, 40, 72]. Other articles [90, 105, 69, 104] used stochastic analysis of specific systems to arrive at their results. In these specific systems, performance criteria like transaction-throughput can be used.

Unfortunately, conclusions based on simulations hold for very specific applications, with a specific platform, and specific transaction characteristics. Conclusions based on stochastic analysis can be a bit more general, if the analysis is parameterized, but still only cover a small part of the general scheduling problem. The performance of the scheduler has to be evaluated for each application area. Real-time databases use performance criteria from real-time scheduling. Two of the most commonly used criteria are the average response time [65, 90, 105], and the percentage of transactions that miss their deadlines [55, 57, 72].

Scheduling results

Liu and Layland [59] introduced two of the most successful real-time scheduling algorithms, earliest deadline first scheduling (EDF) and rate-monotonic scheduling (RMS). Both provide optimal solutions to the single processor scheduling problem, with independent, preemptive tasks that do not share resources, and the performance criterion is the number of tasks that miss their deadline. Unfortunately, most scheduling problems are complicated by additional requirements like dependencies between tasks, or the use of shared resources. A large class of these scheduling problems is known to be NP-hard, and is described by Ausiello, et. al [10]. For example, to combine real-time scheduling with databases the scheduler needs to deal with shared resources (data items). Mok showed in his PhD. thesis that this problem is NP-hard [67]. For the scheduling problems that we discuss in this thesis, heuristics are needed to efficiently find solutions.

Rate-monotonic scheduling consists of an off-line priority assignment, and a straightforward on-line scheduler. Therefore the run-time overhead of RMS is low. A disadvantage of rate-monotonic scheduling is that all scheduling information has to be available off-line. Earliest deadline first scheduling uses the same on-line scheduler, but assigns priorities dynamically during the on-line execution. It provides more flexibility at the cost of increased run-time overhead. Liu and Layland present sufficient feasibility tests [59] for both schedulers. However, using the feasibility test for EDF removes much of EDF's flexibility. The feasibility test can only be performed if the entire task-mix is available off-line. Tasks cannot be added during online execution, or the feasibility test is no longer valid.

Lehocsky, Sha, and Ding present [58] necessary and sufficient feasibility tests for RMS. They argue that RMS has great practical potential. One of their observations is that RMS can be modified to handle synchronization constraints. Indeed, Sha, Rajkumar, and Lehocsky present two scheduling algorithms [85] that deal with tasks that have shared resources, the basic priority inheritance protocol (BPI) and the priority ceiling protocol (PCP). Both adapt RMS, such that critical sections around the use of shared resources can be handled. Interestingly, Chen and Lin [24] show that these adaptations are not unique to RMS, and adapt EDF using the same ideas.

The most successful database scheduler is without doubt two phase locking (2PL) that was first described by Esrawan, Gray, Lorie, and Traiger [30]. 2PL is called "pessimistic": if the scheduler

cannot ensure that the execution of a transaction is serializable, the transaction is blocked. Blocking occurs when transactions access a data item that is in use by other transactions. Long blocking-chains can occur, if blocked transactions previously locked a set of data items. If the system-load is high, this unbounded blocking can cause *trashing*: the throughput of the database goes down radically. Franaszek, Haritsa, Robinson, and Thomasian present the locking with limited wait-depth scheduler [31, 32], that prevents thrashing. Another disadvantage of 2PL is that deadlocks can occur if transactions block each other. A large body of research has been directed at either preventing or detecting and solving deadlocks. Knapp wrote an clear introduction in this subject [48].

As an alternative to "pessimistic" concurrency control, optimistic concurrency control (OCC-pure) was introduced by Kung and Robinson [53]. Even if serializability is not ensured, transactions are allowed to execute. However, when a transaction is completed, a check is made to see if the execution is serializable. If not, the transaction is restarted. OCC-pure wastes processor power when transactions are restarted. More advanced OCC protocols [84, 88, 57] increase the concurrency of the scheduler by having more elaborate serializability-checks. This reduces the number of restarts, hence less processor power is wasted.

Bestavros describes an interesting scheduling regime called speculative concurrency control [14, 15]. It mixes optimistic and pessimistic scheduling by introducing "shadow-executions", transactions are executed in parallel under both scheduling regimes. The most successful execution is committed. The other execution is aborted. Multiple shadow-executions of the same transaction can be created, each time a possible conflict is detected. Speculative concurrency control wastes a lot of processor-time, but does improve performance if processors are not a bottleneck. It is an interesting concept, trading processor-power for concurrency.

2PL and OCC-pure have both been adapted for distributed environments [13, 55, 93, 72, 92]. Distributed 2PL [13, 72] is quite similar to normal 2PL. Usually, each site in the network has its own local lock-server that manages the access of data items, stored at that site. As a result, requesting write-locks becomes an expensive operation when the written data is stored at a remote site.

An important problem that has to be tackled when distributing OCC algorithms is the validation phase. To prevent racing conditions, centralized OCC algorithms treated validation and writing of results as one critical section. This would lead to a major bottleneck in a distributed environment. Combinations of OCC and locking are either used to overcome distribution problems [93, 41], or try to enhance concurrency by offering both mechanisms at the same time [86, 103]. K.w. Lam, Lee, K.y. Lam, and Hung adapt the popular OCC-TI scheduler to distributed environments [55], again using locks. Chapter 5 presents an different adaptation of OCC-TI, which uses a more elaborate blocking mechanism to solve the race-conditions with minimal delays. Distributed commit protocols [51] ensure that transactions are successfully completed at all sites, or are uniformly aborted.

A major source of scheduler overhead in distributed databases is communication, required by the scheduler. Although this is generally recognized [71, 37, 6, 44], database schedulers are usually not designed to minimize the scheduler communication overhead, but to maximize parallelism.

Deadlock detection schemes have to be distributed as well [48]. The schemes are often elaborate and complicated. This is illustrated by several erroneous algorithms that were published (references can be found in [48]).

Real-time database scheduling can be viewed as an extension of real-time scheduling with a more precise look at shared resources, or as an extension to databases with time requirements. Algorithms have been devised using these two viewpoints. Sha, Rajkumar, and Lehocsky apply the priority ceiling protocol [85] to the two phase locking protocol, by requiring "two phase" behavior of tasks. The resulting protocol can be used for off-line scheduling of hard real-time databases. Chen and Lin's extension of EDF with a priority ceiling is more suited for on-line, dynamic scheduling, as the entire

task mix does not have to be known in advance.

OCC-pure is not suited for extension with real-time scheduling mechanisms. When a transaction checks if its execution is serializable, it examines executions of committed transactions. If it finds interferences with a committed transaction, it restarts. The commit of a low-priority transaction can lead to the restart of a high priority transaction, which is then likely to miss its deadline. Schlageter introduced optimistic concurrency control with forward validation (OCC-F) in an early paper [84]. OCC-F offers more concurrency than OCC-pure, by restarting transactions earlier, before they start validation (see also Härder [88]). This makes OCC-F suitable for real-time scheduling, as conflicts between transactions are resolved before either has committed. Hence, a real-time protocol like EDF can be used to decide which transaction is restarted [40]. OCC-F has drawbacks. It requires scheduling information about transactions that have not finished execution. Without constraining the executions of transactions, such scheduling information is not available at this early stage. OCC-F is only implementable if the transactions have access-invariance (their data access is known in advance), or if the validation phase and write phase occur in one, large critical section.

Boksenbaum, Cart, Ferrié, and Pons combine forward and backward validation [21] in one distributed protocol to solve the problems of OCC-F. It is the first article that features dynamic time-stamps, a technique that Lee and Son used as the basis of their famous OCC-TI scheduler [57]. The dynamic time-stamp technique can be used to recognize the serializability of a very large class of schedules. Huang, Stankovic, Ramamritham, and Towsley combine forward validation with locking [41], again solving the problem that scheduling information is not known at an early stage. Both scheduling algorithms allow real-time choices to be made. Since they are both transaction-driven protocols, scenario's exist in which the information that is needed for real-time scheduling is unavailable. In this case, the protocols revert to backward validation or first-come, first-served locking.

1.6 Organization of this thesis

The first chapter contains an introduction into the field of real-time databases, an informal problem description, six example real-time systems that use databases and this small overview.

Chapter 2 shows how real-time database scheduling problems can be specified. It starts by presenting a generic specification method, and adds domain specific knowledge to the method to specify real-time database scheduling problems. Chapter 3 focuses on design issues that play an important role in real-time database scheduling problems. These issues are a recurring theme throughout the thesis. Chapter 4 takes a closer look at the telecom environment. The characteristics of the environment are then used to determine the trade-offs that are described in chapter 3. A database scheduler is designed in accordance with these trade-offs. A small analysis and test-results of an actual implementation complete the chapter. Chapter 5 is not directly inspired by an actual environment, but assumes that a scheduler for a high-conflict environment needs to be designed. The hypothetical characteristics lead to the design of the OCC-BF scheduler. Since this is a new, quite complex scheduler, the chapter contains the algorithm, a correctness proof and possible optimizations. Chapter 6 is motivated by the wish to have predictable performance. The aim of the scheduler design is to allow good performance predictions. Analyzability is acquired at the cost of reduced database flexibility. The chapter contains the SQSL scheduler, extensions to the scheduler, simulation and analysis results. Chapter 7 is based on general safety-critical environments. Hard real-time characteristics are used to simplify the scheduling problem. We design an off-line, hard real-time scheduler, which schedules transactions on the same platform as the applications that access the database. Chapter 8 is loosely based on the HEP sensor system, and more directly on the stock market systems that are described in [2, 25, 23]. The

specific characteristics of those systems enable an important simplification of the database scheduling problem: serializability does not need to be maintained at all times. We design an off-line hard real-time database scheduler that can be scheduled along with the applications that access the database. Chapter 9 takes a look at what has been achieved in this thesis, and what future research can be used to improve the quality of real-time database scheduling. Finally, there are three appendices. Appendix A contains a list of symbols that are used in this thesis. Appendix B describes the pseudo-code that is used in algorithm descriptions. Appendix C describes a few basic algorithms that are referred to in this thesis.

Chapter 2

Specification of scheduling problems

It is generally recognized [77, 27] that problem specification and gathering of user requirements is part of the total design trajectory. Important design decisions are taken when the system requirements and available resources are specified. Since this thesis aims to aid designers in the design of efficient real-time database schedulers, this chapter treats problem specification. The chapter consists of two parts. First, section 2.1 introduces a generic framework for the specification of design problems. The elements that are contained in the problem specification are defined, and their meaning is explained. In section 2.2, this framework is applied to real-time database scheduling problems.

2.1 Specification of design problems

Dasgupta [27] defines a design problem as a set of requirements R. A solution to the design problem is a design D, such that any implementation of D satisfies R. Dasgupta distinguishes between empirical requirements and conceptual requirements. Empirical requirements are requirements that specify externally observable or empirically determinable qualities that are desired for the designed object. Conceptual requirements reflect doctrines, style, esthetics or design philosophies. An example of an empirical requirement is "the total execution time should be less than 5 seconds". A typical conceptual requirement is the "no goto" programming style, or more to the point: the database paradigm.

The goal of our specification framework is to specify clear, unambiguous design problems, and empirical requirements serve that purpose. Problem specifications in this thesis do not directly reason about conceptual requirements.

Observations and constraints

Checking whether empirical requirements hold consists of observing a set of values of the designed object, and checking whether they satisfy all requirements. We distinguish two elements: a set of observations, and a set of constraints on the values of observations. The description of an observation specifies what part of the designed object has to be measured. Constraints specify the legitimate values of observations (i.e. a correctly designed object will satisfy all constraints).

Definition 2.1 *An **observation** is a part of the (behavior of the) object that can be measured.*

Definition 2.2 *A **constraint** is a proposition over the values of observations.*

Note that observations and constraints can be specified that bear no relation to the design problem that should be specified. For example, the outside temperature (an observation) is usually of no concern in real-time database scheduling problems. The sets of observations and constraints that are contained in the problem specification should be based on a carefully chosen model.

Matching mathematical model and real world

We specify design problems by constructing a mathematical model. A good model has to match closely with the part of the real world that it aims to describe. To avoid ambiguities, the matching can be mentioned explicitly in the problem specification.

Observations are represented by problem variables. The value of a problem variable is taken from a pre-defined domain. This value can be assigned by performing a measurement on the object in the real world. With each problem variable, an unambiguous description of the observation that is modeled is associated.

The objective function is an expression that contains a subset of all problem variables, and returns a value from a total order. It should be specified whether the expression has to be maximized or minimized. For example, let T be the set of all transactions that are executed. Suppose for each transaction t, $M(t)$ is 0, if t misses its deadline, and 1, if t meets its deadline. The objective "minimize the number of missed deadlines" is represented by maximization of objective function "$\sum_{t \in T} M(t)$".

Example specification. A short example from real-time scheduling is used to illustrate how observations and constraints can be used in specifications. Examples of objective functions will be given in later sections. In an embedded system, real-time transactions are issued to a database. A transaction t has to finish execution within a certain time-span. Two observations are identified: the response time of t, and the relative deadline of t. These are modeled by the following two problem variables.

Variable	Domain	Description
R_t	seconds	The response time of t.
D_t	seconds	The relative deadline of t.

One constraint specifies that the transaction has to be finished before its deadline.

Constraint	Description
$R_t \leq D_t$	Transaction t meets its deadline.

Free and fixed problem variables

The mapping from real world to mathematical model that is presented above does not yet distinguish between the object that has to be designed, and the (unalterable) environment of the object. This distinction is added by splitting all problem variables into two types: free and fixed problem variables.

A *fixed problem variable* describes the environment of the object, which cannot be influenced by the design. For example, the transactions that arrive at the database are issued by applications that use the database. The database designer has no control over these applications, so the set of transactions that arrive at the database is a fixed problem variable.

Free problem variables describe the object itself, and can be influenced by design decisions. For example, the response time of a transaction depends on the the efficiency of the scheduler, and is therefore a free problem variable.

Fact constraints and goal constraints

Two types of constraints are distinguished. *Fact constraints* hold for each possible design, they cannot be violated. *Goal constraints* can be violated by incorrectly designed objects. Fact constraints can be used to describe the environment in which the object should function. Goal constraints describe the desired behavior of the object. Constraints that only involve fixed problem variables are fact constraints, since they are not influenced by the design. Constraints that involve free problem variables are either fact constraints, or goal constraints.

An example of a fact constraint that involves free problem variables is the following. Suppose the processor speed is represented by free problem variable S (choosing the hardware is part of the design problem). Suppose the work of transaction t is represented by fixed problem variable W. Suppose the time needed to process transaction t is given by free problem variable X (it is a free problem variable, as it is influenced by the choice of the processor). The relation between the processor speed, the work of transaction t and its execution time is now captured by fact constraint "$W/S = X$".

A typical goal constraint is serializability. Suppose the database access order of all transactions is given by list DAO. A so-called serializability graph SG can be constructed by creating a vertex for each transaction. For each pair of vertices A, B, a directed edge $A \rightarrow B$ is added if A and B performed a conflicting data access (in order $A; B$). A data access order satisfies the serializability constraint if its corresponding serializability graph is acyclic. Hence, the resulting goal constraint is: "SG is acyclic".

Value assignment

The value assignment of a problem variable is the result of an observation of part of the designed object. In case of fixed problem variables, this can be viewed as an instantiation of the problem: the actual input is determined. For example, if T is the fixed problem variable that represents the set of transactions, assignment of T occurs before the scheduler can start its execution. In the case of free problem variables, the assignment of variables is used to verify the design, by checking the validity of all goal constraints. The aim of the specification is to serve as a source of information for the designer, and a means to verify the design.

Value assignment occurs quite differently in constraint satisfaction problems (CSPs). Although the specification of a CSP (see [96] for a textbook on constraint satisfaction) looks similar to a specification that is described according to our framework, CSP specifications require a greater level of detail. A CSP also consists of a set of variables and a set of constraints on the values of these variables. However, CSPs are solved by directly assigning values to variables. Once a value assignment is found that satisfies all constraints, the CSP is solved. To use a CSP in the specification of a design problem, a mapping from the assigned values to an implementation of the designed object is needed. This mapping can be very complex, and specifying it is a design problem in itself.

Specifying time

Remember that a design problem specifies the desired behavior of the object that has to be designed. In most cases, there is a temporal aspect to the behavior of an object, events occur in some temporal order. Therefore, the moment that an observation takes place is important. This moment is called the *assignment moment*. Usually, the assignment moment is implicitly contained in the description.

In many problem domains, especially real-time scheduling, assignment moments strongly influence the design. Information about the value of an observation cannot be used before the assignment moment. This provides a hard lower-bound on the moment that decisions can be taken that are based

on this information. Depending on the problem domain, the domain of assignment moments can be very precise (split into micro-seconds) or very rough (an enumerated type that contains "specification, before execution, during execution").

2.2 Real-time database scheduling problems

The specification framework from the previous section is applied to real-time database scheduling problems. We identify the relevant concepts that should be modeled, and give observations and constraints that fit these concepts in our framework. The scheduling problems that are described in this thesis are inspired by the needs of the applications described in chapter 1. The following generic problem statement is applicable to all scheduling problems that we investigate.

Real-time database scheduling problem statement A set of transactions has to be executed on a platform. The executions and the platform have to satisfy a number of functional and temporal requirements. An off-line execution precedes the execution of transactions. This off-line execution has to satisfy a possibly different set of functional and temporal requirements.

The generic problem statement is necessarily vague on a number of points. What are the essential properties of the transactions? How many are there? What does the platform look like? What exactly are those functional and temporal requirements? These questions are answered differently for each application, resulting in a set of unique scheduling problems. To some extent, these problems can be unified by generalization and abstraction. However, solutions to such generalized problems are often less efficient than solutions to very specific scheduling problems. In chapter 3 we give a number of examples that support this claim. To fill in the missing information of the generic problem statement, we use a classification scheme that is similar to Lawler's classification of real-time scheduling problems [56], describing platform characteristics, transaction characteristics and an objective criterion. Our classification is slightly modified to deal with scheduling problems that are related to real-time databases.

The platform is described in section 2.2.2. Characteristics of data are treated in section 2.2.3. Transaction characteristics (similar to task characteristics) are described in section 2.2.4. Lawler's "objective criteria" closely match objective functions, and they are discussed in section 2.2.5. Lawler's classification focuses on off-line scheduling, and the distinction between off-line and on-line execution is not specified clearly. The next section describes the general shape of solutions to scheduling problems, distinguishing between off-line and on-line execution.

2.2.1 Schedulers

An algorithm that serves as a solution to a scheduling problem is called a *scheduler*. A scheduler consists of a set of run-time or pre-runtime processes and data structures. A solution to a scheduling problem can be viewed as an implementation guideline: it contains the algorithms that have to be implemented and descriptions of the resources that can be used in the implementation. A scheduler is correct, if it satisfies all goal constraints that are contained in the problem specification.

We take a look at the execution model that is used for scheduling problems, discuss the goals of the scheduler, and discuss the options of the scheduler.

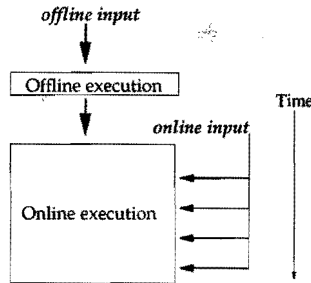


Figure 2.1: LAYERED EXECUTION MODEL

Off-line and on-line execution. We distinguish between the actual execution of the system (called the run-time or on-line execution), and execution that takes place during the setup of the run-time system (called pre-runtime or off-line execution). This distinction leads to the layered execution model that is depicted in figure 2.1. Both the off-line and on-line execution are part of the execution of the scheduler. Time requirements on the off-line execution are typically weak, limited to a bound on the total execution time of the off-line execution. The big-O notation (see for explanation [68], page 52-60) is used to specify this bound.

The input for the off-line execution arrives as one large set of information, at the start of the execution. The input for the on-line execution arrives as a continuous stream, instead of one large lump. Time scales are completely different for the off-line and on-line execution. Often, there are strong temporal requirements on the on-line execution. For example, individual transactions have to finish execution within milliseconds.

Scheduling goals. Schedulers regulate the execution of transactions. The first goal of the scheduler is to compute a *schedule* for a given set of transactions. Real-time database schedules describe the partial order in which transactions access the database, and describe at what moments transactions can start execution, and when they are delayed. We recognize three types of constraints on the schedule: non-interference between transactions, real-time requirements on transaction executions, and requirements on the access of temporal data.

The second goal of the scheduler is to enforce the computed schedule, actually executing or delaying transactions. Schedulers can only enforce the schedule during run-time execution, and execution of the scheduler itself influences the real-time behavior that should be controlled. This additional overhead has to be accounted for when the schedule is constructed. Run-time overhead can be reduced by computing (part of) the schedule off-line, as is often done in hard real-time scheduling.

Availability of scheduling information. The scheduler constructs the schedule using information about the set of transactions that has to be executed. It cannot generate the schedule before the necessary information is available. If the input of the scheduler is modeled by problem variables, the assignment moments of these variables specify when the scheduler can receive its input. We give a classification of the moments at which scheduling information can become available. This classification forms the domain of the assignment moments of problem variables.

- **Specification**. The information is directly contained in the specification. Typically, the hardware architecture (but not necessarily the exact configuration), data access modes, transaction types, and some global restrictions on problem variables (like serializability) are available.
- **Off-line**. Information becomes available after the design, but before on-line execution. This will almost always be the case for the platform description. Classical off-line scheduling algorithms often require extensive amounts of information to be available off-line.
- **On-line, announced**. Information about events becomes available on-line, some time before the events take place. A constraint can restrict the interval in which the information will become available. An example is the interesting hard real-time scheduling mechanism from [35], where all transactions that are received for execution are buffered and executed in batches. Or the announcement scheduling from [42], where transactions announce their execution, in order to facilitate scheduling.
- **On-line, unannounced**. Information about events becomes available just prior to the occurrence of the actual event. For example: information about data accesses is only generated when a transaction is about to perform a data access.
- **On-line, fait accompli**. Information about events usually becomes available after the event has occurred. This is often the case for exact transaction execution durations. Only when a transaction has finished its execution, can this duration be measured. Fait accompli information is the primary source of information of optimistic concurrency control algorithms. It can be very detailed and it is easy to accumulate. OCC schedulers can therefore take sophisticated scheduling decisions. Unfortunately, OCC schedulers have a limited set of manipulation options: they can either execute a transaction, abort it, or restart it. Since the scheduling decisions are based on fait accompli information, delaying transactions is not possible.

Manipulation options of the scheduler. After computing a schedule, the scheduler has to ensure that transactions execute according to that schedule. The manipulation options of the scheduler depend on the status of the execution of the transactions, and the possibilities offered by the platform. There are four ways to manipulate a transaction: execute it, delay it, restart (part of) the transaction, or abort it. In certain states some manipulations can be impossible. For example, when a transaction has already released its results to the rest of the system, it cannot be restarted anymore since its results are possibly in use now.

Delaying an executing transaction can be achieved in two ways. Synchronization points can be built into the execution of a transaction. Whenever a synchronization point is reached, the scheduler is activated, and can decide to delay the transaction. The scheduler has control over the progress of the transaction: it can decide which part of the transaction has to be delayed, and which part can be executed. Therefore, synchronization points are useful to enforce functional requirements like serializability.

Alternatively, a transaction t can be delayed as the result of an outside event, like a delayed user response. Transaction t is pre-empted at an arbitrary point p during its execution. Whether or not t has executed a critical execution step c depends on p, and the speed at which transaction t is executing. This is called a racing condition: either t is pre-empted before it performs c, or not. Generally, the outcome of a racing condition is hard to predict, and racing conditions should be avoided when possible. We conclude that unsynchronized delay of a transaction is not suitable to control the functional behavior of a transaction.

A number of scheduling algorithms require that the scheduler can pre-empt transactions at arbitrary points in time, regardless of the state of these transactions. Unsynchronized delay of transactions is suitable for these algorithms.

2.2.2 Platform description

Three different architectures are treated: centralized, shared-memory and distributed architectures. For each of these architectures we present a set of problem variables that describes the architecture. In all three cases, we limit ourselves to main-memory databases. Main-memory databases have become a practical alternative to disk-based databases, with the increased availability of large and cheap memories. Their speed is superior to disk-based systems, which makes them the ideal choice for systems with high throughput, or tight deadlines. We assume that sufficient memory is available to satisfy the memory requirements of all transactions that are executing, and hence do not model memory requirements.

The execution speed of a transaction depends on the speed of the processor(s) on which it is executed, and the number of execution steps that it requires (also called the amount of *work* required). The speed of a processor is modeled by a problem variable that contains the amount of work that a processor can handle in one time unit. The number of processors that are available influences how much work the system can process in one time unit. Therefore, the number of processors is also modeled by a problem variable. Many schedulers execute transactions in an interleaved fashion on a single processing unit. Whenever the execution of a transaction is pre-empted by another transaction, or a scheduler process, a context switch takes place. Context switches are often time consuming, compared to the average execution length. The overhead caused by a context switch is modeled by a problem variable. Communication in shared-memory architectures and distributed architectures influences the execution time of transactions that need data from more than one processor. In the next three paragraphs we model the architectures separately.

Centralized architecture. Communication between processing units is not applicable, since there is only one processing unit. The fixed problem variables that describe the platform are as follows.

Fixed Variable	**Domain**	**Description**
$Cpower$	work/second	Amount of work that a processor can process in one time unit.
$Cswitch$	seconds	Overhead caused by a context switch.

Shared-memory architecture. The communication in a shared-memory architecture takes place through the shared memory. Typically, accessing the shared memory takes some setup time (latency). Once the access is in progress, data is transferred at a certain rate (throughput). Both latency and throughput are modeled by problem variables. Synchronization between processors is made possible by synchronization primitives (semaphores). Calling these primitives takes time, which is modeled by a problem variable.

Fixed Variable	**Domain**	**Description**
SiteSet	$\mathcal{P}$(identities)	The set of processors.
Ncpus	$\mathbf{N}^+$	The number of processing units.
Cpower	work/second	Amount of work that a processor can process in one time unit
Mlatency	seconds	Latency of the shared memory
Mthoughput	bytes/second	Throughput of the shared memory
Msync	seconds	Overhead caused by the synchronization primitive
Cswitch	seconds	Overhead caused by a context switch

Distributed architecture. Distributed architectures communicate through a network. We abstract from the topology of the network, and assume communication is characterized by the setup time (latency) and the transfer rate (throughput). These observations are modeled by problem variables. Synchronization between processors is realized by sending and receiving empty messages, and needs no separate modeling.

Fixed Variable	**Domain**	**Description**
SiteSet	$\mathcal{P}$(identities)	The set of processors.
Ncpus	$\mathbf{N}^+$	The number of processing units.
Cpower	work/second	Amount of work that a processor can process in one time unit.
Cswitch	seconds	Overhead caused by a context switch.
Nlatency	seconds	Latency of the network.
Nthoughput	bytes/second	Throughput of the network.

Comparison with Lawler's notation

This modeling is quite different from Lawler's description of the hardware. Lawler's description is more focussed on scheduling of large factory environments. The main difference with real-time database scheduling is the time-scale. Real-time transactions are very short, compared to production machine jobs. As a result, the time spent on communication, synchronization, and computation of the schedule is significant, compared to the execution time of one transaction. Therefore, scheduling overhead has to be modeled in more detail than is usual for factories. On the other hand, production machines are often dedicated to a certain operation, which is not the case for processors. Here Lawler's classification was simplified.

2.2.3 Data characteristics

Two kinds of data items are recognized: temporal and non-temporal data items. Temporal data items are treated in chapter 8. In this section we define the characteristics of non-temporal data-items. We assume that the set of non-temporal data items is fixed, and is represented by fixed problem variable *DataSet*. They are characterized by their size, their location, and by the way they can be accessed. Since the location of data items is only relevant in distributed architectures, this is treated separately.

The size of a data item influences the time necessary to read or write the data item. Since these two operations are the most important operations performed on data items, we model the size of data items by a problem variable *DataSize*. We simplify our model by assuming that all data items are of equal size.

Data access. The database offers a fixed set of operations that manipulate data items. Transactions can execute these operations if they need to access the database. A conflict relation is defined on the set of operations (for a more theoretical approach to conflict relations, see [50]). If two operations conflict, a wrong execution order of these operations can possibly violate a consistency requirement. Furthermore, an interference relation is defined on the set of operations. If two operations interfere, interleaved execution of these operations can possibly violate a consistency requirement. In the remainder of this thesis we assume that all data access operations are atomic. Therefore, we do not use the interference relation between data access operations. It is included here, to show that the atomicity assumption on data access operations can be lifted under the right circumstances.

The interference relation is strictly stronger than the conflict relation. It specifies mutual exclusion constraints on data access operations. The conflict relation is used to define functional consistency. This is treated under transaction characteristics. We give an example that explains the use of these relations. Suppose the set of operations is

$$\{read, write, addition\}$$

A read operation on a data item X returns the current value of X. A write operation on X sets the current value, and an addition operation adds a constant to the current value. The conflict relation is now given by the following matrix.

conflict	**read**	**write**	**addition**
read	false	true	true
write	true	true	true
addition	true	true	false

Two read operations can be executed in an arbitrary order, without influencing the final results. Two addition operations can also be executed in an arbitrary order without influencing the final result. All other combinations of operations have two different results, depending on the execution order. The interference relation is given by the following matrix.

inference	**read**	**write**	**addition**
read	false	true	true
write	true	true	true
addition	true	true	true

Since read operations do not change the database state, but only local variables, interleaved execution of two reads will not interfere. An addition operation does change the database state, by reading the old value, adding something and writing it back. If two addition operations are interleaved incorrectly, one of the two additions is lost. Therefore, addition operations can interfere, if they are performed in an interleaved fashion.

The set of operations on data items, and the conflict and interference relations are modeled as fixed problem variables. The set of possible operations is called *Optypes*. Since we allow arbitrary operations to be defined, the domain of *Optypes* is the set of legal identifiers. The set of all access operations is OPS. For each operation, functions *Amode* and *Daccess* define the access mode and the data item that is accessed. Since applications define which data accesses take place, these are fixed problem variables.

Fixed Variable	**Domain**	**Description**
$DataSize$	bytes	The size of a data item.
$DataSet$	$\mathcal{P}$(identifiers)	The set of data items that is stored in the database.
OPS	$\mathcal{P}$(identifiers)	The set of all access operations.
$OPtypes$	$\mathcal{P}$(identifiers)	The set of possible operations.
$Amode$	$OPS \rightarrow OPtypes$	Function that defines access mode of each operation.
$Daccess$	$OPS \rightarrow DataSet$	Function that defines which item an operation accesses.
$ConfRel$	$OPtypes \times OPtypes \rightarrow Bool$	The conflict relation on $OPtypes$.
$InterRel$	$OPtypes \times OPtypes \rightarrow Bool$	The interference relation on $OPtypes$.

Unless specified otherwise, all specifications in this thesis use the following set of operations, conflict and interference relation:

Fact constraint	**Description**
$OPtypes = \{\text{read, write}\}$	Data items can be read and written.
$ConfRel(p,q) = (p = \text{write}) \vee (q = \text{write})$	Only read/read access is non-conflicting.
$InterRel(a,b) = ConfRel(a,b)$	The interference and conflict relation are the same.

The following predicate specifies whether two access operations conflict.

$$conflict(op, op') \equiv Daccess(op) = Daccess(op') \wedge ConfRel(Amode(op), Amode(op'))$$

Location. The actual dataset is stored in main memory. In centralized architectures, and shared-memory architectures, location is not an issue. The following specification holds for distributed architectures. The dataset is divided over the available processors, called *sites*. A common technique that enhances (amongst others) the availability of data is replication [5, 11, 34]. This is not considered in this thesis, each data item is stored at a single site. When a transaction needs to access a data item, it is either stored locally, or on a remote site. The exact division of the dataset strongly determines the execution times of transactions, and is modeled by a problem variable $DataPlace$.

Fixed Variable	**Domain**	**Description**
$DataPlace$	$Items \rightarrow SiteSet$	A function that assigns data items to sites.

Data allocation to sites has a major influence on the execution duration of transactions, and is treated in most database textbooks [12, 51]. In this thesis we assume that data allocation has occurred before the off-line execution starts.

2.2.4 Transaction characteristics

Applications interact with the database by issuing transactions, which are subsequently executed by the scheduler. The execution of a transaction has to satisfy both real-time and database requirements. We divide the description of the real-time requirements into two parts. First, we describe those problem variables that are relevant for almost all real-time scheduling problems. Next, we describe the problem variables that are used in soft, firm and hard real-time scheduling.

Generic real-time characteristics of transactions. Fixed problem variable T describes the set of transactions that is executed by the scheduler. Note that the size of T is unbounded if a continuous stream of transactions arrives at the database. Each transaction t will have a start time and finish time. Since these are determined by the scheduler they are modeled by free problem variables $St(t)$ and $Ft(t)$. Furthermore, each transaction t has an amount of work $W(t)$ that has to be performed, this is a fixed problem variable. The total execution time necessary to execute t is described by fixed problem variable $X(t)$.

Fixed Variable	Domain	Description
T	$\mathcal{P}$(identifiers)	The set of transactions that is executed by the scheduler.
W	$T \to$ work	The amount of work of transaction.
X	$T \to$ seconds	The total execution time of a transaction.

Free Variable	Domain	Description
St	$T \to$ seconds	The start time of transactions.
Ft	$T \to$ seconds	The finish time of transactions.

A few fact constraints relate these problem variables.

Fact constraint	Description
$\forall t \in T : Ft(t) - St(t) \geq X(t)$	A transaction cannot finish before its execution time has passed.
$\forall t \in T : X(t) = W(t)/Cpower$	Relation between processor speed, amount of work and execution time.

Soft real-time, unannounced transactions. Soft real-time databases do not require that all transactions are known before the on-line execution. Transactions arrive at the system in a continuous stream. The set of transactions T is fait accompli information: its definition is complete when the on-line execution finishes. In continuous systems, T is unbounded.

The arrival time of transaction t is described by fixed problem variable $Arr(t)$. An arrived transaction can directly begin execution. Its execution has to be finished before a specified deadline. This deadline is represented by fixed problem variable $Dl(t)$. Transactions are occasionally allowed to miss their deadlines.

Fixed Variable	Domain	Description
Arr	$T \to$ seconds	The moment of arrival of transactions.
Dl	$T \to$ seconds	The deadline of transactions.

Instead of an absolute deadline on the finish time of a transaction, a bound on the response time $R(t) = Ft(t) - Arr(t)$ of transaction t can be specified. The response time is the difference between the finishing time of a transaction, and the arrival time of a transaction. In many cases, $R(t)$ is of more interest than $Arr(t)$ or $Ft(t)$. An advantage of this modeling is that $R(t)$ can be specified using local clocks, instead of a global clock (which is necessary for absolute time points).

Firm real-time, unannounced transactions. Firm real-time databases do not require that all transactions are known before the on-line execution. Transactions arrive at the system in a continuous stream. The set of transactions T is fait accompli information: its definition is complete when the on-line execution finishes. In continuous systems, T is unbounded.

The arrival time of transaction t is described by fixed problem variable $Arr(t)$. An arrived transaction can directly begin execution. Its execution has to be finished before a specified deadline. This deadline is represented by fixed problem variable $Dl(t)$. Transactions that fail to meet their deadlines can be aborted, as their execution becomes valueless. Free problem variable $abort(t)$ specifies whether t is aborted.

Fixed Variable	**Domain**	**Description**
Arr	$T \rightarrow$ seconds	The moment of arrival of transactions.
Dl	$T \rightarrow$ seconds	The deadline of transactions.

Free Variable	**Domain**	**Description**
$abort$	$T \rightarrow$ Boolean	Signifies whether the transaction is aborted.

Goal constraint	**Description**
$\forall t \in T : abort(t) \Rightarrow Ft(t) > Dl(t)$	Transactions that miss their deadline can be aborted.

Hard real-time time-intervals. A classic real-time scheduling problem requires that a well-defined set of transactions are executed in pre-defined time intervals. If a transaction fails to execute within its allotted time-span, the system fails. For each transaction t, this interval is described by fixed problem variables $Est(t)$ and $Dl(t)$ that represent the earliest start time and deadline of t. To give off-line guarantees that the system satisfies the hard real-time requirements, this information becomes available at the start of the off-line execution. Hence, the set of transactions T (which was defined earlier) is finite and its assignment moment is at the start of the off-line execution.

Fixed Variable	**Domain**	**Description**
Est	$T \rightarrow$ seconds	Earliest start time of a transaction.
Dl	$T \rightarrow$ seconds	Deadline of the transactions.

Goal constraint	**Description**
$\forall t \in T : Est(t) \leq St(t) \leq Ft(t) \leq Dl(t)$	Transactions execute within their assigned intervals.

Database requirements on transactions

The schedule that is generated by the scheduler has to satisfy the serializability requirement: the schedule has to be equivalent to a sequential schedule. Depending on the exact equivalence relation, different notions of serializability exist, like conflict-, view- or epsilon-serializability.

All notions of serializability are defined on the order in which database accesses take place. Instead of the entire schedule (that also contains real-time information), we model the sequence of all database accesses by a total order $\prec_A$ on the set of database accesses OPS. This total order is described by a free problem variable $Aorder$ that has type $OPS \times OPS \rightarrow Bool$. Each access operation belongs to a particular transaction. The fixed problem variable tr specifies this relation.

Free Variable	Domain	Description
tr	$OPS \to T$	Function that defines to which transaction an operation belongs.
$Aorder$	$OPS \times OPS \to Bool$	The sequence of all database accesses.

We introduce extra notation to reason about $Aorder$. The nth access in a sequence, described by $Aorder$ is denoted $Aorder(n)$. The index of data access op in the sequence described by $Aorder$ is $ix(Aorder, op)$. The order relation $\prec_A$ is reflected by the index function:

$$p \prec_A q \Leftrightarrow ix(Aorder, p) < ix(Aorder, q)$$

Note that data items can be accessed concurrently, and $Aorder$ is strictly sequential. If two actions a, b are executed concurrently, they can be ordered arbitrarily.

In this thesis, it is assumed that each separate transaction reads and writes each data item at most once. This can be implemented easily and simplifies the descriptions of the scheduling algorithms. The following predicate expresses that all transactions satisfy this behavior.

$$\text{accessonce}(Aorder) \equiv$$
$$\forall p, q \in Aorder : tr(p) = tr(q) \wedge Daccess(p) = Daccess(q) : Amode(p) \neq Amode(q)$$

The different kinds of serializability that were mentioned in chapter 1 can be expressed as constraints on $Aorder$. Note that most scheduling problems in this thesis are only concerned with conflict serializability, view serializability and epsilon serializability are defined to present an overview of the most common consistency constraints. Conflict serializability is a subset of view-serializability, and they do not require additional information. Epsilon-serializability relaxes the serializability requirement by allowing some interferences to occur. To control these interferences, additional information is necessary. Before describing the different serializability concepts, we introduce the notion of a sequential schedule. All serializable schedules are (in some sense) equivalent to such a schedule. A schedule is serializable, if its corresponding sequence of database accesses $Aorder$ satisfies the following conditions. An access order is sequential if the operations of all transactions are consecutive.

$$\text{sequential}(Aorder) = \forall p, g, q \in Aorder : (tr(p) = tr(q) \wedge p \prec_A g \prec_A q) \Rightarrow tr(g) = tr(p)$$

Conflict serializability. Two schedules S and S' are conflict-equivalent, if the access order $Aorder^S$ of S can be obtained from the access order $Aorder^{S'}$ of S' by a sequence of non-conflicting swaps. A swap exchanges the order of two neighboring database accesses. A non-conflicting swap is a swap of two operations that do not conflict (see $ConfRel$). A schedule S is conflict-serializable if a sequential schedule exists that is conflict-equivalent. Conflict serializability of access orders A is specified by the following constraint.

$$\text{swap}(A, A') \equiv A = A' \text{ with } A[i] = A'[i+1] \wedge A[i+1] = A'[i] \wedge \neg\text{conflict}(A[i], A[i+1])$$
$$\text{conflictequiv}(A, A') \equiv A = A' \vee \exists A'' : \text{swap}(A, A'') \wedge \text{conflictequiv}(A'', A')$$
$$\text{conflictserializable}(A) \equiv \exists A' : \text{sequential}(A') \wedge \text{conflictequiv}(A, A')$$

View serializability. Two schedules S and S' are view-equivalent, if the reads-from relation of S and S' is the same, and they contain the same database accesses. The reads-from relation of a schedule

S depends on the corresponding data access order $Aorder$. First, we define a new conflict-relation $ViewConfRel$ that ignores write/write conflicts:

$$ViewConfRel(op, op') \equiv Amode(op) = \text{write} \wedge Amode(op') = \text{read} \wedge$$
$$Daccess(op) = Daccess(op')$$

The reads-from relation is defined on the operations in an access order A, and has the following property.

$$rf(A, op, op') \equiv op \prec_A op' \wedge ViewConfRel(op, op') \wedge \nexists op'' : op \prec_A op'' \wedge rf(A, op'', op')$$

Before we introduce view-equivalence we define the *encapsulation* A_e of access order A. A_e is constructed from A by adding a transaction t_{begin} and t_{end} to A. Transaction t_{begin} precedes all other transactions, and writes each item accessed in A. Transaction t_{end} is preceded by all other transactions, and reads each item accessed in A. Two schedules S, S' are view-equivalent if the reads-from relation of the encapsulated corresponding access orders A_e, A'_e is the same. View-equivalence, and view-serializability of access orders A is defined as follows.

$$\text{viewequiv}(A, A') \equiv \forall op, op' \in A_e : (rf(A_e, op, op') \Leftrightarrow rf(A'_e, op, op')) \wedge op, op' \in A'_e$$
$$\text{viewserializable(A)} \equiv \exists A' : \text{sequential}(A') \wedge \text{viewequiv}(A, A')$$

Epsilon serializability. Epsilon serializability allows more schedules than conflict serializability (and can be extended to allow more schedules than view serializability). The class of allowed data access orders is increased by weakening the conflict-relation that is used in conflict serializability. Each operation op on data "exports" an amount of inconsistency $Exp(op)$. For each operation op an "import-limit" $Imp(op)$ is defined. If the import-limit of transactions that modify the database-state equals 0, only read-only transactions benefit from epsilon serializability. The new epsilon-conflict relation is defined as follows.

$$EpsConfRel(op, op') \equiv ConfRel(op, op') \wedge exp(op) > imp(op') \vee exp(op') > imp(op)$$

Epsilon equivalence is defined in the same way as conflict-equivalence, with the new conflict-relation. Epsilon serializability of access orders A is defined as follows.

$$\text{epsswap}(A, A') \equiv A = A'$$
$$\text{with } A[i] = A'[i+1] \wedge A[i+1] = A'[i] \wedge \neg EpsConfRel(A[i], A[i+1])$$
$$\text{epsilonequiv}(A, A') \equiv A = A' \vee \exists A'' : \text{epsswap}(A, A'') \wedge \text{epsilonequiv}(A'', A')$$
$$\text{epsilonserializable}(A) \equiv \exists A' : \text{sequential}(A') \wedge \text{epsilonequivalent}(A, A')$$

Fixed Variable	**Domain**	**Description**
Exp	$OPS \rightarrow$ inconsistency	The amount of inconsistency that is exported by an access operation.
Imp	$OPS \rightarrow$ inconsistency	The amount of inconsistency that can be imported by an access operation.

Another variant of Epsilon serializability places an import-limit on the entire transaction, instead of the individual operations. We did not investigate this.

Location

In centralized architectures, location of transaction execution is not a major issue. In shared-memory architectures, load balancing between the processors is necessary, but location has no other meaning, since processors are identical. The following specification holds for distributed architectures, where sites differ, since they store different parts of the data set. We assume that transactions execute at one site, which is defined by fixed problem variable $TransPlace$. Data items are fetched from remote sites, before they are processed by transactions. Remote data access is time-consuming, compared to local data access. To optimize the performance of the database, transactions should be executed at the site where their data is stored. In this thesis, we do not consider transaction placement, therefore $Transplace$ is a fixed problem variable. A predicate is introduced that specifies whether data access is local or remote.

$$local(op) \equiv TransPlace(tr(op)) = DataPlace(Daccess(op))$$

The $DataPlace$ and $TransPlace$ variable can be hard to define. Placement of data items can change, as new items are created, and old items are deleted. For effective use of the information, placement of the transaction executions, and their data access is also necessary information. In many occasions, the effort of measuring the value of $DataPlace$ and $TransPlace$ is not worth the benefit.

A useful abstraction is to measure the probability that a data access is local. This statistical information can be obtained through simple experiments, and (as we will show in chapter 4) can steer scheduler design. Each scheduler can calculate this information fait accompli, or the probability can be received as separate input at an earlier stage.

Fixed Variable	Domain	Description
$TransPlace$	$T \rightarrow SiteSet$	This function identifies the site on which a transaction executes.
$Pdatalocal$	[0, 1]	The probability that a data access is local.

Summary of database requirements. The different serializability notions all place restrictions on the allowed data access order. Therefore, they are expressed as goal constraints on the free problem variable $Aorder$.

Goal constraint	Description
conflictserializable($Aorder$)	Access order $Aorder$ is conflict-serializable.
viewserializable($Aorder$)	Access order $Aorder$ is view-serializable.
epsilonserializable($Aorder$)	Access order $Aorder$ is epsilon-serializable.

We defined one fact constraint. This constraint restricts the access pattern of transactions, without restricting the expressiveness of transactions.

Fact constraint	Description
accessonce($Aorder$)	Transactions read and write each data item at most once.

Restricting the input

Many systems, especially hard real-time systems, place extra demands on the transactions that are executed, restricting their generality. Knowledge about these restrictions can simplify the scheduling problem. In hard real-time systems, it is common practice to require that additional information about transactions is provided off-line. Such additional information is specified by adding fact constraints, and possibly a separate set of fixed problem variables that describe the additional input. We give a number of examples. Perhaps the best-known example is the worst-case execution time. Applications specify a worst-case bound $WCX(t)$ on the clean execution time $X(t)$ of each transaction t. Fact constraint $X(t) \leq WCX(t)$ specifies this relation. Worst-case execution times are necessary to give an off-line guarantee that all deadlines will be met during on-line execution.

An example from database scheduling is a restriction on the data access order of transactions. Suppose there exists an acyclic directed graph G, where the vertices represent the data items stored in the database, and the edges represent the order in which the transactions can access the database. A transaction t accesses the database according to graph G, if for all accesses $a \prec b$, no path from b to a exists in G.

The fact constraint "all transactions access the database according to G" severely reduces the generality of the database, but it allows for more efficient scheduling. For example, acyclic data access removes the need for deadlock prevention mechanisms in two phase locking. Another example or restricting the input can be found in chapter 4, where it is specified that around 90% of all transactions is a read-only transaction.

2.2.5 Objective functions

The objective function consists of the performance measure that best describes the objective of the scheduler design. Performance measures use the values of problem variables as input. By weighing one or more problem variables, different performance measures can be defined. We investigate what performance measures are good candidates for objective functions of hard-, firm- and soft real-time database scheduling problems.

Soft real-time systems. In soft real-time databases, the value of a late transaction decreases, but is not instantly zero. Ideally, the objective function is "the total value of all transactions". However, evaluating this performance measure is very impractical. Usually, the real-time performance is measured in the percentage of transactions that meets its deadline.

Objective function: minimize $\sum(t \in T : Ft(t) > Dl(t) : 1)/|T|$

Information about deadlines is not always available, either because the application doesn't specify them, or because the system does not have a global clock that can be used as a reference point. In a system where this information is unavailable, the scheduler can only try to execute transactions as fast as possible, and avoid exceptionally long transaction executions. Objective functions can use the response time distribution of transactions.

Objective function: minimize $\sum_{t \in T} R(t)/|T|$

For example, we can require that the system minimizes the average response time, under the condition that at most 10% of the transactions use more than 4 times the average response time AvR.

Goal constraint	Description
$\sum(t \in T : R(t) \geq 4AvR : 1)/\lvert T\rvert \leq 0.1$	At most 10% of all transactions has more than 4 times the average response time.

Firm real-time systems. In firm real-time databases, the value of a late transaction instantly decreases to zero. The transaction can be aborted without further degrading the system performance. The objective function minimizes the number of transactions that miss their deadline.

