Maintaining Data Consistency in Embedded Databases for Vehicular Systems

by

Thomas Gustafsson

Submitted to the School of Engineering at Linköping University in partial fulfilment of the requirements for the degree of Licentiate of Engineering

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköping 2004
ABSTRACT

The amount of data handled by real-time and embedded applications is increasing. This calls for data-centric approaches when designing embedded systems, where data and its meta-information (e.g., temporal correctness requirements) are stored centrally. The focus of this thesis is on efficient data management, especially maintaining data freshness and guaranteeing correct age on data.

The contributions of our research are updating algorithms and concurrency control algorithms using data similarity. The updating algorithms keep data items up-to-date and can adapt the number of updates of data items to state changes in the external environment. Further, the updating algorithms can be extended with a relevance check allowing for skipping of unnecessary calculations. The adaptability and skipping of updates have positive effects on the CPU utilization, and freed CPU resources can be reallocated to, e.g., more extensive diagnosis of the system. The proposed multiversion concurrency control algorithms guarantee calculations reading data that is correlated in time.

Performance evaluations show that updating algorithms with a relevance check give significantly better performance compared to well-established updating approaches, i.e., the applications use more fresh data and are able to complete more tasks in time. The proposed multiversion concurrency control algorithms perform better than HP2PL and OCC and can at the same time guarantee correct age on data items, which HP2PL and OCC cannot guarantee. Thus, from the perspective of the application, more precise data is used to achieve a higher data quality overall, while the number of updates is reduced.

This work has been supported by (ISIS) Information Systems for Industrial Control and Supervision, and by (CUGS) the national graduate school in computer science.
Acknowledgement

Thanks go to my supervisor and mentor Jörgen Hansson for his guidance in the dark realms of research. With nice discussions and lots of comments on my research he is enlightening a path where I can safely walk toward the ultimate goal of my education—becoming a researcher. His ever-lasting optimism that he is eager to share is very much appreciated. Thanks also go to Simin Nadjm-Tehrani for always being supportive and giving feedback on this thesis.

I would like to thank Anders Göras, Gunnar Jansson, and Thomas Lindberg at Mecel AB, and Jouko Gäddevik, Sven-Anders Melin, and David Holmgren at Fiat-GM Powertrain for valuable input to my research and help with practical matters. The research project would not have reached its current state without help from the master thesis students Martin Jinnelöv, Marcus Eriksson, and Hugo Hallqvist. Thank you.

Last but not least I want to say to dear friends and my family that you are all contributing to any of my successes. Thanks to all RTS LAB members for the nice and friendly atmosphere, and thanks in particular to Mehdi Amirijoo and Aleksandra Tešanović for walking the research path together with me.

Thomas Gustafsson
CONTENTS

Contents

1 Introduction 1
   1.1 Motivation ............................................. 1
   1.2 Contributions ........................................... 3
   1.3 Thesis Outline ......................................... 4

2 Background 5
   2.1 Real-Time System ....................................... 5
       2.1.1 Scheduling ........................................ 7
       2.1.2 Precedence Constraints ......................... 9
       2.1.3 Servers ........................................... 10
   2.2 Databases .............................................. 10
       2.2.1 Transactions ..................................... 11
       2.2.2 Consistency ...................................... 12
       2.2.3 Updating Algorithms ............................ 17
   2.3 Electronic Engine Control Unit ........................ 18
       2.3.1 Data and Transaction Model ................... 20
   2.4 Concurrency Control ................................... 20
       2.4.1 Serializability ................................ 20
       2.4.2 Concurrency Control Algorithms ............. 25
   2.5 Checksums and Cyclic Redundancy Checks .......... 32

3 Problem Formulation 35
   3.1 Software Development and Data Management .......... 35
   3.2 Notations and Assumptions ............................ 36
   3.3 Problem Formulation .................................. 38

4 Updating and Concurrency Control Algorithms 41
   4.1 Database System Prototype ........................... 42
## CONTENTS

4.2 Data Freshness ............................................. 43  
   4.2.1 Example of Data Freshness in Value Domain .... 46  
   4.2.2 Marking of Changed Data Items ................. 47  
4.3 On-Demand Updating Algorithms in Time Domain .... 51  
4.4 ODDFT Updating Algorithm ................................. 52  
4.5 ODBFT Updating Algorithm ................................. 57  
4.6 ODTB Updating Algorithm With Relevance Check ..... 60  
4.7 ODDFT_C Updating Algorithm With Relevance Check .. 64  
4.8 ODKB_C Updating Algorithm With Relevance Check .. 64  
4.9 Supporting Mechanisms and Algorithms .................. 65  
   4.9.1 BeginTrans ........................................ 65  
   4.9.2 ExecTrans ......................................... 65  
   4.9.3 AssignPrio ......................................... 66  
4.10 Multiversion Concurrency Control With Similarity ... 66  
   4.10.1 Relative Consistency ............................... 68  
   4.10.2 MVTO with Similarity .............................. 70  
4.11 Implementation of MVTO-S ................................. 74  
   4.11.1 MVTO-UV .......................................... 74  
   4.11.2 MVTO-UP .......................................... 74  
   4.11.3 MVTO-CHECKSUM ................................... 77  
4.12 Single-version Concurrency Control With Similarity .. 79  
4.13 Implementation of Database System ..................... 81  
   4.13.1 Implementation Details of Concurrency Control .. 83