Objective function: minimize $\sum(t \in T : Ft(t) > Dl(t) : 1)/|T|$

Hard real-time systems. A hard real-time system fails if one of its timeliness constraints is violated. No transaction misses its deadline in a correct implementation. Therefore, the objective functions that are used for soft and firm real-time systems are not suitable for hard real-time systems.

So what are interesting performance measures? A performance measure that is generally interesting is the hardware that is required. Minimizing the hardware that is necessary to implement the system decreases overall system cost. If the number of processors $nCPU$ is a free problem variable, it can be an interesting performance measure. By choosing this measure as an objective function, the optimal solution to the scheduling problem will have a minimal number of processors.

Objective function: minimize $nCPU$

When the hardware is fixed, another objective function is needed. A typical objective function from hard real-time scheduling is maximization of the the maximum amount of work that the system can handle, without violating goal constraints. To measure this maximum, the amount of work that should be handled by the system has to be a free problem variable. Hence, the system designer can decide how many transactions have to be scheduled by the system.

Objective function: maximize $\sum_t \in T : W(t)$

Tuning the objective function

The objective function can be tuned to particular applications. For example, transactions might have a different value to the application: some transactions should never fail, while others are less critical. The performance measure that counts the number of missed deadlines should be replaced by a performance measure that counts the total value of failed transactions.

2.3 Conclusions

The framework that is presented in section 2.1 can be used to eliminate hidden assumptions that are often part of informal problem specifications. It captures the relation between the real world, and the mathematical model in the unambiguous descriptions. Furthermore, their is a clear distinction between the environment and the object that should be designed. The environment is described by fixed problem variables and fact constraints. The object that should be designed is described by free problem variables and goal constraints. The explicit description of the environment serves as a source of information for the designer.

In the second section of this chapter we applied the specification framework to real-time database scheduling problems. A number of standard specifications are introduced that will be used in the next chapters. The aim of this section is to help in the specification of real-time database scheduling problems that can arise in practice. Rather than fixed rules, it offers suggestions and examples.

Chapter 3

Design issues

The design of the scheduler focuses on the satisfaction of the goal constraints, and optimizes the objective function at the same time. In this chapter techniques are presented that aid in this process. The presentation is not exhaustive. Rather, the design issues have been identified during the project of which this thesis is the end-result. These issues reappear in chapters 4 to 8. The contribution of this chapter is the separation of issues & concepts of design from the actual solutions in later chapters. It can be used as a checklist during scheduler design. The following design issues are presented in this chapter.

Targeted problem specification. In section 3.1, the advantages of targeted problem specifications over abstract problem specifications are discussed. It shows with an example that performance improvements can be made, if more detailed problem specifications are given.

Concurrency. Schedulers have to optimize the value returned by the objective function. However, such functions do not specify how the optimization should be realized. Section 3.2 introduces the concept of concurrency, and shows that an increase in concurrency can optimize the soft, firm and hard real-time objective functions from chapter 2.

Overhead. Section 3.3 discusses the impact of scheduler overhead, and ways to reduce this overhead. The treatment of concurrency and scheduler overhead is concluded in 3.4, where a tradeoff between these two concepts is recognized. Information that is contained in the targeted problem specification can be used to decide this tradeoff.

Availability of scheduling information. The moment at which scheduling decisions can be taken depends on the availability of scheduling information. Section 3.5 explains the relation between availability of information and scheduling decisions, both for on-line and off-line scheduling.

3.1 Targeted problem specification

When a real-time scheduling problem is formulated for a given application, there is a substantial amount of freedom. To keep the specification clear, the problem should be formulated in an abstract and short way. Good problem specifications maintain essential details, while abstracting from details that do not influence the final design. However, many abstractions can be chosen.

It is attractive to leave out all details, and specify a very generic real-time scheduling problem. This has two major benefits. First of all, such a problem specification is easy to construct. Second, scheduling solutions to such generic problems can be found in literature. Abstracting from a very

detailed problem description is possible by weakening or removing fact constraints. Since fact constraints are never violated, all solutions to an abstracted problem description still satisfy the detailed problem description. Other abstraction mechanisms involve the creation of a different set of problem variables that abstract from details. Leaving out details can have an adverse effect. Efficient solutions that are based on specific properties will not be designed, if those properties are not described. More general solutions that ignore specific properties are possibly less efficient.

We propose a two-level problem specification. The first part of the specification consists of an abstract version of the scheduling problem. This contains all goal constraints, the objective function, and the problem variables that are necessary to specify them. Chapter 2 contains the abstract problem specifications that are used in this thesis. The second part of the specification contains application-specific details. Additional information is specified by adding fixed problem variables and fact constraints. A promising way of tailoring solutions with this additional information first selects a solution to the abstract problem and then optimizes it for the specific situation. For example, this approach has been followed in the scheduler specification and design in section 3.1.2.

3.1.1 Candidates for detailed specifications

The set of transactions that has to be executed by the database is the input of the database scheduler. This input is often less generic than is specified in the abstract problem specification. For example, generic transactions access arbitrary data items in arbitrary access modes. In a banking system it is highly unlikely that a bank account is written, without being read previously, a fact that is not contained in the abstract problem specification.

Most applications issue a set of transactions that satisfies some specific properties. Therefore, additional constraints on the input are a possible candidate for a more detailed problem specification. Very detailed information about the input is often hard to obtain. For example, calculating the exact execution time of a transaction in advance is intractable for generic transactions. This is related to the halting problem, see [61]. Even for smaller classes of transactions analysis is hard, the execution is influenced by too many factors to be analyzed exactly. Still, a worst-case analysis of the execution time of a certain class of transactions is a common approach, and it offers sufficient information to enable hard real-time guarantees. There are several ways in which additional fact constraints can be specified that provide information about the input that the application sends to the database.

Information about individual transactions. This requires that the transaction set is known in advance. Information about each individual transaction is specified. For example, the access pattern of transactions is known. This includes information about the set of accessed data items of each transaction, the type of accesses that are performed, and the order in which these accesses take place. Such information is hard to obtain and requires a thorough analysis before it can be used for design. These detailed specifications are too large to comprehend without additional abstraction.

Transaction type restrictions. If it is known that the set of possible transactions that is issued is limited to a set of specified parameterized types, the wide range of all possible transactions is reduced. A concrete example is information like "all transactions access the database according to an acyclic access graph". Although this information is very generic, when it is available, it is ensured that two phase locking will not deadlock!

Statistical information. Information about percentages and distributions are available. During actual executions these percentages might slightly differ, but they specify the general trend. For example, if it is known that "95% of all data accesses are addition operations", the optimization of the

addition operation will result in large performance gains. Since statistical information gives probabilities executions with deviating percentages can occur. This has not been taken into account in the specifications in this thesis.

Existential information. Information about the existence of certain characteristics is known. For example, "there exists more than one transaction that can write to data item X". This information makes it impossible to apply the single-writer principle to the transaction that writes X.

Apart from the set of transactions that is input to the scheduler, also the platform on which the scheduler should be implemented can be specified in more detail. Information about the relative speeds of computation, communication and synchronization is important for effective overhead reduction.

3.1.2 Example: statistical information used for optimization

We give an example of a targeted problem specification. An application needs a database that handles a large volume ($\geq 95\%$) of small transactions that write one data item, and a very low volume ($\leq 5\%$) of maintenance transactions that write an unknown number of data items. Both types of transactions read an unknown number of data items. The average maintenance transaction duration is much longer than the average single-write transaction. The intention is to use a distributed architecture. A given transaction allocation mechanism ensures that transactions that write only one data item, execute at the site where this data item is stored.

Specification. The specification first describes the generic scheduling problem, combining a real-time objective function with conflict serializability. In the next part, the problem-specific details are contained. The single-write and local-write property of 95% of all transactions is specified. The relation between execution durations of the transaction types is omitted, and will not influence our design.

platform: distributed	page 22
data: non-temporal, distributed	page 22
transactions: real-time, conflict-serializable, unannounced, deadlines	page 24
objective function: minimize deadlines missed	page 31

Additional detailed description.

Fixed Variable	**Domain**	**Description**
Sw	$T \rightarrow$ Boolean	Announces single-write property.

We introduce two predicates that describe the single-write property, and the local write property of transactions. With these predicates the additional fact constraints are specified.

$$singlewrite(t) \equiv$$
$$|[op | op \in OPS \wedge trans(op) = t \wedge Amode(op) = \text{write}]| = 1$$
$$localwrite(t) \equiv$$
$$\forall op \in OPS : trans(op) = t \wedge Amode(op) = \text{write} : local(op)$$

Fact constraint	Description
$\forall t \in T : Sw(t) \Leftrightarrow singlewrite(t)$	Relation between announcement and actual data access.
$(\sum t \in T : singlewrite(t) : 1)/\lvert T\rvert \geq 0.95$	Percentage of transactions that have the single-write property.
$\forall t \in T : singlewrite(t) \Rightarrow localwrite(t)$	All single-write transactions are local writers.

Design. Each site S has a separate scheduler process that manages all locks of transactions at that site, and manages all data items stored at that site. First, a solution to the generic scheduling problem is selected from literature. We choose distributed two phase locking that uses the two phase commit protocol (see appendix C), extended with earliest deadline first scheduling [59]. Deadlocks are prevented by the wait-die strategy, explained in appendix C. Although possibly not optimal for our specific case, these algorithms are widely used and well understood, and suffice for this example.

Next, the solution is tailored to the specific scheduling problem. The Sw boolean divides the transaction set into single-write transactions and maintenance transactions. Since the single-write transactions are the majority of all transactions (over 95%), successful optimization of their execution will result in significant performance improvements.

Design decision 3.1 *The scheduler will be optimized for efficient treatment of single-write transactions, since these are pre-dominant.*

Since single-write transactions only write local data items, no distributed commit protocol like two phase commit is required. Therefore, the commit phase of single-write transactions is very short, as no inter-site communication is necessary. On average there are more read locks than write locks on data items, since single-write transactions hold at most one write lock. This effect can be strengthened by delaying the write of single-write transactions to the end of the transaction, buffering the value in a shadow-copy. The write lock of single-write transactions t is requested just prior to t's commit. Since no two phase commit protocol is necessary, the single-write transaction can directly unlock the write lock after the write to main memory has been performed.

Design decision 3.2 *The write lock of single-write transactions is postponed until the commit phase, to minimize the blocking probability.*

Suppose single-write transactions read on average N data items, and the presence of maintenance transactions is not considered. As a pessimistic assumption after design decision 3.2, we assume linearly distributed lock-holding durations. Therefore, each active transaction holds on average $(N+1)/2$ read locks and $1/(N+1)$ write locks. There are on average $(N+1)^2/2$ more read locks than write locks in the system. Let lw be the probability that a data item is write locked. The probability that a single-write transaction blocks on any of its reads is $\approx 1-(1-lw)^N \approx N \times lw$. The probability that a single-write transaction blocks on its write is $\approx lw \times (N+1)^2/2$. The ratio of write-blocks and read-blocks is $(N+1)^2/2N$, which is > 2, for $N \geq 1$.

Design decision 3.3 *The scheduler will minimize blocking of write locks of single-write transactions, since this accounts for over 2/3 of all blocking of single-write transactions.*

The write lock on data item X of a single-write transaction t is the last lock that is requested prior to t's commit. We investigate what happens if t is allowed to write, regardless of read locks that are placed on X.

Suppose item X is read locked by a transaction q. Transaction q reads item X only once (see chapter 2). Two scenarios remain: (1) q reads X before t writes it, or (2) vice versa. We require that transaction q reads the data item at the moment that q requires the read lock. This is a common assumption, often found in distributed databases, that does not reduce the generality of the database. Due to this assumption, only scenario (1) is left. Consequently, transaction t is serialized after q. Therefore t's locks should not be released until q is releasing its locks. Furthermore, the value of X, written by t, should be read until q releases its locks.

Transaction t can prevent read access to X by placing a write lock. Ordinarily, the write lock request would block, since q has previously read locked X. However, since it is ensured that q will not read X anymore, t can safely write to X, and commit. We allow t to place its write lock next to the existing read lock, without being blocked. The scheduler has to ensure that other transactions p do not interfere with the execution of q and t. This is accomplished by releasing the locks of p and q after *both* transactions have finished execution. Hence, when transaction t commits and terminates, t does not release any of its locks. Instead, when q unlocks X, it detects the write lock of t that is first in the lock queue. Transaction q unlocks all locks that were held by t. This ensures 2PL behavior. If data item X is read locked by more than one transaction $q_1 ... q_k$, the last transaction q_i that unlocks X also unlocks t's locks.

The delayed unlock rule effectively eliminates all blocking of write locks of single-write transactions by read locks. However, the lock holding duration is not reduced, and thrashing (saturation of the database with held locks) can occur if the delayed unlock rule is applied to a large number of single-write transactions.

Design decision 3.4 *Blocking of write lock requests of single-write transactions by read locks is eliminated by using the delayed unlock rule defined above.*

This completes the design of the optimized scheduling algorithm. Interestingly, Thomas' write rule [13] can be applied to eliminate *all* blocking of write lock requests of single-write transactions. To keep the example as short as possible, this optimization has not been applied.

Algorithm. Algorithm 3.1 describes the optimized two phase locking scheduler. Instead of a standard implementation of locks, the locking procedures are contained explicitly in the algorithm. Transactions access the database by calling the read and write procedures. When a transaction finishes execution, it invokes the commit procedure. One-write transactions finish their execution by calling the write procedure. These procedures are executed in atomic sections. This requirement can be weakened, but serves to keep the example short.

Each site S manages the locks on data items stored at site S by maintaining a set of (accessmode, transaction-id) tuples for each data item X, called $Lockset[X]$. Also, a lock-queue $Q[X]$ is associated with each data item. For each transaction t, $AccessSet(t)$ is the set of held locks. Each transaction t has a unique deadline $Dl(t)$, used in the wait-die procedure. The wait-die procedure checks t against transactions that hold the lock on X, or are enqueued in the lock-queue for X. Wait-die is described in detail in appendix C. It guarantees that cyclic locking does not occur.

For brevity, algorithm 3.1 does not consider the distributed nature of the database. In principle, whenever a transaction calls a read or write procedure, this is a remote procedure call if the data access is remote. Similar, whenever a remote data item is unlocked, the unlock is a remote procedure call.

The read procedure checks the lock status to decide whether the requesting transaction t can immediately read item X. If the lock queue is not empty, or a write lock is placed on X, t's requests is enqueued. The write procedure allows one write transactions t to place a write lock on X, even in

```
read(t,X)
    wait-die(t,X,read)
    if ¬Q(X).empty() ∨ ∃(write, q) ∈ LockSet[X] then Q(X).enqueue(read,t)
    else LockSet[X]+ = (read, t)
        return value[X]

write(t,X,v)
    wait-die(t,X,write)
    if Sw(X) ∧ ∄(write, q) ∈ LockSet[X] then Lockset[X]+ = (write, t)
    else if ¬Q[X].empty() ∨ LockSet[X] ≠ ∅ then Q[X].enqueue(write,t)
        else LockSet[X]+ = (write, t)
    if Sw(X) then value[X]:=v
                if ∄(read, q) ∈ LockSet[X] then unlock AccessSet(t)
                return { committed } to transaction t
    else store (X,v) in shadow table

commit(t)
    if not Sw(t) then two-phase-commit(t)

unlock(t,X)
    Lockset[X]− = {(read, t), (write, t)}
    if ∃q : Lockset[X] = {(write, q)} then unlock AccessSet(q)
    if Lockset[X] = ∅ then grant locks to waiting transactions in Q[X]
```

Algorithm 3.1: OPTIMIZED 2PL SCHEDULER

the presence of read locks. Maintenance transactions are blocked as normal. Furthermore, one write transactions t commit and write their value if the lock is granted. Unlocking is delayed if one write transaction t placed a write lock when a read lock is already placed. The unlock procedure checks that delayed unlocks are performed when appropriate. When a data item X becomes available, one or more transactions from the lock queue are granted the lock. This has not been described in detail, to keep the example short. The commit procedure invokes the two phase commit procedure (see appendix C) only for maintenance transactions.

Correctness. The example focuses on the design of the optimized algorithm, and hence the correctness proof will be short and informal. If required, it can be used as the basis of a more rigorous proof.

It is shown that the schedules that can be generated by the optimized scheduler are conflict equivalent to schedules that can be generated by the two phase locking scheduler. A schedule satisfies two phase locking behavior if the following requirements are met: 1) each data access to a data item X by a transaction t is preceded by a matching lock operation on X, and followed by a matching unlock operation on X. 2) The precondition for a lock operation on a data item X is that the data item is not locked in a conflicting mode. 3) If transaction t locks a data item X, transaction t has not yet performed an unlock operation on any data item Y. If the execution of a transaction t follows these rules, t holds locks on all items that it accesses, just prior to its first unlock operation.

Schedules that are generated by the optimized scheduler satisfy requirements 1) and 3). The only difference between the optimized scheduler and normal two phase locking is the write(t, X, v)

of single-write transactions t. It can ignore the read locks on X that are currently possessed by transactions $q^1...q^k$, thus violating requirement 2. We show that a conflict-equivalent schedule exists that does not violate requirements 1, 2 and 3. Consider data access order S, extended with lock and unlock operations, generated by the optimized scheduler. Consider the sub-sequence that relates to the access of X by transactions t, $q^1...q^k$.

$$Rl_{q^1}(X)R_{q^1}(X)...Rl_{q^k}(X)R_{q^k}(X)....Wl_t(X)W_t(X)......Ul_{q^i}(X)....Ul_{q^j}(X)Ul_t(X)...Ul_t(Y)$$

Since the read lock grant and the read of a transaction occur in an atomic section, it is ensured that all reads of $q^1...q^k$ are performed before $Wl_t(X)$. Furthermore, all locks of t are released after $q^1...q^k$ have unlocked X, in arbitrary order. Between $Rl_{q^1}(X)$ and $Ul_t(X)$, no other transaction p wrote to X, since write locks are exclusive. Between $Wl_t(X)$ and $Ul_t(X)$, no transaction read X, since no new read locks on X are granted, and $q^1...q^k$ have already read X, and transactions read each data item at most once. Finally, $Ul_t(X)$ is the first unlock operation of t.

Therefore, an extended data access order S', equal to S except for the position of $W_t(X)$, is conflict equivalent to S as long as $W_t(X)$ occurs after $Wl_t(X)$ and before $Ul_t(X)$. Let S' be an extended data access order that is conflict equivalent to S, where $W_t(X)$ is positioned just prior to $Ul_t(X)$.

$$Rl_{q^1}(X)R_{q^1}(X)...Rl_{q^k}(X)R_{q^k}(X)....Wl_t(X)......Ul_{q^i}(X)....Ul_{q^j}(X)W_t(X)Ul_t(X)...Ul_t(Y)$$

Finally, note that changing the position of lock and unlock operations preserves conflict equivalence. By moving $Wl_t(X)$ between the last unlock operation of X by transactions $q^1...q^k$, and $W_t(X)$, the execution of t and $q^1...q^k$ satisfy the two phase locking requirements.

$$Rl_{q^1}(X)R_{q^1}(X)...Rl_{q^k}(X)R_{q^k}(X)..........Ul_{q^i}(X)....Ul_{q^j}(X)Wl_t(X)W_t(X)Ul_t(X)...Ul_t(Y)$$

By applying transformations of this kind to the writes of all single-write transactions that ignored read locks, the extended data access order S of the optimized two phase locking scheduler is shown to be conflict equivalent to an extended data access order S' that satisfies the two phase locking rules. Hence, S is conflict serializable.

Conclusion. The execution of single-write transactions has been optimized considerably, by reducing the blocking probability. Write locks of single-write transactions are only visible to other transactions if the delayed unlock rule is invoked. This only occurs if the write of a single-write transaction encounters a conflicting read lock. Since there are less write locks set in the system at any time, the blocking probability of reads is reduced. Write lock requests of single-write transactions no longer block on read locks, due to the delayed unlock rule. Lock holding times have not been reduced, but the real-time response of single-write transactions is improved. These optimizations were possible because additional information about transactions was available.

3.2 Concurrency

Schedulers allow certain data access orders and prevent other orders from occurring. Optimally only those access orders are prevented that violate goal constraints. In practice schedulers are more restrictive, they only allow a subset of all correct data access orders. The set of allowed data access orders of a scheduler is described by free problem variable $ADAO$.

Suppose that a transaction issues a certain data access request. The scheduler has to delay this data access if immediate execution of the data access would result in the data access order

$Aorder \notin ADAO$. Therefore, a large set $ADAO$ has a low probability of delaying transactions. This observation is the rationale behind the following measure of concurrency.

Definition 3.1 *A measure of concurrency is the number of data access orders that are allowed by the scheduler.*

It is recognized [75] that the measure is not very suitable to compare the performance of two schedulers S, S'. The comparison is strengthened by trying to proof either $ADAO_S \subset ADAO_{S'}$, or vice versa. Basically, if neither can be shown, it is not clear which scheduler offers better concurrency.

In general, the probability that a data access request is delayed is inversely proportional to the concurrency that is allowed by the scheduler. For specific applications the relation is more subtle. At any point in time, a finite sequence of data accesses has taken place. This sequence PAO is a prefix of the final access order $Aorder$. A transaction t is delayed if it requests a data access op, such that $PAO; op$ is no longer a prefix of any sequence in $ADAO$.

Specific applications do not request arbitrary data accesses op. Instead, the requested data accesses follow a specific pattern. Typical examples of such patterns are: transactions always read data before writing data, transactions access a small fraction of the entire database, or transactions have a fixed order in which they access data (for example an alphabetical order on their identities).

Information about these data access patterns is captured by the fixed problem variable $RDAO$ that represents the set of all possible *requested* data access orders. If $RDAO \subset ADAO$, all possible requested access orders are allowed access orders, transactions can execute freely. Whenever $RDAO \not\subset ADAO$, it is possible that the scheduler has to execute data accesses in a different order than they are requested. This requires that certain data accesses are delayed.

From the discussion above it should be clear that our measure of concurrency is not accurate if additional information is known about the possible access request patterns. A more precise measure would be the size of $RDAO \cap ADAO$, counting the number of data access orders that can actually occur, taking into account the scheduler that is used, and the application that accesses the database. Even more precise measures would also count the number of re-orderings that are necessary to fit a requested data access order into $ADAO$. This is not further investigated in this thesis. We give a number of examples of types of concurrency that can be offered by a scheduler.

Unbounded concurrency. If all interleavings are possible, a transaction can execute whenever a processor is available. All data accesses are accepted, no scheduler overhead needs to be generated. Unbounded concurrency also accepts data access orders that violate serializability constraints. Therefore, unrestricted transaction execution can lead to unexpected and unwanted results if concurrent transactions conflict.

Theoretically optimal concurrency. As long as only serializable [75, 100] interleavings are possible, database consistency is ensured. Transactions are only delayed if immediate execution would destroy database consistency. Let PAO be the finite sequence of performed data accesses, and op be a requested data access. It has been shown [75] that recognizing whether $PAO; op$ is a prefix of a conflict-serializable data access order is an NP-complete problem.

Approximation: two phase locking. The 2PL approximation uses a set of simple rules to decide whether a data access order is acceptable. These rules can be checked in $O(k)$ time for a transaction that performs k data accesses, but without extensions deadlocks can occur. Additional rules to solve

deadlocks generate more overhead. At least $k+2$ synchronization points are necessary for a transaction that performs k data accesses. The following example shows that 2PL is less concurrent then theoretically optimal concurrency. A correct interleaving of transactions t: $R_t(X)\ W_t(Y)$, and q: $W_q(X)$ is $R_t(X)\ W_q(X)\ W_t(Y)$. This interleaving cannot be generated by 2PL, transaction t will not release the lock on X before it has written Y. Therefore, 2PL will sometimes delay transactions, even if their immediate execution is serializable. To show that 2PL does allow some interleavings of conflicting transactions, a 2PL interleaving of $R_p(X)\ W_p(Z)$ and $R_r(Y)\ W_r(Z)$ is $R_p(X)\ R_r(Y)\ W_p(Z)\ W_r(Z)$. The reads of X and Y are performed concurrently.

Approximation: trivial concurrency. No interferences occur between non-conflicting transactions, so any interleaving of two non-conflicting transactions is correct. Trivial concurrency only allows two transactions p and q to execute in an interleaved fashion, if they are non-conflicting. Otherwise, p and q execute sequentially. Deciding whether a transaction is non-conflicting with concurrently executing transactions takes $O(k)$ time for a transaction that performs k data accesses. Enforcing trivial concurrency requires a synchronization point at the start and end of each transaction.

Trivial concurrency allows less interleavings than 2PL. For example, $R_p(X)\ R_r(Y)\ W_p(Z)\ W_r(Z)$ is an allowed 2PL data access order, but it is rejected by trivial concurrency. Since p and r conflict over Z, only sequential orderings of p and r are accepted by trivial concurrency.

One of the research goals in database scheduling has been to design database schedulers that offer a lot of concurrency. Such schedulers can benefit from multi-processor architectures, as more transactions are allowed to execute in parallel, amongst other benefits. However, there are other factors apart from the concurrency offered by the scheduler, that influence the scheduler choice.

3.3 Scheduler overhead

The response time of a transaction consists of execution of the transaction procedure, waiting time, and time that is spent on the execution of scheduler procedures. Scheduler overhead consists of two parts, decision overhead and enforcing overhead. The scheduler selects an access order $Aorder$ from $ADAO$ that closely matches the access request order. This selection process takes time and is called *decision overhead.* After deciding the data access order, the order needs to be enforced. The communications and synchronizations that are necessary constitute the *enforcing overhead.*

Decision overhead is determined by the quality of the scheduler implementation, and by the concurrency that is offered by the scheduler. For certain $ADAO$ sets, recognizing whether a certain data access leads to a data access order that still belongs to the set can be checked very efficiently and fast. For other $ADAO$ sets, the decision overhead is considerable. Recall that the decision overhead for the maximal $ADAO$ set that consists of serializable schedules is an NP-complete problem [75]. The decision overhead becomes negligible, if $Aorder$ is computed off-line.

Enforcing overhead is counted in the number of synchronization points that are used to enforce the data access order, and the number of communications that are necessary. Similar to decision overhead, enforcing overhead is determined by the concurrency of the scheduler, and the quality of the implementation. For example, 2PL allows transactions to execute partly in parallel, even if they conflict. If it is unknown in advance which data accesses will conflict, a synchronization point is required before each data access. If conflicting transactions are executed sequentially, a synchronization point at the start and end of each transaction suffices. Therefore, trivial concurrency can be implemented with two

synchronization points per transaction. Of course, inefficient implementations of trivial concurrency can add more synchronization points, but these are not required.

3.3.1 Overhead reduction techniques

In chapter 2 the three main sources of both decision and enforcing overhead are described: communication, synchronization and internal computation of the scheduler. The scheduler uses communication to gather the information that it needs for its decision procedures, and to enforce synchronized execution on different sites. Synchronization is used to gather information about the data access order that has previously occurred, and to enforce certain data access orders. Internal computation is used for the decision procedures. We investigate how the scheduler can be constructed, such that this overhead is minimized. The discussion primarily holds for on-line scheduling, but parts of it can also be applied to off-line scheduling with a dispatcher.

Matching communication patterns. In distributed platforms, an important source of overhead is inter-site communication. Scheduler communications typically involve only a few bytes, therefore the communication latency determines the amount of overhead. Scheduler communication overhead can be reduced considerably by attaching scheduling information to messages that are needed for the normal execution of transactions. This is called piggy-backing. Piggy-backing is only possible when scheduler communication and transaction communication coincide. While designing the scheduler, the communication patterns of the scheduler and the transactions should match when possible. Distributed transactions are concluded with the two phase commit protocol, to ensure atomicity. The 2PC protocol sends four sequential messages between the coordinator, and all sites that participate in the transaction. The protocol is further described in appendix C. These messages are good candidates for piggy-backing of scheduler information.

A typical example of a mismatch in communication patterns is the 2PL remote write lock. Suppose transaction t decides to write remote data item X. The scheduler requires a lock request message and a lock grant message at that time. The actual value is not written by t until t commits, so the two messages cannot be piggy-backed and count as scheduler overhead. On the other hand, the communication patterns of 2PL and transaction t match when t decides to read a remote data item. The scheduler again requires two messages, and t requires two messages to request and receive the value. The scheduler communication can be piggy-backed on the transaction communication.

Synchronization points versus concurrency. Synchronization points are another source of overhead, and the number of synchronization points can be reduced to improve scheduler efficiency. There is a trade-off here, because the number of synchronization points determines how many correct interleavings the scheduler can enforce. Suppose a transaction t does not have a synchronization point between two data accesses op, op'. The scheduler cannot delay t after op, and before op'. Therefore, if the scheduler decides to delay op', necessarily op needs to be delayed as well. All schedules that do not delay op, and do delay op' cannot be enforced. Should a synchronization point exist between op and op', such schedules can be enforced.

In order to reduce the number of synchronization points, it should be checked that existing synchronization points are actually necessary to enforce the required level of concurrency. Superfluous synchronization points can be removed. As an example, we previously concluded that trivial concurrency can be enforced with two synchronization points. Suppose each transaction has a synchronization points at the start and finish of its execution. Trivial concurrency is enforced by the following rule. If transaction p conflicts with executing transaction q, p is delayed in its first synchronization

point until q reaches its last synchronization point. Schedulers that enforce trivial concurrency with more than two synchronization points can be optimized.

Optimizing internal computation. Internal computation of the scheduler can be costly. It is important to implement the scheduler using suitable data structures and fast algorithms that operate on them. Furthermore, the number of transaction restarts should be minimized, as each failed transaction execution counts as overhead. There is a relation between the time complexity of the internal computation of the scheduler, and the concurrency that is offered by the scheduler. It goes beyond this thesis to exactly specify the relation, but we have the following remarks.

1. The time complexity of a scheduler that recognizes all serializable schedules is exponential, provided that no-one proves that P=NP [75].
2. The time complexity of a scheduler is bounded from below by the size of its input.
3. The time complexity of a scheduler that recognizes all sequential schedules is $O(1)$ for each transaction.
4. The time complexity of a scheduler that recognizes 2PL schedules is $O(k)$, for k data accesses per transaction. This does not include deadlock detection.

Item 2 gives a weak lower bound on all scheduling algorithms, a bound that is sharp for most on-line scheduling algorithms. Items 3 and 4 are examples of schedulers whose time complexity matches the lower bound, given by item 2. A scheduler that generates sequential schedules only needs the identity of each transaction to define an order. Hence, its time complexity is $O(1)$. A two phase locking scheduler needs all data accesses of a transaction to schedule it. Hence, its time complexity is at least $O(k)$, for k accesses. The lock-queues that are used in two phase locking have $O(1)$ time complexity for each lock request, so indeed the complexity of 2PL schedulers is $O(k)$.

3.4 The overhead/concurrency tradeoff

A tradeoff exists between the concurrency that is offered by the scheduler, and the scheduling overhead. Increasing the concurrency of a scheduler usually results in an increased scheduler overhead. This relation is not linear or even monotonic, it is very well possible to design a scheduler with little concurrency and high overhead. However, the generic relation holds, both for decision overhead and enforcing overhead.

Item 2 from paragraph "Optimizing internal computation" shows that the decision overhead grows at least linearly with the size of the scheduler input. The amount of information that is used by the scheduler determines what schedules can be recognized. For example, to schedule a data access op of transaction t, two phase locking schedulers use information about the availability of the accessed item $Daccess(op)$. Which other transactions q request use of $Daccess(op)$, and which other accesses they make, is not input to the scheduler. Therefore, the unmodified two phase locking scheduler suffers from deadlocks, as wait-cycles cannot be prevented without additional information. In order to overcome deadlocks, additional information is necessary. This increases the time complexity of the algorithm. Alternatively, schedules that potentially lead to deadlocks have to be rejected. This reduces the concurrency.

In paragraph "Synchronization points versus concurrency" the relation between concurrency and the required number of synchronization points is mentioned. To obtain more concurrency, data access orders have to be allowed that require a large number of synchronization points. Again, more concurrency leads to more overhead.

So there is a choice, either select a scheduler with high concurrency at the cost of increased overhead, or a lightweight scheduler that offers little concurrency. When to select which scheduler? The tradeoff can be decided by the application specific information that is contained in the problem specification. We distinguish two parameters that determine the tradeoff decision.

Transaction execution duration. The influence of scheduling overhead depends on the execution duration of transactions. We show this with the following example. Suppose the scheduler needs 10 microseconds to deal with each transaction. If a transaction executes for 1 second, the scheduler overhead is at most 0.0001% of the response time, almost negligible. If transactions are short, and execute 10 microseconds (actually quite long for simple lookup transactions), scheduler overhead is up to 50% of the total response time. If the overhead is a significant percentage of the total response time of transactions, it becomes important to try to reduce it. Therefore, the average length of transactions that are issued by the application gives a good indication whether optimizing scheduler overhead is important, or not.

Total conflict probability. The data access request orders (see section 3.2) that are issued by the application determine the usefulness of a scheduler with high concurrency. In order to characterize data access request orders with one number, we introduce the total conflict probability. The *conflict probability* is defined as the probability that two arbitrary transactions conflict. The *total conflict probability* is defined as the probability that a transaction encounters a conflict during its execution.

If the conflict probability is low, most requested data accesses do not lead to interferences. As long as a transaction does not encounter a conflict, even small $ADAO$ sets like trivial concurrency are sufficient to execute transactions without delays. If the conflict probability is high, a large number of data accesses lead to interferences between transactions. A scheduler that allows a small set of data access orders can only select a poor $Aorder$. Since more complex schedulers can allow a larger class of data access orders, they can probably find better data access orders that incur less transaction delays.

For applications with a high total conflict probability and long average transaction durations, a scheduler that offers a high degree of concurrency should be selected. If the conflict probability is low, and the average transaction duration is short, overhead becomes the limiting factor. A lightweight scheduler is preferable over a complex scheduler, even if it offers less concurrency. Chapter 4 presents a typical low-conflict environment where simple schedulers outperform schedulers that offer a lot of concurrency.

3.5 Available information versus time & quality

Scheduling decisions cannot be made before the information necessary for the decision is available. Although quite trivial, this is easily overlooked, and has a big impact on the scheduler design. The moment that a scheduling decision is made determines the manipulation options that are available to enforce the decision. In general, early scheduling decisions are easier to enforce than late decisions. Therefore it is good practice to take scheduling decisions as soon as all necessary information is available.

3.5.1 On-line scheduling decisions

A question that has to be answered for each real-time database scheduling problem is: "When is sufficient information available to make the scheduling decision?". Different information is necessary for different types of concurrency. Therefore, the concurrency that should be offered by the scheduler determines when scheduling decisions can be made. There is a balancing point here, schedulers have to delay scheduling decisions until the required information is available. However, if they delay their scheduling decisions too long, the generated schedule cannot be enforced anymore. Therefore, the moment that scheduling information becomes available also determines the optimal schedule that can be generated and enforced.

For example, we investigate static locking, as described in chapter 6 and optimistic concurrency control, as described in chapter 4. Both offer trivial concurrency. Trivial concurrency can be enforced with only two synchronization points for each transaction t. So when do the schedulers make the scheduling decisions, related to t? This depends on the availability of scheduling information. Suppose no extra scheduling information is made available by the applications. Information about execution times and access patterns of transactions is observed by the scheduler, unannounced or fait accompli information. Checking whether p conflicts with committed transactions can only be completed after p's access pattern is known, at the second synchronization point. At this point, the manipulation options are limited, either p is committed or p is restarted. Therefore, without additional knowledge, and with the requirement that there are only two synchronization points for one transaction, the resulting scheduler has to be an OCC variant (see also [53] and chapter 4).

If the access patterns of transactions are announced by the application at the first synchronization point, the scheduling decisions can be made at that time. The advantage of this early decision making is that the scheduler has an extra manipulation option, it can decide to delay transactions. A so-called static locking scheduler (see also chapter 6) results. The additional information allows the scheduler to waste less resources. The restart mechanism of the OCC scheduler occupies the processor with useless executions. The static locking scheduler avoids this by preventing transactions to occupy the processor, until successful execution is ensured.

3.5.2 Off-line scheduling decisions

Real-time requirements increase the complexity of scheduler design by limiting the available on-line execution time. Scheduling algorithms have to execute under stringent timing requirements. Their time-complexity has to be low, and they have to be implemented as efficiently as possible, to meet the real-time requirements on a low cost platform. This complexity can be reduced by pre-computing scheduling decisions off-line. During on-line computation, the pre-computed scheduling decision can be retrieved by a fast lookup, instead of making the time-consuming scheduling decision again.

If the entire schedule is computed off-line, the functionality of the on-line scheduler (called the dispatcher) is only to enforce the pre-computed schedule. No schedule decisions are made on-line. Such systems are not very flexible, their functions are completely defined in advance, as well as the moment that these functions will be performed. It can be hard to provide sufficient off-line information, such that the entire schedule is generated off-line. If insufficient scheduling information is available to compute the entire schedule, scheduling decisions have to be taken on-line. However, it can still be possible to pre-compute some scheduling decisions, thus reducing the on-line scheduling overhead.

```
Read(t,X);
    if CI[t, X] then Sch-read(t,X) else Dirty-read(X);

Write(t,X,v);
    if CI[t, X] then Sch-write(t,X,v) else Dirty-write(X,v);
```

Algorithm 3.2: OFF-LINE MODIFICATION OF ARBITRARY SCHEDULERS

Example: combining off-line and on-line scheduling

A real-time database scheduler has to be constructed, which guarantees conflict-serializability, and executes transactions within $[est, dl]$ intervals. The set of transactions T, and their $[est, dl]$ are known in advance. Unfortunately, no worst-case execution times of transactions are available. If transactions miss their deadline, they have to be aborted. A worst-case approximation of the access set of each transaction is available off-line, a transaction t will not use any data item outside set $Maxuse(t)$.

These restrictions are very common in databases. The dataset that is stored in databases is almost always divided into tables. If transactions are generated using SQL-like query languages (see [51]), the tables that will be accessed are known in advance. For example, it might be known that a transaction uses data items from a "customer" table, and it will not access the "supplier" table. The scheduler should minimize the percentage of transactions that miss their deadlines. A centralized architecture is available for on-line execution.

Specification.

platform: centralized	page 21
data: non-temporal	page 22
transactions: firm-real-time, conflict-serializable	page 24
objective function: minimize deadlines missed	page 31

Additional specification of details.

Fixed Variable	Domain	Description
$Maxuse$	$T \to \mathcal{P}(DataSet)$	Transactions access data from this set.

Fact constraint	Description
$\forall op \in OPS: Daccess(op) \in Maxuse(trans(op))$	Transactions do not access data outside their $Maxuse$ set.

Design. Off-line scheduling of the transaction-set is inefficient, since no worst-case execution times are known. Solutions that take all scheduling decisions off-line will ignore the extra information that becomes available on-line. This results in a high percentage of transactions that miss their deadline, or an inefficient use of resources.

Suppose there is an on-line scheduler Sch that solves the generic scheduling problem. We transform it to a scheduler that includes both off-line and on-line computation, and has less on-line overhead. From the $Maxuse$-sets, and the est, dl pairs it can be computed whether a data access to X can

lead to interference. Suppose t accesses X in interval $[a, b]$. If no other transaction t' that overlaps in time has $X \in Maxuse(t')$, the access of t is interference free. We define lookup table $CI[t, X]$ which determines whether access to X by t can lead to interference. Table CI is computed off-line from the $Maxuse$-sets.

$$\text{time-disjunct}(t, t') = dl(t') < est(t) \vee dl(t) < est(t')$$
$$CI[t, X] \equiv \forall t' \in T : t' \neq t \Rightarrow \text{time-disjunct}(t, t') \vee X \notin Maxuse(t')$$

The interference free predicate that is specified above can be weakened, if checks are formulated that show whether a possible conflict between two transactions can lead to non-serializable schedules, or not. This has not been investigated further.

The off-line scheduler modifies the on-line scheduler using the CI-function. Usually, data access is monitored by the on-line scheduler by adapting the access procedures for reading and writing. Procedures "Dirty-read(X)", and "Dirty-write(X,v)" are introduced, that directly read and write data items, without bothering with the on-line scheduler. The on-line scheduler Sch is invoked by calling Sch-read(t,X) and Sch-write(t,X,v). The modified read and write access of transactions are given in algorithm 3.2.

If there are relatively few overlaps between the $Maxuse$-sets, or transactions are often separated in time, the optimization yields a significant performance improvement. Since the CI-function is local to each transaction, it can be implemented efficiently using $O(1)$ lookup tables. The computational overhead of this optimization is marginal.

The optimization can be carried even further, if the CI function evaluates to true for the entire $Maxuse$-set of a transaction t, the lookup can be omitted, and the scheduler is never invoked, no on-line scheduler overhead is generated at all for transaction t. If $CI[t, X]$ is false for most of its domain, an off-line heuristic can decide that the optimization is not useful for the transaction. The off-line scheduler can decide not to apply this optimization, thus avoiding the overhead that is introduced by the lookup.

Conclusions

When and what scheduling information becomes available determines for a large part the scheduler that will be designed. The choice between off-line and on-line scheduling is governed by the availability of information, and it influences the overhead of on-line schedulers. Information about the availability of scheduling information at each point during the execution should be part of every problem specification that concerns real-time database scheduling.

Chapter 4

Scheduling in low-conflict environments

A common feature of database environments is a low conflict-probability. Databases are often large, with millions of data items, and the probability that transactions conflict over the same data items can be close to zero. The real-time database scheduling problem that is defined and studied in this chapter is inspired by scheduling problems from the telecom environment. Mnesia is a real-time database that has been developed by Ericsson, especially for telecom applications [71, 62]. It is equipped with a two phase locking scheduler that uses a wait-die strategy to avoid deadlocks. This strategy reduces the concurrency offered by the scheduler, but adds little scheduler overhead. Furthermore, each transaction is committed using the two phase commit protocol. The Mnesia scheduler has been replaced with the scheduler that is designed in this chapter, to obtain test results. Parts of this chapter have been previously published [18].

The chapter is organized as follows. An environment study in section 4.1 extracts relevant information that will be part of the problem specification. A number of assumptions are made to formalize the arguments, and obtain the problem specification that is given in section 4.2. Section 4.3 explains the scheduler design, and shows how the issues from chapter 3 are used to formulate and make design decisions. The resulting algorithm is presented in section 4.4, analyzed in section 4.6 and test results appear in section 4.8. The results of the chapter are summarized in 4.9.

4.1 Environment study

The telecom environment is large, with several different applications that have completely different characteristics. The application under consideration has been designed to use a distributed database, and has the following characteristics that influence the scheduler design.

Data access of transactions. The database consists of a large number of data items, and transactions access only a few. Therefore, the probability that two transactions access the same data items is low. The exact conflict probability is unknown, especially since the system that is under consideration is still in its experimental phase. Educated guesses place the conflict probability between 0 and 0.0001. We assume that an upper-bound on the conflict probability b is 0.0001.

Transaction durations. Transactions require little computation. Therefore, the overhead posed by the database is a significant part of the transaction response-time. In test-runs with a 2PL scheduler, overhead amounts to 80% of the response time. The importance of overhead depends heavily on the costs of communication and synchronization, compared to the internal computation of transactions. Typically, only a few operations are performed on the value of a data item that is read. We assume

that the amount of work that a transaction performs is at most linear in the number of accessed data items, and that it takes at most 1 millisecond of internal computation for each data item that it writes (this matches the results from test 4.3b).

Data size. Typically, only a few kilobytes of information is stored in a data item. Therefore, the communication costs of transferring the value of data items between sites are dictated by the latency of the network, rather than the throughput.

Transaction arrival process. Transactions arrive unannounced, at a high rate, with exponentially distributed inter-arrival times. The total number of transactions that arrive is more than can be handled by one processor. It is desirable that the database is scalable, i.e. the number of transactions that the database can execute in one time unit should be a linear function of the number of processors. No assumptions are made about the arrival rate of transactions, and the arrival rate is not announced by the application. We do assume that the conflict probability is bounded by a fixed constant b (specified above), independent of the transaction arrival rate.

Access types. A high percentage of transactions is read-only, or perform addition operations. Concurrent read accesses to the same data item do not conflict, and concurrent addition operations can also be executed in an arbitrary order. In this chapter we have not considered optimization of addition operations. We assume that over 90% of all data accesses are read-accesses.

Distributed data access. The database is tailored such that transactions have a high probability of accessing local data items. This is realized by replicating hot-spot data that is used for lookup only, and by a transaction distribution mechanism that dispatches transactions to the site where their data is stored. We assume that over 90% of all data accesses are local.

Real-time requirements. No explicit deadlines are generated for transactions. Instead, the database should try to minimize the response time of transactions.

We make additional assumptions to characterize the platform on which the scheduler is implemented. These figures are derived from actual measurements in a network of SPARC 5 workstations. One-way communication between sites takes on average 1.5ms: $Mlatency = 1.5$ms. Synchronization implemented with local message passing takes on average 0.13ms: $Msync = 0.13$ms. We assume that $Cpower = 1$, which means that the work of transactions equals their execution time.

4.2 Specification

The generic scheduling problem is well-defined. Transactions arrive unannounced at the database. They access non-temporal data that is distributed over a distributed platform. The database should guarantee conflict-serializability, and minimize the response time of transactions.

Generic scheduling problem.

platform: distributed	page 22
data: non-temporal, distributed	page 22
transactions: real-time, conflict-serializable, unannounced	page 24
objective function: minimize average response time	page 30

Additional detailed information.

Fact constraint	Description
$\sum_{t,q\in T:transconflict(t,q)} 1/\lvert T\rvert^2 \leq 0.00001$	Conflicts between transactions are rare.
$W(t) \leq \lvert Aorder(t)\rvert \times 1ms$	Transactions are short.
$\sum_{op\in OPS:Amode(op)=read} 1/\lvert OPS\rvert \geq 0.9$	Most data access is read access.
$\sum_{op\in OPS:local(op)} 1/\lvert OPS\rvert \geq 0.9$	Most data access is local.

4.3 Design

Recall that algorithm 3.1 solves a problem that is quite similar to the problem that is specified in section 4.2. However, algorithm 3.1 uses the fact that transactions *announce* their one-write property. This announcement is not available here, and hence algorithm 3.1 cannot be applied.

The objective of the scheduler that has to be designed is to minimize the transaction response time. The transaction response time depends on the trade-off between the overhead that the scheduler generates, and the processor utilization that the scheduler realizes (see chapter 3). How this trade-off should be made depends on the conflict probability, and the average transaction length. The conflict probability is very low, suggesting that a relatively small class of schedules (like trivial concurrency) suffices to obtain near-optimal processor utilization. Secondly, transaction lengths are short, compared to the overhead that is introduced by synchronization and communication. Therefore, a significant reduction in overhead can be achieved by minimization of the scheduler overhead.

Design decision 4.1 *The scheduler overhead is reduced, at the expense of the concurrency offered by the scheduler. However, the scheduler should offer at least trivial concurrency.*

To enforce trivial concurrency, at least two synchronization points are required for each transaction: at the start, and at the end of its execution. The synchronization points are both required to determine whether a transaction t has started, before a conflicting transaction q has ended execution. The scheduler can enforce its decisions by executing transactions, delaying transactions, or restarting transactions. It can delay transactions in their first synchronization point, or restart them in their second synchronization point.

Scheduling decisions that determine whether a transaction t can execute, or has to be delayed, require that information about $Aorder(t)$ is available. However, $Aorder(t)$ is fait accompli information, so the required information is unavailable in the first synchronization point. Therefore, delaying transactions in their first synchronization point cannot be used to enforce trivial concurrency. We choose to add a restart mechanism to the second synchronization point, essentially building an optimistic concurrency control scheduler [53, 57, 55, 84, 88]. The additional overhead that results from restarts will be low, since the conflict probability is very low.

Design decision 4.2 *A synchronization point is added to the start and finish of each transaction execution. In the second synchronization point, transactions are restarted if their execution does not satisfy trivial concurrency.*

To check whether the execution of a transaction satisfies trivial concurrency, the access-set of each transaction t is administrated by the scheduler. This can be realized without synchronization or communication costs by superimposing scheduler administration procedures on access procedures. Superimposing a procedure F on procedure G means that every time G is executed, F is executed as well, and F can access the same information that G can access. Information about data access of

committed transactions is aggregated, for each data item X a time-stamp is maintained that is equal to the largest time-stamp of all transactions that wrote X. The resulting scheduler is a lightweight variant of the optimistic concurrency control scheduler that Kung and Robinson introduced [53], which we call OCC-light.

The scheduler has to operate in a distributed environment, and we did not yet check whether the communication patterns of the transaction execution, and the scheduler execution match. There are two things to consider. First, the information that is required by the scheduler has to be gathered. The access-set of the transaction t under consideration is gathered by super-imposed scheduler procedures. When transaction t finishes execution and reaches the synchronization point, this information is available locally.

If the access set of t overlaps with the access set of recently committed transaction q that executes on another site, a synchronization between the site where t executes, and where the site where q executes is required. Observe that t and q access at least one common data item. By synchronizing the site where t executes with all sites where t accessed data, a synchronization between t and q is established, since q follows the same protocol. This synchronization can be piggybacked on the two-phase commit protocol, which is used to commit/abort transactions [51], thus avoiding additional message overhead.

Design decision 4.3 *The second synchronization point of t synchronizes with all sites where t accessed data. The messages involved are piggy-backed on the two phase commit protocol.*

The synchronization of t with other transactions uses information about committed transactions q, to decide if t can commit. When t and q are both about to commit, synchronization between the two transactions becomes complicated. Suppose the scheduler decides to delay t, until q commits or restarts. Most probably, q decides to commit (conflict probabilities are very low). Therefore, with a very large probability t has to restart. Should t wait until q commits or restarts? With our main goal (low scheduler overhead) in mind, the question is solved by a simple heuristic. This heuristic also eliminates the need for a deadlock breaking mechanism.

If t and q accessed X in a conflicting way on site S, the following *restart rule* is used. If during the synchronization of t with S, it is discovered that q has synchronized with S, but is not yet committed or restarted, it is assumed that q will commit, and t is restarted. The heuristic can lead to mutual restarts, if t and q conflict on two different sites. However, it is extremely unlikely that this occurs (the probability is about 0.0001^2). By introducing a random wait period for transactions that are restarted in this fashion, life lock caused by mutual restarts is prevented.

Design decision 4.4 *Synchronization between transactions that are both in their second synchronization point is resolved by a restart rule: if p observes that conflicting transaction q started validation, p restarts after a random wait period.*

The two synchronization points of a transaction t are used to determine whether conflicting transactions committed in the interval, defined by the synchronization points. This requires that the first synchronization point synchronizes with all sites where t can access data. Such a synchronization involves a global clock, or a broadcast of messages, a significant overhead to the protocol.

We can eliminate this overhead by observing the following. The communication pattern of OCC-light fits nicely within the communication pattern of transactions. However, it does not match completely: when a transaction reads a remote data item, a value-request message, and a value message are sent. The OCC-light scheduler does not piggyback any information on these messages. By exploit-

ing the free synchronization for remote reads, the scheduler can offer more than trivial concurrency, without any additional synchronization or communication cost.

Design decision 4.5 *The free synchronization that occurs when a transaction reads a remote data item is used to eliminate the need for a global clock protocol, and to increase the concurrency offered by the scheduler.*

Instead of a global clock, each site maintains its own logical clock. Transactions t obtains a start-timestamp from the local logical clock. This synchronizes t with all local events. Whenever a transaction t reads a remote data item at site S, it uses the free synchronization to read the logical clock at site S. This synchronizes the read access of t with all events at site S.

Trivial concurrency ensures (amongst others) that the synchronization order of transactions is equal to the commit order of transactions. Note that the restart-rule of the second synchronization already solves write-write conflicts: when two concurrent transactions both write the same item, at least one restarts. Suppose that a read-write conflict occurs between t and q. Under trivial concurrency, transaction q writes item X, before t starts, or after t commits. If data item X is local, this requirement is easily checked with the two synchronization points of t. For remote reads, we relax this requirement, by observing that conflict-serializability is enforced, if q writes X before t reads X, or after t commits. This weaker requirement can be checked with the extra synchronization point that occurs when the read-request of t is handled.

Design decision 4.6 *The scheduler supports trivial concurrency for locally accessed data items, but a weaker requirement (more concurrency!) is supported for remotely accessed data items. This removes the need for global synchronization at the start of a transaction, local synchronization suffices.*

We shortly summarize the design of the OCC-light scheduler. Problems with concurrent validation, and write-write conflicts are solved by the restart-rule. Local read/write conflicts are solved using the synchronization points at the start and end of each transaction. Trivial concurrency is enforced for local conflicts. Remote read/write conflicts are solved using the free synchronization point just before the read access, and the synchronization point at the end of each transaction. This allows more than trivial concurrency for remote conflicts.

4.4 Algorithm

We implement synchronization points by introducing a single scheduler process on each site. When a transaction synchronizes on a site, it activates the scheduler on the site, and waits for response. The scheduler process at a site S acts as the *coordinator* of transactions that execute at S. Furthermore, the scheduler process at site S also manages access to the data items, stored at S. If t accessed data at site S, the scheduler process is a *participant* of t. The algorithm description consists of two parts. Algorithm 4.1 describes the procedures that directly modify the transaction execution. In algorithm 4.2 the scheduler process at each site is described by a set of messages to which it reacts, the scheduler is essentially a reactive server. The bold-face printed labels (i.e. **Ac(t,X)**) are used in the correctness proof, and are explained in section 4.5. The notation used for these algorithms is further explained in appendix B.

The execution of a transaction t is encapsulated within a procedure "Transaction". This is necessary to restart t, if its validation fails. Furthermore, each transaction t synchronizes with the local scheduler at the start, and end of its execution by invoking "Start" and "End". Data access during

```
Write(t,X,v)
    sites[t]+ = DataPlace(X)
    Ws[t]+ = (DataPlace(X), X, v)

Read(t,X)
    sites[t]+ = DataPlace(X)
    if DataPlace(X) = TransPlace(t)
    then TS:=TS[t]
        Ac(t,X) V:= value[X]
    else scheduler@DataPlace(X) ! {read request, t, X}
        receive { read value, t, X, V, TS } → skip end
    Rs[t]+ = (DataPlace(X), X, TS)
    return V

Transaction(t,Tcode)
    Start-synchronization(t)
    Result:=Tcode(t)
    if End-synchronization(t)=commit then return Result
    else return Transaction(t,Tcode)

Start-synchronization(t)
    scheduler@TransPlace(t) ! {start request, t}
    receive { start ok, TS[t]} → skip
    end

End-synchronization(t)
    scheduler@TransPlace(t) ! { commit request,t,ST[t],sites[t],Rs[t],Ws[t] }
    receive {commit result, Result} → return Result end
```

Algorithm 4.1: MODIFICATION OF TRANSACTION EXECUTION

the execution of a transaction is regulated by procedures "Read" and "Write". Each transaction t maintains the following local data structures.

Start timestamp $TS[t]$**.** This timestamp is defined in the first synchronization point, by reading the local clock. It is used for validation of local read accesses.

Accessed sites set $sites[t]$**.** This set stores the identities of all sites where t accessed data. It is used during validation.

Read sets $Rs[t]$**.** For each site $S \in sites[t]$, the set of read data items at S, with associated timestamp is maintained. The timestamp of locally read data items X equals the start-timestamp of t, which is smaller or equal than $WTS[X]$ when X was actually read: $(TransPlace(t), X, TS) \in Rs[t] \Rightarrow TS = TS[t]$. The timestamp of remotely read data items X equals $WTS[X]$, at the time X was actually read. These time-stamps are used to validate read access to data.

Write sets $Ws[t]$**.** For each site $S \in sites[t]$, the set of written data items at S, together with the written value, is maintained. This is used during validation to enforce the restart-rule, and during

```
St(t) { start request, t }
    t ! { start ok, Clock() }

Ac(t,X) { read request, t, X }
    t ! { read value, t, X, value[X], WTS[X] }

Vs(t) {commit request, t, ST[t], sites[t], Rs[t], Ws[t] }
    validated[t], completed[t], result[t] := ∅, ∅, commit
    foreach S ∈ sites[t] do
        scheduler@S !
            {validate request, t, S, [(X, TS)|(S, X, TS) ∈ Rs[t]], [(X, v)|(S, X, v) ∈ Ws[t]] }

V(t,S) {validate request,t, S, Rs, Ws }
    result:=commit
    foreach (q, Rs', Ws') ∈ Pending do
        if items(Rs) ∩ items(Ws') ≠ ∅ then result:=restart
        if items(Ws) ∩ items(Rs' ∪ Ws') ≠ ∅ then result:=restart
    foreach (X, OTS) ∈ Rs do
        if WTS(X) > OTS then result:=restart
    if result=commit then Pending+ = (t, Rs, Ws)
    scheduler@TransPlace(t) ! {validate result, t, S, result }

{validate result, t, S, result }
    if result[t]=commit then result[t]:=result
    validated[t]+ = S
    if validated[t] = sites[t] then
        Co(t) foreach S ∈ sites[t] do scheduler@S ! { result[t], t, S }

Ad(t,S) { result , t, S }
    Pending− = (t, Rs, Ws)
    if result=commit then
        foreach (X, v) ∈ Ws do
            value[X] := v
            WTS[X] := Clock()
    scheduler@TransPlace(t) ! { completed, t, S }

{ completed , t, S }
    completed[t]+ = S
    if completed[t] = sites[t]
    then erase sites[t], validated[t], completed[t], result[t]
        t ! { commit result, result[t] }
```

Algorithm 4.2: SCHEDULER EVENTS

commit, to write the new values of data items.

At each site S the scheduler maintains the following data structures.

Logical clock. The clock is incremented whenever it is read by calling function "clock()". There is no synchronization between clocks at different sites. The clock is used whenever transactions perform the start-synchronization, and whenever transactions actually write data items.

Write timestamps $WTS[X]$**.** For each data item X that is stored at site S, the write-timestamp $WTS[X]$ is maintained. This timestamp contains the value of the logical clock, at the last time that X was written (and hence WTS is strict monotone increasing). It is used for the validation of read accesses to X.

Pending transactions $Pending$**.** Each transaction t that performed validation at site S, but did not yet commit is represented by three-tuple $(t, Ws, Rs) \in Pending$. Sets Ws and Rs contain the information from sets $Ws[t]$, $Rs[t]$ that pertain to site S. A simple selector function "items(Ws)" returns the set of data items, referred to in Ws or Rs.

Validation result $result[t]$**.** For each transaction t that executes at site S, the scheduler at S maintains the (intermediate) result of its validation. $result[t]$ is either "committed" or "restarted". If $result[t] = committed$ after validation is completed, transaction t commits.

Participant information set $sites[t]$**,** $validated[t]$**,** $completed[t]$**.** These sets store the sites that participate in transaction t, completed validation of t and completed their participation in the execution of t.

The scheduler reacts to the following set of messages. First, a request for a timestamp can arrive, if a new transaction starts execution. Next, read requests arrive if transactions read remote items. The request for validation and commit is the last message issued by transactions. The scheduler invokes the validation of the transaction at all involved sites (note that the scheduler sends a message to itself, if local items are accessed. This has been optimized in the implementation, but for simplicity is not contained in the description of the algorithm). The following messages implement this validation. A validation request initiates the validation on a site. A validation result message is returned. Next, a commit-message (or a restart message) distributes the commit/restart decision to all involved sites. Acknowledgments are returned to the coordinator, which notifies the transaction of the result.

It is possible that a transaction t reads a data item that it has written before. Transaction t should read the value it wrote. Although this is solved in the real implementation, this has been excluded from the presented algorithms, for clarity. The message sent in the end-synchronization contains the $Rs[t]$ and $Ws[t]$ sets.

4.5 Correctness

To ease reasoning about the algorithm, several parts of the algorithm are called events, and have been given uniquely identifying names, printed in bold font. An event can consist of a single program statement (i.e. the **Co(t)** event), or an entire block of statements (i.e. the **V(t,S)** event). All events of one *process* are strictly ordered, but events of different processes can be executed concurrently. Except for local read accesses, all recognized events are handled by scheduler processes. Since there is a single process at each site, except for the **Ac(t,X)** event, all events on a single site are ordered (note that remote read accesses are handled by the scheduler, and are therefore ordered). Since the local **Ac(t,X)** event executed concurrently with other events, it does not access scheduler information. Since it can be executed concurrently with a write operation, local read access can return incorrect data. We recognize the following set of events. To ensure the uniqueness of these events, we assume that restarting transactions select a new transaction identity.

Start event St(t). This event signifies the start synchronization of transaction t. The start timestamp of t is assigned in this event.

Data access event Ac(t,X). This event represents read access to X by transaction t. Local read events do not occur in the scheduler process. Hence, they can interleave with scheduler events at the $TransPlace(t)$. Since the local **Ac(t,X)** event only modifies data structures that are local to t, this does not influence the scheduler execution.

Validation start event Vs(t). This event represents the start of the validation of transaction t. It marks the end of the execution phase of transaction t, and enables all validation events.

Validation event V(t,S). This event represents the validation of transaction t at site S. It enters transaction t in the $Pending$ set at S, and detects conflicts of t and other transactions. Furthermore, it enforces the restart-rule.

Commit event Co(t). This event represents the commit (or restart) of transaction t. It activates the administration events.

Administration event Ad(t,S). This event represents the actual writing of data items by committed transaction t at site S. If transaction t restarts, it represents the clean-up of the data structures at S.

A transaction can be executed more than once, due to restarts. The *successful run* of a transaction is the execution in which the transaction commits. We prove that the successful runs of transactions are conflict-equivalent to the order in which transactions commit. In the proof below, "After E P" means that predicate P is true in all states where event E has occurred. The precedence relation $\prec$ denotes causal dependency. First, we give some preliminary results over the basic execution order of certain events.

Basic transaction-order
$\forall t, X, S$: **St(t)** $\prec$ **Ac(t,X)** $\prec$ **Vs(t)** $\prec$ **V(t,S)** $\prec$ **Co(t)** $\prec$ **Ad(t,S)**

Theorem 4.1 *The successful runs of two transactions p and q are conflict-equivalent to the commit order.*

Proof

1 Either p, q do not conflict, or they do conflict (exhaustive)
2 Suppose p, q do not conflict
2.1 QED. (2, def. conflict-equivalence)
3 Suppose p and q conflict over data item X at site S
3.1 wlog. **V(p,S)** $\prec$ **V(q,S)** (symmetry p, q, mutual exclusion)
3.2 Either **V(q,S)** $\prec$ **Ad(p,S)** or **Ad(p,S)** $\prec$ **V(q,S)** (exhaustive)
3.3 Suppose **V(q,S)** $\prec$ **Ad(p,S)**
3.3.1 q is restarted, contradiction (3,3.1,3.3, restart-rule)
3.4 Suppose **Ad(p,S)** $\prec$ **V(q,S)**
3.4.1 **Co(p)** $\prec$ **Co(q)** (3.4, transaction-order)
3.4.2 Either (q writes X) or (q reads X and p writes X) (3, def. conflict)
3.4.3 Suppose q writes X
3.4.3.1 **Ad(p,S)** $\prec$ **Ad(q,S)** (3.4, transaction-order)
3.4.3.2 QED. (3.4.1, 3.4.3.1, transaction-order)

3.4.4 Suppose q reads X and p writes X
3.4.4.1 Lemma: **Ad(p,S)** $\prec$ **Ac(q,X)**
3.4.4.1.1 Suppose **Ac(q,X)** $\prec$ **Ad(p,S)**
3.4.4.1.2 After **Ac(q,X)**, $(S, X, C) \in Rs[q]$, $WTS[X] \geq C$
(**Ac()**, transaction order, monotony WTS)
3.4.4.1.3 After **Ad(p,S)**, $WTS[X] > C$
(3.4.4, 3.4.4.1.1, 3.4.4.1.2, **Ad()**, def. clock, monotony WTS)
3.4.4.1.4 q restarts, contradiction (3.4, 3.4.4.1.2, 3.4.4.1.3, **V(q,S)**)
3.4.4.2 QED. (3.4.1,3.4.4.1)
□

The restart-rule ensures that the commit events of conflicting transactions are causally ordered. Hence, theorem 4.1 proves conflict serializability of successful runs. Since restarting transactions do not modify the database, they do not invalidate the execution of other transactions. It is possible that restarting transactions cause other transactions to restart as well (due to the simplicity of the restart-rule). Therefore, life lock can occur. Transactions can be restarted infinitely often if a constant stream of conflicting transactions validates. However, the probability of such scenario's is small, since the conflict-probabilities are assumed to be near zero. Transactions can never be deadlocked, since they never wait on other transactions.

4.6 Performance analysis of OCC-light

We derive an expression for the overhead that is generated by the OCC-light scheduler. The expression shows how much overhead is attributed to synchronization and communication.

When transaction t starts, it is embedded in a scheduling procedure. This counts as internal computation overhead, no synchronizations and communications are necessary for this embedding. Instead of closely analyzing the overhead that results from internal computation of the scheduler, we introduce the term Int_{occ-l} that represents all overhead of this type. It is clear that this overhead is dependent on the number of data accesses, and the type of data accesses of transaction t. Furthermore, in our (non-optimal) implementation, it is also dependent on the number of data accesses of concurrently validating transactions. This can be avoided by using improved data structures. The impact of these dependencies have not been analyzed.

At the start of transaction t, it is synchronized with the scheduling process at the site where t executes, introducing one synchronization step $Sync$. Each write-request of t is administrated, but not actually performed. No context switches are necessary, and the administration consists of updating a data-structure local to t. Each local read-request of t is administrated, similar to write-requests. A "dirty" read is performed, which can be pre-empted by writing transactions. No synchronization is required. Remote read-requests involve two sequential messages (value request, and result). The scheduler actions are piggybacked on the normal execution, and extra overhead is minimal. The message-size grows with a few bytes (≤ 40), The validation of t is performed by the scheduler. This introduces two context switches, which count as one synchronization. The global validation is piggybacked on the two phase commit protocol, and does not introduce extra messages. However, if the transaction is restarted, the messages of the two phase commit Protocol that is aborted due to the validation do count as scheduler overhead (2 extra messages). When remote items are written, the transaction-execution needs two sequential messages, these do not count as overhead. The scheduler piggybacks information on these messages. The extra overhead of this is negligible. If t restarts, the failed execution X also counts as overhead. The overhead of the scheduler can now be defined. Let

V be the number of validation phases ($V - 1$ unsuccessful validations, and 1 successful validation). Let V_g be the number of failed validation phases that contain validation of remote data accesses (i.e. $V_g \leq V$, and $V_g = 0$ if transactions only access local data).

$$O_{occ-l} = Sync + (V - 1)X + V \times Sync + V_g \times 2Comm + Int_{occ-l}$$

4.7 Performance analysis of Mnesia's two phase locking scheduler

The prototype database Mnesia is equipped with a lightweight two phase locking scheduler (see also appendix C). We analyze the performance of this scheduler in the same way as we analyzed OCC-light. The 2PL scheduler uses the same embedding of a transaction, as the OCC-light scheduler. The embedding is used for deadlock prevention: transactions that are (possibly) deadlocked are restarted. We introduce term Int_{2PL} to denote all scheduler overhead that results from internal computation.

When a transaction starts, it notifies the scheduler of its existence, a synchronization step. This is used for wait-die deadlock prevention, see appendix C. Read and write access procedures are both modified. Before data access to items X is allowed, the transaction has to obtain a lock on X. This requires a synchronization step. It introduces overhead for local data accesses, and remote writes. This synchronization is piggybacked on remote read requests, and hence does not introduce additional overhead for remote reads. The 2PL scheduler uses the usual two phase commit protocol, and piggybacks the unlock operations on the protocol. Assume that transactions access K_l local items, and read K_{rr} remote items, and write K_{wr} remote items. The deadlock prevention protocol can restart a transaction whenever it performs a data access. The restarted execution so-far, and the scheduler overhead so-far count as overhead. Instead of analyzing this factor, we introduce term $Dprev$ that describes the total time-cost. The overhead generated by Mnesia's 2PL is described as follows.

$$O_{2PL} = (2 + K_l)Sync + 2K_{wr}Comm + Dprev + Int_{2PL}$$

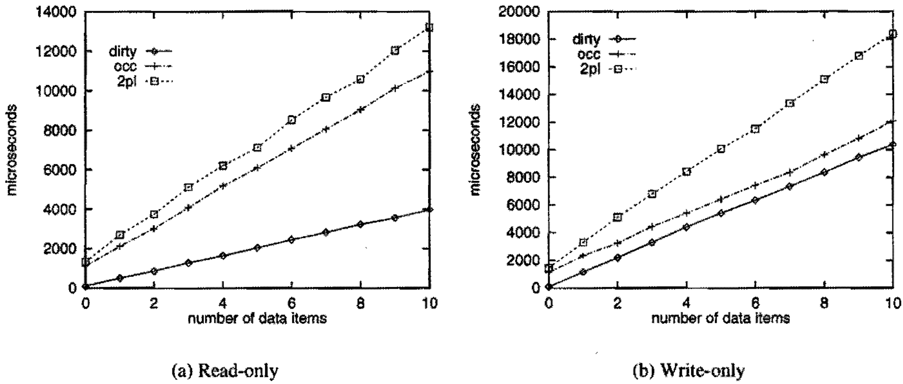
4.8 Test results

A number of experiments were conducted with both schedulers. To this end, the existing 2PL scheduler of the prototype database Mnesia was replaced by an implementation of the OCC-light scheduler. We present the results, and show that they are in accordance with the analysis.

The schedulers have been tested in realistic environments, where background processes and other users occasionally influence the response times of transactions. These influences appear in repeated testing, and are best visible in multi-site tests (see figures 4.4 and 4.5). We have investigated the response time distributions of the tests, and found that the influence of other processes is very chaotic. Most transactions do not suffer any delays, and have an response time that is between 0.5 and 2 times the average. However, a very small percentage (less than 10 in a run of 2 minutes) of transactions show atypical behavior: they are delayed for several seconds. The perturbations in the graph are due to the varying number of atypical transactions in each runs.

The hardware varied from 1 to 8 SPARC 5 workstations, running Solaris. Precise specifications (internal memory, clock speeds) of these machines were unavailable. The same set of workstations was used for the results of a single experiment, but for different experiments, different configurations were used.

A single dot on a curve averages the results from a 10 second test run for single-site tests, to 2 minutes for multi-site tests (to account for the different execution durations of transactions). Each machine that participated in an experiment contained a database site and one or two application processes.



(a) Read-only

(b) Write-only

Figure 4.3: TRANSACTION RESPONSE TIME ON ONE SITE, NO CONCURRENCY

Each site consisted of 100 data items. Each application process executes a sequence of transactions, without delay between consecutive transactions.

Experiment 1: local transactions

Mnesia is configured as a single site database. Experiments are conducted with read-only and write-only transactions that accessed from 1 to 10 arbitrary data-items. These experiments simulate an environment with no conflicts (which is close to the telecom environment), by executing the transactions in a sequential order. We tested the 2PL scheduler, the OCC-light scheduler, and the case where transactions execute at will, without any scheduler overhead (non-serializable executions can occur).

Only one application process issues transactions, so the database executes at most one transaction at a time. The difference in performance between OCC-light and 2PL is directly related to the overhead, generated by the schedulers. No conflicts occur, so no reruns are necessary under OCC-light, $V = 1$ and $V_g = 0$. The expression for the overhead is simplified as follows.

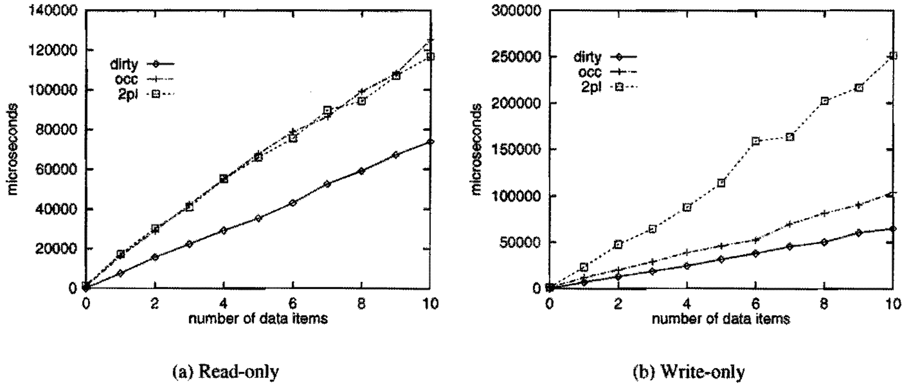
$$O_{occ-l} = 2Sync + Int_{occ-l}$$

No deadlocks have to be prevented, and the overhead of 2PL is as follows.

$$O_{2PL} = (2 + K_l)Sync + Int_{2PL}$$

From the formula's, we see that OCC-light has a constant overhead, if we ignore the overhead that results from internal computation. 2PL has an overhead that increases linearly with the number of accessed items. Therefore, we expect that the difference between OCC-light and 2PL increases linearly.

Indeed, the experimental results from figure 4.3 show that the difference between OCC-light and 2PL gradually increases. In the write-only experiment in figure 4.3b, the overhead of OCC-light is almost constant: the difference between the "dirty" transaction response time and the response time under OCC-light is approximately at 1400 microseconds. The read-only experiment in figure 4.3a shows a different picture, the computation overhead of OCC-light sharply increases with the number



(a) Read-only

(b) Write-only

Figure 4.4: TRANSACTION RESPONSE TIME ON EIGHT SITES

of data items accessed. OCC-light's curve is almost as steep as the curve from 2PL, although 2PL requires more synchronization points. As mentioned before, the internal overhead generated by OCC-light is dependent on the number of data items that is accessed. The validation of a read access requires the checking of the locally stored timestamps with the write-timestamps of data items. Apparently, the data-structures used to store the write-timestamps are not efficiently implemented. The experimental results in figure 4.3 show that 2PL generates more overhead than OCC-light for more data items. This matches our analysis.

Experiment 2: remote data access

The database is configured with 8 sites that each store the same number of data items. At each site, an application process continuously executes transactions. Transactions access from 0 to 10 data items. Again, the OCC-light scheduler, the 2PL scheduler and "dirty" transactions were tested. Figure 4.4a shows the results if transactions only read the database, no conflicts occur. The overhead of OCC-light and 2PL is simplified to the following formula's.

$$O_{occ-l} = 2Sync + Int_{occ-l}$$
$$O_{2PL} = (2 + K_l)Sync + Int_{2PL}$$

Since only 12.5% of all data accesses are local, the influence of factor $K_l Sync$ is small. Furthermore, the time-costs of synchronizations are small compared to the execution time of the transaction (that requires 2 sequential messages for each accessed data item). OCC-light and 2PL should have similar performance, if Int_{occ-l} and Int_{2PL} are similar. Indeed, this matches the test-results from 4.4a. Transactions that access multiple sites are an order of magnitude slower than local transactions, even without extra scheduling overhead, as can be seen from the performance of "dirty" transactions. Figure 4.4b shows the results if transactions only write the database. Conflicts that lead to restarts can occur, so the overhead of OCC-light is described by:

$$O_{occ-l} = Sync + (V-1)X + V \times Sync + V_g \times 2Comm + Int_{occ-l}$$

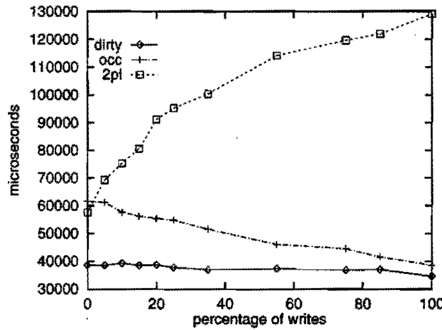


Figure 4.5: TRANSACTION RESPONSE TIME ON 8 SITES, READ/WRITE PERCENTAGE

The total conflict probability is less than 0.5, so the average number of reruns is less than 1. If we use this as a (very pessimistic) approximation the formula simplifies to:

$$O_{occ-l} = 3Sync + X + 2Comm + Int_{occ-l}$$

Here X is the execution time of the transaction, which is linear in the number of data items accessed. However, X does not include any communications (since the writing of the transaction takes place after the validation), therefore X will be small. Hence, we expect that the curve of OCC-light increases slightly faster than the curve of the unscheduled ("dirty") execution. We ignore the restarts that result from the deadlock prevention of 2PL to arrive at the following expression:

$$O_{2PL} = (2 + K_l)Sync + 2K_{wr}Comm + Int_{2PL}$$

As 87.5% of all data accesses is remote, the execution of 2PL introduces almost 2 sequential communications for each data access. These communications are very time-consuming, and we expect that the curve of 2PL increases much faster than OCC-light. Indeed, the results of the analysis match the test results of 4.4b.

Experiment 3: read/write percentage

The database is configured with 8 sites that each store the same number of data items. At each site, an application process continuously executes transactions. Transactions access 6 data items from arbitrary sites. Each access can be a read, or a write, decided arbitrarily. The probability that a data access is a write-access is varied from 0 to 100%.

Figure 4.5 shows the results. The two endpoints (100% reads and 100% writes) do not exactly match the previous tests, due to the fact that the tests were executed on a different set of machines. OCC-light and 2PL have comparable performance for remote reads, but OCC-light outperforms 2PL for remote writes. Interestingly, OCC-light improves its performance if the percentage of writes increases. Apparently, the total conflict probability is so low, such that the overhead of reruns is less than the performance increase that results from less reads and more writes. As observed before, the validation that requires checking of locally stored timestamps against the write-timestamps of data items is not efficiently implemented. This leads to a large computation overhead for each data

item that is read. Since validation of write-write conflicts relies on the restart-rule rather than the comparison of timestamps, this accounts for the increased performance.

4.9 Conclusions

The environment that is described in this chapter requires lightweight schedulers for good performance. Due to the very short transactions, scheduler overhead is a major factor. The OCC-light scheduler that is designed in this chapter minimizes scheduler overhead. Both analysis and tests show that it performs well in low-conflict environments with short transactions. Overhead that results from internal computation is a significant factor of the total overhead of OCC-light for local transactions. Especially the overhead of checking consistency of reads was significant, and should be implemented more efficiently.

Chapter 5

Scheduling in high-conflict environments

When the conflict-probability is high, and execution duration of transactions is long, light-weight schedulers (see chapter 4) perform poorly. More sophisticated schedulers that incur more scheduler overhead can obtain a better performance.

The chapter is organized as follows. In the next section, the environment in which the scheduler should operate is stereo-typed. Assumptions formalize the arguments, such that the problem specification from section 5.2 is obtained. In section 5.3 the scheduler design is discussed. The resulting algorithm in section 5.4 is quite complicated. Section 5.5 offers a correctness proof, as well as proofs for deadlock- and life lock-freedom. We perform a quick analysis of the scheduler overhead in section 5.7, and test results appear in section 5.8. The chapter concludes with some reflections in section 5.9.

Two similar schedulers have been published earlier. DOCC-DASO [55] is an adaptation of OCC-TI that uses locks and a circular validation scheme. The distributed certification protocol from [21] already contains the basic ingredients of OCC-TI (although the article is rather difficult to follow). Its validation protocol is very message-intensive, consisting of several broadcasts, and is only effective if communication between sites uses a bus.

5.1 Environment study

The environment we discuss is not directly inspired by the applications from chapter 1. Still, these application areas are large, and it is likely that high-conflict environments exist within these areas. We are looking for a distributed database design, and we assume the following application characteristics.

Data access of transactions. The database is either quite small, or it contains a subset of "hot data items". Hot data items are accessed more frequently than normal data items. Therefore, the probability that two transactions access the same data items is high. Again, we have no information about actual systems with these characteristics. To bound the conflict probability, we require that it is at least 0.02.

Transaction durations. Transactions are long. A significant amount of computation occurs between data accesses. Therefore, the overhead posed by the database will (if the overhead is reasonable) not be a significant part of the transaction-response time. To formalize this characteristic, we assume that the amount of work that a transaction performs is linear in the number of data items that it accesses. Furthermore, at least 100 internal computation steps are performed, for one data access.

Data size. One a few kilobytes of information are stored in one data item. Therefore, the communication costs of transferring the value of data items between sites are dictated by the latency of the network, rather than the throughput.

Transaction arrival process. Transactions arrive unannounced, at a high rate, with exponentially distributed inter-arrival times. The total number of transactions that arrives is more than can be handled by one processor. It is desirable that the database is scalable, i.e. the number of transactions that the database can execute in one time unit should be a linear function of the number of processors. No assumptions are made about the arrival rate of transactions, and the arrival rate is not announced by the application.

Real-time requirements. No explicit deadlines are generated for transactions. Instead, the database should try to minimize the response time of transactions.

We make additional assumptions to characterize the platform on which the scheduler is implemented. These figures are derived from actual measurements in a network of SPARC 5 workstations. One-way communication between sites takes on average 1.5ms: $Mlatency = 1.5$ms. Synchronization implemented with local message passing takes on average 0.13ms: $Msync = 0.13$ms. We assume that $Cpower = 1$, which means that the work of transactions equals their execution time.

5.2 Specification

The generic scheduling problem is the same as the generic scheduling problem in chapter 4.

Generic scheduling problem.

platform: distributed	page 22
data: non-temporal, distributed	page 22
transactions: real-time, conflict-serializable, unannounced	page 24
objective function: minimize average response time	page 30

Additional detailed information.

Fact constraint	Description
$\sum(t, q \in T : transconflict(t, q) : 1)/\lvert T\rvert^2 \geq 0.02$	The conflict probability is high.
$W(t) \geq \lvert Aorder(t)\rvert \times 10ms$	Transactions are computation intensive.
$CPower = 1$	The work of a transaction equals its execution time.
$Msync = 0.13ms$	Time-cost of a synchronization (actual measurement).
$Mlatency = 1.5ms$	Latency of inter-site communication (actual measurement).

5.3 Design

The objective of the scheduler that has to be designed is to minimize the transaction response time. The transaction response time depends on the trade-off between the overhead that the scheduler generates, and the processor utilization that the scheduler realizes (see chapter 3). How this trade-off should be made depends on the conflict probability, and the average transaction length.

The conflict probability is high, suggesting that a large class of schedules is needed to obtain a good processor utilization. Furthermore, transaction execution durations are long, compared to

basic transaction loop
start: $TI[t] := [0, \infty)$
  $access[t]$:= execute(t, $TI[t]$)
  validation(t, $access$, $TI[t]$)

database access procedure
access(t, X, $mode$, Val)
  $access[t] += (X, mode)$
  $TI[t] \cap = (WTS[X], \infty)$
  if $mode = write$ then $TI[t] \cap = (RTS[X], \infty)$
  if $TI[t] = \emptyset$ then restart(t)
  if $mode = write$ then shadowwrite[t, X]:= Val
    else return value[X]

validation procedures
validation(t, $access[t]$, $TI[t]$)
  select $\tau_t \in TI[t]$
  foreach $q : \exists X : (X, tm) \in access[t] \wedge (X, qm) \in access[q] \wedge ConfRel(tm, qm)$ do
    adjust(t, tm, τ_t, q, qm)
  foreach $(X$, read$) \in access[t]$ do $RTS[X]max = \tau_t$
  foreach $(X$, write$) \in access[t]$ do
    $WTS[X]max = \tau_t$
    value[X] := shadowwrite[t, X]

adjust(t, tm, τ_t, q, qm)
  if tm =read $\wedge$ qm =write then $TI[q] \cap = (\tau_t, \infty)$
  if tm =write then $TI[q] \cap = [0, \tau_t)$
  if $TI[q] = \emptyset$ then restart(q)

restart procedure
restart(t)
  clear data structures of t
  goto start

Algorithm 5.1: OCC-TI SCHEDULER

the overhead that is introduced by synchronization and communication. No significant performance optimizations can be achieved by minimization of synchronizations. Optimization of communication overhead can lead to small performance gains, so should not be ignored.

Design decision 5.1 *Our main goal is to allow a large class of schedules. At the same time, the scheduler overhead should not be excessive.*

One of the most sophisticated database scheduling mechanisms for centralized architectures is optimistic concurrency control with time-stamp intervals (OCC-TI, see [57]). It allows for a large class of schedules, by doing precise bookkeeping of access-orders. This is exactly what we need to satisfy our first design decision.

Design decision 5.2 *The scheduler should recognize the same class of schedules (or more) as OCC-*

TI.

Algorithm 5.1 show the essential parts of the OCC-TI algorithm (see appendix B for notation definitions). The exact manipulations of the data structures have been left out. OCC-TI actually employs two kinds of validation: *backward validation* and *forward validation*. During backward validation, a transaction t validates its execution against the execution of previously committed transactions q. If a transaction has to restart, transaction t *itself* is restarted. During forward validation, a transaction t validates its execution against the execution of uncommitted transactions q. If a transaction has to restart, transaction q is restarted, instead of t. Since the scheduler designed in this section employs the same validation techniques, we call it distributed optimistic concurrency control with backward and forward validation (DOCC-BF). OCC-TI is essentially a centralized scheduler, the entire validation routine is contained in a critical section. Unless this requirement is lifted, the validation will become a major bottleneck in a distributed system.

Design decision 5.3 *The distributed version of OCC-TI should allow concurrent validation of transactions, to avoid a bottleneck in the validation phase.*

Suppose that transactions can concurrently execute the validation procedure of algorithm 5.1. Inconsistencies can occur if transaction p adjusts the time-stamp interval of transaction q, after τ_q has been defined ($\tau_q \in TI[q]$ can be disturbed by this behavior). Therefore, if t selects τ_t, the distributed validation protocol has to ensure that conflicting transactions q do not select τ_q, until t has carried out all adjustments. Therefore, conflicting transactions q have to be delayed in their validation phase. This can be accomplished by *locking* all data items that have been accessed by t for the duration of the validation. To keep the protocol scalable, locks are stored with the data items they guard. Therefore, the distributed validation protocol consists of the following steps: 1) lock all accessed data items of t, 2) perform validation as described in algorithm 5.1 and 3) unlock accessed data items. The communications necessary in step 1 and 3 can be piggy-backed on the two phase commit protocol (see appendix C).

The site where a transaction t executes is called the *coordinator* of t. A site where a transaction t accesses data is called a *participant* of t. The validation is initiated at the coordinator, that notifies all participants of t that step 1 has started. After all locks have been granted, the coordinator of t selects τ_t. Finally, the participants of t modify WTS, RTS, write values, and notify conflicting transactions q that adjustments are made to $TI[q]$.

Design decision 5.4 *The distributed validation protocol uses locks to avoid inconsistent validations.*

The decision to use locks introduces the possibility of distributed deadlock. Using deadlock prevention mechanisms like wait-die (see appendix C) would reduce the concurrency offered by the scheduler. Since a high degree of concurrency is our primary goal, deadlock prevention techniques that require restarts (as used in chapter 4) are not appropriate. The DOCC-DASO scheduler in [55] uses an acyclic validation order to prevent deadlocks. This introduces a considerable amount of communication overhead: if transaction t accessed N sites, step 1 of the validation requires N sequential messages. We observe that a transaction t, waiting on a lock held by q, can complete step 1 of its validation without delays. As long as t is blocked before step 2, q can still adjust t. By changing the locks to so-called *delayed locks*, the coordinators of transactions can gather sufficient information to detect potential distributed deadlocks. For example: suppose transactions t and q both accessed X at site S, and Y at site S'. Suppose t locks X before q, and q locks Y before t. A distributed deadlock has occurred. If t and q complete step 1 without waiting, the coordinators of t and q can detect this cycle: information about the lock-status of X and Y is available.

Design decision 5.5 *The distributed validation protocol uses* delayed locks *that block transactions from starting step 2 of the validation protocol.*

We analyze what information about the lock-status can be gathered by the coordinators of transactions. Consider two transactions t, q that both access X in a conflicting way. Either t obtains the lock before q (thus blocking q from entering step 2 until t completes step 3), or vice versa. Suppose without loss of generality that t obtains the lock before q. The following scenarios are possible.

- Transaction t releases the lock before q accessed X. Hence, transaction t updates $WTS[X]$, $RTS[X]$ before q accesses X. Therefore, when q accesses X, it *backward validates* against t, by adapting $TI[q]$ according to $WTS[X]$, $RTS[X]$ (see algorithm 5.1). The transactions do not need to be aware of each other.

- Transaction q accessed X before t requests the lock. Both the coordinator of t and q can detect that t blocks q from entering step 2. The coordinators of t and q know that both have complete information about the lock-order. Since the knowledge is shared, this is called a *shared conflict*.

- Transaction q accessed X after t requested the lock. Only the coordinator of q detects that t blocks q from entering step 2. Unless t and q conflict over more than one data item, the coordinator of t is unaware of q. Since only q has knowledge about the conflict, this is called an *unshared conflict*.

Therefore, the coordinator of a transaction t distinguishes three types of conflicts with other transactions. First, unshared conflicts that are detected by t prevent t from entering step 2, until the matching unlock operations are executed. Second, shared conflicts can be of two types. If transaction t requests a lock on X before conflicting transaction q requests the lock, we say that t *owns* the conflict with q. If transaction t requests a lock on X after conflicting transaction q requests the lock, we say that t is *owned by* q.

Suppose transaction t both owns q, and is owned by q. Neither transaction is allowed to enter step 2 of the validation, a deadlock occurs. This situation is possible if t and q conflict over more than one data item, at more than one site. The coordinator of t and q both detect this deadlock, since both conflicts are shared. We introduce a total order $<$ on the transactions, to decide which transaction is allowed to enter step 2 first. This order can reflect the real-time priorities of transactions (for example, transactions can be ordered according to their deadlines, such that more urgent transactions are allowed to complete validation first). This has not been tested. The following scenarios describe all possible deadlocks of cycle length two.

- Transaction t owns q, and is owned by q. Without loss of generality assume $t < q$. The coordinator of t and q independently decide that t is allowed to enter step 2 before q.

- Transaction t owns q, and detected an unshared conflict with q. The coordinator of q is unaware that t is blocked. Therefore, q cannot decide to enter step 2 before t, and will wait. To avoid deadlock, t ignores the unshared conflict, and enters step 2 before q.

- Transaction t detects an unshared conflict with q, and transaction q detects an unshared conflict with t. This scenario cannot occur. If t detects an unshared conflict with q over X, t accessed X after q requested a lock to X. Hence, q has completed all data access, before t requests its first lock. Therefore, q cannot detect an unshared conflict, QED.

The most common deadlock of cycle length two is avoided by the rules given above. The distinction between shared and unshared conflicts allows the protocol to solve these deadlocks without additional communication. The coordinator of a transaction t will maintain the sets owned[t] and ownedby[t] that contain the identities of conflicting transactions q. The set unshared[t] contains all identities of conflicting transactions q, where t has detected an unshared conflict. This set is not mentioned in the final algorithm, as it is implemented by the other data structures. In the design of the new scheduler, we will use unshared[t] for clarity.

Design decision 5.6 *The distributed validation protocol uses the deadlock avoidance rules 1 and 2 to avoid all deadlocks of cycle length two.*

Distributed deadlocks of cycle length > 2 can only be detected by introducing additional communications between sites. We observe that a transaction t can only be part of a deadlock cycle if it is blocked: $\text{ownedby}[t] \cup \text{unshared}[t] \neq \emptyset$. Furthermore, transaction t has to block at least one other transaction q: $\text{owned}[t] \neq \emptyset \vee \exists q : t \in \text{unshared}[q]$. Potential deadlock cycles are broken by requiring that a blocked transaction t does not block other transactions q when $q < t$. This is captured by the *reverse rule.*

If transaction t is about to enter validation-step 2, but is blocked by an arbitrary transaction p, and t blocks $q \neq p$, then t reverses the blocking-order with q if $q < t$.

Of course, the coordinator of t notifies the coordinator of q of the decision, such that q is no longer blocked by t. This introduces an inter-site communication. Fortunately, this additional communication is only required if the reverse rule is used to avoid deadlocks.

Design decision 5.7 *The distributed validation protocol uses the reverse rule to establish an acyclic blocking order, thus avoiding deadlocks of cycle length > 2.*

The coordinator of transaction t can only reverse the blocking order with q, if it knows of the existence of q. If $q \in \text{owned}[t]$, the coordinator of t knows that q exists. If $q \notin \text{owned}[t]$, the conflict is unshared. Hence, the coordinator of t cannot use the reverse rule, unless q informs the coordinator of t that it is blocked by t. Therefore, the *notification rule* is introduced.

If transaction t is about to enter validation step 2, but is blocked with only unshared conflicts by q, then t notifies q that it is blocked, if $t < q$.

This requires that the coordinator of t communicates with the coordinator of q, notifying q of t's existence. This introduces an inter-site communication that is only required if the notification rule is applied. At most two sequential inter-site communications (wait notification and reverse wait) are necessary to break distributed deadlocks.

Design decision 5.8 *The distributed validation protocol uses the notification rule to ensure that the reverse rule can be enforced when necessary.*

This completes the design of the DOCC-BF algorithm as presented in the next section. A number of optimizations are possible, which are described in section 5.6.

5.4 The DOCC-BF algorithm

We present the DOCC-BF algorithm in a simplified form to facilitate the correctness proof in section 5.5. First of all, none of the optimizations of section 5.6 are contained in the algorithm. Second, we

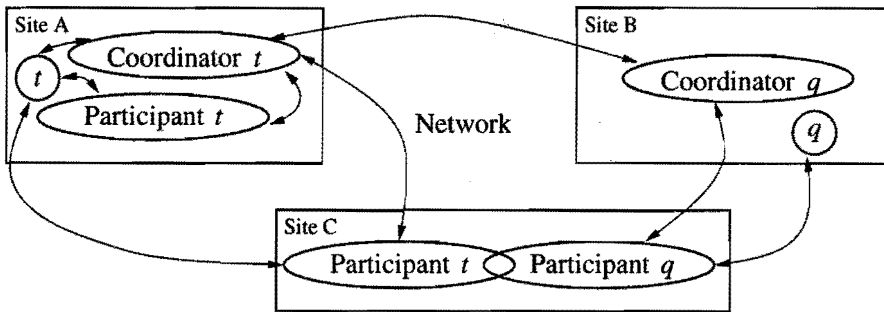


Figure 5.2: AN EXAMPLE SYSTEM CONFIGURATION

assume that all communication between pairs of sites is first-in, first-out (FIFO). Lifting this assumption is possible (and has been done in the prototype implementation), but complicates the description of the algorithm. Third, we abstract from some trivial bookkeeping by assuming that the addresses of all scheduler-processes and transactions are globally known. The address of the process that executes a transaction t is simply t, the address of the scheduler process that serves as t's coordinator (see also next subsection) is denoted $coord(t)$, and the address of the scheduler process that manages data item X is simply X. For simplification, we have assumed that each data item is stored at a separate site. The simplification only serves to ease the correctness proof. In the prototype implementation, this simplification has been removed.

5.4.1 System architecture

At each site S in the network, a scheduler process is created. Such a scheduler process has two main functions. It acts as a coordinator for transactions t that execute at site S, and it acts as a participant for transactions t that access data, stored at site S. In the unoptimized algorithm, the data space of the coordinator of t does not overlap with participants or other coordinators. All communication takes place through messages. The data space of a participant of t can overlap with other participants, and some communication between participants takes place through direct modification of data-structures.

Figure 5.2 gives a possible configuration: transaction t executes at site A, and transaction q executes at site B. Both transactions access data on site C, and transaction t accesses data on site A. Some points of interest: although the coordinator of t and the participant of t on site A are actually the same scheduler process, the unoptimized algorithm does not use this fact. When the coordinator communicates with the participant, the scheduler process actually sends a message to itself! An implementation that uses separate processes for participants and coordinators is possible (and has been drawn in the figure). Second, participant t and participant q at site C are also the same scheduler process. In this case, the algorithm does make use of this fact, and communication between the two participants is implemented by direct modification of data structures.

The execution of a transaction t is encapsulated, such that the scheduler can restart t's execution when necessary. We assume that a transaction t is executed by calling execute(t). Furthermore, the data access routine is modified, such that t communicates with the appropriate scheduling process, when it needs to access data. Finally, we assume that the coordinator of t can stop t's current

basic transaction loop
start: select unique identity t
execute(t)
validation(t)

communication with database
access($t, X, mode, Val$)
enable **Ac(t,X,mode)**
receive Result from $coord(t)$
return Result

validation(t)
enable **Vs(t)**
receive committed from $coord(t)$

Algorithm 5.3: DOCC-BF TRANSACTION MODIFICATION

execution, and start a new one, by issuing command "restart(t)".