5 Performance Evaluations ...................................... 85  
5.1 Methodology of Experiments ............................... 85  
5.2 RADEx++ Experiments .................................... 87  
   5.2.1 Simulator Setup ................................... 87  
   5.2.2 Experiment 1a: Consistency and Throughput With No Relevance Check ......................... 90  
   5.2.3 Experiment 1b: Deriving Only Actuator User Transactions ........................................... 94  
   5.2.4 Experiment 1c: Comparison of Using Binary Change Flag or pa Timestamp ....................... 96  
   5.2.5 Experiment 1d: Transient and Steady States .... 97  
   5.2.6 Experiment 1e: Effects of Blocking Factor .... 98  
   5.2.7 Experiment 1f: Varying Weights .................. 103  
   5.2.8 Experiment 2a: Consistency and Throughput With Relevance Check ............................ 105
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.9</td>
<td>Experiment 2b: Effects of Blocking Factors and Only Deriving Actuator Transactions</td>
<td>106</td>
</tr>
<tr>
<td>5.3</td>
<td>Database Implementation in EECU</td>
<td>110</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Simulator Setup</td>
<td>110</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Experiment 3: Transient and Steady States in EECU</td>
<td>113</td>
</tr>
<tr>
<td>5.4</td>
<td>Database Implementation in PC</td>
<td>114</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Simulator Setup</td>
<td>116</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Experiment 4a: Committed User Transactions</td>
<td>117</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Experiment 4b: Memory Pool Size</td>
<td>122</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Experiment 4c: Priority Levels</td>
<td>123</td>
</tr>
<tr>
<td>5.4.5</td>
<td>Experiment 4d: Overhead</td>
<td>123</td>
</tr>
<tr>
<td>5.4.6</td>
<td>Experiment 4e: Low Priority</td>
<td>124</td>
</tr>
<tr>
<td>5.4.7</td>
<td>Experiment 4f: Transient State</td>
<td>126</td>
</tr>
<tr>
<td>5.5</td>
<td>Observations</td>
<td>126</td>
</tr>
<tr>
<td>6</td>
<td>Related Work</td>
<td>129</td>
</tr>
<tr>
<td>6.1</td>
<td>Updating Algorithms and Data Freshness</td>
<td>129</td>
</tr>
<tr>
<td>6.2</td>
<td>Concurrency Control</td>
<td>133</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions and Future Work</td>
<td>137</td>
</tr>
<tr>
<td>7.1</td>
<td>Conclusions</td>
<td>137</td>
</tr>
<tr>
<td>7.2</td>
<td>Future Work</td>
<td>138</td>
</tr>
<tr>
<td>A</td>
<td>Graphs with Confidence Intervals</td>
<td>149</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Hard, soft and firm real-time tasks. ........................... 7
2.2 A database system. ............................................. 12
2.3 An example of a similarity relation for temperature measurements. ........................................... 16
2.4 The software in the EECU is layered. Black boxes represent tasks, labeled boxes represent data items, and arrows indicate inter-task communication. ........................................... 19
2.5 Data dependency graph in the EECU. ......................... 21
2.6 Example of ghost update. ........................................ 22
2.7 The pseudo-code of an algorithm producing a CRC. ....... 33

3.1 User transactions. The user transaction at the top derives a value, and the user transaction at the bottom sends a derived value to an actuator. ........................................... 38

4.1 Database system that addresses requirement R3. ............ 42
4.2 Two similarity functions mapping to fixed validity intervals and flexible validity intervals. ............................. 44
4.3 A graph that gives the worst-case running time of algorithms OD, ODO, and ODKB. ................................. 54
4.4 The ODDFT algorithm. ........................................... 55
4.5 A schedule of updates generated by ODDFT. .................. 56
4.6 The ODBFT algorithm. ........................................... 58
4.7 The BuildPreGenSchedule algorithm. .......................... 61
4.8 Top-Bottom relevance check algorithm (pseudo-code). .... 62
4.9 Pregenerated schedule by BuildPreGenSchedule for G in figure 2.5. ........................................... 62
4.10 Pseudo-code of the BeginTrans algorithm. ................... 65
4.11 Pseudo-code of the ExecTrans algorithm. ................. 67
4.12 The AssignPriority algorithm. .......................... 68
4.13 Example of error function. .............................. 68
4.14 Example of the write timestamp of versions using MVTO-S. 72
4.15 Determine if two versions are similar by using function f on 
every pair of parents. .................................... 75
4.16 UT τ derives d_{36}. Algorithm CheckSimilarity investigates if 
the existing version of d_{36} is similar to the one τ derives. 75
4.17 CheckSimilarity algorithm. ............................... 76
4.18 Sizes of versions for MVTO-UV and MVTO-UP. ............ 78
4.19 OCC-S validation phase. .................................. 81
4.20 Example of a transaction in the EECU software (C-code). 82
4.21 Implementation details of concurrency control algorithms. 84

5.1 Experiment 1a: Consistency and throughput of UTs (confidence 
intervals are presented in figure A.1). ...................... 91
5.2 Experiment 1a: Ratio of valid committed UTs and total num-
ber of ......................................................... 92
5.3 Experiment 1a: Effects of measuring staleness of data items 
at deadline of UT (confidence intervals are presented in figure 
A.2). ......................................................... 93
5.4 Experiment 1b: Consistency and throughput of UTs that 
only derive leaf nodes. ..................................... 95
5.5 Experiment 1b: Number of generated triggered updates where 
UTs are AUT (confidence intervals are presented in figure A.3). 96
5.6 Experiment 1c: Number of valid committed UTs using either 
the pa timestamp or a change flag to indicate potentially af-
fected data items (confidence intervals are presented in figure 
A.4). ......................................................... 97
5.7 Experiment 1d: Simulation of transient and steady states of 
a system. .................................................... 99
5.8 Experiment 1e: The effect of blockingf on the number of 
valid committed UTs where a UT randomly derives a data 
item. ......................................................... 101
5.9 Experiment 1e: Statistical data for ODDFT using two dif-
f erent blocking factors. .................................... 102
5.10 Experiment 1e: The effect of blockingf on the number of 
valid committed UTs where a UT derives leaf nodes. ....... 103
5.11 Experiment 1f: Varying weights. .......................... 104
5.12 Experiment 2a: Consistency and throughput of UTs with no relevance check on ODDFT and ODKB_V. 107
5.13 Experiment 2a: Consistency and throughput of UTs with a relevance check on triggered updates on ODDFT (confidence intervals are presented in figure A.5.) 108
5.14 Experiment 2b: Performance for updating algorithms that have the possibility to skip updates. 109
5.15 Experiment 2b: Performance for updating algorithms that has the possibility to skip updates (confidence intervals are presented in figure A.6). 111
5.16 Experiment 2b: Performance metrics for ODDFT_C with blocking = 1.5 and ODTB. 112
5.17 Overview of the EECU and engine simulator. 113
5.18 Experiment 3: Performance results of a database implementation in an EECU. The performance metric is the number of cumulative recalculations. 115
5.19 Experiment 4a: Number of UTs committing before their deadlines using single- and multiversion concurrency control algorithms. 118
5.20 Experiment 4a: Number of transactions that can be skipped using ODTB in conjunction with the concurrency control algorithms. 119
5.21 Experiment 4a: A comparison of single-version concurrency control algorithms enforcing relative consistency and multiversion concurrency control algorithms (confidence intervals are presented in figure A.8). 119
5.22 Experiment 4a: Number of restarts of transactions for the concurrency control algorithms. 120
5.23 Experiment 4a: The similarity-aware multiversion concurrency control algorithms using fixed validity intervals. 121
5.24 Experiment 4b: The performance when using different sizes of the memory pool. 123
5.25 Experiment 4c: Number of restarts at different priority levels at an arrival rate of 40 UTs per second. Level one represents the highest priority. 124
5.26 Experiment 4d: An experiment executed on fast and slow computers. 125
5.27 Experiment 4e: Performance of an additional task with the lowest priority issuing a transaction reading 25 data items.  . 127
5.28 Experiment 4f: Every write of a data item changes its value with 450 (confidence intervals are presented in figure A.9).  . 128

A.1 Experiment 1a: Consistency and throughput of UTs.  . . . . 150
A.2 Experiment 1a: Effects of measuring staleness of data items at deadline of UT.  . . . . . . . . . . . . . . . . . . . . . . 151
A.3 Experiment 1b: Number of generated triggered updates where UTs are AUT.  . . . . . . . . . . . . . . . . . . . . . . . 152
A.4 Experiment 1c: Number of valid committed UTs using either the pa timestamp or a change flag to indicate potentially affected data items.  . . . . . . . . . . . . . . . . . . . . . . . . . . 152
A.5 Experiment 2a: Consistency and throughput of UTs with a relevance check on triggered updates using ODDFT.  . . . 153
A.6 Experiment 2b: Performance for updating algorithms that has the possibility to skip updates.  . . . . . . . . . . . . 154
A.7 Experiment 4a: Number of UTs committing before their deadlines using single- and multiversion concurrency control algorithms.  . . . . . . . . . . . . . . . . . . . . . . . . . . . . 155
A.8 Experiment 4a: A comparison of single-version concurrency control algorithms enforcing relative consistency and multiversion concurrency control algorithms.  . . . . . . . . . . 156
A.9 Experiment 4f: Every write of a data item changes its value with 450.  . . . . . . . . . . . . . . . . . . . . . . . . . . 156
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Compatibility matrix for 2V2PL.</td>
<td>31</td>
</tr>
<tr>
<td>3.1</td>
<td>Transaction types that a real-time embedded database system should have support for.</td>
<td>37</td>
</tr>
<tr>
<td>4.1</td>
<td>A summary of updating algorithms.</td>
<td>53</td>
</tr>
<tr>
<td>4.2</td>
<td>Investigation of the robustness of Fletcher's checksum algorithm.</td>
<td>79</td>
</tr>
<tr>
<td>4.3</td>
<td>Investigation of the robustness of CRC-32.</td>
<td>79</td>
</tr>
<tr>
<td>5.1</td>
<td>Parameter settings for database simulator.</td>
<td>89</td>
</tr>
<tr>
<td>5.2</td>
<td>Experiment 1d: Statistical data from transient and steady state simulation.</td>
<td>98</td>
</tr>
<tr>
<td>5.3</td>
<td>Experiment 4a: The number of times the checksum check misses to detect similar values compared to using values in MVTO-UV.</td>
<td>122</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This chapter gives an introduction to the research area of this thesis. The work is part of the project entitled “Real-Time Databases for Engine Control in Automobiles”. Two industrial partners are taking part in this research project: Mecel AB and the SAAB division of Fiat-GM Powertrain. Both companies are working with engine control software for vehicles in general and cars in particular. This thesis addresses data management issues that the industrial partners have identified as challenges during the course of maintaining and developing engine control software.

Section 1.1 gives a motivation for doing the research by introducing data management problems that the industry is facing. Section 1.2 summarizes the research contributions achieved in this research project, and, finally, section 1.3 outlines the thesis.

1.1 Motivation

Computing units are used to control several functional parts of cars, e.g., engine, breaks, and climate control. Every such unit is denoted an electronic control unit (ECU). The ECU software is becoming more complex due to increasing functionality, which is possible because of additional available resources such as memory and computing power. For instance, calculations in the engine electronic control unit (EECU) software have time constraints, which means that the calculations should be finished within given time frames. Thus, the EECU is a real-time system. The control-theoretic aspects of controlling the engine are well understood and implemented as event-based sporadic tasks with soft real-time requirements. However, the
1.1. Motivation

Software in the EECU is complex and consists of approximately 100,000 lines of C code. One reason for this complexity is law regulations put on the car industry to extensively diagnose the system; the detection of a malfunctioning component needs to be done within a certain time after the component breaks [49]. The diagnosis is a large part of the software, up to half of it, and many data items are introduced in the diagnosis. Moreover, the software has a long life cycle, perhaps a decade, and several programmers are involved in maintaining the software. The industrial partners have identified problems with their current approach of developing embedded software. These include:

- Managing data items since they are partitioned into several different data areas—global and application-specific\(^1\). This makes it difficult for programmers to keep track of what data items exist. Also, a data item can accidentally exist in several data areas simultaneously. This increases both CPU and memory usage.