The coordinator of a transaction t sends messages to transaction t. If t requested data access, the coordinator returns the result of the data access. If t requested validation, the coordinator returns a message if t has committed, and can finish execution. If necessary, the coordinator sends restart messages, instead of result/commit messages. Since these communications do not influence the scheduler execution itself, these have not been included in algorithm 5.3 and algorithm 5.5.

5.4.2 Scheduler description

The execution of the distributed scheduler is described as a set of atomic events. An event cannot occur before it is *enabled*. The **Co(t)** event is an exception to this rule, which is further explained in the paragraph below. When an event is enabled, it can occur at any moment, and it will eventually occur. Events can enable other events (in our prototype, an event is enabled by sending a matching message to a scheduler process). We abstract from the contents of messages that enable events: if event b is (indirectly) enabled by event a (i.e. a path of events $a = e_1, ...e_n = b$ exists, such that e_i enables e_{i+1}), event b can directly access the data structures that are modified by a. Each event is uniquely identified by its type, and its parameters, and occurs at most once. The scheduler protocol is initiated by the executions of transactions that can *enable* data access events **Ac(t,X,mode)** and the validation-start event **Vs(t)** (for explanation of the events, see below). All other events are enabled by events. Events are grouped: events that occur at the coordinator of t, and events that occur at the participants of t.

The commit event Co(t). The commit of a transaction t cannot occur before a number of requirements have been met. First of all, transaction t must have finished its execution, and have performed validation on all sites, where t accessed data. Second, all conflicts with transactions q are either resolved, or those transactions q are blocked by t. The order in which these requirements are met is not fixed, and the commit of t can be initiated by three different events (**Vm(t,X)**, **Rw(t,q)** and **Cr(t,q,X)**). To reduce the length of the correctness proof, we assume that the **Co(t)** event *directly follows* the event that initiated it. Both events are executed in one atomic section. This is denoted as follows: if line 14

Ac(t,X,mode)
1 if mode=write then $TI[t, S] \cap = (RTS[X], \infty)$
2 $TI[t, X] \cap = (WTS[X], \infty)$
3 $ac[t, X, mode] := true$
4 enable **Ar(t,X,mode)**

V(t,X)
5 foreach $q : \neg ad[q, X] \wedge \exists tm, qm : ConfRel(tm, qm) \wedge ac[t, X, tm] \wedge ac[q, X, qm]$ do
6 conflicts$[t, X] += (X, q, qm, t, tm)$
7 if $\neg v[q, X]$
8 then owned$[t, X] += q$; ownedby$[q, X] += t$;
9 conflicts$[q, X] += (X, t, tm, q, qm)$
10 $v[t, X] := true$
11 enable **Vm(t,X)**

Ad(t,X)
12 if $ac[t, X, write]$ then $WTS[X] := \max(WTS[X], \tau_t)$
13 if $ac[t, X, read]$ then $RTS[X] := \max(RTS[X], \tau_t)$
14 foreach $q : \neg ad[q, X] \wedge \exists tm, qm : (X, t, tm, q, qm) \in$ conflicts$[q, X] \cup$ conflicts$[t, X]$ do
15 enable **Cr(q,t,X)**
16 $ad[t, X] := true$

Algorithm 5.4: PARTICIPANT CODE

of the TTC procedure in algorithm 5.5 is executed, the **Co(t)** event occurs directly after the event that invoked the TTC procedure. We assume that the **Co(t)** event consists of line 14 and 15 of the TTC procedure.

Participant events

The scheduler stores for each data item X write-timestamp $WTS[X]$, and read-timestamp $RTS[X]$. For each transaction t of which the scheduler is a participant the following sets are maintained. Set conflicts$[t, X]$ contains all transactions that conflict with t over X. Set owned$[t, X]$ contains all transactions that have a conflict with t, and t owns the conflict. Set ownedby$[t, X]$ contains all transactions q that have a conflict with t, and q owns the conflict. Boolean variables are maintained that are set to true if the corresponding events occur: $ac[t, X, mode]$, $v[t, X]$, $ad[t, X]$ are true if events **Ac(t,X,mode)**, **V(t,X)**, **Ad(t,X)** have occurred. See below for the description of the events. Finally, $TI[t, X]$ stores restrictions on τ_t, generated at site X.

The participant should initialize all data structures, the first time that a transaction t accesses data stored at the participant. This initialization is not described in algorithm 5.4. Instead, we assume that all data structures are correctly initialized.

Data access event Ac(t,X,mode). This event is executed on the participant of t that stores data item X. Its functionality is similar to the data access procedure in algorithm 5.1. It performs a backward validation, by adjusting $TI[t, X]$ in accordance with $mode$, and $WTS[X]$, $RTS[X]$. Furthermore, the occurrence of the event is administrated in $ac[t, X, mode]$. Finally, it enables the **Ar(t,X,mode)** event.

Validation event V(t,X). This event is executed at the participant of t where X is stored. It enables the **Vm(t,X)** at the coordinator. Furthermore, it detects all owned, ownedby and unshared conflicts of t, regarding X. The occurrence of the validation event is administrated in $v[t, X]$.

Administration event Ad(t,X). The administration event is executed at the participant of t where X is stored. It updates $WTS[X]$, $RTS[X]$. Furthermore, it enables **Cr(q,t,X)** events for each transaction q that has been blocked by t. The occurrence of the administration event is administrated in $ad[t, X]$.

Coordinator events

The scheduler stores for each transaction t its timestamp τ_t (initially undefined), the allowed timestamp interval $TI[t]$, the transactions it owns owned[t], and that own t: ownedby[t]. Furthermore, it maintains the set of detected conflicts conflicts[t], the set of resolved conflicts resolved[t], and the set of transactions that reversed wait order reversed[t] to t. The set of transactions that t has decided to wait for is waitfor[t]. Set notified[t] contains the transactions that have been notified that t is waiting. The set of participating sites is maintained in $sites[t]$. Furthermore, the boolean $vm[t, X]$ specifies whether the validation event for site X has occurred, and the boolean $vs[t]$ specifies whether the validation start event has occurred.

If all conflicts between transactions t, q are resolved, both transactions can commit. We introduce shorthand existsunresolved(t, q) that signifies whether t has detected an unresolved conflict between t and q: existsunresolved$(t, q) \equiv \exists X, m, m' : (X, t, m, q, m') \in$ conflicts[t] $-$ resolved[t]. We assume that conflicts are described in a uniform notation, although this is not expressed in the algorithm. Therefore, the following equivalence relation holds: $(X, q, qm, t, tm) \equiv (X, t, tm, q, qm)$. We will use both notations to describe the same conflict between t and q. The *merge rule* specifies whether a transaction t has merged all validation information of all participants.

Definition 5.1 *Merge rule*
$allmerged(t) \equiv vs[t] \wedge \forall X \in sites[t] : vm[t, X]$

The *commit rule* specifies whether a transaction that satisfies the merge rule can commit. A transaction t is not allowed to commit, as long as it is blocked by other transactions q. It remains blocked until either q restarts (not described in the algorithm), q reverses the blocking order, or the conflicts between t and q are resolved.

Definition 5.2 *Commit rule*
$cancommit(t) \equiv$
$\forall q : existsunresolved(t, q) \wedge$
$(t < q \wedge (q \in owned[t] \cup reversed[t])) \vee (q \in owned[t] \wedge q \notin ownedby[t] \cup waitfor[t])$

The *notification rule* specifies which transactions q have to be notified by transaction t, if t satisfies the merge rule, but cannot commit. All transactions q with $q > t$ that block t and are unaware of t have to be notified to prevent deadlock. The set of transactions that has to be notified is given by the following expression.

Definition 5.3 *Notification rule*
$tonotify(t) \equiv$
$[q \mid existsunresolved(t, q) \wedge t < q \wedge q \notin owned[t] \wedge$
$q \notin ownedby[t] \cup notified[t]]$

```
Ar(t,X,mode)
1  TI[t]∩ = TI[t, X]
2  sites[t]∪ = {X}

Vs(t)
3  foreach X ∈ sites[t] do enable V(t,X)
4  vs[t] := true

Vm(t,X)
5  conflicts[t]∪ = conflicts[t, X]
6  owned[t]∪ = owned[t, X]
7  ownedby[t]∪ = ownedby[t, X]
8  vm[t, X] := true
9  if allmerged(t) ∧ ¬cancommit(t) then
10     foreach q ∈ tonotify(t) do notified[t]+ = q; enable Wn(q,t)
11     foreach q ∈ toreverse(t) do waitfor[t]+ = q; enable Rw(q,t)
12 TTC(t)

TTC(t)
13 if allmerged(t) ∧ cancommit(t) then
14     Co(t) choose τ_t ∈ TI[t]
15         foreach X ∈ sites[t] do enable Ad(t,X)

Wn(t,q)
16 if τ_t = undefined then
17     conflicts[t]∪ = [(X, t, tm, q, qm)|(X, t, tm, q, qm) ∈ conflicts[q]]
18     waitfor[t]+ = q
19     enable Rw(q,t)

Rw(t,q)
20 if τ_t = undefined then
21     reversed[t]+ = q
22     TTC(t)

Cr(t,q,X)
23 if τ_t = undefined then
24     conflicts[t]∪ = [(Y, t, tm, q, qm)|(Y, t, tm, q, qm) ∈ conflicts[q]]
25     if (X, t, write, q, read) ∈ conflicts[t] then
26         TI[t]∩ = (τ_q, ∞); resolved[t]+ = (X, t, write, q, read)
27     if (X, t, write, q, write) ∈ conflicts[t] then
28         TI[t]∩ = (τ_q, ∞); resolved[t]+ = (X, t, write, q, write)
29     if (X, t, read, q, write) ∈ conflicts[t] then
30         TI[t]∩ = [0, τ_q); resolved[t]+ = (X, t, read, q, write)
31     TTC(t)
```

Algorithm 5.5: COORDINATOR CODE

The *reverse rule* specifies which transactions q take precedence over t, even if t blocks q. This rule becomes effective if t satisfies the merge rule, but cannot commit. Transaction t reverses the wait

order with all transactions q with $q < t$, and t blocks q. The set of transactions with whom the wait order should be reversed is given by the following expression.

Definition 5.4 ***Reverse rule***

$toreverse(t) \equiv$
$[q \mid existsunresolved(t, q) \wedge q < t \wedge q \in owned[t] \wedge$
$q \notin ownedby[t] \cup waitfor[t]]$

At the start of the execution of a transaction t, the coordinator of t needs to be notified of t's existence, to initialize the data structures that pertain to t. This initialization has not been included in the algorithm 5.5, instead we assume that all data structures are correctly initialized.

Access resolution event Ar(t,X,mode). The access resolution event is executed at the coordinator of t. It updates $TI[t]$ with $TI[t, X]$. At this point, the scheduler decides to restart t, if $TI[t] = \emptyset$. Restarts of transactions have not been described in the algorithm. The identity of participant X is stored in $sites[t]$. Finally, the event reactivates the transaction execution by returning the result of the data access (the algorithm does not describe this result).

Validation start event Vs(t). The validation start event is executed at the coordinator of t. It enables **V(t,X)** events at all participants X of t. The occurrence of the validation start event is administrated by the coordinator.

Validation merge event Vm(t,X). The validation merge event is executed at the coordinator of t. It merges the validation information of the **V(t,X)** event with other validation information in the coordinator of t. The occurrence of the event is administrated. Furthermore, the TTC-procedure is invoked.

procedure TTC(t). The TTC procedure (short for "try to commit") is not an event, but is invoked by **Rw()**, **Cr()** and **Vm()** events. As long as there is a participant Y for which **Vm(t,Y)** has not yet occurred, no actions are taken. Otherwise, the notification rule is invoked. This can lead to the enabling of **Wn(q,t)** events. If t satisfies the commit-rule, it commits, executing the **Co(t)** event in one atomic section (the **Co(t)** event is an exception to normal events that is explained above). If t cannot commit, the reverse rule is invoked. This can lead to the enabling of **Rw(q,t)** events.

Commit event Co(t). The commit event is executed at the coordinator of t. It is directly follows a **Wn()**, **Cr()** or **Vm()** event, in the same atomic section. It defines the timestamp τ_t and enables **Ad(t,X)** events for all participants X.

Conflict resolution event Cr(t,q,X). The conflict resolution event is executed at the coordinator of t. It ensures that the coordinator of t is aware of all conflicts between t and q. Furthermore, it resolves the conflict between t and q over X, according to the rules specified in algorithm 5.1. Finally, it invokes the TTC(t) procedure. The conflict resolution event directly accesses conflicts[q]. This information is included (by the coordinator of q) in the message that enables the administration event. In turn, the participant that executes the administration event includes the information in the message that enabled the conflict resolution event.

Reverse wait event Rw(t,q). The reverse wait event is executed at the coordinator of t. It notifies the coordinator that the blocking order between t and q is reversed. Furthermore, it invokes the TTC(t) procedure.

Wait notification event Wn(t,q). The wait notification event is executed at the coordinator of t. It notifies the coordinator of t that q is blocked by t. If t has not yet executed **Co(t)**, the **Rw(q,t)** event is enabled. Otherwise, no actions take place. Since the wait notification event directly accesses conflicts[q], this information is included in the message that enables the wait notification event.

5.5 Correctness

We proof that all schedules generated by DOCC-BF are conflict-serializable. Each schedule is conflict-equivalent to the timestamp-order $<_\tau$ that is generated by the algorithm, if the following restrictions on the time-stamps (taken from OCC-TI [57]) are maintained: 1) If t reads X before q writes X then $\tau_t < \tau_q$. 2) If t writes X before q reads X then $\tau_t < \tau_q$. 3) If t writes X before q writes X then $\tau_t < \tau_q$. Since the time-stamps are chosen from the respective TI-intervals, this translates to restrictions on the TI intervals. Requirement $\tau_t < \tau_q$ is satisfied if $TI[t] \subset [0, \tau_q) \vee TI[q] \subset (\tau_t, \infty)$. We prove that the DOCC-BF algorithm detects all conflicts between pairs of transactions t and q, and that each conflict is resolved exactly once, by adapting either $TI[t]$ or $TI[q]$. Suppose t and q have a conflict over data item X. We distinguish two cases. Either t or q (suppose t, without loss of generality) has performed the **Ad(t,X)** event before q performs the conflicting **Ac(q,X,mode)** event. Since **Ad(t,X)** updates $WTS[X]$ or $RTS[X]$, $TI[q]$ is modified in the matching **Ar(q,X,mode)** event. Alternatively, **Ac(t,X,mode)** $\prec$ **Ad(q,X)** $\wedge$ **Ac(q,X,mode')** $\prec$ **Ad(t,X)**. This case requires more elaborate analysis. The remainder of the correctness proof deals with this interleaving. We start with some preliminary results and definitions. We say that an event e *belongs to* transaction t, if the first part of e's parameter set refers to t. For example, event **Ac(t,X,read)** belongs to t, event **Rw(t,q)** belongs to t, and event **Cr(q,t,X)** belongs to q.

Definition 5.5 *An event e* belongs to *transaction t if the first part of e's unique identity refers to t.*

All data structures are monotone increasing according to the subset-ordering (i.e. an element added to a set is never removed). Therefore, if an element is not contained in a set after the transactions have finished execution, it was not contained in that set in earlier stages of the execution. By considering the state of the data structures after the execution of transactions is completed, we can abstract from different execution orders that have the same functionality.

Definition 5.6 *A system state is t-*maximal, *if no event that belongs to transaction t is enabled, and no event that belongs to t can be enabled in future states.*

We consider the data structures that pertain to t and q, in states that are both t-maximal and q-maximal.

Intuitively, if a state is t-maximal, and t did not complete its execution, t is either deadlocked or life locked. Also, if a state is t-maximal and q-maximal, and a conflict between t and q is not yet resolved, the protocol is not preserving conflict serializability. A number of basic properties of algorithms 5.3, 5.5 and 5.4 are presented below. They are used in the correctness proof, as well as in the deadlock and life lock freedom proofs.

1. **Basic transaction execution**. Each transaction t will execute an arbitrary number of data access events **Ac(t,X,mode)**, and exactly one validation start event **Vs(t)** (*algorithm 5.3*). Each data access **Ac(t,X,mode)** is followed by an access resolution event **Ar(t,X,mode)**: **Ac(t,X,mode)** $\prec$ **Ar(t,X,mode)** (*line 4 Ac(t,X,mode)*). For each accessed site X, one validation event follows the validation start, and one validation merge follows the validation: **Vs(t)** $\prec$ **V(t,X)** $\prec$ **Vm(t,X)**

(*line 3 Vs(t), line 11 V(t,X)*). The commit event **Co(t)** occurs after all validation events **Vm(t,X)**: **Vm(t,X)** $\prec$ **Co(t)** (line 13 TTC(t), definition $allmerged(t)$). For each validation event **V(t,X)**, an administration event **Ad(t,X)** follows (line 3 **Vs(t)**, line 15 TTC(t). All administration events are preceded by the commit event: **Co(t)** $\prec$ **Ad(t,X)** (line 13, 15 TTC(t)).

Ac(t,X,mode) $\prec$ **Ar(t,X,mode)** $\prec$ **Vs(t)** $\prec$ **V(t,X)** $\prec$ **Vm(t,X)** $\prec$ **Co(t)** $\prec$ **Ad(t,X)**

2. $\cup_{X \in sites[t]}$conflicts$[t, X] \subseteq$ conflicts$[t]$
   $\cup_{X \in sites[t]}$owned$[t, X] \subseteq$ owned$[t]$
   $\cup_{X \in sites[t]}$ownedby$[t, X] \subseteq$ ownedby$[t]$

   After **V(t,X)**, conflicts$[t, X]$ is constant (*line 7, 10 V(t,X)*). After **Vm(t,X)**, conflicts$[t, X] \subseteq$ conflicts$[t]$ (*line 5 Vm(t,X)*). Property 1 and line 3 **Vs(t)** show that **V(t,X)** and **Vm(t,X)** occur for all $X \in sites[t]$. Since conflicts$[t]$ is monotone, the result follows. Similar reasoning proves the result for owned$[t, X]$ and ownedby$[t, X]$.

3. $q \in$ owned$[t] \Leftrightarrow t \in$ ownedby$[q]$

   This follows from line 8 **V(t,X)**, **V(q,X)** event, and property 2.

4. **Ac(t,X,tm)** $\prec$ **Ad(q,X)** $\wedge$ **Ac(q,X,qm)** $\prec$ **Ad(t,X)** $\wedge$ $ConfRel(tm, qm) \Rightarrow (X, t, tm, q, qm) \in$ conflicts$[q] \cup$ conflicts$[t]$

   This follows from line 5 of **V(t,X)**, **V(q,X)**, and property 2.

5. $\exists(X, t, tm, q, qm) \in$ resolved$[t] \Rightarrow [(Y, t, tm, q, qm) | (Y, t, tm, q, qm) \in conflicts[q]] \subseteq$ conflicts$[t]$)

   Since $(X, t, tm, q, qm) \in$ resolved$[t]$, **Cr(t,q,X)** occurred (*line 26, 28, 30Cr(t,q,X)*). Hence the result (*line 24Cr(t,q,X)*).

6. $q \in$ owned$[t] \Rightarrow \exists(X, q, qm, t, tm) \in$ conflicts$[t]$

   This follows from line 6, 8 **V(t,X)**, and property 2.

7. $q \in$ ownedby$[t] \cup$ waitfor$[t] \Rightarrow \exists(X, q, qm, t, tm) \in$ conflicts$[t]$
   This follows from the following two statements.
   $q \in$ ownedby$[t] \Rightarrow \exists(X, q, qm, t, tm) \in$ conflicts$[t]$ follows from line 8, 9 **V(t,X)**.
   $q \in$ waitfor$[t] \Rightarrow \exists(X, q, qm, t, tm) \in$ conflicts$[t]$ follows from line 17, 18 **Wn(q,t)**, line 11 **Vm(t,X)**, line 10 **Vm(t,Y)** and definition of $tonotify(q)$ and $toreverse(t)$.

8. $q \in$ reversed$[t] \Rightarrow t < q \wedge \exists(X, q, qm, t, tm) \in$ conflicts$[q]$

   In event **Rw(t,q)**, q is added to reversed$[t]$. Two scenario's are possible. First, **Rw(t,q)** can be enabled by line 11 **Vm(q,X)**. From the definition of $toreverse(q)$ the result follows. Second, **Rw(t,q)** can enabled by line 19 **Wn(q,t)**, which in turn is enabled by line 10 **Vm(t,X)**. From the definition of $tonotify(t)$, and line 17, 18 **Wn(q,t)** the result follows.

9. $q \in$ reversed$[t] \Leftrightarrow t \in$ waitfor$[q]$

   In event **Rw(t,q)**, q is added to reversed$[t]$. Two scenario's are possible. First, **Rw(t,q)** can be enabled by line 11 **Vm(q,X)**, the result directly follows. Second, **Rw(t,q)** can enabled by line 19 **Wn(q,t)**, the result follows from line 18 **Wn(q,t)**.

10. Sets conflicts[t], ownedby[t], owned[t], reversed[t], waitfor[t] and resolved[t] are constant after **Co(t)**.

    The only events that modify these data structures are **Vm(t,X)**, **Wn(t,q)**, **Rw(t,q)** and **Cr(t,q,X)**. Of these events, **Vm(t,X)** cannot occur after **Co(t)** (property 1). The other events will not modify data structures, after **Co(t)** has occurred (lines 16, 20, 23). Line 20 and 23 have been added to simplify the proof-obligation, the protocol ensures that **Rw(t,q)** and **Cr(t,q,X)** events do not occur after **Co(t)**. This proof has been omitted to keep the correctness proof as short as possible.

The following lemma states that if t commits without solving a conflict over X with q, q will solve the conflict.

Lemma 5.7 *Suppose*
(A1) ***Ac(t,X,tm)*** $\prec$ ***Ad(q,X)*** *and* ***Ac(q,X,qm)*** $\prec$ ***Ad(t,X)***.
(A2) $ConfRel(tm, qm)$
(A3) ***Cr(t,q,X)*** *does not occur*
Then ***Cr(q,t,X)*** $\prec$ ***Co(q)***

Proof
Two cases: $\exists(Y,t,tm,q,qm) \in$ conflicts[t] or $\nexists(Y,t,tm,q,qm) \in$ conflicts[t] (exhaustive)
1 Suppose $\nexists(Y,t,tm,q,qm) \in$ conflicts[t]
2 $(X,t,tm,q,qm) \in$ conflicts[q] (1, (A1), property 4)
3 $q \notin$ ownedby[t] (1, property 7)
4 $t \notin$ owned[q] (3, property 3)
5 $t \notin$ reversed[q] (1, property 8)
6 $\forall(Y,t,tm,q,qm) \in$ conflicts[q] : $(Y,t,tm,q,qm) \in$ resolved[q] (4,5,commit rule)
7 QED. (2, 6, **Cr(t,q,X)**, property 10)
Second case:
1 Suppose $\exists(Y,t,tm,q,qm) \in$ conflicts[t]
2 $\exists(Y,t,tm,q,qm) \in$ conflicts[t] $-$ resolved[t] (1, property 4 and 5, (A1-A3))
Should transaction t resolve a conflict, (X,t,tm,q,qm) *is added to conflicts*[t]. *This conflict is not resolved by t (A3), due to statement 1 t has information about at least one unresolved conflict.*
3 $t < q \Rightarrow q \in$ owned[t] $\cup$ reversed[t] (2, commit rule)
4 $q < t \Rightarrow q \in$ owned[t] $\wedge\ q \notin$ ownedby[t] $\cup$ waitfor[t] (2, commit rule)
5 $t < q \Rightarrow t \in$ ownedby[q] $\cup$ waitfor[q] (3, property 3 and 9)
6 $q < t \Rightarrow t \in$ ownedby[q] $\wedge\ t \notin$ owned[q] $\cup$ reversed[q] (4, property 3 and 9)
7 $\exists(Y,t,tm,q,qm) \in$ conflicts[q] (5,6, property 7)
8 $\forall(Y,t,tm,q,qm) \in$ conflicts[q] : $(Y,t,tm,q,qm) \in$ resolved[q] (5, 6, 7, commit rule)
9 $[(X,t,tm,q,qm) | (X,t,tm,q,qm) \in$ conflicts[t]] $\subseteq$ conflicts[q] (7, 8, property 5)
10 $(X,t,tm,q,qm) \in$ conflicts[q] (9, property 4)
11 QED. (8, 10)
□

The following lemma states that if t resolves a conflict with q, q does not resolve conflicts with t. First, we observe that an **Cr(t,q,X)** event only resolves a conflict, if τ_t is undefined (line 23, or property 10).

Lemma 5.8 *If an* ***Cr(t,q,X)*** *event occurs before* ***Co(t)***, *no* ***Cr(q,t,Y)*** *event occurs before* ***Co(q)***.

Proof

1 Suppose **Cr(t,q,X)** $\prec$ **Co(t)** $\wedge$ **Cr(q,t,Y)** $\prec$ **Co(q)**
2 **Co(q)** $\prec$ **Ad(q,X)** $\prec$ **Cr(t,q,X)** $\prec$ **Co(t)** (1, property 1, line 15 **Ad(q,X)**)
3 **Co(t)** $\prec$ **Ad(t,Y)** $\prec$ **Cr(q,t,Y)** $\prec$ **Co(q)** (1, property 1, line 15 **Ad(t,Y)**)
4 **Co(q)** $\prec$ **Co(q)**, contradiction (2,3)

□

The following theorem proves the result. All conflicts are resolved exactly once. Furthermore, if transactions t, q have more than one conflict, all conflicts are resolved by the same coordinator.

Theorem 5.9 *All conflicts between transactions t, q are resolved exactly once, by the same coordinator.*

Proof

Property 4 shows that all conflicts between t and q are detected. Lemma 5.7 shows that at least one coordinator of t, q resolves the conflicts. Finally, lemma 5.8 proves that either t or q solves all conflicts between t, q, not both. This proves the result.

□

5.5.1 Deadlock freedom

A set transactions T is deadlocked if no transaction in T can commit, all transactions $t \in T$ have previously enabled **Vs(t)**, and there are no enabled events left. We assume that no new transactions enter the system (to avoid confusion with life lock).

Theorem 5.10 *The DOCC-BF algorithm is deadlock free.*

Proof

1 Suppose a set transactions T is deadlocked: $\forall t \in T : allmerged(t) \wedge \neg cancommit(t)$
2 Choose $t \in T : \forall q \in T : t \leq q$
3 $\exists(X, q, qm, t, tm) \in$ conflicts$[t]$ − resolved$[t]$: $q \notin$ owned$[t] \cup$ reversed$[t]$
(1, 2, commit rule)
4 $t \notin$ ownedby$[q] \cup$ waitfor$[q]$ (3, property 3, 9)
5 $q \notin tonotify(t)$ (1, line 10 **Vm(t,X)**)
If $q \in tonotify(t)$ during execution of line 10 ***Vm(t,X)****, q is added to notified$[t]$. Therefore, after execution of line 10* ***Vm(t,X)****, $q \notin tonotify(t)$. Since t is deadlocked, line 10* ***Vm(t,X)*** *has been previously executed.*
6 $q \in$ ownedby$[t] \cup$ notified$[t]$ (3, 5, notification rule)
Two cases:
7.1 Suppose $q \in$ ownedby$[t]$
7.2 $t \in$ owned$[q]$ (7.1, property 3)
7.3 $\exists(Y, t, tm, q, qm) \in$ conflicts$[q]$ (7.2, property 6)
7.4 $t \in$ waitfor$[q]$ (1, 5, 7.2, 7.3, reverse rule, line 11 **Vm(q,X)**, line 18 **Wn(q,t)**)
7.5 contradiction (4, 7.4)
Second case:
8.1 Suppose $q \in$ notified$[t]$
8.2 $t \in$ waitfor$[q]$ (1, 8.1, line 10 **Vm(t,X)**, line 18 **Wn(q,t)**)
8.3 contradiction (4, 8.2)

□

5.5.2 Life lock freedom

The DOCC-BF algorithm is life lock free if the domain of transaction identities is well-founded and non-dense (for example: the natural numbers), and that each transaction identity is only used once.

A transaction t is life locked, if its progress is blocked by an infinite number of transactions q that each block t for a finite time (if transaction q blocks t for an infinite time, it is a deadlock situation, not a life lock). It is shown that the protocol is life lock free under the assumptions that are stated above. First, we prove that all conflicts with transactions q that are detected by transaction t, are either detected during t's validation (the conflict over X is stored in conflicts[t, X]), or $q < t$. Therefore, only a finite number of transactions can block t, and therefore t can eventually commit. The protocol is life lock free.

Lemma 5.11 $(X, t, tm, q, qm) \in \mathit{conflicts}[t] \Rightarrow (X, t, tm, q, qm) \in \mathit{conflicts}[t, X] \vee q < t$

Proof

1 Suppose $(X, t, tm, q, qm) \in$ conflicts[t] $\wedge$ $(X, t, tm, q, qm) \notin$ conflicts[t, X] $\wedge$ $q > t$

2 **Wn(t,q)** does not occur (1, line 10 **Vm(q,X)**, notification rule)

3 QED. (1,2, only **Vm(t,X)**, **Wn(t,q)** can modify conflicts[t])

□

Lemma 5.12 *$|\mathit{conflicts}[t]|$ is finite*

Proof

Since the domain of transaction identities is well-founded and non-dense, the set $[q|q < t]$ is finite. The result follows from lemma 5.11, and the fact that |conflicts[t, X]| is finite.

□

Theorem 5.13 *The DOCC-BF algorithm is life lock free.*

Proof

Life lock occurs if a transaction t is blocked by an infinite set of other transactions q. A transaction t is only blocked by a transaction q if t and q have a conflict, and t detects this conflict. This means that the size of conflicts[t] is unbounded, if t is life locked. Lemma 5.12 proves that conflicts[t] is finite, hence t is not life locked.

□

5.6 Optimizing the algorithm

The algorithm that we described so far resulted from an extension of OCC-TI to a distributed environment. Modifications were applied to eliminate critical sections over multiple sites. No attempts have been made to optimize the resulting algorithm. In this section, several optimizations are introduced that improve the scheduler considerably. Apart from Thomas' write rule, these optimizations have not been used in the tests presented in section 5.8.

Applying Thomas' write rule

When two transactions p and q have a write-write conflict over X at site S, conflict serializability introduces the requirement "$\tau_p < \tau_q \Leftrightarrow$ **Ad(p,S)** $\prec$ **Ad(q,S)**" (the order in which transactions write X at site S should match the serialization order). In OCC-TI, a rather arbitrary choice is made, the requirement is strengthened with "**Co(p)** $\prec$ **Co(q)** $\Leftrightarrow$ **Ad(p,S)** $\prec$ **Ad(q,S)**" (i.e. if p commits before q, and p, q have a write-write conflict, $\tau_p < \tau_q$). Although easy to enforce, this additional requirement seriously reduces concurrency.

The optimization consists of two parts. First, we do no longer fix the order of write-write conflicts in the order that p and q commit. Second, write-write conflicts are resolved by applying Thomas' write rule (see for example [13]). This means that if $\tau_t < \tau_q$, and q writes a data item X before t, t's write is not performed. Since τ_q is administrated in $WTS[X]$, the rule becomes:

If transaction t wants to write data item X, and $\tau_t < WTS[X]$, then the write is not performed.

This simple rule eliminates the need to resolve write-write conflicts in the rest of the algorithm! Therefore, write-write conflicts are removed from the conflict-relation, and the **Ad(t,S)** event is modified with Thomas' write rule. The resulting schedule is no longer conflict-serializable, but still view-serializable. There is one requirement that has to be fulfilled, before Thomas' write rule can be applied: all time-stamps have to be unique. The reason for this is the following non-serializable scenario: two transactions p, q with identical time-stamps write the same items, on two sites A and B. On site A, p precedes q, on site B, q precedes p. If the second write is ignored, the database is inconsistent. If the second write is carried out, the database is still inconsistent. This symmetry is broken if all time-stamps are unique, if $p < q$, p's second write is ignored, and q's second write is performed. Uniqueness is easily realized by extending the timestamp for this check with the transaction identity (in a lexicographic ordering).

Combining multiple Cr() events

The previous optimization allows us to apply a second optimization. Suppose transactions p and q have a set of conflicts with each other, at several participants. Without loss of generality assume that p commits before q. After the first **Cr(q,p,X)** event, τ_p is available in the coordinator of q. Therefore, all read-write and write-read conflicts can be resolved directly. All other **Cr(q,p,Y)** events can be ignored.

Rerouting commit-information

Read-write and write-read conflicts between p and q can be resolved by q if τ_p is available. Instead of routing this information through the **Ad(p,X)** event, **Co(p)** can directly enable all **Cr(q,p,S)** events that relate to shared read-write or write-read conflicts.

If this optimization is combined with bundling of **Cr()** events, and Thomas' write rule, only one **Cr(q,p)** event is necessary for all shared read-write and write-read conflicts, and no **Cr()** event is necessary for write-write conflicts. Furthermore, it requires only one communication (between the two coordinators), instead of two (communication over the shared participant-site). Unshared conflicts still have to be routed over the **Ad()** event, since the necessary information is not available in **Co(p)**. However, a set of unshared conflicts can still be combined into one **Cr()** event. All other **Cr()** events are simply ignored.

Implicitly resolving conflicts

If transactions p and q have a conflict with requirement $\tau_p < \tau_q$, it can be implicitly resolved by the resolving of other conflicts. For example, suppose that previous validation resulted in requirements $\tau_p < 10$ and $20 < \tau_q$. The conjunction of these two requirements implies that the requirement $\tau_p < \tau_q$ holds! To apply this optimization, information about already resolved conflicts has to be stored at the sites where transactions access data. This requires that the (current) timestamp interval of a transaction t is piggy-backed on all communication with participants of t. Participants S store the value of $TI[t]$, at the last data access at S. This old $TI[t]$ value is called $TI[t, S]$. If a transaction t detects a conflict with transaction q during validation, the conflict is ignored if $TI[t, S] \cap TI[q, S] = \emptyset$.

Exploiting coordinator/participant site-sharing

In the presented algorithm, we have assumed that coordinators and participants execute on different sites, and all communication occurs through messages. In reality, data is often stored at the site where a transaction executes. This fact can be used to optimize local data accesses.

First of all, if a transaction t accesses a local data item, the **Ac()** and the **Ar()** event can be combined into one atomic event. Similarly, the local **V()** and **Vm()** can be combined with the **Vs()** event. Next, the event that commits t can be combined with the local **Ad()** event. With these optimizations, validating and committing a local transaction t is reduced to a single event, if t is not blocked. If t is blocked, an additional conflict resolution event is still required.

Second, suppose t locally accesses X. After t's local validation, transaction q accesses X in a conflicting mode. Normally, q would have an unshared conflict with t. With a small adaptation of the validation of q, this conflict can be changed to a shared conflict. Since q's validation of X occurs on the coordinating site of t, q can directly modify the data structures of the t's coordinator. Thus, q can ensure that the conflict is known to both t and q, the conflict over X is shared. Furthermore, there is a good probability that the conflict can be resolved implicitly, since the exact $TI[t]$ interval is available.

Exploiting coordinator/coordinator site-sharing

Similar to coordinator/participant overlap, two types of optimizations are possible when the coordinators of different transactions execute on the same site. First, unnecessary communication overhead can be avoided. Second, the availability of additional information and manipulation options allows for more scheduling freedom.

Suppose transactions p and q have the same coordinator. The following events can be combined, should they occur. The **Wn(p,q)** event can be combined with the **Rw(q,p)** event. The event that commits p can be combined with the **Cr(q,p)** event, and the **Rc(p)** event can be combined with the **It(q,p)** event. The wait order of transactions p and q that share the same coordinator can be changed by a few statements. Therefore, we relax the commit-condition of a transaction by the following rule: *Transaction p can commit if for each conflict between p and a transaction q over a data item X, either a conflict resolution took place in coordinator p, or p placed a delayed block on q, **or** p **and** q **share the same coordinator.***
Should a transaction p commit "out of turn" with this optimization, the conflicts between p and q are directly resolved by adapting $TI[q]$.

Online choice of pessimistic or optimistic scheduling

When p accesses a data item X that has just been validated by (conflicting) transaction q, it can be beneficial to wait for q's commit. Suppose p reads a data item X, to be written by q. If p reads X before q writes it, the resulting requirement is $\tau_p < \tau_q$. If p waits with data access until q has written X, the resulting requirement is $\tau_q < \tau_p$. By implicitly trying to resolve the conflict at access time, a guaranteed restart can be avoided by choosing the right course of action.

An example: suppose $\tau_p \in [30, 40]$ and $\tau_q \in [10, 20]$, due to earlier resolved requirements. If p reads before q writes X, either p or q has to be restarted. If p waits until q has written X, the conflict is implicitly resolved!

There are gray areas, where it is not clear whether p should wait, or continue: the TI intervals of p and q overlap. p can wait until τ_q is determined, before deciding to read before q writes, or after q has written. If p decides to read before q writes, then the waiting was unnecessary.

5.7 Performance analysis

Suppose transaction t reads $r = r_l + r_r$ data items, and writes $w = w_l + w_r$ data items (r_l and w_l denote local access, r_r and w_r denote remote access). A synchronization occurs at the start, and end of each transaction execution. The invocation of an **Ac**() event either requires a communication (if the item is stored remotely) or a synchronization (if the item is stored locally). Similar, the invocation of the **Ar**() event requires a communication, or (the same) synchronization. Communications related to remote reads are piggy-backed on the value request, and do not introduce additional overhead. Finally, control is returned to the transaction, which requires a synchronization.

During validation, the invocation of the **V**(), **Vm**(), **Ad**() and **Fi**() events are piggy-backed on the two phase commit protocol. The communications necessary for the **V**() and **Vm**() invocation only count as scheduler overhead if the transaction is restarted. If a transaction is not influenced by the execution of other transactions (no restarts occur), the overhead is described as follows.

$$O_{occbf} = (2 + r_l + w_l + r + w)Sync + w_r Comm + Int_{occbf}$$

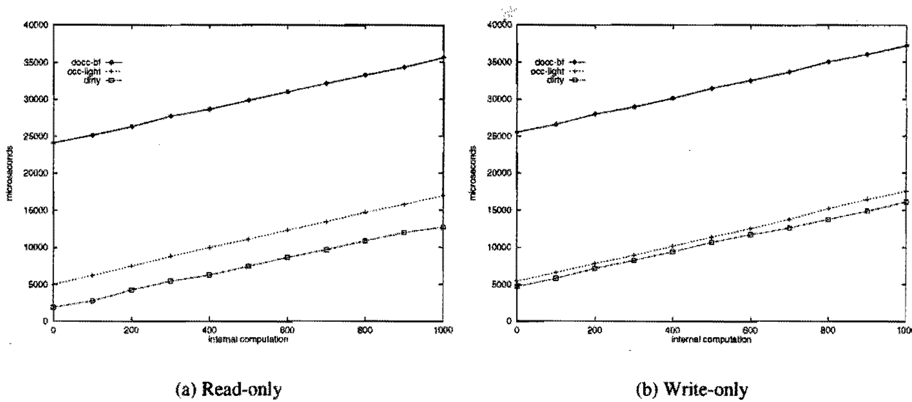
The set-manipulations that are used in the implementation of DOCC-BF introduce a significant internal-computation overhead that outweighs the overhead of synchronizations. This is partly caused by unoptimized implementation of DOCC-BF, and partly caused by the Erlang language that lacks efficient set-manipulation procedures.

The deadlock-prevention protocol can introduce two additional sequential messages: the wait notification and the reverse-wait. Similar to two phase locking, wait-chains of multiple transactions can occur (not contained in the performance analysis). If a transaction restarts, its execution so far counts as scheduler overhead. The overhead of a transaction that restarts V_r times is worst-case if the transaction completed execution every time, and it was only restarted just before the deadlock-prevention protocol allowed it to commit.

$$O_{occbf} = V_r * ((2 + r_l + w_l + r + w)Sync + (w_r + 4)Comm + Int_{occbf} + X)$$

5.8 Test results

DOCC-BF has been tested in a realistic environment, where background processes and other users occasionally influence the response times of transactions. These influences reappear in repeated testing, and are best visible in multi-site tests (see figure 5.7).



(a) Read-only

(b) Write-only

Figure 5.6: ONE SITE, NO CONCURRENCY

The hardware on which the Mnesia database was running varied between 1 and 8 SPARC 5 workstations, running Solaris. DOCC-BF is compared with the OCC-light scheduler (that has been designed in chapter 4), to show how the different designs lead to different performance.

A single dot on a curve averages the results from a 10 second test run for single-site tests, to 2 minutes for multi-site tests (to account for the different execution durations of transactions). Each machine that participated in an experiment contained a database site and one or two application processes. Each site consisted of 100 data items. Each application process executes a sequence of transactions, without delay between consecutive transactions.

5.8.1 Experiment 1: local transactions

This experiment serves to show that DOCC-BF imposes a significant overhead to the system. Mnesia is configured as a single site database. Experiments are conducted with read-only and write-only transactions that accessed 4 arbitrary data-items. Each transaction spends a fixed amount of time performing computation, varying from 0 to 200 computation loops before each data access. One computation loop occupies the processor for a limited amount of time (depending on the processor speed). Figures 5.6a and b show the test results. The experiment simulates an environment where transactions execute sequentially. The scheduler overhead is described by the following simplified formulas (derivation of scheduler overhead of OCC-light in chapter 4).

$$O_{docc-bf} = 2Sync + 2(r + w) \times Sync + Int_{docc-bf}$$

$$O_{occ-l} = 2Sync + Int_{occ-l}$$

The experiment shows that DOCC-BF has a considerable overhead that results from internal computation, the difference between OCC-light and DOCC-BF is almost constantly around 18000 microseconds. This cannot be attributed to the extra synchronization costs, which are (when 4 data items are accessed) in the order of $130 \times 8 \approx 1000$ microseconds.

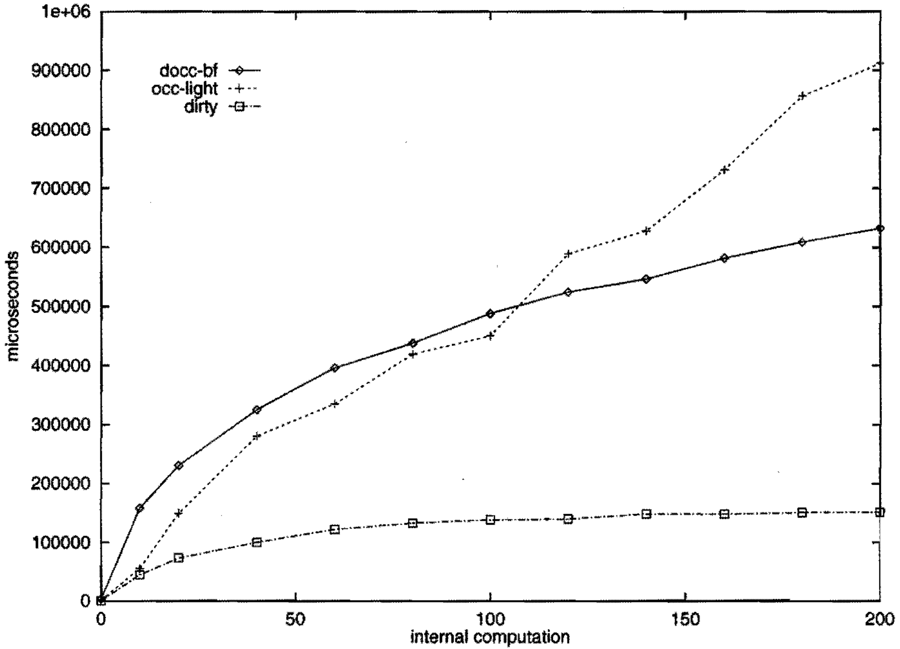


Figure 5.7: EIGHT SITES, WRITE ONLY

Since the absolute difference in performance is constant, the relative difference in performance between DOCC-BF and OCC-light becomes smaller, as the execution duration of transactions becomes longer. For transactions that perform no computation (0 computation loops), the factor between OCC-light and DOCC-BF is 5, while the factor is decreased to 2.25 if transactions perform 200 computation loops before each data access.

5.8.2 Experiment 2: remote data access, large conflict probability

The database is configured with 8 sites that each store the same number of data items. At each site, an application process continuously executes transactions. Transactions write 4 data items, and the number of internal computation loops that they perform is varied between 0 and 200 computation loops. Again, the DOCC-BF scheduler is compared with the OCC-light scheduler. In the formula below V denotes the number of validation phases that are required by OCC-light, and V_g denotes the number of validation phases that contained validation of remote data accesses (i.e. $V_g \leq V$). In the DOCC-BF formula, V_r denotes the number of reruns required by DOCC-BF.

$$O_{occ-l} = Sync + (V-1)X + V \times Sync + V_g \times 2Comm + Int_{occ-l}$$

$$O_{docc-bf} = V_r * ((2 + r_l + w_l + r + w)Sync + (w_r + 4)Comm + Int_{docc-bf} + X)$$

Although the two formulas given above cannot be exactly analyzed, they share a common factor X: $(V-1)X$ and $V_r X$ (we deliberately choose V_r different from V, to signify that these are two independent quantities). The validation check of DOCC-BF (dynamic adjustment of time intervals) rejects far less interleavings than the validation check of OCC-light. Therefore, $V_r \ll V-1$. As a result, if X increases, the overhead of OCC-light will have a steeper increase than the overhead of DOCC-BF.

Figure 5.7 shows the test results. As the execution-time of transactions becomes longer, the importance of scheduler overhead reduces, while the importance of concurrency increases. This can be seen from the figure, OCC-light outperforms DOCC-BF for short transactions, but is outperformed by DOCC-BF, if transactions are long. This is in accordance with our expectations, DOCC-BF offers more concurrency than OCC-light. Therefore, it outperforms OCC-light in high-conflict environments with long transactions.

5.9 Conclusions

The optimistic concurrency control algorithm OCC-TI [57] was adapted to a distributed environment. The main problem of distributing OCC-TI is the efficient regulation of concurrent validation at multiple sites. Our solution avoids bottlenecks by using a validation method that avoids unnecessary delays when possible: transactions are only blocked if further progress can lead to incorrect results. Therefore, transactions execute without any delay, until they are about to commit. At that point, a transaction can be blocked by a concurrently validating transaction. Postponing the moment that a transaction is blocked has two advantages. A transaction is unblocked, if the reason for the block (a concurrently validating transaction) leaves the system. Therefore, postponing the block reduces the total blocking duration. Second, protocols that use blocking have to avoid deadlocks: cyclic blocking. Deadlocks are especially difficult to solve in distributed systems, where information about blocking cycles is distributed over multiple sites. By postponing the blocking moment, (part of) the information about blocking cycles is collected by the coordinators of transactions. Therefore, the DOCC-BF algorithm is able to solve deadlocks of cycle length two without additional communication.

The bookkeeping that is necessary for the deadlock-prevention protocol is complicated by this delayed blocking. Future work should try to simplify the description of the algorithm. In this chapter, the focus has been on the correctness of the new protocol, rather than efficient implementation of the algorithm.

During implementation, it became clear that much more knowledge about the implementation language is required, than is generally found in the language documentation. Some language constructs were very time-consuming, while others were much faster. No indication about the relative speeds of the constructs was available. Since such information is essential for a fast implementation, a significant amount of time was spent on testing the (average) speed of language constructs. This information should be available beforehand, something that is lacking for most languages.

Even with an inefficient implementation, the goal of the DOCC-BF algorithm was met: it outperforms OCC-light in the high-conflict environment that was described at the beginning of this chapter. If the execution duration of transactions is long enough, the high scheduler overhead of DOCC-BF becomes a minor factor. DOCC-BF offers more concurrency than OCC-light, and can therefore utilize the available processors more efficiently than OCC-light.