- Making sure data is updated such that accurate calculations of control variables and diagnosis of the system can be done.

- Using CPU and memory resources efficiently allowing to choose cheaper devices which cuts costs for the car manufacturers.

Data freshness in an ECU is currently guaranteed by updating data items with fixed frequencies. Previous work proposes ways of determining fixed updating frequencies on data items to fulfill freshness requirements [30, 36, 44, 71]. This means that a data item is recalculated when it is about to be stale, even though the new value of the data item is exactly the same as before. Hence, the recalculation is essentially unnecessary and resources are not utilized efficiently.

Databases are used in many different applications to organize and store data [8]. They consist of software modules that take care of issues related to application-specific data (see section 2.2, Databases, for more details), e.g., transaction management and secondary memory storage. The benefits from using a database are clear; by keeping data in one place it is more easily accessible to many users, and the data is easier maintained compared to if it was partitioned, each partition residing on one isolated computer.

\(^1\)In the context of an EECU software, an application is a collection of tasks responsible for one main part of the engine, e.g., control of fuel and the related problem of knocking.
Queries can be issued to retrieve data to analyze. It is the task of the database to parse the query and return a data set containing the requested data. Databases are often associated with the storage of thousands of Mb of data and advanced query languages such as SQL. The distinguished feature of a database is the addition and deletion of data items, i.e., the data set is dynamic. However, the benefits of a database, especially as a complete system to maintain data, can of course be applied to systems with a fixed data set. The goal of this thesis is to investigate how a database can give advantages in maintainability and resource utilization in real-time applications. An engine electronic control unit provided by the industrial partners is used as a real-life example. In this research project we assume that no secondary memory is available and query languages are not needed. Thus, the database in this research project should be seen as a data repository since the requirements on data storage is quite different from ordinary large-scale and general databases.

An important part of a database is concurrency control. In fact, the concurrency control affects the relative consistency of data used by calculations. Relative consistency means that data read by calculations should be derived from the same state of the external environment. Calculations can interrupt each other giving problems in maintaining relative consistency. Consider a calculation that has only read a subset of the required data items. Another calculation preempts the first and in the interim values of the read set of the first calculation change. Now when the first calculation continues and reads the remaining data items, the read data items are derived based on different states in the environment. Hence, the values are not relatively consistent. This thesis contains an evaluation on how concurrency control algorithms affect relative consistency and presents new concurrency control algorithms that can guarantee the usage of relatively consistent values in calculations.

1.2 Contributions

The contributions of the research project are:

- Novel updating algorithms that are used to maintain data freshness. They have the ability to automatically adapt the number of needed updates to the current state of the external environment.

- Validity and consistency of produced values of data items are inves-
tigated. It is found that multiversion concurrency control algorithms can, together with an updating algorithm, guarantee relative consistency. Using an updating algorithm that can skip unnecessary calculations the proposed multiversion concurrency control algorithms perform better than traditional single-version concurrency control algorithms that are not guaranteeing relative consistency. Traditional single-version concurrency control algorithms can guarantee relative consistency by introducing restart of transactions. Our experiments show that the multiversion concurrency control algorithms provide considerably better performance than the single-version algorithms.

1.3 Thesis Outline

Chapter 2, Background, introduces real-time systems and scheduling of real-time tasks, database systems and their modules, concurrency control algorithms, and serializability and similarity as correctness criterion. An EECU is also described in this chapter.

Chapter 3, Problem Formulation, presents the difficulties the industrial partners have found in developing large and complex ECU software. Notation and assumptions of the system used throughout this research project are presented. Finally, the problem formulation of this research project is stated.

Chapter 4, Updating and Concurrency Control Algorithms, starts by introducing data freshness in the value domain. This kind of data freshness is then used in updating algorithms whose purpose are to make sure the value of data items is up-to-date when they are used. Novel multiversion concurrency control algorithms that guarantee relative consistency are also presented in this chapter.

Chapter 5, Performance Evaluations, shows how updating algorithms and concurrency control algorithms perform in a setting similar to a real-world system. This is achieved by using a discrete event simulator, a simulated environment running on a real-time operating system, and a database system with updating algorithms running on a subset of data in an EECU.

Chapter 6, Related Work, positions the work and the results reached in this research project to previous work done in the area of real-time databases.

Finally, chapter 7, Conclusions and Future Work, concludes this thesis and gives directions for future work.
Chapter 2

Background

This chapter gives preliminaries in the areas of real-time systems and databases theory in sections 2.1 and 2.2. In section 2.3 a description is given of an electronic engine control unit (EECU) that is used as a real-life example of a real-time embedded system throughout the thesis. The construction of data updating algorithms described later in this thesis is based on an analysis of the software in an EECU. Concurrency control algorithms are presented in section 2.4. Checksums and cyclic redundancy checks are described in section 2.5.

2.1 Real-Time System

A real-time system consists of tasks, where some/all have time constraints on their execution. It is important to finish a task with a time constraint before its deadline, i.e., it is important to react to an event in the environment before a predefined time. A task is a sequence of instructions executed by a CPU in the system. In this thesis, only single-CPU systems are considered. Tasks can be either periodic, sporadic, or aperiodic [37]. A periodic task is periodically made active, e.g., to monitor a sensor at regular intervals. A sporadic task can be made active at any time after a time duration since the last execution of the task, i.e., sporadic tasks have a minimum interarrival rate. An example of a sporadic task in the EECU software is the ignition time of the spark plug to fire the air fuel mixture in a combustion engine. The shortest time between two invocations of this task for a particular cylinder is 80 ms since, assuming the engine cannot

\footnote{The term task and real-time task are used interchangeably in this thesis.}
run faster than 6000 rpm and has 4 cylinders, the ignition only occurs every second revolution. Aperiodic tasks, in contrast to sporadic tasks, have no limits on how often they can be made active. Hence, both sporadic and aperiodic tasks are invoked occasionally and for sporadic tasks we know the smallest possible amount of time between two invocations.