Chapter 6

Predictable scheduling

The performance of schedulers has been the object of analytic study for many years [69, 65, 103, 90, 91, 105, 104]. Generally, it is hard to analyze schedulers that have been optimized for high performance. So far, most papers derive an expression for the average response time of a transaction [69, 65, 103, 90, 91, 105, 104] and in rare cases approximations of higher moments are derived [83, 82]. This thesis is part of STW project "Construction and performance of real-time transactions", and focuses on construction of high-performance schedulers. The accompanying thesis of S.A.E. Sassen [82] treats the performance analysis of optimistic concurrency control in depth. Expressions for the response time distribution of transactions in several environments are derived using (primarily) a decomposition-aggregation method.

In this chapter we show that the complexity of the performance analysis can be significantly reduced by a careful scheduler design. A family of so-called single queue static locking (SQSL) scheduling algorithms is presented. These schedulers have been designed with analyzability in mind. The design of the algorithms is not directly inspired by a specific environment, therefore the chapter does not contain an environment study. The scheduling problem is specified in section 6.1 and the scheduler designs are found in section 6.2. The performance analysis is given in section 6.3. This analysis has previously been published [17]. Simulations show the validity of the analysis in section 6.4, and relate the performance of schedulers that have not been analyzed. Finally, section 6.5 concludes the chapter.

6.1 Specification

The algorithms are designed for a shared-memory architecture. As the analysis and simulations in the second part of this chapter do not consider scheduler overhead, the relative speeds of communication and synchronization have not been specified. Transactions arrive in a continuous stream, and announce the dataset that they access, as well as their deadline and a best-case and worst-case approximation of their execution time. If a transaction misses its deadline, it becomes useless to the application, and it can be discarded. The scheduler should minimize the number of deadlines that are missed. A non-functional requirement that is not specified in our framework is analyzability: together with the algorithm design, a performance analysis is required. To analyze the system, additional assumptions about the scheduler input are required. These are given in section 6.3.

Generic scheduling problem.

platform: shared-memory	page 21
data: non-temporal	page 22
transactions: firm real-time, conflict-serializable, unannounced	page 24
objective function: minimize missed deadlines	page 31

Additional detailed information.

Fixed Variable	**Domain**	**Description**
$Access$	$T \to \mathcal{P}(DataSet)$	Transactions announce required data items.
Wx	$T \to$ seconds	Announced worst-case execution time of transactions.
Bx	$T \to$ seconds	Announced best-case execution time of transactions.

Fact constraint	**Description**
$\forall t \in T : Bx(t) \leq X(t) \leq Wx(t)$	The execution time of a transaction is correctly announced.
$\forall op \in OPS : Daccess(op) \in Access(trans(op))$	Transactions access items within their announced set.

6.2 A family of scheduling algorithms

The design of the scheduling algorithm has to satisfy two objectives: minimization of the number of missed deadlines, and it should be analyzable. To design a scheduler such that its real-time performance can be analyzed, it should be understood what properties of a scheduling algorithm make it difficult to analyze its performance. This depends both on the system (scheduler plus environment) under consideration, and the analysis method used. Since the system under consideration can suffer from transient overloads (no upper limit is specified on the number of transactions that arrive each second), hard real-time analysis does not yield useful results. Analysis methods that can deal with such transient overloads, and give probabilistic performance guarantees can be found in queuing theory. Markov models are a well-understood analysis tool, that has been used with success in stochastic operations research.

If a Markov model (see for a textbook [38]) can be constructed of the system under consideration, analysis techniques exist that derive the response time distribution of transactions. This gives a direct indication whether transactions meet their deadlines, or not. Markov models consist of a set of states, and state transitions. States should represent the system at different moments in time, and they have to satisfy the Markov property: they should be memoryless. This means that the past states of the system do not influence the next transition, only the current state influences the next transition of the system.

All systems that can be represented by a sequence of states can be represented by a Markov model. By including the entire history in the system state, the Markov property is trivially satisfied. As long as the size of the resulting Markov model is within certain bounds, these models can be analyzed in reasonable time. With the current state of the art, Markov models with up to 1000 states can be analyzed easily, and depending on the extra computing power available this can be scaled a few orders higher. This gives us a good indication whether a certain scheduler is analyzable using Markov models: as long as the state space of the resulting Markov model is within reasonable bounds, the

Markov analysis is feasible. Our goal will be to design a scheduler that can be analyzed using Markov models.

Design decision 6.1 *The scheduler that will be designed should be analyzable with Markov models.*

This decision restricts the available design choices. All scheduler optimizations that lead to infeasible large state spaces are excluded. Instead of designing just one scheduler, we design a family of schedulers. The first, basic scheduler is analyzed using Markov theory, while the more advanced schedulers stepwise introduce optimizations that improve performance, but also require large state spaces for their analysis. The analysis that holds for the basic schedulers will provide a worst-case analysis for the optimized schedulers.

Design decision 6.2 *A family of schedulers will be designed. The basic schedulers should be analyzable with Markov models, and the advanced schedulers should guarantee a higher performance.*

6.2.1 SQSL-soft

Before we introduce the new scheduler, let's look at the size of the state space that is required for a normal two phase locking scheduler. At any moment, each transaction t holds k_t locks. This information is relevant to the performance of each transaction, it determines the block-probability and the remaining execution time of each transaction. Suppose at most n transactions execute at the same time, and each will acquire k data items. In the order of k^n states are necessary to describe all possible states. For $k = 10$ and $n = 6$ this requires 1.000.000 states already, while information about lock-holding times and waiting transactions is not yet included. A scheduling algorithm is necessary that can be described by less states.

Two phase locking has at least k scheduler invocations, if a transaction accesses k data items. At each scheduler invocation, the dependencies of a transaction with other transactions are different: the number of held locks is different. This leads to a large state space. In order to reduce the state space, we will design a scheduler that requires only 2 scheduler invocations to schedule a transaction. As a side effect, this reduces the synchronization overhead of the scheduler, since scheduler invocations are used to synchronize transactions. Overhead is not further considered in this chapter.

Design decision 6.3 *The scheduler requires only 2 scheduler invocations for each transaction.*

Static locking is a variant of two phase locking that offers trivial concurrency. Instead of requiring locks one-at-a-time, all locks are requested at the start of each transaction. This reduces the state space considerably: a transaction either holds no locks, or all required locks, there is no in between.

Design decision 6.4 *The scheduler uses static locking to guarantee trivial concurrency.*

Once a transaction has acquired all locks, it can run to completion without further interference of other transactions. Unfortunately, transactions t can be life locked by a continuous stream of transactions that lock at least one of the data items that t needs. To remedy this drawback of static locking, we use a first-come, first-served (FCFS) strategy. A transaction that is waiting for execution is never overtaken by transactions that arrive after it.

Design decision 6.5 *The scheduler uses a single queue to store transactions that are waiting for execution.*

```
{ arrive, t, Access, Bx, Wx, Dl }
    Q.enqueue(t,Access, Bx, Wx, Dl)
    TTE()

{ leave, t, Access, Bx, Wx, Dl }
    fP+ = 1
    Locked− = Access
    TTE()

TTE()
    while ¬ Q.empty() ∧ fP > 0 do
        (t,Access,Bx,Wx,Dl):= Q.head()
        if Access ∩ Locked = ∅ then
            Q.dequeue()
            Locked∪ = Access
            fP− = 1
            sX.enqueue(t,Access,Bx,Wx,Dl)
        else break
```

Algorithm 6.1: SQSL-SOFT SCHEDULER

Concluding, transactions that arrive at the scheduler are executed first-come, first-served. The transaction at the head of the waiting queue can execute if all locks are available, and a processor is available. The remaining question is: how can this scheduling algorithm fit on a shared memory architecture? We have chosen for a setup that bears resemblance to process farming. One processor is dedicated to scheduling. This is called the *scheduler processor.* All other processors are dedicated to the execution of transactions, and are called *worker processors.* Transactions are passed from the scheduler processor through the shared-memory to the worker processors. A worker processor executes one transaction at a time, or is idle. This simple setup ensures that the arrival of new transactions in the system does not delay transactions that are already in execution.

This completes the design of the first scheduler. Since the designed scheduler does not abort transactions that miss their deadline, it is only suitable for soft real-time scheduling. Hence, we call the designed scheduler SQSL-soft: single queue static locking with soft deadlines. The SQSL-soft scheduler does not use the best-case and worst-case approximations of the transaction execution time, and does not use deadline information. Since the information is not used, it will not influence our Markov model. The more advanced schedulers will use this information, and require larger Markov models.

Algorithm

Time spent on scheduling decisions does not influence the execution time of transactions, since there is a dedicated scheduling processor. The time spent on scheduling is not taken into account in the analysis in section 6.3 and is assumed to be negligible compared to the execution durations of transactions. Transactions arrive at the scheduler processor, that receives an "arrive" message for each transaction. The scheduler processor places them in the shared memory if they can be executed. Worker processors retrieve transactions from the shared memory and execute them without further delay. Worker

```
{ start, t, Access, Bx, Wx, Dl }
   t()

{ finish, t, Access, Bx, Wx, Dl }
   sF.enqueue(t,Access, Bx, Wx, Dl)
```

Algorithm 6.2: SQSL-SOFT WORKER

processors execute one transaction at a time. If they are not executing a transaction, they are *idle*.

The scheduler maintains a set *Locked* that is the union of all *Access*-sets of executing transactions, and a FCFS-queue Q that stores transaction that are waiting for execution. Queues have the following methods operating on them: enqueue(X), dequeue(), head() and empty(). The enqueue method adds element X at the back of the queue. The dequeue method discards the first element of the queue. The head method returns the value of the first element of the queue. Finally, the empty method returns a boolean that signifies whether the queue is empty.

The number of idle processors is maintained in fP. In the shared memory, a queue sX of transactions that can be executed is stored, and a queue sF of finished transactions is stored. Simple polling algorithms check the state of the shared-memory queues sX, sF. These polling algorithms are not presented. We assume that if a transaction is finished, the scheduler processor is notified through a "leave" message, generated by the polling algorithm. Likewise, a worker processor that is unoccupied receives a "start" message from its local polling algorithm, if a transaction is retrieved from the sX queue. Algorithm 6.1 describes the algorithm that is executed on the dedicated scheduling processor. The execution of worker processors is described in algorithm 6.2.

Correctness. The correctness of the SQSL-soft algorithm is rather trivial: a transaction t does not execute concurrently with conflicting transactions, therefore the data access order is conflict-equivalent to the commit-order of transactions. Deadlock freedom is ensured, since transactions do not lock data items, unless they can run to completion, and release their locks without interferences. Life lock freedom is ensured by the single wait-queue. A transaction t is never delayed by transactions that arrive after t. Eventually, all transactions that arrived before t have finished execution and released their locks, and t can execute.

6.2.2 SQSL-firm

The second scheduler, SQSL-firm makes explicit use of information about deadlines and best-case execution times. The poor behavior of SQSL-soft under overload conditions (see test results in section 6.4) is corrected: late transactions are discarded instead of executed.

We define τ_{now} as the current time. A transaction t will miss its deadline if it does not start execution before $Dl(t) - Bx(t)$. Therefore, a waiting transaction t is discarded if $\tau_{now} > Dl(t) - Bx(t)$. If a transaction t starts execution before $Dl(t) - Wx(t)$ it is guaranteed to meet its deadline. If t starts execution after $Dl(t) - Wx(t)$ and before $Dl(t) - Bx(t)$, it either fails or succeeds, depending on $X(t)$. Should $Bx(t) = Wx(t)$, SQSL-firm executes transaction t only if t meets its deadline.

A timer is introduced for each processor. It has one parameter "activation", that specifies when the timer generates a local timeout-message "timeout". This timer is necessary to activate the scheduler when a transaction misses its deadline. If the exact execution duration $X(t)$ is unknown in advance,

```
{ start, t, Access, Bx, Wx, Dl }
    activation:= Dl
    active:= (t, Access, Bx, Wx, Dl)
    t()

{ finish, t, Access, Bx, Wx, Dl }
    activation:= ∞
    sF.enqueue(t,Access, Bx, Wx, Dl)

{ timeout }
    activation:= ∞
    (t, Access, Bx, Wx, Dl):=active
    abort(t)
    sF.enqueue(t,Access, Bx, Wx, Dl)
```

Algorithm 6.3: SQSL-FIRM WORKER

i.e. $Bx(t) < Wx(t)$, then it is possible that a transaction t misses its deadline while it is executing. The algorithm for worker processors is modified to handle this scenario. Algorithm 6.3 describes this modified worker processor algorithm. It sets the timer to the deadline of the transaction that is executing, and maintains the properties of the executing transaction in tuple "active". The scheduler aborts t when it receives the timeout message.

The algorithm for the scheduler processor is also modified, the new algorithm is given by algorithm 6.4. The TTE() procedure now deals with deadlines. First of all, the queue is only considered if at least one worker processor is idle ($fP > 0$). Second, transactions t at the head of the queue are discarded if they are guaranteed to miss their deadline ($\tau_{now} > Dl(t) - Bx(t)$). Otherwise, the scheduler checks if t is allowed to start execution, which is similar to SQSL-soft. If $\tau_{now} < Dl(t) - Bx(t)$ and t cannot execute, the timer of the scheduler processor is set to $Dl(t) - Bx(t)$. If $\tau_{now} = Dl(t) - Bx(t)$, the scheduler is reactivated by a timeout, and it discards t. We assume that τ_{now} has progressed $\epsilon > 0$ time units before procedure TTE() is invoked. The "start" and "finish" messages are handled like before.

6.2.3 SQSL-MLF

The first-come, first-served strategy of SQSL-firm can be improved by introducing a priority-queue. Transactions that are near their deadlines have precedence over other transactions. By taking the required execution time into account, the scheduler gives priority to transactions t with a low remaining laxity. The laxity is the time that transaction t can wait without missing its deadline: $Dl(t) - \tau_{now} - X(t)$. This is known as minimum laxity first (MLF), or shortest time to extinction (STE) [74]. Since $X(t)$ is unknown to the scheduler, the worst-case approximation $Wx(t)$ is used instead, which gives a pessimistic approximation of the laxity. By replacing the first-come, first-served queue with a priority queue, the SQSL-MLF algorithm is obtained from SQSL-firm. The priority queue has the same methods as a normal queue, but enqueue inserts elements $(t, Access, Dl, Wx)$ according to $Dl - \tau_{now} - Wx$ (smallest values first).

Transactions in the priority queue are dependent: if transaction t appears before q in the priority queue, $Dl(t) - X(t) \leq Dl(q) - X(q)$. Due to this dependency, it is no longer possible to repre-

```
{ timeout }
    TTE()

TTE()
    while ¬ Q.empty() ∧ fP > 0 do
        (t,Access,Bx,Wx,Dl):= Q.head()
        if τ_now > Dl(t) − Bx(t) then
            Q.dequeue()
        else if Access ∩ Locked = ∅ then
                Q.dequeue()
                Locked∪ = Access
                fP− = 1
                sX.enqueue(t,Access,Bx,Wx,Dl)
            else break
    if fP = 0 ∨ Q.empty then activation:= ∞
    else (t, Access, Bx, Wx, Dl) := Q.head()
        activation := Dl-Bx
```

Algorithm 6.4: SQSL-FIRM SCHEDULER

sent the state of the priority queue by a single number. The resulting state space is too large to use straightforward Markov analysis techniques.

6.2.4 SQSL-MLF with queue-skipping

The single queue becomes a bottleneck, if the transaction that is first in the queue is blocked for a long time. Processors can be idle, even while there are transactions waiting in the queue. To remedy this behavior queue-skipping is introduced, a transaction can execute out-of-turn if its execution does not delay the SQSL-MLF execution of transactions that it overtakes. We do allow that a skipping transaction t prevents other transactions q from skipping the queue. When the queue-skipping requirement is applied to FCFS queues, the analysis of the scheduler without queue skipping can serve as a worst-case analysis. However, this does not hold for the SQSL-MLF scheduler, since the queue is ordered according to the laxity. Therefore, it is possible that a transaction t skips the queue, just prior to the arrival of an urgent transaction q. In this worst-case scenario, the queue skipping optimization can decrease the performance of the system. However, on average the queue skipping optimization does increase performance (see section 6.4).

Let $GRBT$ be the guaranteed remaining blocking time of transaction q at the head of the waiting queue: there exists a transaction t that conflicts with q, which will continue execution for at least $GRBT$ time. Transaction q misses its deadline, if $\tau_{now} > Dl(q) - Bx(q) - GRBT$. This new requirement replaces test $\tau_{now} > Dl(q) - Bx(q)$. Since failed transactions leave the queue at an earlier stage, the time that they block other transactions in the queue is reduced. Therefore, the new requirement increases the overall performance. In fact, if the exact execution time is known in advance ($\forall t : Bx(t) = Wx(t)$), transactions at the head of the queue that are not immediately discarded will meet their deadline. To compute $GRBT$, a time-stamp $Held[X]$ is introduced for each data item X, that specifies the earliest moment that X can become available again.

Calculating whether a transaction t delays transactions that it overtakes can be quite involved,

if the calculation is exact. Instead, we simplify the queue-skipping rule such that it becomes less computationally intensive (but will allow less queue skipping than the exact queue skipping rule). Two scenarios are identified, in which a transaction t is allowed to "skip" the queue and start execution.

First, if t is certain to finish execution before the transaction q that is at the head of the queue can start execution or misses its deadline, then executing t out of order will not delay the SQSL-MLF scheduling of the queue. Second, if t uses a different dataset than other transactions in the queue, and sufficient worker processors are available, executing t will not delay other transactions in front of t.

Let w^i denote the ith transaction that is waiting in the queue (i.e. w^1 is the head of the queue). Now w^k is allowed to execute if the following two conditions are met. First, w^k should not conflict with transactions that are already in execution: $Access(w^k) \cap Locked = \emptyset$. Second, w^k should not violate the queue skipping conditions that were introduced above. There are two ways of satisfying them. Either w^k does not conflict with any transaction that is in front of w^k, and sufficient processors are available: $\forall i \in [1, k) : Access(w^i) \cap Access(w^k) = \emptyset \wedge fP \geq k - 1$, or transaction w^k has a total execution time that is shorter than the guaranteed blocking time, and at least one processor is available: $Wx(w^k) < GRBT \wedge fP \geq 1$.

The SQSL-MLF scheduler with queue skipping is obtained from the SQSL-MLF scheduler by replacing the TTE() procedure with algorithm 6.5. Furthermore, an extra method item(ix) is added to the priority queue, which returns the value of the ix^{th} item in the queue, and a new method remove(ix) removes the ix^{th} element from the queue.

6.2.5 Scheduler discussion

The SQSL-soft scheduler is a combination of static locking and first-come, first-served execution. These algorithms are well understood, and SQSL-soft is easy to implement. We have made no distinction between read and write operations, but this can be added without problems by modifying the $Access \cap Locked = \emptyset$ check. Due to the simplicity of SQSL-soft, correctness, deadlock freedom and life lock freedom proofs are straight-forward. Since SQSL-soft is aimed at soft real-time systems, it does not abort transactions that missed their deadlines. This lacking feature, together with the single queue can form a serious bottleneck. Long waiting queues can lead to the scenario where transactions start their execution after they have missed their deadline. This results in a low firm real-time performance.

SQSL-firm solves this problem by discarding transactions that can no longer meet their deadlines. This reduces the time spent on executing transactions that fail their deadline, thus increasing the time spent on executing transactions that meet their deadline. SQSL-firm uses the best-case execution time to determine whether a transaction can still meet its deadline. If this approximation is poor, the algorithm will often decide to execute transactions that will not meet their deadline. The performance of SQSL-firm increases if the difference between best-case and actual execution duration decreases. In the optimal case ($Bx(t) = X(t)$) transactions are always successful if they start execution.

SQSL-MLF favors urgent transactions that are close to being discarded over non-urgent transactions. This increases the firm real-time performance of the system, but introduces complex dependencies between the transactions in the waiting queue. To include these dependencies in a Markov model would increase the size of the state space several orders of magnitude, making straight-forward Markov analysis infeasible.

The queue-skipping optimization to SQSL-MLF improves performance on average, but introduces a considerable computation overhead. Without queue-skipping, the SQSL-MLF scheduler only needs to check the transaction at the head of the waiting queue. With queue-skipping, the access sets of at most $Ncpus - 1$ transactions are accessed whenever it is possible that a new transaction is executed.

```
TTE()
    while ¬ Q.empty() ∧ fP > 0 do
        (t,Access,Bx,Wx,Dl):= Q.head()
        GRBT := max_{X∈Access} Held[X] − τ_now
        if τ_now > Dl − Bx − GRBT then
            Q.dequeue()
        else if Access ∩ Locked = ∅ then
            fP− = 1
            Q.dequeue()
            foreach X ∈ Access do Held[X]:= τ_now + Bx(t)
            Locked∪ = Access
            sX.enqueue(t,Access,Bx,Wx,Dl)
        else skipping()
            break
    if fP = 0 ∨ Q.empty then activation:= ∞
    else (t, Access, Bx, Wx, Dl) := Q.head()
        activation := Dl-Bx

skipping()
    i:=1;
    while i≤ min(Q.length()-1, fP) do
        (t,Access,Bx,Wx,Dl) := Q.item(i+1)
        if τ_now > Dl − Bx
        then Q.remove(i)
        else if (Wx < GRBT ∨ Access ∩ (∪_{j∈[1,i]} Q.item(j).Access = ∅) ∧ Access ∩ Locked = ∅
            then fP− = 1
                Q.remove(i)
                foreach X ∈ Access do Held[X]:= τ_now + Bx(t)
                Locked∪ = Access
                sX.enqueue(t,Access,Bx,Wx,Dl)
            else i+=1
```

Algorithm 6.5: SQSL-MLF WITH QUEUE SKIPPING

Since scheduler overhead has been ignored in our simulations, SQSL-MLF with queue skipping outperforms SQSL-MLF. The advantage of queue skipping is a better processor utilization. The main bottleneck of SQSL-MLF is removed by the optimization: if the transaction at the head of the queue is blocked for a long time, it can be overtaken by unblocked transactions that appear after it in the queue.

6.3 Performance analysis

Markov analysis can be used to analyze the first two schedulers SQSL-soft and SQSL-firm. In this section we present the analysis of SQSL-soft. The analysis of SQSL-firm is similar to the analysis of SQSL-soft. Subsection 6.3 discusses the changes that are necessary to use the SQSL-soft analysis for SQSL-firm.

Markov model of SQSL-soft

The SQSL-soft scheduler has a simple structure, with straightforward dependencies between transactions. A first-come, first-served queue holds transactions that are waiting for a processor to execute them. Processors can be idle, even if transactions are waiting. In such a case, the transaction at the head of the waiting queue conflicts with an executing transaction.

By making a few additional assumptions about the input that arrives at the system, the system performance can be analyzed using Markov models. The assumptions are as follows. The arrival of transactions is a Poisson process with parameter λ. This is a common assumption that accurately models a wide range of applications. Arguments that support the validity of these assumptions can be found in [47] or [80]. Furthermore, execution times of transactions are independent and exponentially distributed with rate μ. Up to n transactions can be executing at the same time and the queue is unbounded. Since one processor is dedicated to scheduling, $n = Ncpus - 1$. We assume that the database stores a fixed number d of data items, and that each transaction accesses a data items. Finally, all items have an equal probability of being accessed.

System states. Under the presented assumptions, the system state is completely described by the tuple (i, j), where i is the number of executing and j the number of waiting transactions. When the number of executing transactions is lower than the number of available processors ($i < n$) and the number of waiting transactions is positive ($j > 0$), the first transaction in the queue has a data conflict with at least one executing transaction. If all processors are executing transactions, it is unknown whether the first transaction has a data conflict. These observations are in fact extra pieces of information that are incorporated into the state description. They lead to the following definition of the states (i, j).

$i = 0$ and $j = 0$:	The system is empty.
$1 \leq i < n$ and $j \geq 0$:	i mutually non-conflicting transactions are executing, and j transactions are waiting, of which the first (if any) has a conflict with at least one of the i executing transactions.
$i = n$ and $j \geq 0$:	i mutually non-conflicting transactions are executing and j transactions are waiting.

Some probabilities. Let $B(i)$ be the probability that a transaction has a data conflict with one or more out of i executing transactions. If transaction t at the head of the queue has a data conflict with at least one of i executing transactions, $B(i-1 \mid i)$ is the probability that t still has a data conflict with at least one of the remaining $i - 1$ executing transactions, after one of the i executing transactions has left. Conflict probabilities $B(i)$ and $B(i - 1 \mid i)$ are:

$$B(i) = 1 - \binom{d - ai}{a} \Big/ \binom{d}{a} \text{ and } B(i - 1 \mid i) = \frac{B(i-1)}{B(i)}.$$

These equations hold as long as $a(i + 1) < d$, which is the case for all realistic purposes. The expression for $B(i-1 \mid i)$ was derived by applying Bayes' formula $B(i-1 \wedge i) = B(i-1 \mid i) \cdot B(i)$. Here $B(i-1 \wedge i)$ is the probability that a transaction conflicts with at least one out of $i-1$ transactions and also conflicts with at least one out of i transactions; the $i - 1$ transactions are a subset of the i transactions. Therefore, $B(i - 1 \wedge i) = B(i - 1)$ and the result follows. As the complements of $B(i)$ and $B(i - 1 \mid i)$ are often used, we define $A(i) = 1 - B(i)$ and $A(i - 1 \mid i) = 1 - B(i - 1 \mid i)$.

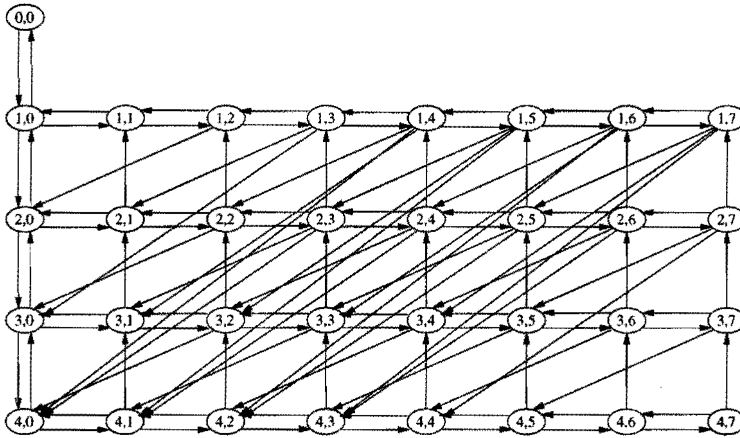


Figure 6.6: A MODEL OF STATIC QUEUEING FOR $n = 4$

Markov property. The processing of transactions can be described by a continuous time Markov chain with state descriptor (i, j). This follows from the exponential, memoryless inter-arrival and execution times, the fixed number of items used by each transaction, and the fact that all items have an equal probability of being accessed. The future state of the system depends on the current state (i, j) and not on the previous states: the Markov property holds.

Transitions of the Markov model. We analyze what state transitions are possible in the model. Transactions arrive at the system with rate λ. If there are no waiting transactions, at least one processor is free, and i transactions are executing, with probability $B(i)$ the arriving transaction is blocked and with probability $A(i)$ is allowed to execute:

$$(i, 0) \rightarrow (i, 1) \text{ with rate } \lambda B(i) \text{ if } i < n.$$

$$(i, 0) \rightarrow (i + 1, 0) \text{ with rate } \lambda A(i) \text{ if } i < n.$$

If the number of waiting transactions j is greater than zero, or no processor is available ($i = n$), the arriving transaction enters the queue:

$$(i, j) \rightarrow (i, j + 1) \text{ with rate } \lambda \text{ if } i = n \text{ or } j > 0.$$

Second, if $i > 0$ transactions are executing, a transaction finishes its execution at rate $i\mu$. If the queue is empty, a finished transaction is not replaced:

$$(i, 0) \rightarrow (i - 1, 0) \text{ with rate } i\mu \text{ if } i > 0.$$

If at least one transaction is waiting and $i = n$, with probability $B(n - 1)$ the first transaction in the queue remains blocked at a transaction completion epoch. With probability $A(n - 1)$ the first transaction is not blocked and can start executing:

$$(n, j) \rightarrow (n - 1, j) \text{ with rate } n\mu B(n - 1) \text{ if } j > 0.$$

$$(n, j) \to (n, j-1) \text{ with rate } n\mu A(n-1) \text{ if } j > 0.$$

If $j > 0$ and $i < n$ just before a transaction completes execution, with probability $B(i-1 \mid i)$ the first transaction f in the queue remains blocked:

$$(i, j) \to (i-1, j) \text{ with rate } i\mu B(i-1 \mid i) \text{ if } i < n \text{ and } j > 0.$$

With probability $A(i-1 \mid i)$ f begins execution. Now if a processor is still available, and the queue is not empty, the transaction r that is next in line is available for execution. With probability $B(i)$ it is blocked:

$$(i, j) \to (i, j-1) \text{ with rate } i\mu A(i-1 \mid i)B(i) \text{ if } i < n \text{ and } j > 1.$$

With probability $A(i)$, r is allowed to execute. Now if yet another processor is available, and the queue is still not empty, a third transaction is available for execution. With probability $B(i+1)$ it is blocked:

$$(i, j) \to (i+1, j-2) \text{ with rate } i\mu A(i-1 \mid i)A(i)B(i+1) \text{ if } i < n-1 \text{ and } j > 2.$$

With probability $A(i+1)$ the transaction executes. As long as there is still a processor available and the new first transaction in the queue does not conflict with the transactions in execution, the scheduler admits a new transaction to a free processor. The transitions that can arise and their rates are included in the following, summarizing expression. When $j > 0$, a departure can cause the following state transitions

$$(i, j) \to (i-1+k, j-k)$$

$$\begin{aligned}
&\text{with rate } n\mu B(n-1) \quad \text{if } i = n \text{ and } k = 0\\
&\text{with rate } n\mu A(n-1) \quad \text{if } i = n \text{ and } k = 1\\
&\text{with rate } i\mu B(i-1 \mid i) \quad \text{if } i < n \text{ and } k = 0\\
&\text{with rate } i\mu A(i-1 \mid i)\prod_{m=0}^{k-2} A(i+m) \quad \text{if } i < n \text{ and } k = \min\{j, n-i+1\} > 0\\
&\text{with rate } i\mu A(i-1 \mid i)\prod_{m=0}^{k-2} A(i+m)B(i-1+k) \text{ if } i < n \text{ and } 0 < k < \min\{j, n-i+1\}.
\end{aligned}$$

The convention is used that $\prod_{m=0}^{\ell} = 1$ if $\ell < 0$.

Figure 6.6 shows the possible transitions when $n = 4$, and queues are at most 7 transactions long. Observe that the system is cyclic, even and odd states can be defined by counting the distance to (0,0). All transitions are between an even and an odd state. Figure 6.7 gives a close-up of the transitions to and from (1,6), with their intensities.

Steady-state distribution

Let vector π denote the steady-state distribution of the Markov model described above. Then $\pi(i, j)$ is the probability that in the long run the system is in state (i, j). The steady state distribution is used in section 6.3 to compute the moments of the response time. We are primarily interested in the first two moments: the average and the variance of the response time.

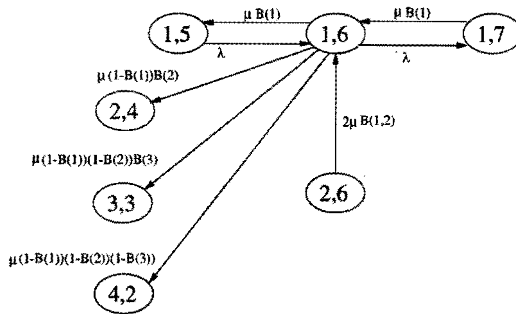


Figure 6.7: TRANSITIONS TO AND FROM (1,6) IF $n = 4$

Approximating the steady state by truncating the state space. Balance equations can be derived from the expressions in the previous section, by applying the "rate out of state (i, j) = rate into state (i, j)" principle. A Markov generator Q is constructed by combining all transitions defined above. Solving the balance equations $\pi Q = 0$ gives the steady state distribution π of the Markov chain.

Markov generator Q is a matrix of infinite size, since the number of states is unbounded. This makes it difficult to solve the balance equations. We computed the steady state probabilities numerically from the balance equations by truncating the state space and thus bounding the size of Q, at a sufficiently high number J of waiting transactions. The probability that transactions arrive in states where $\geq J$ transactions are waiting should be negligible. Therefore, J is related to parameters n, λ, μ, a and d.

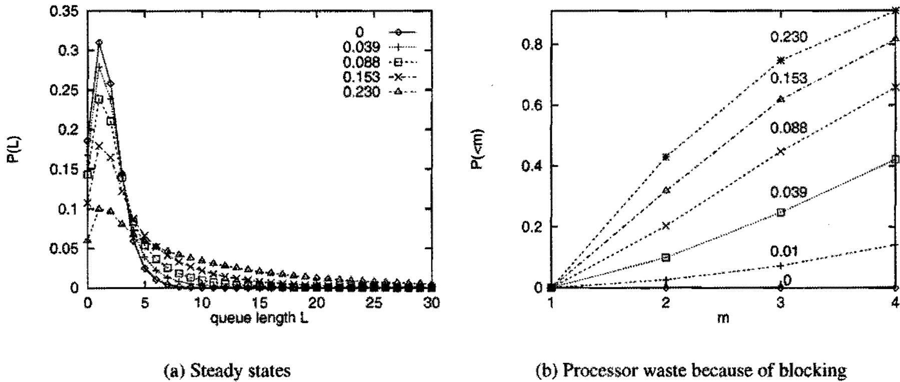
The matrix geometric approach instead of truncation. The steady state probabilities can be computed without truncating the state space, by using the matrix geometric approach [70]. Let level j represent all states with j waiting transactions, and define π_j as the vector $(\pi(0, j) \ldots \pi(n, j))$. Then steady state distribution π_j has the geometric form:

$$\pi_j = \pi_1 R^{j-1} \quad \text{for } j \geq 1.$$

Matrix R is the unique solution to a matrix equation of order $n+1$. This is the equation $\sum_{k=0}^{n+1} R^k A_k = 0$ where the A_k's are specific $n \times n$ sub-matrices of Markov generator Q. Matrix R can be solved numerically by successive substitution, starting with $R = 0$.

Steady state vector π_0 is also determined by R. Unfortunately, for most choices of n an explicit expression for R cannot be obtained, because of the high order of the matrix equation. So even when Neuts's matrix geometric approach is used, the steady state probabilities can only be computed numerically.

Drawing conclusions from the steady state. In Figure 6.8(a), the steady state distributions of the total number of transactions in the system are drawn for different conflict probabilities. The parameters used are $\lambda = \frac{5}{3}$, $\mu = 1$ and $d = 100$. The number a of data items used by a transaction was varied between 0 and 5 to obtain the different conflict probabilities. The figure shows that high conflict probabilities result in longer queues, as some transactions have to wait because of a conflict.



(a) Steady states

(b) Processor waste because of blocking

Figure 6.8: STEADY STATES AND PROCESSOR USAGE FOR $n = 4$ AND VARIOUS a

In Figure 6.8(b) the effect of blocking on processor usage has been plotted. Depicted is the percentage of time that $< m$ processors are executing transactions, given that there are $\geq m$ transactions in the system, for $m = 1$ to 4. The different graphs correspond to blocking probability 0, 0.01, 0.039, 0.088, 0.153 and 0.230. If the conflict probability is 0, transactions never wait if a processor is available. When conflicts become more frequent, the usefulness of a third and fourth processor drops dramatically. For example, in a system with conflict probability 0.153, $\approx 60\%$ of the time that there are transactions waiting for execution, at most two processors are executing transactions, the other processors are idle. Adding processors hardly increases the performance of the system, the high conflict probability limits the number of transactions that are allowed to execute concurrently. Increasing the processing speed does result in a significant performance increase. Therefore, in systems where the conflict probability is high, a few powerful processors result in more performance gains than a large number of less powerful processors.

The steady state of the system gives information about the throughput, average queue-lengths and processor activity. If execution times differ substantially, knowledge of the average execution time is not sufficient to guarantee that deadlines are met. Information about the distribution of the response time is needed.

The response time distribution

The distribution of the response time S of a transaction is usually described by the moments of the response time. We aim to find $E[S^k]$, the k-th moment of S, for $k \geq 1$. The average response time $E[S]$ of a transaction can be computed using Little's rule $E[L] = \lambda E[S]$. Here $E[L]$ is the average number of transactions in the system and is obtained from the steady state probabilities: $E[L] = \sum_{i+j>0}(i + j)\pi(i, j)$.

For $E[S^k]$ with $k > 1$ no direct relation between $E[S^k]$ and $E[L]$, $E[L^2]$ up to $E[L^k]$ is known for systems like this. To derive an expression for $E[S^k]$ we define a recursive relation that depends on $E[S^l]$ ($l < k$), and depends on moments of exponentially distributed stochastic variables. This

process is described below.

A recursive relation for the response time. We follow the path of an arbitrary transaction through the model, from arrival to departure. With a 'path' we mean the states that are reached during the presence of the transaction under consideration. Tuple $[i, j]$ describes the situation where i transactions are in execution and $\geq j$ transactions are waiting in the queue. The tuple (i, j) refers to the system state as defined before. Define

$$S_{[i,j]} = \text{the time until a transaction } t \text{ leaves the system, when } i \text{ transactions are executing, and } j-1 \text{ transactions are ahead of } t \text{ in the queue.}$$

If $j = 0$, the transaction under consideration is in execution. When the system is in state (i, j) after an arrival, $S_{[i,j]}$ is the response time of the newly arrived transaction. Important is the observation that $S_{[i,j]}$ does not depend on transactions that arrive at the system after the transaction under consideration. This follows from the property of the SQSL-soft scheduler that transactions waiting in the queue cannot be overtaken.

The consequence of this observation is that arrivals of other transactions need not be considered when $E[S^k_{[i,j]}]$ is computed. This enables us to concentrate on the departures only. Let X_i be the time till the next departure when i transactions are executing. X_i is exponentially distributed with rate $i\mu$. Let $p_{[i,j][m,\ell]}$ be the probability that the next departure leads to a state with m transactions in execution and $\ell - 1$ transactions present in the queue ahead of the transaction under consideration. Then

$$S_{[i,j]} = X_i + S_{[m,\ell]} \qquad \text{with probability } p_{[i,j][m,\ell]} \text{ for all } [m, \ell].$$

This is a recursive relation, as $m + \ell = i + j - 1$. Therefore, the moments of $S_{[i,j]}$ can be computed from the moments of $S_{[m,\ell]}$ with $m + \ell < i + j$. Once a transaction is in execution, its service time is exponentially distributed with mean $1/\mu$. Thus the boundary condition for the recursion is $S_{[i,0]} = X$ for all $i > 0$, where X is exponentially distributed with parameter μ.

Let $a_{(r,\ell)(i,j)}$ be the probability that a transition to state (i, j) is caused by an arbitrary transaction t that sees state (r, ℓ) on arrival. An expression for t's response time S is found by conditioning on state (r, ℓ) and by using the PASTA [101] property:

$$S_{[i,j]} \qquad \text{with probability} \sum_{(r,\ell):i+j=r+\ell+1} \pi(r, \ell)a_{(r,\ell)(i,j)}.$$

Deriving the moments of the response time. The moments of the response time are derived directly from the recursive relation. Two important rules are used to find $E[S^k]$ for $k \geq 1$.

- **Choice**. The transaction follows path l with k-th moment $E[S^k_l]$, or path m with k-th moment $E[S^k_m]$. The probability that path l is taken is p. Then

$$E[S^k] = pE[S^k_l] + (1 - p)E[S^k_m].$$

- **Addition**. The transaction first follows path l with duration S_l, followed by path m with duration S_m. Then

$$E[S^k] = E[(S_l + S_m)^k].$$

Based on these rules, the moments of S can be found using dynamic programming.

Example. Suppose $n > 2$ and we need the second moment of the response time of transaction t that is first in the queue while two transactions are in execution (situation [2, 1]). Transaction t must wait until a transaction leaves. This time is represented by random variable X_2 which is exponentially distributed with parameter 2μ. After the departure, t remains blocked with probability $B(1 \mid 2)$. Otherwise t begins execution. Using both choice and addition rule, the expression becomes:

$$\begin{aligned} E[S^2_{[2,1]}] &= A(1 \mid 2)E[(X_2 + S_{[2,0]})^2] + B(1 \mid 2)E[(X_2 + S_{[1,1]})^2] \\ &= A(1 \mid 2)(E[S^2_{[2,0]}] + 2E[S_{[2,0]}]E[X_2]) + \\ &\quad B(1 \mid 2)(E[S^2_{[1,1]}] + 2E[S_{[1,1]}]E[X_2]) + E[X_2]^2. \end{aligned}$$

To find $E[S^2_{[2,1]}]$, the first and second moment of X_2, $S_{[2,0]}$ and $S_{[1,1]}$ must be known. In general, for $E[S^k_{[i,j]}]$, moments $E[S_{[m,\ell]}], \ldots, E[S^k_{[m,\ell]}]$ with $m + \ell < i + j$ and the first k moments of exponentially distributed variables are needed. With the choice rule, the second moment of the response time of an arbitrary customer is found.

$$E[S^2] = \sum_{(r,\ell)} \pi(r,\ell) \sum_{(i,j):i+j=r+\ell+1} a_{(r,\ell)(i,j)} E[S^2_{[i,j]}].$$

Approximating the response time distribution by fitting

We use a mixture of exponentially distributed variables that has the same moments of S to approximate the response time distribution. The way this mixture is chosen influences the quality of the approximation. Denote the stochastic variable corresponding to the chosen mixture by $\hat{S}$. Then $P(S \leq x)$, the probability that a transaction meets its deadline, is approximated by $P(\hat{S} \leq x)$. We say the distribution of $\hat{S}$ is fitted to the moments of S. We used the two-moment fit as described in [94]. The fitting procedure can fit a distribution to any combination of $E[S]$ and $E[S^2]$.

Adapting the Markov model for SQSL-firm

SQSL-firm uses information about $Dl(t) - Bx(t)$ to decide whether a transaction t is discarded. This can be incorporated in the model, without increasing the state space, if the following assumptions are added. First, it is assumed that the actual execution times of transactions match the best-case approximation that is input to the scheduler: $\forall t : Bx(t) = X(t)$. Second, we assume that the initial laxity $Dl(t) - Bx(t)$ is exponentially distributed with parameter d. This preserves the memoryless property of the states.

Algorithm 6.4 discards late transactions when they arrive at the front of the queue. This is functionally equivalent to an immediate discard of a transaction, if its slack time becomes less than 0 (although the second implementation requires more computational overhead). The transaction discard introduces a set of new state transitions. Only transactions that are in the queue can miss their deadline, because of the slack time definition. If there is more than one waiting transaction ($j > 1$), a transaction that is not first in line can miss its deadline:

$$(i, j) \rightarrow (i, j-1) \text{ with rate } (j-1)d$$

This transition can also result from an execution completion event, so the rates have to be combined by simple addition.

More complex transitions can result if the first transaction in the queue misses its deadline. The next transaction in line is no longer blocked, and might be able to execute instantly. Therefore, up to

$n-1$ transitions are possible if the first transaction misses its deadline, since at most $n-1$ processors are idle if the queue is not empty. Suppose p transactions enter execution $(0 \leq p \leq n-i)$. If the waiting queue is not empty after the transition, the possible transitions are described as follows.

$$(i, j+1) \rightarrow (i+p, j-p) \text{ with rate } dB(i+p)\Pi_{x=0}^{p-1}A(i+x)$$

If the queue is empty after the transition, the possible transitions are:

$$(i, p+1) \rightarrow (i+p, 0) \text{ with rate } d\Pi_{x=0}^{p-1}A(i+x)$$

We assume that $B(n) = 1$, as no processors are available for execution.

The path through the system. There are two ways of determining the success probability of a transaction. The first way is to replay the analysis of SQSL-soft with the new transitions, and to use the resulting distribution $P(X < c)$ to check $P(X < Y)$, where Y is exponentially distributed with parameter d. The second way is to determine $P(\text{success})$ directly when following the path of a transaction through the system. In each state $[i, j]$, there is a probability $P(d_{[i,j]})$ that the transaction under consideration misses its deadline.

$$P(d_{[i,j]}) = \frac{d}{i\mu + jd + d}$$

The total probability that a transaction meets its deadline can be computed with numerical methods, not unlike the way the moments are computed. For example, suppose state $[i, j]$ can change into state $[i, j-1]$ or $[i-1, j]$ with probability p and $1-p$. The success probability $P(s_{[i,j]})$ is now:

$$P(s_{[i,j]}) = (1 - P(d_{[i,j]}))(pP(s_{[i,j-1]}) + (1-p)P(s_{[i-1,j]}))$$

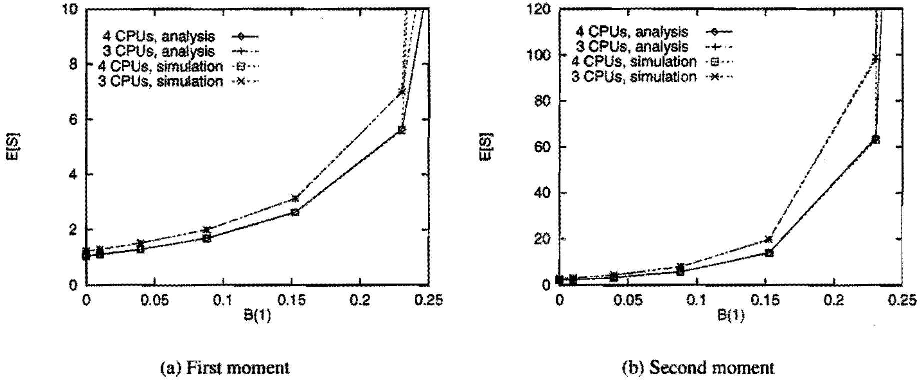
This eliminates the need for the approximate fitting procedures.

6.4 Comparison with simulation results

All four SQSL schedulers were simulated using discrete time simulation. The simulations use the same assumptions about the input of the system as the analysis in 6.3. In addition, it is assumed that transactions exactly announce their execution time: $\forall t : Bx(t) = Wx(t)$, and that the initial laxity of each transaction $Dl(t) - Bx(t)$ is exponentially distributed with parameter 4μ. Subsection 6.4.1 compares the results from the Markov analysis with the simulations. Subsection 6.4.2 compares the performance of all four SQSL schedulers, using simulations.

6.4.1 Simulation compared to fitting

A simulation of the system has been programmed, and several test runs were made. Again, $\lambda = \frac{5}{3}$ and $\mu = 1$ was used. In Figure 6.9, the simulation and analysis results for $E[S]$ and $E[S^2]$ are shown. The results show that the truncation of the state space results in inaccuracies when the system is not stable. For $n = 4$ we used moments $E[S]$ and $E[S^2]$ from our analysis to approximate $P(S \leq x)$ for $x = 1, 3, 5$ and 7. Conflict probability $B(1)$ was varied from 0 to 0.230, corresponding to 0 to 5 data accesses in a database with 100 data items. We also estimated $P(S \leq x)$ by simulation. The results of both analysis and simulation are given in table 6.1. The simulated values are the midpoints of a 95% confidence interval with a width smaller than 0.02. Table 6.1 shows that the fitting procedure gives an excellent approximation of the response time distribution.



(a) First moment

(b) Second moment

Figure 6.9: ANALYSIS AND SIMULATION FOR $n = 3$ AND $n = 4$

6.4.2 Comparison of the schedulers

The four SQSL schedulers were tested in a number of simulations, which varied in the conflict probability. Note that for SQSL-soft and SQSL-firm, simulation and analysis results match. In figure 6.10 the arrival rate of transactions is varied between 0.5μ and 3.5μ. The conflict probability between two transactions is fixed to 0.39, and the number of processors is 4.

The SQSL-soft scheduler loses all real-time performance if the system becomes overloaded. A long queue of waiting transactions builds, as arrival of transactions is higher than the throughput of the system. The other schedulers do not suffer from this problem, as they discard late transactions, long queues can never build. The increase in performance is remarkable, even at low arrival rates a performance difference of 35% exists between SQSL-soft and SQSL-firm. In fact, SQSL-firm hardly discards transactions for low arrival rates. It appears that by discarding a few "bad" transactions, almost all other transactions meet their deadlines.