The correct behavior of a real-time system depends not only on the values produced by tasks but also on the time when the values are produced [11]. A value that is produced too late can be useless to the system or even have dangerous consequences. A task is, thus, associated with a deadline relative to the start time of the task. Note that a task has an arrival time when the system is notified of the existence of the ready task, and a start time when the task starts to execute. Tasks are generally divided into three types:

- **Hard real-time tasks.** The missing of a deadline of a task with a hard requirement on meeting the deadline has fatal consequences on the environment under control. For instance, the landing gear of an aeroplane needs to be ejected at a specific altitude in order for the pilot to be able to complete the landing.

- **Soft real-time tasks.** If the deadline is missed the environment is not severely damaged and the overall system behavior is not at risk but the performance of the system degrades.

- **Firm real-time tasks.** The deadline is soft, i.e., if the deadline is missed it does not result in any damages to the environment, but the value the task produces has no meaning after the deadline of the task. Thus, tasks that do not complete in time should be aborted as late results are of no use.

The deadlines of tasks can be modeled by utility functions. The completion of a task gives a utility to the system. These three types of real-time tasks are shown in figure 2.1. A real-time system can be seen as optimizing the utility the system receives from executing tasks. Thus, every task gives a value to the system, as depicted in 2.1. For instance, for a hard-real time system the system receives an infinite negative value if the task misses its deadline.

A task can be in one of the following states: ready, running, and waiting. The operating system moves the tasks between the states. When several tasks are ready simultaneously, the operating system picks one of them, i.e.,
schedules the tasks. The next section covers scheduling of tasks in real-time systems. A task is in the waiting state when it has requested a resource that cannot immediately be serviced.

### 2.1.1 Scheduling

A real-time system consists of a set of tasks, possibly with precedence constraints that specify if a task needs to precede any other tasks. A subset of the tasks may be ready for execution at the same time, i.e., a choice has to be made which task should be granted access to the CPU. A *scheduling algorithm* determines the order the tasks are executed on the CPU. The CPU is allocated to a selected task and this process is called dispatching. Normally, the system has a real-time operating system (RTOS) that performs the actions of scheduling and dispatching. The RTOS has a queue of all ready tasks from which it chooses a task and dispatches it.

A feasible schedule is an assignment of tasks to the CPU such that each task is executed until completion [11]. The problem of constructing a schedule on \( m \) processors given a set of tasks with precedence constraints is
known to be NP-complete [53]. Hence, this intractable problem cannot be given an exact solution at run-time (online), and for large instances not even off-line. However, the problem of sequencing tasks with release times and deadlines on one processor is NP-complete in the strong sense and solvable in pseudo-polynomial time if the values on release times and deadlines are bounded [19].\(^2\) Hence, there are polynomial algorithms to schedule a set of tasks with precedence constraints on one processor. Tasks have priorities reflecting their importance and the current state of the controlled environment. Scheduling algorithms that assume that the priority of a task does not change during its execution are denoted static priority algorithms [37]. Moreover, a schedule can be preemptive, i.e., a task can interrupt an executing task, or nonpreemptive, i.e., a started task runs to completion or until it becomes blocked on a resource before a new task can start to execute.

Under certain assumptions it is possible to tell whether a construction of a feasible schedule is possible or not. The two most known algorithms of static and dynamic priority algorithms are rate monotonic (RM) [45] and earliest deadline first (EDF) [31] respectively. The rate monotonic algorithm assigns priorities to tasks based on their period times. A shorter period time gives a higher priority. The priorities are assigned before the system starts and remain fixed. EDF assigns the highest priority to the ready task which has the closest deadline. The ready task with the highest priority, under both RM and EDF, is executing.

Under the assumptions, given below, A1–A5 for RM and A1–A3 for EDF, there are necessary and sufficient conditions for a task set to be successfully scheduled by the algorithm. The assumptions are [37]:

A1 Tasks are preemptive at all times.

A2 Only process requirements are significant.

A3 No precedence constraints, thus, tasks are independent.

A4 All tasks in the task set are periodic.

A5 The deadline of a task is the end of its period.

\(^2\)The problem has the following problem instance [19]: Set \(T\) of tasks and, for each task \(t \in T\), a length \(l(t) \in \mathbb{Z}^+\), a release time \(r(t) \in \mathbb{Z}_0^+\), and a deadline \(d(t) \in \mathbb{Z}^+\). The question is whether there is a one-processor schedule for \(T\) that satisfies the release time constraints and meets all the deadlines.
Under the assumptions A1–A5 the rate monotonic scheduling algorithm gives a condition on the total CPU utilization that is sufficient to determine if the produced schedule is feasible. The condition is \( U \leq n(2^{1/n} - 1) \), where \( U \) is the total utilization of a set of tasks and \( n \) is the number of tasks [11, 37]. The total utilization \( U \) is calculated as the sum of fractions of task computation times and task period times, i.e., \( U = \sum_{\forall \tau \in T} \frac{wcet(\tau)}{P(\tau)} \), where \( T \) is the set of tasks, \( wcet(\tau) \) the worst-case execution time of \( \tau \), and \( P(\tau) \) the period time of task \( \tau \). Note that if \( U \) is greater than the bound given by \( n(2^{1/n} - 1) \) then there may exist a schedule that is schedulable, but if \( U \) is less than the bound, then it is known to exist a feasible schedule, namely the one generated by RM.