The performance of the more complex schedulers is higher than SQSL-firm, although the gain

	$P(S \leq 1)$		$P(S \leq 3)$		$P(S \leq 5)$	
$B(1)$	Fit	Sim.	Fit	Sim.	Fit	Sim.
0	0.61	0.61	0.95	0.95	0.99	0.99
0.010	0.59	0.59	0.94	0.94	0.99	0.99
0.039	0.54	0.54	0.90	0.90	0.98	0.98
0.088	0.45	0.45	0.83	0.83	0.95	0.95
0.153	0.32	0.32	0.68	0.68	0.85	0.85
0.230	0.16	0.17	0.41	0.42	0.59	0.60

Table 6.1: RESULTS FOR THE RESPONSE TIME DISTRIBUTION

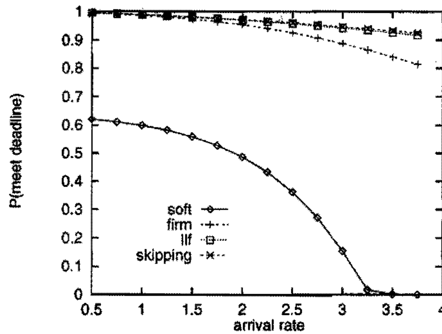
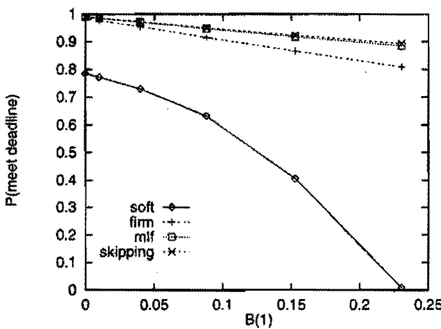


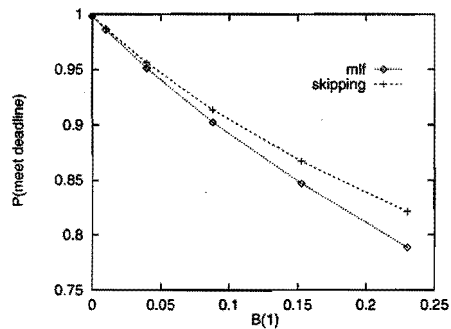
Figure 6.10: COMPARING SCHEDULING METHODS, N=4, P(CONFLICT)=0.39

in performance becomes smaller, with each optimization. This behavior is expected, since each optimization is guaranteed to improve or at least not degrade system performance. In figure 6.11a, the arrival rate of transactions is fixed to 2μ, and the conflict probability is varied between 0 and 0.230, which is translated to access of 0 to 5 data items in the database. The results are similar to the results of 6.10: if the conflict probability increases, the system can handle less transactions in parallel. Therefore, the system becomes overloaded at lower arrival rates.

The advantage of queue skipping is shown in figure 6.11b. Both pure MLF scheduling and MLF scheduling with queue skipping are depicted. The parameters of the simulation ($n = 8$ and $\lambda = 4\mu$) result in a system where a lot of processor power is wasted because of blocking. Especially in these circumstances queue skipping is advantageous.



(a) $n = 4, \lambda = 2\mu$



(b) $n = 8, \lambda = 4\mu$

Figure 6.11: VARIOUS CONFLICT PROBABILITIES

6.5 Conclusions

The accuracy of the straightforward Markov analysis shows that it is indeed possible to design *analyzable* real-time database schedulers. This can substantially reduce the effort that is necessary to obtain reliable and efficient performance evaluations of database schedulers. To date, most performance evaluations are based on simulations and testing, which is time-consuming and often costly. For example, a single simulation run of SQSL-soft required around 8 hours of computation, while our analysis takes a few minutes on the same workstation. On the other hand, mathematical analysis of existing schedulers is very demanding. By showing that schedulers can be designed to reduce the complexity of the analysis we offer a third alternative, that is possibly more cost-effective.

The SQSL schedulers presented offer a choice to a real-time database designer. The first schedulers are straightforward to implement, and impose little overhead on the system. The more elaborate schedulers have a better processor usage, which increases the database performance. However, these schedulers need more information about transactions, and require more computation time to use this information in their scheduling decisions.

MLF scheduling uses ordered insertion in the queue, and the queue skipping test uses expensive set intersections. The overhead that these mechanisms impose on the system is higher than the overhead caused by either the SQSL-soft scheduler or the SQSL scheduler with deadlines. Depending on the transaction complexity, straightforward schedulers that impose little overhead on the system might actually be better suited to a particular environment. However, this requires a more involved modeling than has been presented in this chapter, since the overhead has to be modeled in detail.

Analysis of the first two schedulers is possible because of the specific nature of the SQSL scheduler, combined with important assumptions that enable us to use a memoryless model. Further research could be directed at improving the analysis methods, such that more complex schedulers can be handled. Another direction can be to closer investigate the relation between the scheduler design, and its analysis. A combined approach could lead to clear, predictable database schedulers.

Chapter 7

Off-line database scheduling

Off-line, hard real-time scheduling systems are used in safety critical systems with real-time demands. Their real-time behavior has to be totally predictable. This makes it difficult to program these systems. To somewhat ease the programming effort, extend an off-line, hard real-time programming environment with database functionality. The hard real-time database functionality is implemented by a pre-processor of the main scheduler, such that the off-line scheduler does not need to be modified. Part of this work has appeared earlier in [36].

7.1 Environment study

The hard real-time environment under consideration is the DEDOS programming environment [39]. Its off-line scheduler guarantees hard real-time requirements, but the system does not offer database functionality. At this time, the DEDOS programming environment and its scheduler are still under development (see [99, 98]). In this chapter we add database functionality by adding a pre-processing step that translates database requirements into hard real-time requirements. Since the internal structure of the DEDOS scheduler remains unchanged, the pre-processing step can also be applied to other scheduling systems that offer a similar interface.

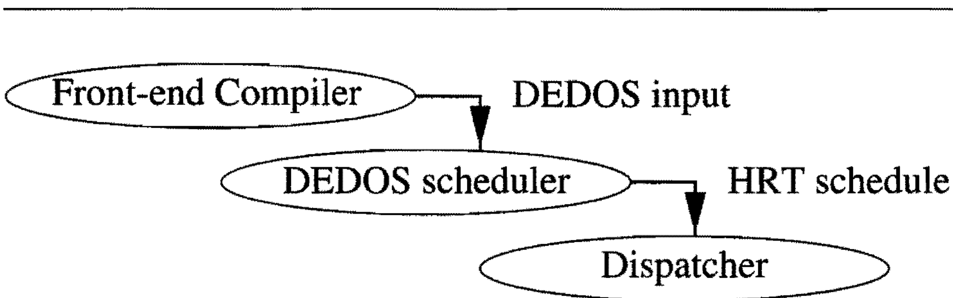


Figure 7.1: OVERVIEW OF THE DEDOS SYSTEM

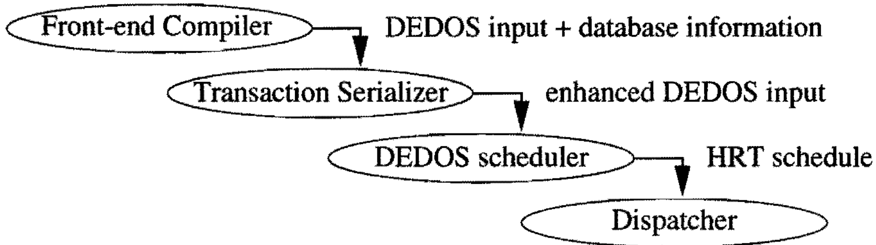


Figure 7.2: OVERVIEW OF THE EXTENDED DEDOS SYSTEM

Description of DEDOS

Figure 7.1 gives a short overview of the DEDOS system. A front-end compiler translates a high level language into input for the DEDOS scheduler. The DEDOS scheduler uses this input and the hardware description to generate a schedule. This schedule is executed on the hardware by a dispatcher.

The DEDOS scheduling input consists of a hardware model and a software model. We omit the hardware description, since it will not influence our database extension, assignment to hardware is handled by the DEDOS scheduler. This includes the scheduling of communication between execution units. Programs consist of a collection of objects OB. An object is a collection of data, together with methods that can be used to inspect or modify this data. Objects can communicate through method calls or asynchronous communication, irrespective of physical location of objects.

The execution of a method can be pre-empted at pre-defined points. These pre-emption points define non-interruptible pieces of code, called execution units, or units in short. Set B contains all execution units. Function BOB determines for each execution unit to which object it belongs. Function Wx determines for each execution unit the worst-case execution time. Function St determines the start time for each execution unit. B, BOB and Wx are produced by the front-end compiler, St is computed by the scheduler.

Timing requirements are expressed by time precedence constraints. Set TPC contains all time precedence constraints. A time precedence constraint relates the start times of two execution units: if a and b are execution units then time precedence constraint (a, s, f, b) specifies the following requirement on St:

$$St(a) + s \leq St(b) \leq St(a) + f$$

Normal precedence constraints can be expressed with time precedence constraints in the following way: "$a \prec b$":

$$(a, Wx(a), \infty, b)$$

Similar, a time precedence constraint (a, s, f, b) implies $a \prec b$ if and only if $s \geq Wx(a)$.

Transaction-serializer interface

Figure 7.2 shows the DEDOS system after it has been extended with database functionality. The transaction serializer will be further defined in this section. We assume that the front-end compiler has already been modified such that information about transactions and database accesses are available

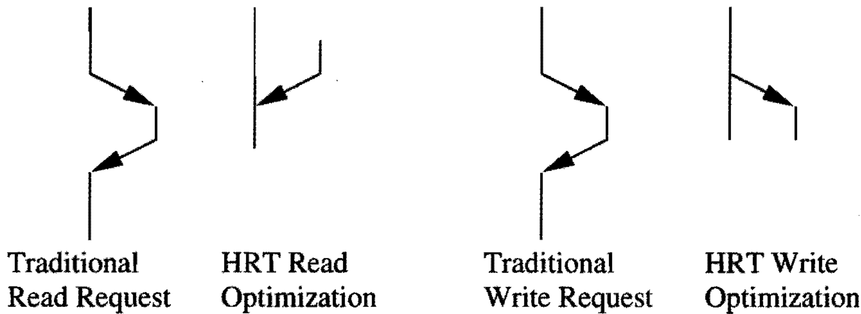


Figure 7.3: OPTIMIZING HARD REAL-TIME DATA ACCESS

to the transaction serializer. We describe the extra input that results from adding information about transactions and database accesses.

A transaction is modeled by an object with a single start method. Set T describes the set of transactions, and is a subset of BOB. Data items are modeled by objects with a read and write method. Set $DataSet$ describes the set of data items. Set OPS contains all read and write accesses, and is a subset of B. Functions tr, $Amode$ and $Daccess$ (together with OPS defined in chapter 2) define for each execution unit $op \in OPS$ to which transaction op belongs, the type of data access and the data item that op accessed. The fixed problem variables T, $DataSet$, tr and OPS are determined by the front-end compiler.

Hard real-time data access optimization

Figure 7.3 shows the classic way of invoking read and write methods in a message sequence chart (MSC). Message sequence charts [79, 64, 63] visualize communication protocols in an intuitive way. The vertical axis represents the flow of time. A vertical bar represents the execution of one or more execution units in a single process. Parallel bars thus represent parallel executions. Arrows correspond to messages between processes.

The traditional read/write request is initiated by a request message from the transaction, to the process that maintains the data item. The request is handled, and the result or acknowledgment is returned. Since the DEDOS environment schedules all execution in advance, the read access request becomes superfluous, leading to the optimized read access. And because the DEDOS environment offers a dependable platform, no acknowledge for the write request is necessary. These optimizations are carried out by the front-end compiler that translates data access operations to communications between the transaction object and the data item object.

Therefore, if transaction t reads data item X, an asynchronous communication from the data item object X to the transaction object t is defined. Similar, if transaction t writes data item X, an asynchronous communication from the transaction object t to the data item object X is defined. Since all data access is now asynchronous, this allows for a substantial increase in concurrent execution of transactions.

Figure 7.4: TWO EXAMPLE GRAPHS

Database functionality

The transaction serializer ensures that the execution of transactions is conflict serializable. However, the order in which execution units are executed is determined by the DEDOS scheduler. The transaction serializer can enforce a certain execution order $a \prec b$ by supplying the corresponding time precedence constraint $(a, Wx(a), \infty, b)$ to the DEDOS scheduler. The set of time precedence constraints that is supplied to the DEDOS scheduler is denoted TPC'. This set has to include all time precedence constraints that have been specified by the application programmer, so $TPC \subseteq TPC'$. Furthermore we restrict the generality of time precedence constraints: time precedence constraints cannot refer to internal execution units of transactions (they can refer to the first and last execution unit of a transaction). This restriction is necessary to preserve the atomicity constraint of transactions.

Abstract representation of the database scheduling problem. To focus on the scheduling problem, we look for an abstract representation of the database scheduling problem that is solved by the transaction serializer. It is known from literature [73] that schedules can be tested for conflict serializability by constructing a so-called *conflict-graph*. A conflict-graph uses vertices to represent transactions and edges to represent precedence constraints between conflicting data accesses. A schedule is conflict-serializable if the corresponding conflict-graph is acyclic.

Suppose two transactions p, q perform at least one conflicting data access. Either (p, q) or (q, p) is a directed edge in the conflict-graph. Which edge is added depends on the actual execution order that is determined by the DEDOS scheduler. Unless a time precedence constraint in TPC' enforces $p \prec q$ (or vice versa), the DEDOS scheduler chooses an arbitrary order. Arbitrary orders cannot be represented by a conflict-graph, hence it does not suffice to describe our scheduling problem.

We generalize the conflict-graph to a *mixed graph*. A mixed graph $G = (V, E^u, E^d)$ consists of a set of vertices V, a set of undirected edges E^u, and a set of directed edges E^d ($E^u \cap E^d = \emptyset$). Similar to the conflict graph, vertices represent transactions and edges represent conflicts between transactions. A directed edge (a, b) specifies that the order between the conflicting operations a, b is fixed by a time precedence constraint. An undirected edge (a, b) specifies that the DEDOS scheduler can choose an arbitrary order. For undirected edges the relation $(a, b) = (b, a)$ holds throughout the chapter, and the conversion is not explicitly mentioned in the presented algorithms. A cycle in the mixed graph occurs if a path $p_1 ... p_k = p_1$ $(k > 2)$ exists such that $p_i \in V$ $(1 \leq i \leq k)$ and either $(p_i, p_{i+1}) \in E^u$ or $(p_i, p_{i+1}) \in E^d$. If the mixed graph is acyclic, the DEDOS scheduler can only generate conflict-serializable schedules (since each directed edge in the mixed graph corresponds to a time precedence constraint).

An extension of the mixed graph is necessary to deal with cyclic dependencies between pairs of transactions. If transactions p, q perform at least two conflicting data accesses, the access order has to be specified, since a cycle of length two can occur. Since the mixed graph allows for at most one edge between two vertices, this cyclic dependency cannot be detected without extending the graph definition. We *label* undirected edges that correspond to pairs of transactions that have two or more

conflicts with each other. Let lab be the set of labeled edges.

Usually, the mixed graph $G = (V, E^u, E^d)$ that is generated from the input of the transaction serializer (the conflict relation, and TPC) is cyclic. The task of the transaction serializer is to find a (partial) orientation $E^{u'}, E^{d'}$, such that the resulting mixed graph $G' = (V, E^{u'}, E^{d'})$ is acyclic, and all labeled edges have received an orientation ($E^{u'} \cap lab = \emptyset$). In effect, the transaction serializer orients a number of undirected edges (replaces them by directed edges), and adds the corresponding time precedence constraints to TPC'.

Towards an objective function. Adding time precedence constraints can restrict the solution space of the DEDOS scheduler. Since the internal structure of the DEDOS scheduler is unknown, we have to assume that restricting the solution space decreases the probability that a feasible solution is found. Therefore, we investigate if the number of undirected edges $|E^{u'}|$ in the resulting graph G' is a good objective function.

Suppose that a set of independent directed paths exist in $G = (V, E^u, E^d)$, with total length k. Then the maximum number of undirected edges in $G' = (V, E^{u'}, E^{d'})$ is at most $|V| - k - 1$. For $k = 0$, this is proven as follows: the $|V| - 1$ undirected edges form a spanning tree of G. Adding a single edge introduces a cycle. If a directed path $P = p_1 ... p_l$ exists, no undirected path can exist between two vertices on P in G' (since G' is acyclic). Hence, the maximum number of edges consists of a spanning tree on $V - P$, plus a single edge to P: $|V| - |P|$. There are at most $|V|^2 - |V|$ edges. Generally, the majority of the edges in the final solution G' has to be directed to eliminate cycles. We conclude that $|E^{u'}|$ is not suitable as an objective function.

Even if edges are directed, the remaining solution allows for a certain amount of concurrency. Consider figure 7.4. The left graph allows for more concurrency: vertex a and c can execute in parallel, while the right graph enforces strict sequential computation. To strengthen our earlier point: function $|E^u|$ does not distinguish between these two graphs! A well-known objective function from hard-realtime scheduling (the sum of all completion times [76]) does distinguish between these two graphs. With a slight modification it can serve as the objective function. Completion times of transactions are determined by the DEDOS scheduler, and are not available to the transaction serializer. Instead, we calculate transaction completion time, assuming that all transactions have unit execution length.

Definition 7.1 *For a given graph $G = (V, E^u, E^d)$, define Cg_i ($i \in V$) as the length of the longest directed path in G with endpoint i.*

Now function $\sum_{i \in V} Cg_i$ (abbreviated $\sum Cg_i$) does distinguish between the two graphs in figure 7.4. The left graph has $\sum Cg_i = 1$, which is better than $\sum Cg_i = 3$, for the right graph. Furthermore, if $Cg_i = 0$ for all transactions i, all transactions can execute in parallel (no precedence constraints delay transactions). More general, if $\sum Cg_i$ is low, compared to the total number of transactions, it means that the level of concurrency is high. This suggests that minimization of $\sum Cg_i$ is suitable as an objective function. Since the transaction serializer uses no knowledge about the scheduling regime of the DEDOS scheduler, both proposed objective functions are rather crude heuristics. Their performance should be tested by actual scheduling of applications on the DEDOS platform.

Summarizing, time precedence constraints and conflicts relations are translated to a labeled mixed graph G. An acyclic mixed graph G' is generated by finding a partial orientation of G. The directed edges in G' are translated to time precedence constraints that are (together with previously defined time precedence constraints) passed to the DEDOS scheduler. In this chapter, the main focus is on identification of the scheduling problem. The problem specification specifies the predecessor-heuristic as its objective function. Finding an acyclic graph that optimizes the objective function $\sum_{i \in T} Cg_i$ is

NP-complete, as we show by mapping the

$$J2|chains, p_{ij} = 1|\sum C_i$$

onto our problem in subsection 7.3.1. We will call this the *graph orientation problem* in the remainder of this chapter. More complex heuristics (for example, a combination of the two mentioned heuristics) and accompanying algorithms can return solutions that improve the probability that the DEDOS scheduler returns a feasible schedule. This has not been investigated.

7.2 Specification

The transaction serializer is added as a pre-processor to the DEDOS scheduler. Hence, all platform details are hidden from the transaction serializer. The transaction serializer influences the execution of transactions by adding time precedence constraints to TPC'.

Generic scheduling specification.

platform: DEDOS scheduler	section 7.1
data: non-temporal	page 22
transactions: conflict-serializable	page 24
objective function: $\sum_{i \in T} Cg_i$	section 7.1

Additional specification.

Fixed Variable	Domain	Description
OB	$\mathcal{P}$(identifiers)	The set of all objects.
B	$\mathcal{P}$(identifiers)	The set of execution units.
Wx	$B \to \mathbf{R}^+$	Worst case execution time function.
TPC	$\mathcal{P}(B \times \mathbf{R} \times \mathbf{R} \times B)$	Set of time precedence constraints.

Free Variable	Domain	Description
TPC'	$\mathcal{P}(B \times \mathbf{R} \times \mathbf{R} \times B)$	Set of time precedence constraints created by transaction serializer.
St	$B \to \mathbf{R}^+$	Start times of execution units

Fact constraint	Description
$TPC \subseteq TPC'$	The transaction serializer only adds constraints.
$\forall (a,s,f,b) \in TPC' : St(a) + s \leq St(b) \leq St(a) + f$	Scheduler enforces TPCs.
$\forall a, b \in OPS : (a,s,f,b) \in TPC' \wedge s \geq Wx(a) : a \prec_{Aorder} b$	The data access order is determined by timed precedence constraints.

```
1 V, E^u, E^d, lab := T, ∅, ∅, ∅
2 foreach a, b ∈ OPS : a < b ∧ conflicts(a, b) do
3         if (a, b) ∈ E^u then lab+ = (a, b)
4         else E^u+ = (a, b)
```

Algorithm 7.5: CONFLICT DETECTOR

7.3 Scheduler design and algorithms

The transaction serializer has to ensure that the order in which transactions execute their access operations is conflict serializable. Although it cannot directly determine the start times of data access operations, it can specify time precedence constraints between data access operations. Normal precedence constraints can be expressed by time precedence constraints, so a conflict serializable order can be enforced. The transaction serializer continues to add time precedence constraints to TPC' and thus exchanging undirected edges for directed edges in the mixed graph, until G is acyclic. The transaction serializer is divided in four steps, to manage the complexity of the scheduling problem.

1. **Conflict detector** The conflict detector explicitly defines the solution space of the transaction serializer. For each transaction t, a vertex is added to mixed graph G. For each pair of conflicting execution units a, b undirected edge (a, b) is added to mixed graph G. If an undirected edge is added more than once (two transactions have two or more conflicts with each other), the edge is labeled by adding it to set lab.

2. **Application PC extractor** The application PC extractor eliminates cycles in G by replacing undirected edges with directed edges, in accordance with the time precedence constraint-set TPC. This does not exclude feasible solutions.

3. **Heuristic unit** The heuristic unit eliminates the remaining cycles in G (thus creating G') using heuristics that possibly eliminate feasible solutions.

4. TPC' **constructor** The TPC' constructor performs a straightforward translation from the mixed graph, to time precedence constraints.

7.3.1 Conflict detector

The conflict detector generates the initial mixed graph $G = (V, E^u, E^d)$ and its labeling lab. The vertices represent transactions, undirected edges represent conflicts between transactions, and directed edges represent serialization order. Initially, $V = T$, and $E^u = E^d = lab = \emptyset$. We use a straightforward algorithm that checks the conflict relation for each pair of data accesses to determine E^u. Algorithm 7.5 describes this. At the same time, algorithm 7.5 labels undirected edges, if they represent more than one conflict. Since each pair of data accesses is investigated (line 2) the time complexity of the conflict detector is $O(|OPS|^2)$. This complexity can be reduced if the conflict relation can be enumerated. However, since there can be $O(|OPS|^2)$ conflicts, this will not improve the worst-case execution time.

```
1 foreach p, q ∈ B do M[p, q] := (−∞, ∞)
2 foreach (p, s, t, q) ∈ TPC do
3       M[p, q]⊕ = (s, t)
4       M[q, p]⊕ = (−s, −t)
5 for k := 1 to ⌈log|B|⌉ do
6       foreach p, q, h ∈ B do M[p, q]⊕ = M[p, h] ⊗ M[h, q]
7 TPC* := ∅
8 foreach p, q ∈ B : p ≤ q do
9       (s, f) := M[p, q]
10      TPC* + = (p, s, f, q)
```

Algorithm 7.6: TAKING TRANSITIVE CLOSURE OF TPC

Application precedence constraint extractor

The application precedence constraint extractor generates a set of precedence constraints APC from TPC and Wx. These precedence constraints are used to direct edges in G in accordance with the time precedence constraints that are supplied by the application. The application precedence constraint extractor executes the following steps.

1. Take the transitive closure of TPC: TPC^*.
2. Generate precedence constraints from TPC^*: APC.
3. Adapt G in accordance with APC.

Two kinds of time precedence constraints are considered, both of which are contained in the set TPC: time precedence constraints that are specified explicitly in the application program, and time precedence constraints that result from sequential execution of execution units. Set TPC^* represents the transitive closure of the partial order specified by set TPC. Algorithm 7.6 maintains a two-dimensional array M. At each position $M[p, q]$, the time precedence constraint between operation p and q is maintained as a tuple (s, f). The following two rules are used to combine information from different time precedence constraints.

- **Transitivity** The transitivity rule combines sequences of timed precedence constraints.

$$(a, s, f, b) \otimes (b, s', f', c) = (a, s + s', f + f', c)$$

- **Combination** If two time precedence constraints are specified between a pair of execution units, they can be combined into one.

$$(a, s, f, b) \oplus (a, s', f', b) = (a, max(s, s'), min(f, f'), b)$$

Transitive closure of TPC. Algorithm 7.6 first initializes array M by combining all time precedence constraints TPC. Since there are $|B|$ execution units, the transitivity rule has to be applied $\lceil log|B|\rceil$ times to all possible combinations in M, to complete the transitive closure of TPC. After the kth execution of line 6, entry $M[p, q]$ contains the timed precedence constraint between p and q that is the combination of all possible sequences of time precedence constraints of length 2^k

```
1 APC := ∅
2 foreach (a, s, f, b) ∈ TPC* : a, b ∈ OPS ∧ conflict(a, b) ∧ s ≥ Wx(a) do
3       APC+ = (tr(a) ≺ tr(b))
4 foreach (a, s, f, b) ∈ TPC* : a, b ∈ OPS ∧ conflict(a, b) ∧ f ≤ −Wx(b) do
5       APC+ = (tr(b) ≺ tr(a))
```

Algorithm 7.7: GENERATING PCS FROM TPCS

or less. Therefore, line 6 has to be executed $\lceil log|B|\rceil$ times, to complete the transitive closure of TPC. Finally, the results in M are stored as set TPC^*. The time complexity of algorithm 7.6 is $O(log|B| \times |B|^3)$, since the foreach-statement on line 6 checks $O(|B|^3)$ possible combinations.

Generate precedence constraints. Algorithm 7.7 generates the set of application precedence constraints from TPC^*. If an access operation b starts after a is guaranteed to finish ($St(a) + Wx(a) \leq St(b)$), this implies that $tr(b)$ is serialized after $tr(a)$. If b finishes before a starts ($St(a) \geq St(b) - Wx(b)$), the reverse holds: $tr(b) \prec tr(a)$. In all other cases, executions of access operations are allowed to overlap, and precedence constraints cannot be extracted. The time complexity of algorithm 7.7 is $O(|B|^2)$, as a time precedence constraint is specified between each pair of execution units. This complexity can be reduced to $O(|OPS|^2)$ by suitable data-structures that eliminate the need to consider time precedence constraints that do not relate to pairs of data access operations.

Adapt G. Algorithm 7.8 adapts the mixed graph by replacing undirected edges by directed edges. Although the DEDOS scheduler can never generate a schedule if contradicting time precedence constraints are specified (both $a \prec b$ and $b \prec a$), the check in line 5 stops the algorithm from proceeding at this early stage. Performing the checks $(a, b) \in E^u$ and $(a, b) \in E^d$ can be done in $O(1)$, if the right data structures are used (for example adjacency matrixes, see [26]). The time complexity of algorithm 7.8 is $O(|T|^2)$, since it considers each element of APC at most once.

Heuristic unit

Finding a set of directed edges $E^{d'}$ that replace undirected edges from E^u, such that the $G = (V, E^{u'}, E^{d'})$ is acyclic, and $\sum_{i \in V} Cg(i)$ is minimal is NP-complete. We show that the

$$J2|chains, p_{ij} = 1|\sum C_i$$

```
1 foreach (a ≺ b) ∈ APC do
2    if (a, b) ∈ E^u
3    then E^u −= (a, b)
4         E^d += (a, b)
5    else if (b, a) ∈ E^d then report incorrect specification
```

Algorithm 7.8: ADAPTING THE MIXED GRAPH WITH APC

job-shop scheduling problem that is known to be NP-hard [95], can be polynomially reduced onto our problem.

Complexity of the graph orientation problem. The $J2|chains, p_{ij} = 1|\sum C_i$ job-shop problem is described as follows. Mixed graph $G_j = (V_j, E_j^u, E_j^d)$ consists of a set of operations V_j. Each job a has to be executed on two resources $r1$, $r2$. These executions are represented by operations $a_1, a_2 \in V_j$. The order in which these operations have to occur depends on the job, and is represented either by directed edge $(a_1, a_2) \in E_j^d$ or $(a_2, a_1) \in E_j^d$. Each job requires an execution time of 1 on each resource. Each resource can only execute one operation at a time. Operations a_i, b_i on the same resource are connected by an undirected edge $(a_i, b_i) \in E_j^u$. The order in which jobs can be executed is restricted by so-called chains, that determine a partial order on the execution of jobs. If job a, with $(a_i, a_p) \in E_j^d$ precedes job b with $(b_k, b_l) \in E_j^d$ in the partial order, this is represented by directed edge $(a_p, b_k) \in E_j^d$. An execution order should be defined by giving an orientation of E_j^u, such that the resulting schedule minimizes $\sum C_i$, the sum of all completion times of jobs.

The mapping from instances of the job-shop problem to instances of finding an orientation of mixed graph $G = (V, E^u, E^d)$ is as follows: $V = V_j$, $E^u = E_j^u$, $E^d = E_j^d$. Furthermore, all undirected edges in E^u are labeled ($lab = E^u$), to ensure that they receive an orientation. Solution $G' = (V, E^{u'}, E^{d'})$ is an acyclic directed graph. Since the reduction is almost a one-on-one mapping, the corresponding orientation is a solution to the job-shop scheduling problem. As jobs have a unit execution time, $\sum Cg_i = \sum C_i$. Therefore, an optimal solution to the mixed graph orientation problem implies an optimal solution to the job-shop scheduling problem.

Solution methods. Several approaches are possible to find solutions that optimize the objective function. The problem can be formulated as a constraint satisfaction problem (for a discussion of CSPs, see chapter 2). The cycle remover that is described in algorithm 7.10 can be used for reduction of the solution space. Algorithm 7.9 computes the objective function for a given result. The objective function is discussed below.

Genetic algorithms received their name from a strong similarity of biological processes [66]. For a more pragmatic approach to genetic algorithms and a description of other solution methods, see [1]. The optimization problem can be transformed into a problem that can be solved with genetic algorithms. Two transformations take place. The objective function is modified to serve as the so-called *fitness function*. Infeasible solutions that still allow cyclic dependencies are penalized: $total$ is increased with a sufficiently large constant C_{error}. This constant should be higher than any $\sum Cg_i$ that belongs to a feasible solution. For example, $C_{error} = |V|^2$ suffices, as it can be shown that $\sum Cg_i \leq |V|(|V| - 1)/2$. Checking whether cycles in the mixed graph and labeled undirected edges still occur is possible in $O(V + E)$ time with standard DFS [26]. In this fashion, the genetic selection mechanisms will prefer feasible solutions.

The mixed graph is written as a string of 0/1/2 values (a "genome"). Each character ("gen") in the string represents an initially undirected edge. Value 0 and 2 represent the opposite orientations of a directed edge, and 1 represents an edge that remains undirected. By encoding only the initially undirected edges, instead of the entire graph, the string length is reduced. The fitness function has to reconstruct the entire graph from the string, and the set of initially directed edges. The modified version of algorithm 7.9 can then be used as the fitness-function that is required in genetic algorithms, since it penalizes incorrect solutions.

```
1 foreach v ∈ V do Cg[v] := −1
2 F, DONE, total := ∅, ∅, 0
3 foreach v ∈ V : ∄v' ∈ V : (v', v) ∈ E^d do F+ = v; Cg[v] := 0
4 while F ≠ ∅ do
5     choose v ∈ F; F− = v; DONE+ = v
6     total+ = Cg[v]
7     foreach v' ∈ V : (v, v') ∈ E^d do
8         Cg[v']max = Cg[v] + 1;
9         if ∀(v'', v') ∈ E^d : v'' ∈ DONE then F+ = v'
```

Algorithm 7.9: COMPUTING THE OBJECTIVE FUNCTION

Computing the objective function. The objective function $\sum Cg_i$ is computed in algorithm 7.9 by using a modified breadth first search (BFS) on a forest [26]. At any time, a "front" F of vertices exists that can be processed. A vertex v can be processed, if all its predecessors q ($(q, v) \in E^d$) have been processed. Initially, F contains all vertices that have no predecessors. For each vertex v, $Cg[v]$ is computed by taking the maximum of $Cg[v'] + 1$, for all predecessors v' of v (line 8). The algorithm stores each processed vertex in set $DONE$. Each vertex is added exactly once to set F. Therefore, the while-loop in line 4 is executed $|V|$ times. The foreach statement in line 7 checks each outgoing edge of the vertex that is under consideration. Using suitable data-structures (adjacency lists, see [26]), each outgoing edge is found in $O(1)$ time. Similar, the checks in line 9 can be resolved in $O(1)$ if the sets $DONE$ and E^d are suitably chosen. Therefore, the time complexity of the entire traversal is $O(|V| + |E^d|)$.

Cycle remover. Suppose in mixed graph G a path of directed edges exists from a to b. If $(b, a) \in E^d$, graph G is cyclic, and no serializable schedule can be constructed. The application has to be redefined. If $(a, b) \in E^u$, the mixed graph is cyclic. However, by replacing undirected edge (a, b) by directed edge (a, b), the cycle is avoided. Heuristics can avoid these simple cycles by invoking the cycle remover directly after an edge (a, b) is added to the mixed graph. The cycle remover uses depth first search (DFS) [26] to find these implicitly directed edges and replace them.

```
DFS(v, a)
1 visited[v]:=true
2 if (v, a) ∈ E^u then
3     E^u− = (v, a)
4     E^d+ = (a, v)
5 foreach (v, v') ∈ E^d : ¬visited[v'] do DFS(v', a)

cycleRemover((a, b))
1 foreach v ∈ V do visited[v]:= false
2 DFS(b, a)
```

Algorithm 7.10: CYCLE REMOVER

1 $TPC' := TPC$
2 foreach $a, b \in OPS : conflict(a, b) \wedge (a, b) \in E^d$ do
3 $\quad TPC' += (a, Wx(a), \infty, b)$

Algorithm 7.11: TPC' CONSTRUCTOR

Algorithm 7.10 shows the modified DFS and the main algorithm. The DFS traverses the graph along directed edges only, and starts in node b. At each node, it is checked whether undirected edges connect to vertex a. Such undirected edges are replaced by correctly oriented directed edges. The time complexity of a DFS traversal is $O(|V| + |E^u + E^d|)$.

7.3.2 TPC' Constructor

The TPC' constructor translates precedences between transactions (directed edges in the mixed graph) to time precedence constraints on conflicting access operations. Therefore, even if transaction t precedes transaction q in the accompanying data access order, part of the execution of t and q can occur in parallel: only the conflicting data accesses are ordered. Algorithm 7.11 describes how the translation is performed.

7.4 Conclusions

In this chapter, database functionality is added to DEDOS. DEDOS is a hard real-time scheduling system that uses time precedence constraints to specify timing behavior of non-interruptible execution units. Conflict serializability is enforced by adding a pre-processor to the system. This pre-processor (the transaction serializer) does not use any information about the DEDOS scheduler other than the interface. Therefore, the preprocessor can be used in other contexts as well. The transaction serializer can add database functionality to any hard real-time scheduler that specifies timing behavior with time precedence constraints on execution units.

By taking advantage of the hard real-time functionality of the DEDOS platform, several optimizations are possible. These optimizations considerably enhance possible parallelism within a single transaction. Since on-line read-requests are superfluous in a hard real-time environment (the request is known off-line, and the read operation can be scheduled without on-line request), all read operations of a transaction can occur in parallel. Since DEDOS offers a dependable platform, writes never fail. No two phase commit protocol is required. This reduces the time that a transaction needs to commit.

The chapter primarily focuses on the definition of the interfaces of the transaction serializer, and the identification of the remaining scheduling problem. A number of algorithms construct an abstract representation of the scheduling input, denoted as a mixed graph. Proposals are made to solve the remaining scheduling problem with either CSP solvers, genetic algorithms or other variants of local search methods.

Chapter 8

Temporal consistency in hard real-time databases

Real-time systems that directly interact with their environment through sensors and actuators maintain a "view" of the environment. This view should be consistent and relevant, i.e. the data used to build the view should be measured recently. We observe that maintaining a consistent model is not a purpose in itself: these "temporal consistency" requirements only need to hold when the model is accessed by decision-taking procedures. In this chapter we investigate what new requirements arise in these systems, and we propose a way to implement these requirements in hard real-time databases.

The structure of this chapter is as follows. In the next section, we show how databases can help in maintaining a model of the environment, and we investigate how temporal requirements can be specified. Section 8.2 gives a problem specification that expresses these new requirements. The scheduler design is explained in section 8.3, resulting in the scheduler that is presented in section 8.4. The results of applying the scheduler to an example system are given in section 8.5. Finally, in section 8.6 the benefits and problems of our new specification method and scheduler implementation are discussed.

8.1 Environment study

The database should store data that is received from sensors (sensor data), and data that is derived from sensor data (derived data). Data that is received from sensors describes the environment at a certain point in time. Data items that describe the continuously changing environment are called *temporal data items*. Non-temporal data do not directly reflect the environment, and they are treated in the normal way. The traditional serializability requirement does not suffice for temporal data items, since it does not consider the passage of time. In this chapter, so-called *transaction-based temporal consistency* is introduced, which replaces serializability for temporal data items. This is a generalization of a previously introduced temporal consistency concept, that is best described as *item-based temporal consistency*, see [8].

Often, applications require sensor data or derived data of a certain age. Such age requirements can be satisfied by implementing a set of refresh transactions: transactions that write new values into the database. When these refresh transactions are executed depends on the age requirements of transactions. We define *temporal requirements* that transactions place on the temporal data that they read in the database.

Refresh transactions

All database accesses are made by application software that issues transactions. More than one application can make use of the same real-time database. The database fulfills a supporting role: it provides up-to-date data to the applications, and ensures that concurrent data access does not lead to perceived system states that violate consistency requirements. Applications supply the database with a set of *transaction!refresh*. The real-time database can execute these whenever temporal data has to be refreshed. Three transaction categories describe what transactions can be executed.

Sensor transactions. These refresh transactions read out sensors, and store the results in the database. They do not read temporal data, and the temporal data items they write are called sensor items. The database can execute sensor transactions when necessary.

Derive transactions. These refresh transactions use information already present in the database to derive new information (draw conclusions). They read temporal data and the temporal data items they write are called derived items. The database can execute derive transactions when necessary. A good example of a derive transaction is the correlation of results from multiple sensors, to obtain a complete view of the environment.

User transactions. These transactions are issued by the applications directly, and do not write temporal data items. They either control actuators, write to screens or write to non-temporal parts of the database. The database ensures that temporal requirements of these transactions are satisfied.

To eliminate confusion, we explicitly state the difference between transactions and transaction types. A *transaction* refers to an actual execution, in which the database is accessed. Each transaction has a *transaction type* that defines what algorithm the transaction executes, and what requirements have to be fulfilled. In this chapter we focus on the temporal requirements that are specified by transaction types. If transaction t is of transaction type TT, we say t is an *invocation* of TT. Function $Ttype(t)$ returns the transaction type of transaction t. Exactly one refresh transaction type is provided for each temporal data item (note that one refresh transaction can refresh more than one item).

Definition 8.1 *Each temporal item X is refreshed by transactions of type TT_X.*

Due to this *single-writer* property, and since the database itself can determine when refresh transactions are executed, serializing refresh transactions is easy. This removes the need for complex serializability constraints [100]. Furthermore, we assume that there exists a global clock τ_{now} that applications use to specify their real-time requirements, and that the database can use to satisfy them.

Temporal data items

Temporal data reflects (part of) the environment at a certain point in time. When a temporal data item is updated, this point in time changes. We make this mechanism explicit by introducing instances of temporal items.

Definition 8.2 *Instance*
Instance x of temporal item X consists of a value $x.v$, and a time of measurement $x.\tau$.

Intuitively, $x.\tau$ is the moment that $x.v$ accurately reflects the state of the environment. $x.v$ can reflect other moments as well, but this is not ensured. Each time a new value is written to a data item, a new instance is created. When instances are no longer useful, they can be destroyed or archived. To

facilitate the specification, we assume that the set of instances at moment τ_{now} is given by function $Insts(\tau_{now})$, and that instances are never destroyed: $Insts(\tau)$ is monotone increasing in τ according to subset ordering. Often, requirements are specified on the age of instances, instead of on the time of measurement. The age of an instance x at time τ is defined as follows.

$$age(x, \tau) = \tau - x.\tau$$

Time of measurements. The time of measurement $x.\tau$ is defined by the transaction t that writes x. For sensor items, $x.\tau$ can be directly related to the moment that the sensor was read. For derived items Y, $y.\tau$ is not exactly defined, because the item is never measured! Since it is necessary to reason about the age of derived items, the transaction t that writes y should define an approximative time of measurement $y.\tau$ (see section 8.1).

Data access of transactions. Multiple instances x of the same data item X can be present in the database at the same time. We allow transactions to read more than one instance of the same data item (necessary if the transaction wants to extrapolate the value of the item in time). For each transaction type TT, we define the sets $IN(TT)$ and $OUT(TT)$ that contain the data items accessed by invocations of TT, as well as the number of instances accessed from these items. We assume that transactions write at most one instance of each data item. The type of one tuple in $IN(TT)$ is $Dtup = DataSet \times \mathbf{N}^+$.

Definition 8.3 ***Transaction input***

$$IN(TT) = \{(X, n) \mid \textit{invocation } t \textit{ of } TT \textit{ reads } n \textit{ instances of } X\}$$

Definition 8.4 ***Transaction output***

$$OUT(TT) = \{X \mid \textit{invocation } t \textit{ of } TT \textit{ writes to } X\}$$

As an abbreviation X is equivalent to $(X, 1)$. For example, suppose invocations of transaction type TT access one instance of items X, Y, and 3 instances of item Z. The invocations write to data items P, Q. This is specified by:

$$IN(TT) = \{X, Y, (Z, 3)\}$$
$$OUT(TT) = \{P, Q\}$$

Since it is possible that a transaction invocation reads more than one instance of the same data item X, we identify the read instances with an additional number 1 to k. Read instance i of item X is denoted x_i. To avoid confusion, we require that this ordering reflects the time of measurements of the instances: $\forall i, j \in [1, k] : i < j \Rightarrow x_i.\tau < x_j.\tau$.

Definition 8.5 ***Identifying instances***

The kth instance of data item X read by a transaction t is denoted x_k.

Absolute temporal requirements

Each transaction type specifies properties of the instances that are read by its invocations. For each instance the minimum and maximum allowed age of that instance can be specified as well. If the database satisfies maximum age requirements, it is ensured that transactions read significant, fresh data. Freshness of data is not the only thing that counts. Information about the future can be stored in the database, and certain data should not be read before a certain time. If the database satisfies minimum age requirements, it is ensured that transactions do not read data that is not yet valid. This

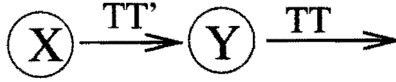


Figure 8.1: A DERIVATION PATH

prevents transactions from acting too soon. Suppose $(X, n) \in IN(TT)$. For each instance x_i ($i \in [1, n]$), transaction type TT can specify that the age of x_i should be in some interval $[a, b]$.

Definition 8.6 ***Absolute requirements***
The absolute temporal requirement $TT.abs(x_i) = [a, b]$ *specifies that the age of instance* x_i*, read by an invocation* t *of* TT*, should lie in interval* $[a, b]$.

It is allowed to use ∞ and $-\infty$ if upper or lower bounds need not be specified. As an abbreviation,

$$t.abs(X) = \{[a^1, b^1], .., [a^k, b^k]\}$$

is equivalent to k separate absolute temporal requirements for instances $x_1..x_k$.

The set of all absolute temporal requirements of a transaction type TT is denoted $ABS(TT)$. A single absolute temporal requirement is denoted as a four-tuple (x, n, a, b), which is equivalent to $TT.abs(x_n) = [a, b]$. The type of an absolute temporal requirement is

$$Tabs = DataSet \times \mathbf{N}^+ \times \mathbf{R} \times \mathbf{R}$$

Relative temporal requirements

To get a consistent view of the environment, a transaction needs to read instances with bounded age-differences. To extrapolate the value of a data item, a transaction needs to read instances of the same data item that are representative for the item's evolution: age differences need to be significant. Transaction type TT can specify upper and lower bounds on the age difference between instances that are read by its invocations. Suppose $(X, n), (Y, m) \in IN(TT)$. For each pair x_i, y_j ($i \in [1, n], j \in [1, m]$) transaction type TT can specify that the difference in age should lie within an interval $[a, b]$.

Definition 8.7 ***Relative requirements***
The relative temporal requirement $TT.rel(x_i, y_j) = [a, b]$ *specifies that the difference in age of instances* x_i *and* y_j *read by an invocation* t *should lie within interval* $[a, b]$.

The set of all relative temporal requirements of a transaction type TT is denoted $REL(TT)$. A single relative temporal requirement is denoted as a six-tuple (x, n, y, m, a, b), which is equivalent to $TT.rel(x_n, y_m) = [a, b]$. The type of a relative temporal requirement is called

$$Trel = DataSet \times \mathbf{N}^+ \times DataSet \times \mathbf{N}^+ \times \mathbf{R} \times \mathbf{R}$$

Specifying requirements on intermediate instances

In figure 8.1, invocations of transaction type TT read derived data item Y, and TT can specify temporal requirements on instances of Y. Whenever a transaction t of type TT reads an instance y, the

time of measurement $y.\tau$ is used to check these requirements. Since Y is a derived data item, $y.\tau$ is an approximation. If the approximation does not suffice for transaction type TT, we allow that TT specifies temporal requirements on the data items that are used to derive Y. Therefore, TT can specify temporal requirements on instances x, read by invocations of TT' that write Y. To avoid aliasing and naming problems, x is identified by explicitly naming the derivation path: $x \rightarrow y$. We allow transactions to specify absolute and relative temporal requirements on intermediate instances (like x in our example). However, as we will find out in 8.1, extra restrictions will be placed on these specifications.

Temporal correctness

Now all the temporal requirements of transactions have been specified, we define the temporal correctness properties of the real-time database. We assume that a selection mechanism exists, such that transactions read a correct set of instances, if this set exists. The temporal requirements of a transaction t that is an invocation of TT should hold at the start of t: $St(t)$. We define two predicates that specify whether the set of available instances at time τ satisfies the absolute and relative temporal requirements of transaction type TT. These predicates are then used to define temporal correctness.

$$\text{absolute}(TT, \tau) \equiv \forall (x, k, a, b) \in ABS(TT) : \exists x_k \in Insts(\tau) : a \leq \tau - x_k.\tau \leq b$$

$$\text{relative}(TT, \tau) \equiv \forall (x, k, y, l, a, b) \in REL(TT) : \exists x_k, y_l \in Insts(\tau) : a \leq x_k.\tau - y_l.\tau \leq b$$

Definition 8.8 ***Temporal correctness*** *The database (represented by function $Insts$) has the temporal correctness property, if each executed transaction t reads instances x that satisfy the absolute and relative temporal requirements of the transaction type $Ttype(t)$ of t.*

$$\begin{aligned} &temporallycorrect(T) \equiv \\ &\quad \forall t \in T : absolute(Ttype(t), St(t)) \wedge relative(Ttype(t), St(t)) \end{aligned}$$

Intermediate instances

Suppose in figure 8.1 TT needs to specify an absolute time constraint on $x \rightarrow y$,

$$TT.abs(x \rightarrow y) = [a, b]$$

According to the semantics of absolute temporal requirements, this means

$$(St(t) - (x \rightarrow y).\tau) \in [a, b] \text{ for all invocations } t$$

So when invocation t starts execution, it reads an instance y. Instance y is created from an instance x that satisfies an additional temporal requirement. It is not clear if this requirement conflicts with an absolute temporal constraint on x that is specified by TT':

$TT'.abs(x) = [c, d]$ and $TT.abs(x \rightarrow y) = [a, b]$
mean
$St(t') - x.\tau \in [c, d]$ and $St(t) - x \rightarrow y.\tau \in [a, b]$.

The time of measurement $x.\tau$ is now dependent on both $St(t')$, and $St(t)$. Since $St(t')$ and $St(t)$ are defined by the scheduling system, it is hard to determine beforehand whether the two specifications contradict each other. Therefore, it is hard to write good specifications that involve intermediate

instances. Two alternatives were suggested to circumvent this problems. The first alternative is to remove temporal requirements on intermediate instances altogether. If the temporal requirements of a deriving transaction type TT need to be modified, a new transaction type TT^m can be constructed. TT^m is executed with the new temporal requirements. This leads to a lot of double work, and high processor loads.

Instead, we choose to allow *strengthening* of temporal requirements. The semantics of temporal requirements on intermediate instances have to be modified. The rationale is that these modified semantics have a clearer meaning. Better specifications and implementations are the result.

Definition 8.9 ***Absolute temporal consistency requirements on intermediate instances***
Let $x \rightarrow ... \rightarrow y$ be a derivation path of y, let $Y \in IN(TT)$, $X \in IN(TT')$. The absolute temporal requirement $TT.abs(x \rightarrow ... \rightarrow y) = [a, b]$ means $\forall t' : TT' = Ttype(t') \Rightarrow (St(t') - x.\tau) \in [a, b]$.

If a transaction type TT specifies temporal requirements on an intermediate instance x, read by transaction type TT', it can be seen as strengthening the temporal requirements of invocations of transaction type TT'. So the combination

$$TT.abs(x \rightarrow ... \rightarrow y) = [a, b]$$

$$TT'.abs(x) = [c, d]$$

can be interpreted as

$$TT'.abs(x) = [c, d] \cap [a, b]$$

Temporal requirements on intermediate instances can be treated as normal temporal requirements. The source of the temporal requirements of TT' is traceable, since the strengthened temporal requirements are specified separately. In the remainder of this chapter, we do not consider temporal requirements on intermediate instances. Relative temporal requirements are restricted to instances, read by the same transaction. Since relative temporal requirements are not related to the start-time of a transaction, no further change of semantics is necessary.

Time of measurement functions

Temporal correctness is defined on the age of instances, accessed by transactions. Transactions that write instances of temporal items use a *time of measurement function* (tom-function) to define the time of measurement of these instances. Transactions can write multiple data items, each with their separate time of measurement. Therefore, tom-functions are specified for each data item separately. The tom-function of data item X can be denoted $tom(X)$. The tom-function of data item X, which is written by invocations t of TT can use information about the t.o.m. of instances read by t, as well as start time $St(t)$ and finishing time $Ft(t)$ to determine its result. The specific function depends on the semantics of TT. A number of typical tom-functions are given in the following list.

Sensor items. If the precise moment of measurement of a sensor is unknown, the time of measurement of instance x of sensor item X is assumed to be the start of the invocation of sensor transaction t of type TT:
$x.\tau = St(t)$

Derived items: one critical input. Transactions t of type TT derive item Y, using a number of data items. One data item X is time-critical: it is very important that instances x, read by t are fresh. The t.o.m. of the time-critical item determines the t.o.m. of the derived item:
$y.\tau = x.\tau$

Derived items: extrapolation of data. Transactions t of type TT extrapolate item X, and write the future value of X in data item F. The transactions t use linear extrapolation, and access two instances of X. To bound the error in the extrapolation, the time of measurement of the future value equals the t.o.m. of the most recent instance of X that is read, plus a constant:
$f.\tau = max(x_1.\tau, x_2.\tau) + c$

Multiple output instances. If transactions of type TT write to more than one data item, the t.o.m. of the corresponding instances need not be equal. The previous example is extended; transactions of type TT result in three different extrapolations F, G, H. Each instance extrapolates X to a different point in the future, and has a different error margin:

$$f.\tau = max(x_1.\tau, x_2.\tau) + c_1$$
$$g.\tau = max(x_1.\tau, x_2.\tau) + c_2$$
$$h.\tau = max(x_1.\tau, x_2.\tau) + c_3$$

No general rules exist for the tom-function of derive transactions. Tom-functions need to be specified carefully for each derived data item. Since the transaction programmer can specify the value of the derivation, it should also be possible to specify the tom-function. For sensor items, specifying the tom-function is easy if the actual time of measurement can be read from the global clock. Still, since sensory data is often pre-processed before it is entered into the database, some offsets can be necessary.

Removing instances

To keep the memory requirements of the database low, instances should be removed as soon as possible. The maximal useful age MUA_X can be computed from the upperbounds of absolute temporal requirements that are specified on X. This is not further described in this chapter.

8.2 Specification

The main contribution of this chapter is the precise specification method of temporal requirements that have been discussed in the previous section. To show that these specifications describe implementable systems, we test them on a well-understood platform. It is assumed that user transactions are scheduled on a centralized architecture using rate-monotonic scheduling [59]. Refresh transactions are executed on the same platform: each refresh transaction type is executed periodically. Since this chapter introduces new concepts that deal with temporal requirements of transactions, the basic specification does not specify a complete scheduling problem, since all goal-constraints are explicitly specified below.

Basic specification.

platform: centralized	page 21
data: temporal	see below
transactions: hard real-time*, user+refresh	page 24, see below
objective function: minimize refresh overhead	see below

* The real-time scheduling of user transactions is not part of the scheduling problem, the emphasis is on the satisfaction of temporal requirements. Instead of goal constraints, the real-time requirements on user transactions are presented as fact constraints.

Additional specification. Off-line, a set of transaction-types TTs of both user and refresh transactions is offered to the scheduler. A number of functions specify for each transaction-type the input, output, and temporal requirements. Two functions *miniage*(X) and *maxiage*(X) are provided for each data item X. If transaction t writes instance X, these functions provide an upper bound and a lower bound on the range of $St(t) - x.\tau$ (this is called the "initial age" of x):

$$miniage(X) \leq St(t) - x.\tau \leq maxiage(X)$$

Since $x.\tau$ is computed by $tom(X)$, the *miniage* and *maxiage* functions replace the tom-functions. These bounds on the tom-functions are necessary to provide hard real-time guarantees. The *miniage* and *maxiage* functions use information about the t.o.m. of the instances that are read by refresh transaction t to compute their result. All information about transaction-types is available off-line.

Refresh transactions are scheduled using rate-monotonic scheduling [59]. Rate-monotonic scheduling expects as input a set of requirements on the period P_{TT} and deadline D_{TT} of periodic transactions of type TT:

$$P_{TT} + D_{TT} \leq \Delta_{TT}.$$

A common assumption in rate-monotonic scheduling is $P_{TT} = D_{TT}$. Therefore, the remainder of the chapter only refers to Δ_{TT}. Temporal requirements are mapped to rate-monotonic requirements. This means that Δ_{TT} has to be computed for each refresh transaction type TT. The scheduler should minimize the load that results from executing refresh transactions. In [59], an expression for the load is derived. We apply this to the load, generated by refresh transactions. If the worst-case execution time $Wx(TT)$ of transactions of type TT is known, this load is given by (remember that the period of TT is $\leq 0.5\Delta_{TT}$):

$$\sum_{TT \in TTs} 2Wx(TT)/\Delta_{TT}$$

User transaction-types TT can be identified by the fact that they do not write temporal data items: $OUT(TT) = \emptyset$. Since this chapter focuses on the execution of refresh transactions, the set of executed user-transactions T^u is a fixed problem variable that is not used during scheduling of refresh transactions.

Fixed Variable	**Domain**	**Description**
TTs	$\mathcal{P}$(identities)	The set of all possible transaction-types.
IN	$TTs \rightarrow \mathcal{P}(Dtup)$	Defines the input of a transaction-type.
OUT	$TTs \rightarrow \mathcal{P}(Dtup)$	Defines the output of a transaction-type.
ABS	$TTs \rightarrow \mathcal{P}(Tabs)$	Defines the absolute temporal requirements of a transaction-type.
REL	$TTs \rightarrow \mathcal{P}(Trel)$	Defines the relative temporal requirements of a transaction-type.
miniage	$DataSet \rightarrow \mathbf{R}$	Defines a lower-bound on the range of tom-functions.
maxiage	$DataSet \rightarrow \mathbf{R}$	Defines an upper-bound on the range of tom-functions.
T^u	$\mathcal{P}$(identities)	The set of executed user transactions.
$Ttype$	$T^u \rightarrow TTs$	The transaction-type of each executed user transaction.

Refresh transactions are executed by the database. Therefore the set of refresh transactions T^r is a free problem variable. Together, they create the set of available instances $Insts$. For each instance x free problem variable $x.\tau$ defines its time of measurement (unannounced information).

Free Variable	Domain	Description
T^r	$\mathcal{P}$(identities)	The set of executed refresh transactions.
$Ttype$	$T^r \rightarrow TTs$	The transaction-type of each executed refresh transaction.
$Insts$	$\mathbf{R} \rightarrow \mathcal{P}(DataSet \times \mathbf{R})$	Returns the set of available instances at a specific time.
τ	$DataSet \times \mathbf{R} \rightarrow \mathbf{R}$	Defines the time of measurement of instances.

We intentionally used the same identifier for the $Ttype$ function for both user transactions and refresh transactions, such that $T = T^u \cup T^r$ can be used as a shorthand. Strictly speaking, T is a free problem variable, since the database designer can influence T^r. A fact-constraint specifies how $Insts$ relates to the execution of refresh transactions in T^r. Each refresh transaction t of type TT writes a set of instances that is available after t has finished execution. This is expressed by the following predicate, which also states that the time of measurements of these instances are in accordance with the *miniage* and *maxiage* functions.

$$
\begin{aligned}
&haswritten(t, TT) \equiv \\
&\quad \forall X \in OUT(TT) : \forall \tau \in [Ft(t), \infty) \exists x \in Insts(\tau) : \\
&\qquad miniage(X) \leq St(t) - x.\tau \leq maxiage(X)
\end{aligned}
$$

Fact constraint	Description
$\forall t \in T^r : haswritten(t, Ttype(t))$	Refresh transactions add instances to the database.

For all transaction invocations, we require that their temporal requirements are satisfied. The following predicates specify absolute and relative temporal correctness. Together with the *haswritten* property, it proves that the execution of refresh transactions ensures that the database is temporally correct.

Goal constraint	Description
$\forall t \in T : \text{temporallycorrect}(t)$	Transaction invocations are temporally consistent.

The objective function minimizes the load of refresh transactions. Transaction type TT specifies refresh transactions is $OUT(TT) \neq \emptyset$. Therefore, the load of refresh transactions is given by the following function.

Objective function: minimize $\sum_{TT \in TTs \wedge OUT(TT) \neq \emptyset} 2Wx(TT)/\Delta_{TT}$

8.3 Scheduler design

A straightforward "always-fresh" approach is used to ensure that all temporal requirements of all transactions can always be met. This means that at all moments, all temporal requirements of all transaction-types are met. In effect, we strengthen the temporal correctness requirement to:

$$
\begin{aligned}
&\text{alwaysfresh}(T) \equiv \\
&\quad \forall \tau \in [0, \infty), t \in T : \ \text{absolute}(Ttype(t), \tau) \wedge \text{relative}(Ttype(t), \tau)
\end{aligned}
$$

Together with the *haswritten* property of refresh transactions, bounds on the refresh rates of data items can be derived. A refresh transaction type TT can write several data items X. For each data item X, a required refresh rate Δ_X is computed. By taking the minimum of all required refresh rates Δ_X, Δ_{TT} can be computed:

$$\Delta_{TT} = min_{X \in OUT(TT)} \Delta_X$$

Required refresh rates

When a transaction t of type TT reads an instance x_i, it requires that $St(t) - x_i.\tau$ is contained in a certain *age interval* $[a, b]$. This age interval is defined by absolute and relative temporal requirements (see below). Due to the always-fresh approach, this is strengthened to

$$\forall \tau \in [0, \infty) : \exists x \in Insts(\tau) : \tau - x.\tau \in [a, b]$$

Therefore, the maximal allowed age difference between two successive updates of X is equal to the minimal width $b - a$ of any age interval $[a, b]$. Let x' be the instance, written after instance x. Suppose the refresh rate of X is Δ_X. With the *haswritten* property, bounds on the age-difference $x'.\tau - x.\tau$ can be found.

$$x'.\tau - x.\tau \leq \Delta_X + maxiage(X) - miniage(X)$$

If $\delta_X = b - a$ is the width of an age interval $[a, b]$, then the maximal allowed age difference between successive instances of data item X is at most δ_X. By combining this with the bound on the age difference that is derived above, the following requirement on the refresh rate of X is obtained.

$$\Delta_X \leq \delta_X - maxiage(X) + miniage(X)$$

Computing age intervals. Relative temporal requirements can be translated to absolute temporal requirements. These absolute temporal requirements then specify age intervals, that can be used to derive bounds on Δ_X. An example, suppose $(x, k, y, j, a, b) \in REL(TT)$. By placing absolute temporal requirements on both x_k and y_j, the relative temporal requirement can be satisfied: $(x, k, 10, 20), (y, j, 10 - a, 20 - b) \in ABS(TT)$. We calculate: $10 - (10 - a) \leq x_k.\tau - y_j.\tau \leq 20 - (20 - b)$, therefore $x_k.\tau - y_j.\tau \in [a, b]$): the relative temporal requirement is satisfied. This transformation is presented below in its generalized form.

Unrealistic absolute temporal requirements can be specified that require special attention. Suppose that data item X is used to store information from a remote sensor. The initial age of all instances x is at least 10: *miniage*(X)=10. An absolute temporal requirement $(x, k, 0, 8)$ is unrealistic: the database can never provide instances of X with $x.\tau \leq 8$. To express this, the lower bound of all absolute temporal requirements on X is strengthened to at least *miniage*(X).

The set of absolute temporal requirements that results from these transformations is called the set of $[l, l+\delta]$ intervals. For each instance x_i, read by invocations of transaction type TT, an age interval $[l_{x_i}, l_{x_i} + \delta_{x_i}]$ is defined. If the database scheduler ensures that at all moments τ, at least one instance x satisfies $\tau - x.\tau \in [l_{x_i}, l_{x_i} + \delta_{x_i}]$, the database satisfies the always-fresh property. We add the requirement that all age intervals of k instances of the same data item, read by the same transaction, have the same width: $\forall i, j \in [1, k] : \delta_{x_i} = \delta_{x_j}$. Instead of writing δ_{x_i}, we write δ_X. This reduces the complexity of determining $[l, l + \delta]$ intervals, and does not result in higher refresh rates Δ_X (since Δ_X depends on the smallest δ_{x_i}). The age intervals are constructed by applying the following rules.

- **Absolute temporal requirements**
  If $(x, i, a, b) \in ABS(TT)$
  then $a \leq l_{x_i}$
  $l_{x_i} + \delta_X \leq b$.

- **Relative temporal requirements**
  If $(x, i, y, j, e, f) \in REL(TT)$
  then $l_{y_j} + \delta_Y - f \leq l_{x_i}$
  $l_{x_i} + \delta_X \leq l_{y_j} - e$

- **Minimal instance age**
  The *haswritten* property specifies that the age of an instance x is at any time at least *miniage*(X). Therefore, we require that $l_{x_i} \geq$ *miniage*(X).

A set of $[l, l + \delta]$ intervals that satisfies these equations can be found using some search strategy, as is explained in section 8.3. A straightforward selection mechanism can use the age intervals at run-time to decide which instances transactions should read.

Finding good solutions

The *miniage*(X) and *maxiage*(X) function use the $[l, l + \delta]$ intervals of the transaction type TT_X as input. Furthermore, the results of the *miniage*(X) and *maxiage*(X) function are used in the computation of $[l, l + \delta]$ intervals of transaction types TT', whose invocations read X. This suggests that there is an order in which transaction types should be processed. If transaction type TT reads X, which is written by TT', then TT' should be processed before TT. However, cyclic dependencies are allowed in the specification of refresh transactions. For example, a transaction that writes to X is allowed to read X as well. In these cases, an (possibly too large) approximation of the $[l, l + \delta]$ intervals is used for the determination of *miniage*(X), *maxiage*(X). We will not consider cyclic dependencies in the remainder of this chapter.

The set of $[l, l + \delta]$ intervals that belongs to a transaction type should maximize the Δ_X values. High Δ_X values result in high Δ_{TT} values, which in turn minimize the objective function:

$$\sum_{TT \in TTs \wedge OUT(TT) \neq \emptyset} 2Wx(TT)/\Delta_{TT}$$

The objective function cannot be evaluated before all Δ_{TT} values have been defined. Instead of directly evaluating the objective function, we use an optimization function f (defined below). Function f only considers the local problem of constructing $[l, l + \delta]$ intervals for a single transaction type. It is constructed such that an optimal solution to the sub-problem can be found with non-linear programming techniques (see section 8.4). However, the resulting solution is not necessarily optimal for the global problem of minimizing the load that is generated by refresh transactions. Different heuristics that directly optimize the entire system load, instead of considering one transaction at a time, can lead to better results (at the costs of increased computational costs).

Optimization function f. The optimization function only considers the construction of $[l, l + \delta]$ intervals for a single transaction type TT. Function f is chosen such that the product of all Δ_X values ($(X, k) \in IN(TT)$) is maximized. This ensures that solutions are found that have large Δ_{TT_X} values, and no extremely small Δ values are found. In order to maximize $\Pi_{(X,k) \in IN(TT)} \Delta_X$, the size of the $[l, l + \delta]$ intervals should be as large as possible. Therefore, the optimization function of the

```
1  foreach X ∈ DataSet do Δ_X := ∞
2  G:=constructDependencyGraph(TTs)
3  topOrder:=topologicalSort(G)
4  for i:= 1 to |TTs| do
5      TT := topOrder[i]
6      Intervals=analyzeTransactionType(TT, Δ)
7      foreach [l_x, l_x + δ_x] ∈Intervals do Δ_X := min(Δ_X, δ_x − maxiage(X) + miniage(X))
8  foreach TT ∈ TTs : OUT(TT) ≠ ∅ do Δ_TT := min_{Y∈OUT(TT)} Δ_Y
```

Algorithm 8.2: MAIN ALGORITHM

transaction type analysis can be described as a product of all δ variables. As a first attempt, function f' is defined as follows.

$$f' = \Pi_{(X,k)\in IN(TT)}(\delta_X - maxiage(X) + miniage(i))$$

Undesired anomalies appear, if term $\delta_X - maxiage(X) + miniage(i)$ is smaller than 0. The additional constraint $\forall(X,k) \in IN(TT) : \delta_X \geq maxiage(i) - miniage(i)$ prevents them from occurring. Function f' is monotone increasing in each of the δ_i variables. We modify it to make it strict monotone increasing. Let ϵ be an infinitely small, positive constant. The optimization function f is defined as follows.

$$f = \Pi_{(X,k)\in IN(TT)}(\delta_X - maxiage(X) + miniage(i) + \epsilon)$$

An optimization that is not computationally expensive is applied to the heuristic. If consideration of previous transactions established the requirement $\Delta_X \leq c$ (for some constant c), increasing δ_X beyond $c + maxiage(X) - miniage(X)$ does not improve the final solution. Therefore, the requirement $\delta_X \leq c + maxiage(X) - miniage(X)$ is added to the set of constraints that have to be satisfied. Note that shrinking δ_X can lead to an increase of δ_Y, if the transaction type under consideration has specified a relative temporal requirement between X and Y. Therefore, better solutions can result if δ_X is bounded.

8.4 Algorithm

Algorithm 8.2 shows the main algorithm that translates temporal requirements to rate-monotonic requirements. The refresh rates Δ_X are defined incrementally, at each moment maintaining an optimistic approximation of the final value.

Lines 2 and 3, define the order in which transaction types are processed. The function "construct-DependencyGraph" in line 2 constructs a directed graph $G = (V, E)$ that represents the reads-from dependencies between transaction types. The set of vertices V represent transaction types ($V = TTs$). A directed edge (TT, TT') exists in E, if and only if $\exists X, k : X \in OUT(TT) \wedge (X, k) \in IN(TT')$. Function "constructDependencyGraph" is straightforward and not further described. If the transaction types are processed according to a topological order of G, all $[l, l + \delta]$ intervals of TT_X are defined, before either $miniage(X)$ or $maxiage(X)$ is computed. Function "topologicalSort" in line 3 returns an ordered array of transaction types (for an efficient topological sort algorithm, see [26]).

Lines 4-7 process transaction types according to the topological order. The analysis of a single transaction type TT is handled by procedure "analyzeTransactionType" (line 6), which is treated in

```
analyzeTransactionType(TT, Δ)
1  Cset := constructLinearConstraints(TT, Δ)
2  s^start := simplex-phase-I(Cset)
3  DD := constructDeltaDependencies(Cset)
4  Cset' := ∅
5  foreach (X, Y) ∈ DD do
6      Cset' += "δ_X − maxiage(X) + miniage(X) = δ_Y − maxiage(Y) + miniage(Y)"
7  s^nearopt := simplex-phase-II(Cset ∪ Cset', Σ_{(X,k)∈IN(TT)} δ_X, s^start)
8  s^opt := gradient-projection-method(Cset, f, ∇f, s^nearopt)
```

Algorithm 8.3: ANALYSIS PROCEDURE

section 8.4. It returns a set of $[l_x, l_x + \delta_X]$ intervals for instances x, read by invocations of TT. Line 7 adapts Δ_X such that $\Delta_X \leq \delta_X - maxiage(X) + miniage(X)$ is satisfied. Finally, line 8 computes the Δ_{TT} values from the Δ_X values.

Transaction type analysis

The transaction type analysis in algorithm 8.3 considers one transaction type TT in isolation. Line 1 defines the boundary conditions of the solution space, by invoking procedure "constructLinearConstraints". This procedure translates all temporal requirements and boundary conditions to a set of linear constraints $Cset$. The following rules are applied by the procedure to construct $Cset$.

- **Avoid anomalies in optimization function, no negative values**
  $\forall (X, n) \in IN(TT) : 0 \leq \delta_X - maxiage(X) + miniage(X)$
- **Use optimistic approximation as upper bound on δ values**
  $\forall (X, n) \in IN(TT) : \delta_X \leq \Delta_X + maxiage(X) - miniage(X)$
- **Absolute temporal constraints**
  $\forall (x, i, a, b) \in ABS(TT)$:
  $l_{x_i} \geq a$
  $l_{x_i} + \delta_X \leq b$
- **Relative temporal constraints**
  $\forall (x, i, y, j, a, b) \in REL(TT)$:
  $l_{x_i} \geq l_{y_j} + \delta_Y - f$
  $l_{x_i} + \delta_X \leq l_{y_j} - e$
- **Least initial age requirement**
  $\forall (X, n) \in IN(TT), i \in [1, n] : l_{x_i} \geq miniage(X)$

To find a feasible solution s^{start}, phase I of the simplex method [60] is executed in line 2. An optimization problem is obtained, which has a non-linear optimization function f, a set $Cset$ of linear constraints on the solution space, and a feasible solution s^{start}. Several solution methods exist for such problems. We use the gradient-projection method from [60]. The gradient of f in any solution vector x (denoted $\nabla f(x)$) is defined as follows. Let X^1 to X^n be the list of data items accessed by

invocations of TT.

$$\nabla f(x) = (\\ \Pi_{(X,k)\in IN(TT),X\neq X^1}(\delta_X - maxiage(X) + miniage(X) + \epsilon),\\ \ldots,\\ \Pi_{(X,k)\in IN(TT),X\neq X^n}(\delta_X - maxiage(X) + miniage(X) + \epsilon))$$

The gradient projection method requires a feasible solution (a solution that satisfies all constraints) as a starting point. Instead of using the arbitrary starting point s^{start}, we find a feasible solution that is already near the optimum in line 3-7.

In line 3, procedure "constructDeltaDependencies" determines which δ's are dependent. Two variables δ_X, δ_Y are dependent, if they both appear in the same constraint, or if they are constrained by variables that are dependent (i.e. $\delta_X \leq a$, $a \leq b$, $b \leq \delta_Y$ implies that δ_X, δ_Y are dependent). If δ_X and δ_Y are dependent, tuple (X, Y) appears in result DD. The dependencies amongst δ variables can be computed efficiently by using a union-find algorithm [26].

We find a good feasible solution $s^{nearopt}$, by using the simplex method (line 7) on the following adapted problem. The simplex method can only optimize linear functions, so instead of f, function $g = \sum_{(X,k)\in IN(TT)} \delta_X$ is chosen as the optimization function. Maximization of the linear optimization function g also maximizes f, if all δ_X variables are independent. If a set of δ variables is dependent, g does not give satisfactory results. For example, if $\delta_X + \delta_Y = 10$, an optimal solution for g is $\delta_X = 0$, $\delta_Y = 10$, which results in a very poor value for f. Suppose *miniage*(X)=*maxiage*(X) and *miniage*(Y)=*maxiage*(Y). Function f is optimal if $\delta_X = \delta_Y = 5$. A set of additional linear constraints $Cset'$ is added to (partly) remedy the shortcomings of g. For each dependent pair $(X, Y) \in DD$, a linear constraint is added in line 6: $\delta_X - maxiage(X) + miniage(X) = \delta_Y - maxiage(Y) + miniage(Y)$ This ensures that the poor solution that was found in the previous example cannot occur. Although a heuristic, in many cases the solution $s^{nearopt}$, found by phase II of the simplex method (line 7) will be on (some of) the surfaces where the maximum is found.

Finally, the optimal solution s^{opt} is found by invoking the gradient projection method in line 8, with $s^{nearopt}$ as a starting point. Since the optimization function f is strict monotone increasing in each of the δ dimensions, and linear equations bound a convex domain, there are no local maxima in which the gradient projection method can get stuck. Invoking the gradient projection method with s^{start} as starting point also results in an optimal solution. However, the gradient projection method can be computationally expensive, which motivates our two step approach.

8.5 Complete example: air traffic control

We look at a small part of an air traffic control system. The positions $P1$, $P2$ of two planes are measured by sensor transactions $rad1$, $rad2$. User transaction ipw checks $P1$, $P2$ for proximity, and issues an immediate proximity warning if necessary. Flight information is received from the planes by sensor transactions $inf1$, $inf2$, and stored in $F1$, $F2$. Two derive transactions $extra1$, $extra2$ use the position and flight information to extrapolate the future plane positions, stored in $E1$, $E2$. Finally, user transaction $nupw$ checks the future plane positions $E1$, $E2$, and gives a non-urgent proximity warning if necessary.

Transactions $rad1$, $rad2$ produce instances of $P1$, $P2$ that are between 0 and 20 milliseconds old. Transactions $inf1$, $inf2$ produce instances that are between 10 and 20 milliseconds old. User transaction ipw requires that it receives instances of $P1$, $P2$ that are between 0 and 50 milliseconds

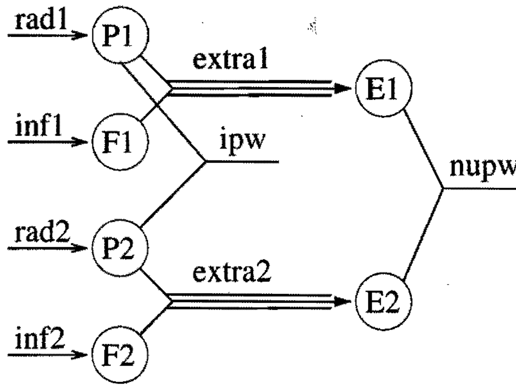


Figure 8.4: AIR TRAFFIC EXAMPLE

old. The derive transactions $extra1, extra2$ read three instances of $P1$, $P2$ and three instances of $F1$, $F2$. The age of these instances should be 50, 250 and 450 milliseconds, where and imprecision of 50 milliseconds is allowed. The time of measurement of derived data items $E1$, $E2$ depends on the input:

$$E1.\tau = \lfloor \sum_{i \in [1,3]} (P1_i.\tau + F1_i.\tau)/6 \rfloor - 5000$$

$$E2.\tau = \lfloor \sum_{i \in [1,3]} (P2_i.\tau + F2_i.\tau)/6 \rfloor - 5000$$

User transaction $nupw$ requires instances of $E1$, $E2$ that lie between 4000 and 5000 milliseconds in the future. Furthermore, the difference between the t.o.m. of $E1$ and $E2$ should be less than 200 milliseconds. A graphical representation of the system is shown in figure 8.4. In the corresponding specifications, we leave out transactions $rad2, inf2, extra2$, as they are equivalent to $rad1, inf1, extra1$.

$IN(rad1) = \{\}$ $\quad$ $IN(inf1) = \{\}$
$OUT(rad1) = \{P1\}$ $\quad$ $OUT(inf1) = \{F1\}$
$maxtom(P1) = 20$ $\quad$ $maxtom(F1) = 20$
$mintom(P1) = 0$ $\quad$ $mintom(F1) = 10$
$IN(nupw) = \{E1, E2\}$ $\quad$ $IN(ipw) = \{P1, P2\}$
$OUT(nupw) = \{\}$ $\quad$ $OUT(ipw) = \{\}$
$REL(nupw) = \{(E1, 1, E2, 1, -200, 200)\}$ $\quad$ $ABS(ipw) = \{(P1, 1, 0, 50), (P2, 1, 0, 50)\}$
$ABS(nupw) = \{(E1, 1, -5000, -4000), (E2, 1, -5000, -4000)\}$
$IN(extra1) = \{P1 : 3, F1 : 3\}$ $\quad$ $OUT(extra1) = \{E1\}$
$ABS(extra1) = \{(P1, 1, 0, 100), (P1, 2, 200, 300), (P1, 3, 400, 500),$
$(F1, 1, 0, 100), (F1, 2, 200, 300), (F1, 3, 400, 500)\}$
$maxtom(E1) = \lfloor \sum_{i \in [1,3]} (HIA_{P1_i} + HIA_{F1_i})/6 \rfloor - 5000$
$mintom(E1) = \lfloor \sum_{i \in [1,3]} (LIA_{P1_i} + LIA_{F1_i})/6 \rfloor - 5000$

Implementation

Algorithm 8.2 constructs a topological order that is in accordance with the dependency graph, as depicted in figure 8.4. Since the dependency graph does not enforce a total order, more than one topological order is possible, and the algorithm selects an arbitrary one:

$$rad1; rad2; inf1; inf2; ipw; extra1; extra2; nupw$$

The analysis of the sensor transactions is straightforward, since they do not read temporal data. The analysis procedure for ipw first constructs the following constraint-set:

$$20 \leq \delta_{P1} \qquad 20 \leq \delta_{P2}$$
$$0 \leq l_{P1} \qquad 0 \leq l_{P1}$$
$$l_{P1} + \delta_{P1} \leq 50 \qquad l_{P2} + \delta_{P2} \leq 50$$

The union-find loop determines that δ_{P1}, δ_{P2} are independent, therefore, solution $s^{nearopt}$ is optimal:

$$l_{P1} = l_{P2} = 0 \qquad \delta_{P1} = \delta_{P2} = 50$$

In the main algorithm, $\Delta_{P1} = \Delta_{P2} = 50 - 20 = 30$ is defined. Analysis of $extra1, extra2$ is similar to the analysis of ipw, except that $\Delta_{P1} = 30$ is used to add constraint $\delta_{P1} \leq 30 + 20$. Again, since $extra1, extra2$ specify no relative requirements, $s^{nearopt}$ is an optimal solution, resulting in:

$$l_{P1_1} = 0 \qquad l_{P1_2} = 200 \qquad l_{P2_3} = 400 \qquad \delta_{P1} = 50$$
$$l_{F1_1} = 10 \qquad l_{F1_2} = 200 \qquad l_{F2_3} = 400 \qquad \delta_{F1} = 90$$

In the main algorithm $\Delta_{P1} = 30$ and $\Delta_{F1} = 90 - 10 = 80$. Furthermore, since the $[l, l+\delta]$ intervals are defined for $extra1$, $miniage(E1)$ and $maxiage(E1)$ can now be computed:

$$miniage(E1) = \lfloor(50 + 250 + 450 + 100 + 290 + 490)/6\rfloor - 5000 = -4729$$
$$maxiage(E1) = \lfloor(0 + 200 + 400 + 10 + 200 + 400)/6\rfloor - 5000 = -4799$$

The analysis of $extra2$ is similar to the analysis of $extra1$. Finally, $nupw$ is analyzed. Although $nupw$ specified a relative requirement on its two inputs, $s^{nearopt}$ is optimal, due to the symmetry of our example.

$$l_{E1} = l_{E2} = -4799 \qquad \delta_{E1} = \delta_{E2} = 200$$
$$\Delta_{E1} = \Delta_{E2} = 200 - (-4729 - -4799) = 130.$$

We show that although the transaction analysis method finds optimal solutions for each separate transaction, the topological order can influence the final result. Suppose the following topological order was selected:

$$rad1; rad2; inf1; inf2; extra1; extra2; nupw; ipw$$

In the analysis of $extra1, extra2$, the result of the analysis of ipw ($\Delta_{P1} = \Delta P2 = 30$) is not yet available. Therefore, $maxiage(E1)$=$maxiage(E2)$= -4700, instead of -4729. Finally, this results in a higher refresh rate for $extra1, extra2$: $\Delta_{E1} = \Delta_{E2} = 101$, instead of 130! As a first heuristic in selecting the best topological order, user transactions should appear as early as possible in the topological order, since they do not influence tom-functions.

The algorithm generates rate-monotonic requirements from the computed Δ values. In the table below these requirements are summarized, together with the age at which instances can be removed.

This can be computed by considering the upper and lower bounds of age intervals, together with the update frequencies. For example: instances of data item $P1$ can be removed if $age(P1) \geq 430$, since the highest lower bound on $P1$ is 400, and the update frequency is 30. Therefore, if the age of an instance of $P1$ is 430, it is ensured that a second instance of $P1$, with age ≥ 400 exists: the oldest instance can be removed.

Requirement	Description
$P_{rad1} + D_{rad1} \leq 30$	Poll sensor
$P_{rad2} + D_{rad2} \leq 30$	Poll sensor
$P_{inf1} + D_{inf1} \leq 80$	Poll sensor
$P_{inf2} + D_{inf2} \leq 80$	Poll sensor
$P_{extra1} + D_{extra1} \leq 130$	Extrapolation
$P_{extra2} + D_{extra2} \leq 130$	Extrapolation
$age(P1) \geq 430$	Remove old instance $P1$
$age(P2) \geq 430$	Remove old instance $P2$
$age(F1) \geq 490$	Remove old instance $F1$
$age(F2) \geq 490$	Remove old instance $F2$
$age(E1) \geq -4599$	Remove old instance $E1$
$age(E2) \geq -4599$	Remove old instance $E2$

8.6 Conclusions

The model presented in this chapter allows for a precise specification of temporal constraints of transactions. It is sufficiently powerful to support prediction models that require multiple instances of the same data item. Other issues related to prediction are treated as well: instances can have time of measurements that lie "in the future". Since their values are predicted, the age of these predicted instances is essentially negative when they are first created. Therefore, it might be possible that instances "are not yet valid". Minimum age requirements are introduced to deal with these cases.

The always-fresh algorithm that accompanies the model shows that efficient implementations of systems, specified in the model, actually exist. The always-fresh approach is not efficient in systems where temporal data is rarely accessed, and the algorithm should be modified to handle such systems. Less computation-intensive "on demand" approaches can reduce the refresh overhead (see [2]). Furthermore, the algorithm finds local optima for each separate transaction, instead of optimizing the entire system at once. Whether such a global approach is feasible (from a computational point of view) should be investigated.

Specification of intermediate instances have a separate semantics, something that is not very elegant. A semantics that encompasses both normal temporal requirements and temporal requirements on intermediate instances can simplify the model. This new semantics should lead to clear system specifications, and should be implementable.

The model can fully specify hard real-time databases, where all requirements are always met by the system. In soft real-time and firm real-time databases, the database is allowed to violate temporal requirements, although it does degrade performance of the database. Penalty-functions and alternative actions have to be incorporated in the model, to support such systems.

Chapter 9

Conclusions

The topic of this thesis is optimization of real-time database schedulers. Since the problem specification is the starting point for any design, specification of real-time database scheduling problems is treated in depth. The boundary conditions of the design are described by so-called fact constraints. These express important characteristics of the scheduler input and the platform on which the scheduler operates. The objectives of the design are given by so-called goal constraints and the objective function. The goal constraints specify requirements that have to be satisfied by the scheduler. The performance of the scheduler is measured by the objective function. This is described in chapter 2.

The remainder of the thesis deals with the design of real-time database schedulers. Chapters 4 to 6 describe the design of soft and firm real-time schedulers. These schedulers either optimize performance, or ensure that the scheduler is analyzable. Chapters 7 and 8 deal with hard real-time schedulers. These chapters focus on the identification of new concepts and problem recognition.

9.1 Performance

Many of the applications that are described in chapter 1 push the capabilities of databases to their limits. Telecommunication systems provide communication channels with extremely large capacity demands. High-energy physics experiments generate unimaginable amounts of data in milliseconds, and banking systems serve millions of customers. All applications have real-time requirements, either dictated by demands of impatient customers, by available buffer space or by physical limitations. Independent of the underlying reasons, these systems have one thing in common: satisfying the real-time requirements they pose on databases is only possible by efficient use of a high-performance platform.

Performance in real-time databases is defined differently for soft, firm and hard real-time. However, their performance measures all depend on the response times of transactions. The response time is the time between the request for execution of the transaction and the completion of the execution. Reducing the transaction response times increases the performance of soft, firm and hard real-time databases. Therefore, the response time of a transaction is one of the most important performance measures in this thesis.

The approach of this thesis is to increase the performance of the database by optimizing the real-time database scheduler. The scheduler determines the order in which transactions are allowed to execute. This has a direct impact on the performance of the database: transactions can be delayed considerably (which results in poor response times), if the scheduler does not allow a high degree of concurrency. Furthermore, the overhead of the scheduler also increases the response times of

transactions. Both scheduler overhead and the degree of concurrency offered by the scheduler have to be optimized.

There is a relation between the degree of concurrency that is offered by the scheduler and the overhead that the scheduler imposes on transactions. This relation is complex, and requires more research, for a better understanding of the performance of real-time database schedulers. In this thesis, a number of observations are made that indicate that schedulers that offer a high degree of concurrency impose more scheduler overhead on transactions. To arrive at this result, we investigate the sources of scheduler overhead.

The scheduler fulfills two tasks: computing and enforcing the schedule. Note that on-line schedulers perform both tasks at the same time, such that the distinction is not always clear. For example: when a transaction is blocked on a lock request, two phase locking both enforces and decides on the schedule. To compute the schedule, the schedule requires information about the execution of transactions (such as data access patterns and execution durations). Collecting this information introduces overhead, especially if the required information is distributed over multiple sites. Using the information to decide on a schedule also introduces overhead. Generally, a scheduler that offers a high degree of concurrency requires detailed information about transactions. This can require inter-site communication. Furthermore, the decision procedures that are used by these schedulers require more internal computation. This sounds straightforward, but it is not trivial. The exact relation between the information and computation time that is needed by a scheduler and the degree of concurrency that it can offer is unknown.

To enforce the schedule, the scheduler has to synchronize transactions (for example, by using a locking strategy) according to the schedule. A direct relation exists between the number of synchronization points within transactions and the number of different schedules that can be enforced by the scheduler. If a scheduler offers a high degree of concurrency, it can generate a large number of schedules. Therefore, it will require a large number of synchronization points within each transaction. Summarizing, scheduler overhead consists of internal computation (to compute the schedule), local synchronization (to enforce the schedule) and inter-site communication (to gather information and to enforce the schedule).

9.2 Targeted design

In certain environments, the transaction response time is mainly determined by the scheduler overhead. The Telecom environment in chapter 4 is a typical example of such an environment. In other environments, the scheduler overhead is negligible compared to the computational requirements of transactions. It stands to reason that these different environments require a different scheduler. The Telecom environment benefits from a lightweight scheduler that adds little overhead. If overhead is negligible, a high degree of concurrency is more important than little overhead. Indeed, the experiment in chapter 5 shows that OCC-light outperforms DOCC-BF if transaction executions are short (overhead is not negligible), while DOCC-BF performs much better if transaction executions are long (overhead is less important).

We conclude that scheduler performance is highly dependent on the environment in which the scheduler operates. With this observation in mind, it seems logical to design the scheduler *especially* for the environment it will operate in. This requires a more detailed description of the database scheduling problem. We propose a two-level problem specification. The first part of the problem specification contains the abstract problem specification, without detailed information about the environment in which the scheduler will operate. It allows the database designer to use existing solutions

from literature. The second part of the problem specification contains application-specific details. It allows the database designer to optimize its design to the actual application that will use the scheduler. This targeted problem specification allows for the design of high performance schedulers. For example, chapter 4 and 5 specify the same abstract problem, but the application-specific details differ. This explains the different designs of OCC-light and DOCC-BF.

9.3 Analysis and design

Chapter 6 takes the idea of targeted scheduler design one step further. It is observed that the real-time performance of database schedulers is often hard to analyze. Schedulers are optimized for high performance, and these optimizations can complicate the analysis. Two alternatives to analysis are simulation studies and measuring the performance in actual systems. These alternatives have drawbacks too: getting results from simulations or prototypes is often time-consuming and costly. In chapter 6 a third alternative is presented: designing the scheduler such that its analysis is straightforward. By avoiding constructs in the scheduler that are hard to analyze, it proves possible to create an analyzable database scheduler and to give a Markov analysis of its performance in a number of environments. The SQSL scheduler demonstrates the feasibility of the approach. Further work in this area should investigate more closely the relation between the analyzability of a scheduler, and the way that it is designed. The design of the SQSL scheduler is based on the requirement that the performance can be analyzed with Markov models as the only analysis tool. This places strong restrictions on the types of schedulers that can be analyzed. This range of analysis tools can be extended to avoid some of these restrictions. Probably, the analysis and design of more generic and more powerful schedulers is possible if clear relations between analysis and design are established.

9.4 Off-line database scheduling

In chapter 7, the off-line scheduler of the hard real-time DEDOS system is extended with database functionality. A pre-processor (the transaction serializer) is added to the DEDOS scheduler. It translates database requirements into hard real-time requirements. Duplicated functionality in the DEDOS scheduler and the transaction serializer is avoided. Therefore, the transaction serializer focuses on satisfying conflict serializability. All real-time constraints and reliability constraints are satisfied by the DEDOS scheduler. As a result, the standard database protocols of reading and writing data can be simplified considerably, which increases the concurrency that is allowed by the system. The simplified scheduling problem is still NP-hard. It can be approximated with a number of techniques like constraint satisfaction, or genetic algorithms. Chapter 7 shows that databases can be added to existing hard real-time scheduling systems, without requiring an entirely new scheduler. The transaction serializer does not use any information about the internal structure of the DEDOS scheduler, or the platform on which the DEDOS scheduler executes the applications. Therefore, the transaction serializer can be added to other hard real-time schedulers that offer a similar interface as the DEDOS scheduler. Chapter 7 identifies the scheduling problem and gives an initial solution for the transaction serializer. Further research should try different approaches to the scheduling problem, and experiment with test applications, to investigate the performance of the transaction serializer.

9.5 Temporal consistency

The classic database paradigm of reading and writing according to a serializable schedule does not suffice for all real-time applications. Applications that directly interact with a continuously changing outside world need to store data that describes this environment. This temporal data grows old as time passes and it loses its value to the application. Since serializability is a consistency requirement that does not have a notion of time, traditional databases do not support such temporal data. The application has to ensure that temporal data is refreshed. To relieve the application programmer of the burden of refreshing temporal data, the functionality of the database is extended. In chapter 8 we introduce the notion of transaction-based temporal consistency. Each transaction can specify a time interval for each data item that it reads. The database ensures that the values that are actually read describe the state of the environment during these time intervals.

Transaction-based temporal consistency is a concept that closely reflects the real needs of applications. Different processes within an application can use the same data for different purposes. Understandably, the timing requirements that belong to those different processes can differ as well. This is reflected by transaction-based temporal consistency: each transaction can specify independent time intervals for each data item that it reads. Transactions can require more than one value of the same data item, corresponding to different time intervals. This can be useful to interpolate or extrapolate values. Therefore, the specification method is suitable for systems that use prediction.

There are still extensions necessary before transaction-based temporal consistency can be applied in a general setting. Transactions can specify their temporal requirements. However, it is unclear what should happen if the database cannot fulfill these requirements. Further research should extend transaction-based temporal consistency to deal with soft- and firm real-time systems.

9.6 A final word

Over thirty years of research into real-time scheduling and database scheduling has resulted in a large number of scheduling algorithms. Real-time, distributed database scheduling can use results from both fields as sources of inspiration. To effectively use these results, the relations between real-time algorithms and database algorithms have to be investigated more closely. The concept of concurrency is not directly related to real-time performance measures. The degree of concurrency that is offered by a scheduler is measured in the number of different schedules that can be generated. Real-time performance measures refer to start and finishing times of transactions. That there is a relation between the degree of concurrency and the real-time performance has been shown in this thesis. However, the exact relation is complex and not known. Better understanding is required before a unified theory of real-time, distributed database scheduling can emerge.

Appendix A

List of symbols

Variable	Domain	Description
$Cpower$	work/second	Amount of work that a processor can process in one time unit.
$Cswitch$	seconds	Overhead caused by a context switch.
$SiteSet$	$\mathcal{P}$(identities)	The set of processors.
$Ncpus$	$\mathbf{N}^+$	The number of processing units.
$Mlatency$	seconds	Latency of the shared memory
$Mthoughput$	bytes/second	Throughput of the shared memory
$Msync$	seconds	Overhead caused by the synchronization primitive
$Nlatency$	seconds	Latency of the network.
$Nthoughput$	bytes/second	Throughput of the network.
$DataSize$	bytes	The size of a data item.
$DataSet$	$\mathcal{P}$(identifiers)	The set of data items that is stored in the database.
OPS	$\mathcal{P}$(identifiers)	The set of all access operations.
$OPtypes$	$\mathcal{P}$(identifiers)	The set of possible operations.
$Amode$	$OPS \rightarrow OPtypes$	Function that defines access mode of each operation.
$Daccess$	$OPS \rightarrow DataSet$	Function that defines which item an operation accesses.
$ConfRel$	$OPtypes \times OPtypes \rightarrow Bool$	The conflict relation on $OPtypes$.
$InterRel$	$OPtypes \times OPtypes \rightarrow Bool$	The interference relation on $OPtypes$.
$DataPlace$	$Items \rightarrow SiteSet$	A function that assigns data items to sites.
T	$\mathcal{P}$(identifiers)	The set of transactions that is executed by the scheduler.
St	$T \rightarrow$ seconds	The start time of transactions.
Ft	$T \rightarrow$ seconds	The finish time of transactions.

Variable	Domain	Description
W	$T \rightarrow$ work	The amount of work of transaction.
X	$T \rightarrow$ seconds	The total execution time of a transaction.
Arr	$T \rightarrow$ seconds	The moment of arrival of transactions.
Dl	$T \rightarrow$ seconds	The deadline of transactions.
Est	$T \rightarrow$ seconds	Earliest start time of a transaction.
tr	$OPS \rightarrow T$	Function that defines to which transaction an operation belongs.
$Aorder$	$OPS \times OPS \rightarrow Bool$	The sequence of all database accesses.
Exp	$OPS \rightarrow$ inconsistency	The amount of inconsistency that is exported by an access operation.
Imp	$OPS \rightarrow$ inconsistency	The amount of inconsistency that can be imported by an access operation.
$TransPlace$	$T \rightarrow SiteSet$	This function identifies the site on which a transaction executes.
$Pdatalocal$	[0, 1]	The probability that a data access is local.

Appendix B

Pseudo-code conventions

The algorithms that are described in this thesis are presented in pseudo-code. *Pseudo-code* cannot be directly compiled, but focuses on the algorithmic aspects, rather than on organizational aspects (like historically the case for object orientation). Furthermore, it contains mathematical formulas, which are usually expressed much more verbose in traditional programming languages. It provides a powerful way to describe algorithms. This appendix describes the most important language conventions that have been used.