The sufficient and necessary conditions for EDF still hold if assumptions A4 and A5 are relaxed. EDF is said to be optimal for uniprocessors [11,37]. The optimality lies in the fact that if there exists a feasible schedule for a set of tasks generated by any scheduler, then EDF can also generate a feasible schedule. As for RM, there exists a condition on the total utilization that is easy to check. If \( U \leq 1 \), then EDF can generate a feasible schedule.

When the system is overloaded, i.e., when the requested utilization is above one, EDF performs very poorly [11, 60]. The domino effect occurs because EDF executes the task with the closest deadline, letting other tasks to wait, and when the task finishes or terminates, all blocked tasks might miss their deadlines. Haritsa et al. introduce adaptive earliest deadline (AED) and hierarchical earliest deadline (HED) to enhance the performance of EDF in overloads [29].

### 2.1.2 Precedence Constraints

Precedence constraints can be taken care of by manipulating start and deadlines of tasks according to the precedence graph\(^3\) and ready tasks. The adjusted tasks are sent to the EDF scheduler and it is ensured that the tasks are executed in the correct order. A description of the algorithm for manipulating time parameters can be found in [11].

Another method to take care of precedence constraints is the PREC1 algorithm described in [37]. The precedence graph is traversed bottom-up from the task that is started, \( \tau \), and tasks are put in a schedule as close to the deadline of \( \tau \) as possible. When the precedence graph has been traversed,\(^3\)A precedence graph is a directed acyclic graph describing the partial order of the tasks, i.e., which tasks need to be executed before other tasks.
tasks are executed from the beginning of the constructed schedule.

2.1.3 Servers

The dynamic nature of aperiodic tasks makes it hard to account for them in the design of a real-time system. In a hard real-time system, where there is a need to execute soft aperiodic real-time tasks, a server can be used to achieve this. The idea is that a certain amount of the CPU bandwidth can be allocated to aperiodic tasks without violating the execution of hard real-time tasks. A server has a period time and a capacity. Aperiodic tasks can consume the available capacity for every given period. For each server algorithm, there are different rules for recharging the capacity. The hard real-time tasks can either be scheduled by a fixed priority scheduler or a dynamic priority scheduler. Buttazzo gives an overview of servers in [11].

An interesting idea presented by Chetto and Chetto is the earliest deadline last server [12]. Tasks are executed as late as possible and in the meantime aperiodic tasks can be served. An admission test can be performed before starting to execute an arrived aperiodic task. Period times and WCET of hard real-time tasks need to be known. Tables are built that holds the start times of hard real-time tasks. Thomadakis discusses algorithms that can make the admission test in linear time [65].

2.2 Databases

A database stores data and users retrieve information from the database. A general definition of a database is that a database stores a collection of data representing information of interest to an information system, where an information system manages information necessary to perform functions of a particular organization\footnote{In [8] an organization is any set of individuals having the same interest, e.g., a company. We use the broader interpretation that an organization also can be a collection of applications/tasks in a software storing and retrieving data.} [8], whereas a database is defined as a set of named data items where each data item has a value in [9]. Furthermore, a database management system (DBMS) is a software system able to manage collections of data, which have the following properties [8].

- \textit{Large}, in the sense that the DBMS can contain hundreds of Mb of data. Generally, the set of data items is larger than the main memory of the computer and a secondary storage has to be used.
Background

- **Shared**, since applications and users can simultaneously access the data. This is ensured by the concurrency control mechanism. Furthermore, the possibilities for inconsistency are reduced since only one copy of the data exists.

- **Persistent**, as the lifespan of data items is not limited to single executions of programs.

In addition, the DBMS has the following properties.

- **Reliability**, i.e., the content of a database in the DBMS should keep the data during a system failure. The DBMS needs to have support for backups and recovery.

- **Privacy/Security**, i.e., different users known to the DBMS can only carry out specific operations on a subset of the data items.

- **Efficiency**, i.e., the capacity to carry out operations using an appropriate amount of resources. This is important in an embedded system where resources are limited.

A database system (DBS) can be viewed to consist of software modules that support access to the database via database operations such as Read($x$) and Write($x$, val), where $x$ is a data item and val the new value of $x$ [9]. A database system and its modules are depicted in figure 2.2. The transaction manager receives operations from transactions, the transaction operations scheduler (TO scheduler) controls the relative order of operations, the recovery manager manages commitment and abortion of transactions, and the cache manager works directly on the database. The recovery manager and the cache manager is referred to as the data manager. The modules send requests and receive replies from the next module in the database system.

The database can either be stored on stable storage, e.g., a hard drive or in main-memory. A traditional database normally stores data on a disk because of the large property in the list above.

2.2.1 Transactions

A transaction is a function that carries out database operations in isolation [8, 9]. A transaction supports the operations Read, Write, Commit and Abort. All database operations are enclosed within the operations begin of transaction (BOT) and end of transaction (EOT). All writings to data items
2.2. Databases

Figure 2.2: A database system.

within a transaction have either an effect on the database if the transaction commits or no effect if the transaction aborts. A transaction is well-formed if it starts with the begin transaction operation, ends with the end transaction operation, and only executes one of commit and abort operations.

The properties atomicity, consistency, isolation, and durability (abbreviated ACID) should be possessed by transactions in general [8]. Atomicity means that the database operations (reads and writes) executed by a transaction should seem, to a user of the database, to be executed indivisibly, i.e., all or nothing of the executed work of a finished transaction is visible. Consistency of a transaction represents that none of the defined integrity constraints on a database are violated (see section Consistency (section 2.2.2)). Execution of transactions should be carried out in isolation meaning that the execution of a transaction is independent of the concurrent execution of other transactions. Finally, durability refers to that the result of a successful committed transaction is not lost, i.e., the database must ensure that no data is ever lost.

2.2.2 Consistency

Transactions should have an application-specific consistency property, which gives the effect that transactions produce only consistent results. A set of integrity constraints is defined for the database as predicates [8,9]. A database state is consistent if, and only if, all consistency predicates are true.