B.1 Expressions

Expressions can be simple constants (numeric, boolean or strings), variables, or functions applied to expressions ('3+5','not true', 'f(x)', etc.). Such expressions are found in most programming languages. In addition, an object-oriented style of function invocations (for example "Q.empty()") is allowed. A number of expressions are used that are not commonly supported by current programming languages:

- $X \in Y, X \subset Y, \emptyset$, **etc.** Sets are treated as type constructors, and are treated as is usual in mathematics.
- $|X|$ Returns the cardinality of set X.
- (X, Y) Tuples are treated as type-constructors, as is often found in functional languages. They are similar to Pascal and C records, but the fields are nameless, and specified by the order of their appearance. We allow tuples with two or more fields of arbitrary types, i.e. $(3, 'a', [1, 3], A)$ is allowed.
- **a[X],b[Y,Z]** As any self-respecting programming environment, multi-dimensional arrays are valid expressions.
- **doit(X,Y)** Function invocations and procedure calls are denoted as is common in most third generation programming languages.
- [$X|EXPRESSION$] This notation is called list-comprehension. It generates a set of elements that match X, and for which EXPRESSION evaluates to true. For example $[(a, b)|a \in [1, 3, 5] \wedge b \in [2, 4] \wedge a < b]$ generates set $\{(1, 2), (1, 4), (3, 4)\}$. Note that the order is arbitrary.

- { **acknowledge,** t, X } Messages between processes consist of an identifier ("acknowledge"), and a list of parameters, similar to tuples.

B.2 Statements

Pseudocode consists of statements that are executed in a sequential fashion. Sequences of statements are either separated by ';', or each statement is written on a separate line. Many statements are similar to Pascal, Basic, or C statements:

- **X := EXPRESSION;** The assignment statement, the variable X is assigned the value that is defined by the expression.

- **(X,Y) := EXPRESSION;** Assignments to tuples are allowed. In this particular example, the expression has to result in a two-tuple. The first field is assigned to X, the second field is assigned to Y.

- **VARIABLELIST := EXPRESSIONLIST;** This is a multi-assignment statement. It is assumed that assignments are evaluated from left to right. Example: X,Y := 1,2; is equivalent to X:=1; Y:=2;

- **X ⊗= EXPRESSION;** Here, ⊗ can be any arbitrary operator. This is also an assignment statement, and is functionally equivalent to X := X ⊗ (EXPRESSION).

- **for X:= EXPRESSION to EXPRESSION2 do STATEMENTLIST;** This statement is the standard for-loop, as is found in Pascal.

- **while (EXPRESSION) do STATEMENTLIST;** This statement is similar to the while-statement of Pascal and C.

- **foreach VARIABLELIST∈ X : EXPRESSION do STATEMENTLIST;** The statementlist is executed for each possible combination of assignments to values in the variable-list, for which the expression evaluates to true. The order of execution is left unspecified. Example: "foreach $p, q \in [1, 2, 3] : p < q$ do print p,q;" will print 3 pairs of numbers, but the order in which they are printed is not specified, i.e. " 1,3 2,3 1,2" is a possible output.

- **process ! message** This statement specifies the asynchronous sending of a message to other processes. The ! statement is non-blocking.

- **receive MATCHEXPRESSION → STATEMENTLIST end** A match-expression is a normal expression that contains unbound variables. When a message arrives, it is checked against the match-expression. If a variable-binding can be found, the corresponding statementlist is executed. One receive statement can actually have multiple MATCHEXPRESSION/STATEMENTLIST pairs. The first match is selected. The receive statement is blocking, as long as no matching message arrives, the process waits. This notation is very similar to the Erlang language [7].

```
TwoPhaseCommitCoordinator(t)
    foreach p ∈ Participants[t] do p ! { prepare, t }
    state:= ok
    foreach p ∈ Participants[t] do
        receive {ok,t} → skip
            {abort,t} → state:=abort
        end
    if state=ok then commit else abort
    foreach p ∈ Participants[t] do p ! {state,t}
    foreach p ∈ Participants[t] do receive {finished,t} → skip end
```

Algorithm B.1: TWO PHASE COMMIT COORDINATOR, PROCEDURAL STYLE

B.3 Concurrent programs

A concurrent program consists of a set of processes. The code of a process can be represented by a normal imperative program (similar to a pascal procedure), or a set of responses to message arrivals. A response to a message arrival is denoted by the message, followed by the code that is executed when the message arrives. If a process can be described in this representation it is called a *reactive process*. We give an example for each style of representation.

The code of algorithm B.1 specifies the behavior of the coordinating process in the Two Phase Commit protocol (further explained in appendix C) in an imperative style. The code of algorithm B.2 specifies the behavior of participating processes in the Two Phase Commit protocol, in a reactive style. Which style is used for which algorithms depends on the type of algorithm, and personal taste. The main scheduler protocols in this thesis are presented as reactive servers, as they manage more than one transaction in an interleaved fashion. This is more easily expressed in the reactive style, than in the imperative style.

```
{ prepare, t }
    coordinator ! {localstate(t), t}

{ ok,t }
    commit t
    coordinator ! {finished,t}

{ abort,t }
    abort t
    coordinator ! {finished,t}
```

Algorithm B.2: TWO PHASE COMMIT PARTICIPANT, REACTIVE STYLE

Appendix C

Basic algorithms

A number of algorithms have been so influential that they are found in almost every database scheduler. This appendix presents some of the most common algorithms that are used in this thesis.

C.1 Locking

Binary semaphores [29] were introduced in the late sixties, but are better known as "locks". A lock has two access methods, called "lock" and "unlock". The first process p that invokes the lock method continues execution as normal. Process p is said to be *holding* the lock. When process p invokes the unlock method, process p *frees* the lock. Whenever a process q invokes the lock method while the lock is held by another process p, q's execution is suspended until p frees the lock. If more than one process is suspended when p frees the lock, only one process can resume execution (FIFO order).

In databases, locks have more functionality than standard binary semaphores. They differentiate between read-locks and write-locks. Read-locks can be shared: more than one transaction can hold a read-lock at the same time. To avoid starvation of transactions that place write-locks, all lock requests are treated in FIFO order. I.e. if a transaction is blocked on a write-lock, all subsequent read-lock requests are blocked as well.

Algorithm C.1 specifies the lock and unlock procedures for a single lock. Processes that requests locks send lock-requests to a so-called *lock server*, and wait for a return message, the lock-grant. The lock server maintains the following datastructures.

- Number of placed read-locks $nRLocks$. This counts the number of read locks that are set.
- Number of requested&held write-locks $nWLocks$. This counts the number of write locks that are set, plus the number of blocked write lock invocations. If $nRlocks = 0 \wedge nWLocks > 0$, a write lock is currently placed. If $nWLocks > 0$, incoming read-lock requests are not granted. If $nRLocks + nWLocks > 0$, incoming write-lock requests are not granted.
- Lock queue Q. This queue stores the lock-requests that have not yet been granted.

Whenever the unlock method is invoked, either $nRLocks$ or $nWLocks$ is decremented. If $nWLocks$ is decremented, possibly more than one read-lock request is granted, if there are a number of read lock requests at the head of the lock queue.

```
lock-request(t,mode)
    lockserver ! { lock, t, mode }
    receive { grant, t } → skip end

unlock-request(t)
    lockserver ! { unlock, t }
    receive { ack, t } → skip end

{ lock, t, mode }
    if mode=read then
        if nWLocks > 0 then Q.enqueue({t,read})
        else nRLocks+ = 1
            t ! { grant, t }

    else nWLocks+ = 1
        if nWLocks > 1 ∨ nRLocks > 0 then Q.enqueue({t,write})
        else t ! { grant, t }

{ unlock, t }
    if nRlocks > 0 then nRlocks− = 1
    else nWlocks− = 1
    if nRlocks = 0 ∧ ¬Q.empty() then
        (Next,Mode) := Q.dequeue()
        Next ! { grant, Next }
        if Mode = write then nWlocks+ = 1
        else nRlocks+ = 1
            while ¬Q.empty()∧ Mode=read do
                (Next,Mode) := Q.first()
                if Mode=read then (Next, Mode) := Q.dequeue()
                                  Next ! { grant, Next }
                                  nRlocks+ = 1
    t ! { ack, t }
```

Algorithm C.1: A LOCK IMPLEMENTATION

C.2 Wait-die deadlock breaking

The wait-die procedure is used as a straight-forward deadlock breaking mechanism. It assumes that a total order $<$ is defined on the set of transactions. Whenever a transaction t requests a lock on data item X in a certain mode tm, it is only allowed to wait if all conflicting, preceding transactions q in

```
wait-die(t,X,tm)
    if ∃(q, qm) ∈ LockSet[X] ∪ toset(Q[X]) : ConfRel(tm, qm) ∧ q < t then restart t
```

Algorithm C.2: WAIT-DIE PROCEDURE

the lock queue satisfy $q > t$. Otherwise, t is restarted ("die"), and releases all its locks.

In algorithm C.2, we have assumed that $LockSet[X]$ contains tuples (t, tm), of transaction identities t, and their locking mode tm. These transactions hold locks on X. Furthermore, we assume that function "toset($Q[X]$)" returns the set of (t, tm) pairs of transactions t that are waiting for a lock on X, and are stored in locking queue $Q[X]$. The wait-die procedure checks these data structures and restarts the transaction if necessary.

C.3 Two phase locking

The two phase locking protocol (2PL) is the dominant database scheduler in industry. Developed in the seventies [30], it is the first database scheduler that provided conflict serializable schedules that allowed for a reasonable amount of concurrency. Its computational overhead is negligible, but its synchronization overhead is linear in the number of accessed data items.

Each transaction t executes according to the protocol that consists (as the name implies) of two phases. In the *locking phase* (or growing phase) transaction t is not allowed to unlock any locks. Before t accesses a data item X, it has to request an appropriate lock (read or write) on X. In the *unlocking phase* (or shrinking phase) t is not allowed to lock new data items. Before t finishes execution, it has to unlock all its locked data items.

The 2PL scheduler has two functions. First, it maintains a lock for each data item X, stored in the database. Second, without extensions, the 2PL protocol can suffer from deadlocks. One possible solution is to have a separate process check for deadlocks, and break them (see for an overview [48]).

At the start of the unlocking phase, transaction t holds locks to all data items that it accesses. By ordering transactions according to the start of their unlocking phase, a conflict-equivalent sequential schedule can be found easily. This straight-forward correctness, combined with a simple implementation is one of the reasons that 2PL is the defacto industry standard.

C.4 Two phase commit

The two phase commit protocol is used to reliably commit distributed transactions t. The algorithm is initiated by the *coordinating site*, named the coordinator. The sites on which the transaction t that is committing has executed are called *participants* and are stored in set $Participants[t]$. We have assumed that each participant can invoke a function "localstate(t)" that determines whether t is allowed to commit at the participant, it results in either "ok" or "abort". Furthermore, the algorithm does not contain the code that is executed when a transaction is actually committed to the database, or aborted.

Basically, the execution consists of two phases. In the first phase, the coordinator asks the participants whether everything is ok, and if the participants are ready to commit. If all participants are ready to commit, the coordinator decides to go ahead, commits, and notifies all participants that they can commit the results of the transaction to the database. Finally, the coordinator terminates once it has received acknowledgments of all participants.

If errors occur (for example, one participant has crashed and does not respond), several possible scenario's exist. These are further described in database textbooks like [51], we limit ourselves to the normal two phase commit execution. The algorithms for both the coordinator and the participants are described in appendix B, algorithms B.1 and B.2.

Bibliography

[1] E. Aarts and J. Lenstra, editors. *Local Search in Combinatorial Optimization*. Discrete Mathematics and Optimization. John Wiley & Sons, Baffins Lane, Chichester, England, 1997.

[2] B. Adelberg, B. Kao, and H. Garcia-Molina. Database support for efficiently maintaining derived data. *Proc. of the 5th Int. Conf. on Extending Database Technology*, pages 223–240, 1996.

[3] D. Agrawal and A. El Abbadi. Locks with constrained sharing. *ACM 089791-352-3/90/0004/0085*, pages 85–93, 1990.

[4] D. Agrawal, A. El Abbadi, and R. Jeffers. An approach to eliminate transaction blocking in locking protocols. *Proc. of the 11th conf. on Principles of Database Systems*, pages 223–235, 1992.

[5] P. Apers. *Query processing and data allocation in distributed database systems*. PhD thesis, Mathematical Centre, Kruislaan 413, Amsterdam, The Netherlands, 1983.

[6] E. Argante. *CoCa: a model for parallelization of high energy physics software*. PhD thesis, Eindhoven University of Technology, Den Dolech 2, Eindhoven, The Netherlands, 1998.

[7] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG*. Prentice Hall, Campus 400, Maylands Avenue, Hemel Hempstead, Hertfordshire, England, 1996.

[8] N. Audsley, A. Burns, M. Richardson, and A. Wellings. Absolute and relative temporal constraints in hard real-time databases. *Proc. of the Fourth Euromicro Workshop on Real-time systems*, pages 148–153, 1992.

[9] N. Audsley, A. Burns, M. Richardson, and A. Wellings. Data consistency in hard real-time systems. *Informatica*, (19):223–234, 1995.

[10] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, C. Spaccamela, A. Mar, and M. Protasi. *Approximate solution of NP-hard optimization problems*. Springer-Verlag, 1998.

[11] E. Bakker and J. van Leeuwen. The optimal placement of replicated items in distributed databases on tree-like networks. Technical Report RUU CS91-23, Utrecht University, P.O. Box 80.089, Utrecht, The Netherlands, 1991.

[12] D. Bell and J. Grimson. *Distributed Database Systems*. International computer science. Addison-Wesley, Wokingham, England, 1992.

[13] P. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, pages 185–221, 1981.

[14] A. Bestavros. Speculative concurrency control. *Tech. Report BUCS-TR-93-002, Computer Science Department, Boston University, MA*, 1993.

[15] A. Bestavros. Multi-version speculative concurrency control with delayed commit. *Proc. of the 1994 Int. Symposium on Computers and their Applications*, 1994.

[16] M. Bodlaender. *Scheduler optimization in real-time distributed databases*. PhD thesis, Eindhoven University of Technology, Den Dolech 2, Eindhoven, The Netherlands, 1999.

[17] M. Bodlaender, S. Sassen, P. van der Stok, and J. van der Wal. The response time distribution in a multi-processor database with single queue static locking. In *Proceedings of the 4th Int. Workshop on Parallel and Distributed Real-Time Systems*, pages 118–121, 1996.

[18] M. Bodlaender and P. van der Stok. Design issues of an efficient distributed database scheduler for telecom. In *Proceedings of the 7th Int. Conf. on Computer Communications and Networks*, pages 897–904. IEEE, 1998.

[19] M. Bodlaender and P. van der Stok. A transaction-based temporal data model that supports prediction in real-time databases. In *Proc. of the 10th Euromicro Workshop on Real Time Systems*, pages 197–203. IEEE, 1998.

[20] M. Bodlaender, P. van der Stok, and S. Son. A transaction-based temporal data model for real-time databases. *Proc. of the Joint Workshop on Parallel and Distributed Real-Time Systems*, pages 149–158, 1997.

[21] C. Boksenbaum, M. Cart, J. Ferrié, and J. Pons. Concurrent certifications by intervals of timestamps in distributed database systems. *IEEE Transactions on Software Engineering*, pages 409–419, 1987.

[22] G. Buckley and A. Silberschatz. Beyond two-phase locking. *Journal of the ACM*, 32(2):314–326, April 1985.

[23] R. Chandra and A. Segev. Managing temporal financial data in an extensible database. *Proc. of the 19th VLDB Conference*, pages 302–313, 1993.

[24] M. Chen and K. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *IEEE Transactions on Parallel and Distributed Computing*, 1(2):184–194, 1990.

[25] M. Cochinwala and J. Bradley. A multidatabase system for tracking and retrieval of financial data. *Proc. of the 20th VLDB Conference*, pages 714–721, 1994.

[26] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT press, McGraw-Hill Book Company, Cambridge, Massachusetts, 1990.

[27] S. Dasgupta. *Design Theory and Computer Science*. Processes and Methodology of Computer Systems Design. Cambridge University Press, Trumpingtom street, Cambridge, England, 1991.

[28] A. Datta and I. Viguier. Providing real-time response, state recency and temporal consistency in databases for rapidly changing environments. *Information Systems*, 22(4):171–198, 1997.

[29] E. Dijkstra. The structure of "the"-multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[30] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[31] P. Franaszek, J. Haritsa, J. Robinson, and A. Thomasian. Distributed concurrency control with limited wait-depth. In *Int. Conf. on Distributed Computing Systems*, pages 160–167. IEEE, 1992.

[32] P. Franaszek, J. Haritsa, J. Robinson, and A. Thomasian. Distributed concurrency control based on limited wait-depth. In *Transactions on Parallel and Distributed Systems*, pages 1246–1264. IEEE, 1993.

[33] J. Freytag, D. Maier, and G. Vossen, editors. *Query Processing for Advanced Database Systems*. Data Management Systems. Morgan Kaufmann Publishers, San Mateo, California, 1994.

[34] H. Garcia-Molina. *Performance of Update Algorithms for Replicated Data*. Computer Science: Distributed Database Systems. UMI Research Press, Ann Arbor, Michigan 48106, 1981.

[35] L. George and P. Minet. Condition de faisabilit'e et temps de r'eponse maximum pour un syst'eme transactionnel r'eparti temps r'eel. In *Proceedings of the Real-time Systems Conference*, 1995.

[36] R. Golsteijn. A database extension for dedos. Master's thesis, Eindhoven University of Technology, Den Dolech 2, Eindhoven, The Netherlands, 1997.

[37] V. Gottemukkala and T. Lehman. Locking and latching in a memory-resident database system. *Proc. of the 18th VLDB Conference*, pages 533–544, 1992.

[38] G. Grimmett and D. Stirzaker. *Probability and Random Processes*. Clarendon Press, Oxford, 1989.

[39] D. Hammer, E. Luit, O. van Roosmalen, P. van der Stok, and J. Verhoorsel. Dedos: A distributed real-time environment. *IEEE Journal of Parallel and Distributed Technology*, 2(4):32–47, 1994.

[40] J. Haritsa, M. Carey, and M. Livny. Dynamic real-time optimistic concurrency control. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 94–103, 1990.

[41] J. Huang, J. Stankovic, K. Ramamritham, and D. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *Proceedings of the Very Large Databases Conference*, pages 35–46, 1991.

[42] P. Jansen and E. Wijgerink. Flexible scheduling by deadline inheritance in soft real-time kernels. In *Multimedia 96 Conference*, pages 323–330, 1996.

[43] W. Jonker and L. Nieuwenhuis. Overview of databases requirements for intelligent networks. *Proc. of the 3rd Joint Workshop on Parallel and Distributed Real-Time Systems*, pages 160–161, 1995.

[44] A. Kägi, D. Burger, and J. Goodman. Efficient synchronization: Let them eat qolb. In *24th Ann. Int. Symposium on Computer Architecture*, pages 170–179. ACM, 1997.

[45] W. Kim, D. Reiner, and D. Batory, editors. *Query Processing in Database Systems*. Springer-Verlag, Berlin, 1985.

[46] Y. Kim and S. Son. Supporting predictability in real-time database systems. *IEEE Real-Time Technology and Applications Symposium*, pages 38–48, 1996.

[47] L. Kleinrock. *Queueing Systems*, volume 1. John Wiley & Sons, 1975.

[48] E. Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys*, 19(4):303–328, 1987.

[49] H. Kopetz. *Design Principles for Distributed Embedded Applications*. Real-time Systems. Kluwer Academic Publishers, Boston, 1997.

[50] H. Korth. Locking primitives in a database system. *Journal of the ACM*, 30(1):55–79, Januari 1983.

[51] H. Korth and A. Silberschatz. *Database System Concepts*. McGraw-Hill, 1991.

[52] V. Kumar, editor. *Concurrency Control Mechanisms in Centralized Database Systems*. Prentice Hall, Englewood Cliffs, 1996.

[53] H. Kung and J. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, june 1981.

[54] T. Kuo and A. Mok. Ssp: a semantics-based protocol for real-time data access. *Proceedings 14th real-time systems symposium*, pages 76–86, 1993.

[55] K.w. Lam, V. Lee, K.y. Lam, and S. Hung. Distributed real-time optimistic concurrency control protocol. *Proc. of the 4th Joint Workshop on Distributed and Real-Time Systems*, pages 122–125, 1996.

[56] E. Lawler, J. Lenstra, A. Rinnooy Kan, and D. Shmoys. Sequencing and scheduling: Algorithms and complexity. *Logistics of Production and Inventory*, Handbooks in Operations Research and Management Science(4):445–522, 1993.

[57] J. Lee and S. Son. Using dynamic adjustment of serialization order for real-time database systems. *Proc. of the 14th Real-Time Systems Symposium*, pages 66–75, 1993.

[58] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. *Proceedings of the IEEE Real-time Systems Symposium*, pages 166–171, 1989.

[59] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20:46–61, 1973.

[60] D. Luenberger. *Linear and nonlinear programming*. Addison-Wesley Publishing Company, Amsterdam, The Netherlands, 2 edition, 1989.

[61] D. Mandrioli and C. Ghezzi. *Theoretical Foundations of Computer Science*. John Wiley & Sons, New York, 1987.

[62] H. Mattsson, H. Nilsson, and C. Wikström. Mnesia - a distributed robust dbms for telecommunications applications. *Lecture Notes in Computer Science*, 1551:152–163, January 1999.

[63] S. Mauw. The formalization of message sequence charts. *Computer Networks and ISDN Systems*, (28):1643–1657, 1996.

[64] S. Mauw and M. Reniers. An algebraic semantics of basic message sequence charts. *The Computer Journal*, 37(4):269–277, 1994.

[65] D. Menasc'e and T. Nakanishi. Optimistic versus pessimistic concurrency control mechanisms in database management systems. *Information Systems*, 7(1):13–27, 1982.

[66] M. Michell. *An Introduction to Genetic Algorithms*. Complex Adaptive Systems. The MIT Press, Cambridge, Massachusetts, 1996.

[67] A. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environments*. PhD thesis, MIT, Cambridge, Mass., 1983.

[68] B. Moret and H. Shapiro. *Algorithms from P to NP*. The Benjamin/Cummings Publishing Company, Redwood city, 1991.

[69] R. Morris and W. Wong. Performance analysis of locking and optimistic concurrency control algorithms. *Performance Evaluation*, (5):105–118, 1985.

[70] Neuts. *Matrix-Geometric Solutions in Stochastic Models*. The Johns Hopkins University Press, Baltimore, 1981.

[71] H. Nilsson and C. Wikström. Mnesia - and industrial dbms with transactions, distribution and a logical query language. In *Proceedings of the Int. Symposium on Cooperative Database Systems for Advanced Applications*, 1996.

[72] Ö. Ulusoy and G. Belford. Real-time lock-based concurrency control in distributed database systems. *Proc. of the IEEE Conf. on Distributed Computing Systems*, pages 136–143, 1992.

[73] M. Özsu and P. Valduriez. *Principles of distributed database systems*. Prentice-Hall International Editions, Englewood Cliffs, 1991.

[74] S. Panwar, D. Towsley, and J. Wolf. Optimal scheduling policies for a class of queues with customer deadlines to the beginning of service. *Journal of the Association for Computing Machinery*, 35(4):832–844, October 1988.

[75] C. Papadimitriou. The serializability of concurrent database updates. *Journal of the Association for Computing Machinery*, 26(4):631–653, october 1979.

[76] M. Pinedo. *Scheduling, Theory, Algorithms and Systems*. Industrial and Systems Engineering. Prentice Hall, Englewood Cliffs, 1995.

[77] P. Pratt and J. Adamski. *Database Systems Management and Design*. Boyd & Fraser publishing company, 3 edition, 1994.

[78] K. Ramamritham and C. Pu. A formal characterization of epsilon serializability. *IEEE Trans. on Knowledge and Data Engineering*, 7(6):997–1007, 1995.

[79] M. Reniers. *Message Sequence Chart Syntax and Semantics*. PhD thesis, Eindhoven University of Technology, Den Dolech 2, Eindhoven, The Netherlands, 1999.

[80] S. Ross. *A first Course in Probability*. Macmillan Publishing Company, 1988.

[81] K. Salem, H. Garcia-Molina, and J. Shands. Altruistic locking. *ACM Transactions on Database Systems*, 19(1):117–165, 1994.

[82] S. Sassen. *Multi-server feedback queues for optimistic concurrency control*. PhD thesis, Eindhoven University of Technology, Den Dolech 2, Eindhoven, The Netherlands, 1998.

[83] S. Sassen and J. van der Wal. The response-time distribution in a real-time database with optimistic concurrency control and exponential execution times. In *Proceedings of the 15th International Teletraffic Congress*, pages 145–156, 1997.

[84] G. Schlageter. Optimistic methods for concurrency control in distributed database systems. *Proceedings of the 7th Conference on Very Large Databases*, pages 125–130, 1981.

[85] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[86] A. Sheth and M. Liu. Integrating locking and optimistic concurrency control in distributed database systems. *Proceedings of the IEEE Conf. on Distributed Computing Systems*, pages 89–99, 1986.

[87] J. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo. Implications of classical scheduling results for real-time systems. *Computer*, june 1995.

[88] T. Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, 1984.

[89] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases, Theory, Design and Implementation*. Database Systems Applications. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1993.

[90] A. Thomasian. Performance evaluation of centralized databases with static locking. *IEEE Transactions on Software Engineering*, pages 346–355, 1985.

[91] A. Thomasian. Performance analysis of locking policies with limited wait depth. *Performance Evaluation Review*, 20(1):115–127, 1992.

[92] A. Thomasian. Distributed optimistic concurrency control methods for high-performance transaction processing. *Transactions on Knowledge and Data Engineering*, 10(1):173–189, 1998.

[93] A. Thomasian and E. Rahm. A new distributed optimistic concurrency control method and a comparison of its performance with two-phase locking. *Proc. of the Int. Conf. on Distributed Computing Systems*, pages 294–301, 1990.

[94] H. Tijms. *Stochastic Models, an Algorithmic Approach*. John Wiley & Sons, Chichester, 1994.

[95] V. Timkovsky. Is a unit-time job shop not easier than identical parallel machines? *Discrete Applied Mathematics*, 85:149–162, 1998.

[96] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, Harcourt & Company, London, 1993.

[97] J. van de Pol, J. Hooman, and E. de Jong. Formal requirements specification for command and control systems. In *Proc. of the Conf. on Engineering of Computer Based Systems*, pages 37–44, 1998.

[98] P. van Gorp and A. Gouder de Beauregard. Resource assignment in hierarchical multiple resource systems. In *Proceedings of the Conf. on Parallel and Distributed Processing Techniques and Applications*, pages 1008–1014, 1998.

[99] J. Verhoorsel. *Pre-Run-Time Scheduling of Distributed Real-Time Systems.* PhD thesis, Eindhoven University of Technology, Den Dolech 2, Eindhoven, The Netherlands, 1995.

[100] K. Vidyasankar. Unified theory of database serializability. *Fundamenta Informaticae*, 14:147–183, 1991.

[101] R. Wolff. Poisson arrivals see time averages. *Operations Research*, 30:223–231, 1982.

[102] M. Xiong, J. Stankovic, K. Ramamritham, D. Towsley, and R. Sivasankaran. Maintaining temporal consistency: Issues and algorithms. *Proc. of the First International Workshop on Real-Time Databases: Issues and Applications*, pages 2–7, 1996.

[103] P. Yu and D. Dias. Analysis of hybrid concurrency control schemes for a high data contention environment. *IEEE Transactions on Software Engineering*, 18(2):118–129, 1992.

[104] P. Yu and D. Dias. Performance analysis of concurrency control using locking with deferred blocking. *IEEE Transactions on Software Engineering*, 19(10):982–996, 1993.

[105] P. Yu, D. Dias, and S. Lavenberg. On the analytical modeling of database concurrency control. *Journal of the ACM*, 40(4):831–872, 1993.

Index

2PC, *see* two phase commit
2PL, *see* two phase locking

abstraction mechanism, 33
access mode, 23
access operation, 23
access pattern, 34
addition operation, 23
age interval, 130
AGV, *see* automatic guided vehicle
allowed data access orders, 39
always-fresh, 129
announced, 20, 45
application precedence constraint extractor, 116
architecture
- centralized, 21
- distributed, 2, 22
- shared-memory, 2, 21, 89

arrival time, 26
assignment moment, 17
ATM, *see* automatic teller machine
automatic guided vehicle, 6
automatic teller machine, 6

basic priority inheritance, 10
belongs to, 77
BFS, *see* breadth first search
BPI, *see* basic priority inheritance
breadth first search, 119

C&C, *see* command and control system
command and control system, 7
commit rule, 74
communication, 22, 42
communication pattern, 42, 52
concurrency
- bounded, 40
- theoretically optimal, 40
- trivial, 41, 42, 51
- two phase locking, 40

concurrency control, 2
- optimistic, 11, 45, 51
- pessimistic, 10
- speculative, 11

conflict, 9
- shared, 69
- unshared, 69

conflict detector, 115
conflict probability, 44
conflict relation, 23
conflict serializability, 27
conflict-graph, 112
consistency, 1
- temporal, 9
  - data-based, 9
  - transaction-based, 9

constraint, 15
- fact, 17
- goal, 17

constraint satisfaction problem, 17, 118
context switch, 21
coordinator, 53, 68, 151
CSP, *see* constraint satisfaction problem
cycle remover, 119

data allocation, 8, 24
data item, 22
- non-temporal, 22
- temporal, 121

database, 1
- centralized, 9
- disk-based, 9
- distributed, 2, 9
- main-memory, 9
- real-time, 1, 2
- shared-memory, 2

database management system, 3
database queries, 3

dataset, 3, 22
DBMS, *see* database management system
deadline, 26
deadlock, 43, 52, 68
deadlock avoidance rule, 70
decision overhead, 41
DEDOS, *see* mine pump
delayed lock, 69
delayed unlock rule, 37
depth first search, 119
derived data, 121
derived item, 122
design problem, 15
DFS, *see* depth first search
dispatcher, 45
DOCC-BF, 68
DOCC-DASO, 65

earliest deadline first, 10
earliest start time, 26
EDF, *see* earliest deadline first
enabled, 72
enforce, 19
enforcing overhead, 41
epsilon serializability, 28
event, 56, 72
execution time, 25
execution units, 110

fait accompli, 20
FCFS, *see* first-come, first-served
FIFO, *see* first-in, first-out
financial trading application, 6
finish time, 25
first-come, first-served, 91
first-in, first-out, 71
fitness function, 118
flat database management system, 4
FTA, *see* financial trading application

genetic algorithms, 118
gradient projection method, 133
graph orientation problem, 114
growing phase, 151

haswritten, 129
HEP, *see* high energy physics
high energy physics, 5

instance, 122
  intermediate, 124
instrument, 6
interference relation, 23
internal computation, 42
invocation, 122
IPA, ii

job-shop problem, 118

life lock, 52, 81
local-write property, 35
lock, 149
  read, 149
  write, 149
locking phase, 151

manipulation options, 20
Markov model, 90
Markov property, 90
matrix geometric approach, 101
merge rule, 74
message sequence chart, 111
minimum laxity first, 94
mixed graph, 112
MLF, *see* minimum laxity first
Mnesia, 49
MSC, *see* message sequence chart

notification rule, 70, 74

objective function, 16, 30
observation, 15
OCC-light, 52
OCC-TI, 67
off-line, 19, 109
on-line, 19
owned by, 69
owns, 69

PABX, *see* private automatic branch exchange
participant, 53, 68, 151
PCP, *see* priority ceiling protocol
performance measure, 30
piggy backing, 42
pork bellies, 7
pre-emption point, 110
pre-runtime, 19

priority ceiling protocol, 10
private automatic branch exchange, 5
problem statement, 18
problem variable, 16
  fixed, 16
  free, 16
process farming, 92
processor speed, 21
pseudo-code, 145

query optimization, 4
query optimizer, 4

racing condition, 20
rate-monotonic, 10, 128
read operation, 23
reads-from relation, 28
real-time
  firm, 31, 90
  hard, 31, 109
  soft, 30
refresh transaction, 122
requirement
  firm real-time, 3
  hard real-time, 2
  soft real-time, 3
response time, 25
response time distribution, 90
restart rule, 52
reverse rule, 75
RMS, *see* rate-monotonic

schedule, 4, 19
  conflict serializable, 112
  serializable, 8
  two phase locking, 38
scheduler, 18
scheduler overhead, 41, 51
scheduler processor, 92
scheduling decision, 45
  off-line, 45
  on-line, 45
scheduling problem, 18
sensor data, 121
sensor item, 122
serializability, 9, 27
  conflict, 9
  epsilon, 9
  view, 9
shortest time to extinction, 94
shrinking phase, 151
simplex method, 133
single-write property, 35
single-writer property, 122
site, 24
SQSL-firm, 93
SQSL-MLF, 94
SQSL-soft, 92
start time, 25
state, 90
static locking, 45, 91
STE, *see* shortest time to extinction
steady state, 102
synchronization, 42
synchronization point, 20, 42, 51

t-maximal, 77
telecom, 5, 49
temporal consistency
  item-based, 121
  transaction-based, 121
temporal correctness, 125
temporal requirement
  absolute, 123
  relative, 124
Thomas' write rule, 82
thrashing, 37
time of measurement, 123
time of measurement function, 126
time precedence constraint, 110
timer, 93
tom-function, *see* time of measurement function
topological sort, 132
total conflict probability, 44
transaction, 1, 122
  characteristics, 24
  derive, 122
  sensor, 122
  user, 122
transaction manager, 4
transaction serializer, 110
transaction type, 122
trashing, 11
two phase commit, 42, 151

two phase locking, 10, 151

unannounced, 20
unlock, 149
unlocking phase, 151
user interface, 3
user request, 3

validation
  backward, 68
  forward, 68
view serializability, 27
view-Δ similarity, 9

wait-die, 150
work, 21, 25
worker processor, 92
worst-case execution time, 30
write operation, 23

雪梅, Xuemei, i

Summary

Database systems maintain a set of data and regulate access to that data. The task of the database scheduler is to determine the (partial) order in which transactions are allowed to read and write the database. This order has to satisfy certain correctness criteria. The most common requirement is serializability: the order of reading and writing is functionally equivalent to a sequential order.

This thesis investigates how the real-time performance of database schedulers can be optimized. Real-time performance is a function of the response times of transactions: the time between arrival of the transaction in the database, and the return of the result. We observe that the transaction response time depends on the environment in which the scheduler operates. Both the platform on which transactions are executed, and the applications that generate the transactions are important. For example, if the applications generate computationally demanding transactions, restarting a transaction substantially increases the transaction response time. Several existing schedulers restart the execution of a transaction to break deadlocks. This suggests that such schedulers are not suitable in the described environment. With a number of examples it is shown that a detailed description of the environment can be used to optimize the scheduler.

We determine what characteristics of a database scheduler most influence its performance. Traditionally, the emphasis is on efficient processor utilization. This is accomplished by maximizing the amount of parallelism allowed by the scheduler. We show that in several environments the overhead generated by the scheduler is more important than parallelism. In these environments maximizing the amount of parallelism will result in less performance gains than minimizing the scheduler overhead.

The sources of overhead are investigated. Since we limit ourselves to main-memory databases, three parts are distinguished: communication between computers, synchronization of local processes and internal scheduler computation. It seems that there is a relationship between scheduler overhead and the amount of parallelism that is allowed by the scheduler. Schedulers that offer a lot of parallelism have to synchronize more often between concurrently executing transactions to avoid non-serializable schedules, than schedulers that offer less parallelism. To decide whether transactions can execute concurrently, schedulers that offer a lot of parallelism require a large amount of information about transactions at an early stage.

These concepts are applied in a number of environments. The OCC-light scheduler is designed to function in a typical Telecom environment. Both synchronization overhead and communication overhead is minimized in the OCC-light scheduler. OCC-light is an adaptation of the classical optimistic concurrency control algorithm. The adaptations simplify the algorithm such that the overhead that is imposed on transactions is reduced. It turns out that OCC-light performs well in low-conflict environments.

The DOCC-BF scheduling algorithm is designed to function in high-conflict environments, by maximizing the offered parallelism. To achieve this, DOCC-BF uses the popular dynamic time-stamp mechanism that became famous in the centralized OCC-TI scheduler [57]. Since DOCC-BF operates on a distributed platform, OCC-TI has been extended with a distributed validation protocol. This pro-

tocol ensures that the validation is consistent, deadlock free and it minimizes the number of required sequential communications.

A totally different environment is given by the hard real-time scheduling system DEDOS [39]. Hard real-time scheduling systems are usually not flexible, and hence hard to program. To extend the expressiveness of the DEDOS programming environment, a pre-processor is created. This pre-processor adds database functionality to the DEDOS system. Several optimizations to the database protocols can be realized, since real-time functionality and reliability are already features of the DEDOS system. Hence, the database protocols can be streamlined considerably.

The thesis finishes with the introduction of temporal consistency for real-time databases. In several environments, databases are used to store information about continuously changing objects. This information becomes old and loses its value as time passes. The database should ensure that old data is refreshed when necessary. We introduce new concepts, give a specification method that can be used to design systems that use temporal consistency, and show that implementations of specified systems are possible.

Samenvatting

Database systemen beheren een set data en controleren de toegang tot die data. De taak van de database scheduler is de (partiële) volgorde te bepalen waarin de transacties de database mogen lezen en schrijven. Die volgorde moet aan bepaalde consistentie-eisen voldoen. De meest gebruikelijke eis is serialiseerbaarheid: de volgorde van lezen en schrijven is functioneel equivalent met een sequentiële volgorde.

In dit proefschrift wordt onderzocht hoe de real-time performance van database schedulers kan worden geoptimaliseerd. Dit houdt in dat de reactietijd van transacties (de tijd tussen het arriveren van de transactie in de database, en het teruggeven van het resultaat) aan bepaalde voorwaarden moet voldoen. We observeren dat de performance van een scheduler voor een belangrijk deel afhangt van de omgeving van de scheduler. Zowel het platform waarop de scheduler de transacties uitvoert, als de applicaties die transacties genereren zijn van belang. Bijvoorbeeld, als applicaties zeer rekenintensieve transacties genereren, moet de scheduler ervoor zorgen dat de executie van een transactie altijd succesvol is. Verscheidene schedulers herstarten de executie van een transactie als geen geschikt schedule gevonden kan worden. Met behulp van een aantal voorbeelden laten we zien dat een nauwkeurige beschrijving van de omgeving van de scheduler aanknopingspunten biedt die gebruikt kunnen worden om de scheduler te optimaliseren.

Daarnaast wordt onderzocht welke kenmerken van een database scheduler de meeste invloed hebben op de performance. Traditioneel ligt de nadruk sterk op efficiënt processor-gebruik. Dit wordt bereikt door de hoeveelheid parallellisme die wordt toegestaan door de scheduler te maximaliseren. We laten zien dat in bepaalde omgevingen de overhead van de scheduler meer van belang is dan parallellisme. In deze omgevingen levert maximalisering van de hoeveelheid geboden parallellisme minder performance-winst op dan minimalisering van de scheduler overhead.

We onderzoeken waaruit scheduler overhead bestaat. Omdat we ons beperken tot main-memory databases, onderscheiden we drie onderdelen: communicatie tussen computers, synchronisatie tussen locale processen en interne scheduler berekeningen. Het blijkt dat er een relatie is tussen scheduler overhead en geboden parallellisme. Schedulers die veel parallellisme bieden moeten vaker synchroniseren tussen gelijktijdig executerende transacties dan schedulers die weinig parallellisme bieden. Dit is noodzakelijk om niet-serialiseerbare executie-volgordes te voorkomen. Om te beslissen of transacties gelijktijdig kunnen executeren moet schedulers die veel parallellisme bieden, relatief veel informatie over transacties hebben. Daarnaast is het belangrijk dat deze informatie op een vroeg tijdstip beschikbaar komt, zodat de scheduler voldoende tijd heeft om de juiste beslissingen te nemen.

Deze concepten worden in een aantal omgevingen toegepast. Na onderzoek van een typische Telecom omgeving ontwerpen we de OCC-light scheduler. Communicatie- en synchronisatie overhead zijn geminimaliseerd in de OCC-light scheduler. OCC-light is een aanpassing van het klassieke Optimistisch Concurrency Control mechanisme. De aanpassingen simplificeren het algoritme zodanig dat de overhead op transacties wordt teruggebracht. Hierdoor presteert OCC-light goed in een conflict-arme omgeving met kort executerende transacties.

Het DOCC-BF scheduling mechanisme is ontworpen om in conflict-rijke omgevingen goede prestaties te leveren. Daarom biedt DOCC-BF veel parallellisme. Hiervoor maakt DOCC-BF gebruik van het populaire dynamische time-stamp mechanisme dat bekend werd door OCC-TI (zie [57]). Aangezien DOCC-BF werkt op een platform dat bestaat uit meerdere machines, is OCC-TI uitgebreid met een gedistribueerd validatie-protocol. Dit protocol zorgt voor consistentie van de validatie, deadlock preventie en minimaliseert het aantal communicatie stappen dat nodig is.

Weer een geheel andere omgeving vormt het hard real-time scheduling systeem DEDOS [39]. Hard real-time schedule systemen zijn niet flexibel en daarom moeilijk te programmeren. Om de uitdrukkingskracht van het DEDOS scheduling systeem te vergroten construeren we een pre-processor voor de DEDOS scheduler. De pre-processor voegt database functionaliteit toe aan het scheduling systeem. Aanzienlijke optimalisaties van de database protocollen kunnen gerealiseerd worden daar het hard real-time scheduling systeem reeds real-time en reliability eigenschappen heeft.

Het proefschrift sluit af met de introductie van temporele consistentie. In veel gevallen worden databases gebruikt om informatie over een continu veranderende omgeving op te slaan. Deze informatie wordt oud en verliest zijn waarde naarmate de tijd verstrijkt. Het is de taak van de database om ervoor te zorgen dat oude informatie ververst wordt. We introduceren nieuwe concepten, geven een specificatie-methode en laten zien dat implementaties van zodanig gespecificeerde systemen mogelijk zijn.

Curriculum Vitæ

Maarten Bodlaender werd op 24 mei 1970 geboren in Bennekom, gemeente Ede. Reeds op jeugdige leeftijd begon hij aan zijn informatica-opleiding, zodat hij naast de Atheneum B opleiding op het Marnix college te Ede, 's avonds zijn eerste Basic programmaatjes schreef.

Na de middelbare school volgde informatica aan de rijksuniversiteit Utrecht. In eerste instantie specialiseerde Maarten zich in kunstmatige intelligentie, expertsystemen en neurale netwerken. Uiteindelijk maakte hij toch de ommezwaai naar de afstudeerrichting algoritmiek, met als specialisatie gedistribueerde systemen. Onder begeleiding van afstudeerdocent dr. Gerard Tel verrichtte hij onderzoek naar het uitbreiden van de toepassingsmogelijkheden van wave algoritmen. Een wave algoritme is in essentie een communicatie-protocol in een netwerk van computers. Zijn afstuderen werd in augustus 1994 voltooid.

Van oktober 1994 tot februari 1999 verrichtte hij zijn promotie-onderzoek aan de Technische Universiteit Eindhoven, onder begeleiding van dr. Peter van der Stok. De titel van het STW-project waar dit onderzoek deel van uitmaakte "Construction and performance of real-time transactions" geeft een aardige indicatie van het onderzoek dat verricht werd. Maarten verdeelde zijn tijd tussen constructie van real-time distributed database schedulers, en het analyseren of meten van hun performance.

Tegenwoordig werkt Maarten bij het Philips natuurkundig laboratorium, in Eindhoven.

Titles in the IPA Dissertation Series

The State Operator in Process Algebra
J. O. Blanco
Faculty of Mathematics and Computing Science, TUE, 1996-1

Transformational Development of Data-Parallel Algorithms
A. M. Geerling
Faculty of Mathematics and Computer Science, KUN, 1996-2

Interactive Functional Programs: Models, Methods, and Implementation
P. M. Achten
Faculty of Mathematics and Computer Science, KUN, 1996-3

Parallel Local Search
M. G. A. Verhoeven
Faculty of Mathematics and Computing Science, TUE, 1996-4

The Implementation of Functional Languages on Parallel Machines with Distrib. Memory
M. H. G. K. Kesseler
Faculty of Mathematics and Computer Science, KUN, 1996-5

Distributed Algorithms for Hard Real-Time Systems
D. Alstein
Faculty of Mathematics and Computing Science, TUE, 1996-6

Communication, Synchronization, and Fault-Tolerance
J. H. Hoepman
Faculty of Mathematics and Computer Science, UvA, 1996-7

Reductivity Arguments and Program Construction
H. Doornbos
Faculty of Mathematics and Computing Science, TUE, 1996-8

Functorial Operational Semantics and its Denotational Dual
D. Turi
Faculty of Mathematics and Computer Science, VUA, 1996-9

Single-Rail Handshake Circuits
A. M. G. Peeters
Faculty of Mathematics and Computing Science, TUE, 1996-10

A Systems Engineering Specification Formalism
N. W. A. Arends
Faculty of Mechanical Engineering, TUE, 1996-11

Normalisation in Lambda Calculus and its Relation to Type Inference
P. Severi de Santiago
Faculty of Mathematics and Computing Science, TUE, 1996-12

Abstract Interpretation and Partition Refinement for Model Checking
D. R. Dams
Faculty of Mathematics and Computing Science, TUE, 1996-13

Topological Dualities in Semantics
M. M. Bonsangue
Faculty of Mathematics and Computer Science, VUA, 1996-14

Algorithms for Graphs of Small Treewidth
B. L. E. de Fluiter
Faculty of Mathematics and Computer Science, UU, 1997-01

Process-algebraic Transformations in Context
W. T. M. Kars
Faculty of Computer Science, UT, 1997-02

A Generic Theory of Data Types
P. F. Hoogendijk
Faculty of Mathematics and Computing Science, TUE, 1997-03

The Evolution of Type Theory in Logic and Mathematics
T. D. L. Laan
Faculty of Mathematics and Computing Science, TUE, 1997-04

Preservation of Termination for Explicit Substitution
C. J. Bloo
Faculty of Mathematics and Computing Science, TUE, 1997-05

Discrete-Time Process Algebra
J. J. Vereijken
Faculty of Mathematics and Computing Science, TUE, 1997-06

A Functional Approach to Syntax and Typing
F. A. M. van den Beuken
Faculty of Mathematics and Informatics, KUN, 1997-07

Ins and Outs in Refusal Testing
A.W. Heerink
Faculty of Computer Science, UT, 1998-01

A Discrete-Event Simulator for Systems Engineering
G. Naumoski and W. Alberts
Faculty of Mechanical Engineering, TUE, 1998-02

Scheduling with Communication for Multiprocessor Computation
J. Verriet
Faculty of Mathematics and Computer Science, UU, 1998-03

An Asynchronous Low-Power 80C51 Microcontroller
J. S. H. van Gageldonk
Faculty of Mathematics and Computing Science, TUE, 1998-04

In Terms of Nets: System Design with Petri Nets and Process Algebra
A. A. Basten
Faculty of Mathematics and Computing Science, TUE, 1998-05

Inductive Datatypes with Laws and Subtyping – A Relational Model
E. Voermans
Faculty of Mathematics and Computing Science, TUE, 1999-01

Towards Probabilistic Unification-based Parsing
H. ter Doest
Faculty of Computer Science, UT, 1999-02

Algorithms for the Simulation of Surface Processes
J.P.L. Segers
Faculty of Mathematics and Computing Science, TUE, 1999-03

Stellingen

bij het proefschrift

Scheduler Optimization in Real-Time Distributed Databases

van

MAARTEN PETER BODLAENDER

—9—

Nederlands wordt steeds minder geschikt om als voertaal te dienen in een wetenschappelijke omgeving. De eis dat een proefschrift een Nederlandse samenvatting bevat dient daarom te vervallen.

—10—

In veel communicatie media is het medium zelf het onderwerp van een significant percentage van alle gecommuniceerde berichten. Deze observatie kan gebruikt worden om dit soort berichten effectief te comprimeren.

—11—

De primaire bron van frustratie is onvermogen.

Corollary: als je niet gefrustreerd bent, kan je kennelijk nog beter presteren.

—12—

De verbetering van communicatiemedia heeft het aantal tragische liefdes sterk doen toenemen.

ISBN 90 386 0681 8