Consistency constraints can be constructed for the following types of consistency requirements: internal consistency, external consistency, temporal consistency, and dynamic consistency. Below each type of consistency is described [39].
• **Internal consistency** means that the consistency of data items is based on other items in the database. For instance, a data item Total is the sum of all accounts in a bank, and an internal consistency constraint for Total is true if, and only if, Total represents the total sum.

• **External consistency** means that the consistency of a data item depends on values in the external environment that the system is running in.

• **Temporal consistency** means that the values of data items read by a transaction are sufficiently correlated in time.

• **Dynamic consistency** refers to several states of the database. For instance, if the value of a data item was higher than a threshold then some action is taken that affects values on other data items.

It is important to notice that if the data items a transaction reads have not changed since the transaction was last invoked, then the same result would be produced if the transaction was executed again. This is under the assumption that calculations are deterministic and time invariant. The invocation is unnecessary since the value could have been read directly from the database. Furthermore, if a calculation is interrupted by other more important calculations, then read data items might origin from different times, and, thus, also from different states of the system. The result from the calculation can be inconsistent although it is finished within a given time.

This important conclusion indicates that there are two kinds of data freshness consistency to consider: absolute and relative. Absolute consistency means that data items are derived from values that are valid when the derived value is used; relative consistency means that derived data items are derived from values that were valid at the time of derivation, but not necessarily valid when the derived value is used. Ramamritham introduces absolute and relative consistency for continuous systems [55] and Kao et al. discuss the consistency for discrete systems [35]. A continuous system is one where the external environment is continuously changing, and a discrete system is one where the external environment is changing at discrete points in time. In both [55] and [35], the freshness of data items is defined in the time domain, i.e., a time is assigned to a data item telling how long a value of the data item is considered as fresh.
Absolute consistency, as mentioned above, maps to internal and external consistency, whereas relative consistency maps to temporal consistency. The following two subsections cover absolute and relative consistency definitions in the time domain and value domain respectively.

Data Freshness in Time Domain

Physical quantities do not change arbitrarily and, thus, engineers can use this knowledge by assuming an acquired value is valid a certain amount of time. The validity of data items using the time domain has been studied in the real-time community [5,7,15,23,24,35,36,44,50,55,66,67].

A continuous data item is said to be absolutely consistent with the entity it represents as long as the age of the data item is below a predefined limit [55].

Definition 2.2.1 (Absolute Consistency). Let \( x \) be a data item. Let \( timestamp(x) \) be the time when \( x \) was created and \( avi(x) \), the absolute validity interval, be the allowed age of \( x \). Data item \( x \) is absolutely consistent when:

\[
\text{current\_time} - timestamp(x) \leq avi(x).
\] (2.1)

Note that a discrete data item is absolutely consistent until it is updated, because discrete data items are assumed to be unchanged until their next update. An example of a discrete data item is engineRunning that is valid until the engine is either turned on or off. Thus, since a discrete data item is valid for an unknown time duration, it has no absolute validity interval.

There can be constraints on the values being used when a value is derived. The temporal consistency of a database describes such constraints, and one constraint is relative consistency stating requirements on data items to derive fresh values. In this thesis we adopt the following view of relative consistency [35].

Definition 2.2.2 (Relative Consistency). Let validity interval for a data item \( x \) be defined as \( VI(x) = [\text{start}, \text{stop}] \subseteq \mathbb{R} \), and \( VI(x) = [\text{start}, \infty] \) if \( x \) is a discrete data item currently being valid. Then, a set of data items \( RS \) is defined to be relatively consistent if

\[
\bigcap\{VI(x_i) | \forall x_i \in RS\} \neq \emptyset.
\] (2.2)
The definition of relative consistency implies a derived value from RS is valid in the interval when all data items in the set RS are valid. The temporal consistency, using this definition, correlates the data items in time by using validity intervals. This means that old versions of a data item might be needed to find a validity interval such that equation 2.2 holds. Thus, the database needs to store several versions of data items to support this definition of relative consistency. Datta and Viguire have constructed a heuristic algorithm to find the correct versions in linear time [15]. Kao et al. also discuss the subject of finding versions and use an algorithm that presents the version to a read operation that has the largest validity interval satisfying equation 2.2.

Data Freshness in Value Domain

Kuo and Mok present the notion of similarity as a way to measure data freshness and then use similarity in a concurrency control algorithm [39]. Similarity is a relation defined as: \( f : D \times D \rightarrow \{true, false\} \), where \( D \) is the domain of data item \( d \). The value of a data item is always similar to itself, i.e., the relation is reflexive. Furthermore, if a value of data item \( d, v_1(d) \), is similar to another value of data item \( d, v_2(d) \), then \( v_2(d) \) is assumed to be similar to \( v_1(d) \). This is a natural way to reason about similar values. If value 50 is similar to value 55, it would be strange if value 55 is not similar to value 50. Thus, relation \( f \) is symmetric. The relation in figure 2.3 is reflexive, symmetric and transitive, but a similarity relation does not need to be transitive. The similarity relation \( |old - new| \leq \text{bound} \) is reflexive since \( old = new \iff |old - old| \leq \text{bound} \), and symmetric since \( |old - new| \leq \text{bound} \iff |new - old| \leq \text{bound} \), but not transitive since, e.g., \( |5 - 7| \leq 3, |7 - 9| \leq 3 \), but \( |5 - 9| \not\leq 3 \).

The intervals where two temperatures are considered to be similar might be entries in a lookup table, thus, all temperatures within the same interval result in the same value to be fetched from the table, motivating why similarity works in real-life applications. Transactions can use different similarity relations involving the same data items.

It should be noted there are other definitions of relative consistency than definition 2.2.2. Ramamritham defines relative consistency as the time-stamps of data items being close enough in time, i.e., the values of the data items originate from the same system state [55]. The difference between the two described ways to define relative consistency is that in definition 2.2.2 values need to be valid at the same time, but in [55] the values need to be
2.2. Databases

created at roughly the same time. Algorithms presented in this thesis use data freshness in the value domain by using similarity relations which has the effect of making data items to become discrete since the value of data items are updated only due to changes in the external environment. The definition of relative consistency (definition 2.2.2) is aimed at describing relative consistency for discrete data items, and is, thus, the definition we use.

\[
f(t_1, t_2): \\
\text{if } t_1 < 50 \text{ and } t_2 < 50 \\
\quad \text{return true} \\
\text{else if } t_1 \geq 50 \text{ and } t_1 < 65 \text{ and } t_2 \geq 50 \text{ and } t_2 < 65 \\
\quad \text{return true} \\
\text{else if } t_1 \geq 65 \text{ and } t_1 < 95 \text{ and } t_2 \geq 65 \text{ and } t_2 < 95 \\
\quad \text{return true} \\
\text{else if } t_1 \geq 95 \text{ and } t_1 < 100 \text{ and } t_2 \geq 95 \text{ and } t_2 < 100 \\
\quad \text{return true} \\
\text{else if } t_1 = 100 \text{ and } t_2 = 100 \\
\quad \text{return 100} \\
\text{else} \\
\quad \text{return false}
\]

Figure 2.3: An example of a similarity relation for temperature measurements.

Epsilon-serializability also uses a form of similarity [54]. Epsilon-serializability is used in concurrency control to relax the serializability criterion (see section 2.4) and transactions are allowed to import inconsistencies or export inconsistencies as long as they are bounded. The degree of error in read values or written values is measured by an upper bound on how much a value possibly can change when concurrent transactions are using it.

Wedde et al. use similarity to reduce the number of invocations of transactions to reduce system workload [68]. Here, the similarity relation \( f \) is defined to be a bound on how much two values of a data item can differ and still consider the values as similar.

Ramamritham et al. have investigated data dissemination on the Internet, where the problem of clients reading dynamic data from a server is discussed [16, 47, 56]. Dynamic data is characterized by rapid changes and the unpredictability of the changes, which makes it hard to use prediction
techniques to fetch/send data at predetermined times. The data should have *temporal coherency* between the value at the server and the value at the client. In this context, temporal coherency is defined as the maximum deviation between the client value and the server value of a data item. Ramamritham et al. note that the deviation could be measured over a time interval and temporal coherency is then the same as absolute consistency as defined in definition 2.2.1 [16, 47, 56]. However, the deviation can be measured in units in the value of a data item. This is then the same as that used by Wedde et al. [68].

Data can be fed to clients in two ways. Either by the server pushing values when conditions are fulfilled, e.g., the new value of a data item has changed more than a given bound from the last sent value of the data item to the client, or by the client pulling values from the server. In order to achieve good temporal coherency, algorithms that combine push and pull techniques have been proposed by Ramamritham et al. [16, 56]. A feedback control-theoretic approach is investigated in [47].

### 2.2.3 Updating Algorithms

In order to keep data items fresh according to either of the data freshness definitions given above, on-demand updating of data items can be used [5, 7, 15, 22–24]. A triggering criterion is specified for every data item and the criterion is checked every time a data item is involved in a certain operation. If the criterion is true, then the database system takes the action of generating a transaction to resolve the triggering criterion. Thus, a triggered transaction is created by the database system that executes before the triggering transaction\(^5\) continues to execute. Considering data freshness, the triggering criterion coincides with the data freshness definition and the action is a read operation, i.e., the updating algorithms either use data freshness defined in the time domain by using absolute validity intervals or in the value domain by using a similarity relation. Formally, we define the triggering criterion as follows.

**Definition 2.2.3 (On-Demand Triggering).** Let \(O\) be operations of a transaction \(\tau\), \(A\) an action, and \(p\) a predicate over \(O\). On-demand triggering is defined as checking \(p\) whenever \(\tau\) issues an operation in \(O\) and taking \(A\) if and only if \(p\) is evaluated to true.

\(^5\)A triggering transaction is the transaction that caused the action of starting a new transaction.
2.3 Electronic Engine Control Unit

A vehicle control system consists of several electronic control units (ECUs) connected through a communication link normally based on CAN [64]. A typical example of an ECU is an engine electronic control unit (EECU). In the systems of today, the memory of an EECU is limited to 64Kb RAM, and 512Kb Flash. The 32-bit CPU runs at 16.67MHz.\(^6\)

The EECU is used in vehicles to control the engine such that the air/fuel mixture is optimal for the catalyst, the engine is not knocking,\(^7\) and the fuel consumption is as low as possible. To achieve these goals the EECU consists of software that monitors the engine environment by reading sensors, e.g., air pressure sensor, lambda sensor in the catalyst, and engine temperature sensor. Control loops in the EECU software derive values that are sent to actuators, which are the means to control the engine. Examples of actuator signals are fuel injection times that determine the amount of fuel injected into a cylinder and ignition time that determines when the air/fuel mixture should be ignited. Moreover, the calculations have to be finished within a given time, i.e., they have deadlines, thus, an EECU is a real-time system. All calculations are executed in a best effort way meaning that a calculation that has started executes until it is finished. Some of the calculations have deadlines that are important to meet, e.g., taking care of knocking, and these calculations have the highest priority. Some calculations (the majority of the calculations) have deadlines that are not as crucial to meet and these calculations have a lower priority than the important calculations.

The EECU software is layered, which is depicted in figure 2.4. The bottom layer consists of I/O functions such as reading raw sensor values and transforming raw sensor values to engineering quantities, and writing actuator values. On top of the I/O layer is a scheduler that schedules tasks. Tasks arrive both periodically based on time and sporadically based on crank angles. The tasks are organized into applications that constitute the top layer. Each application is responsible for maintaining one particular part of the engine. Examples of applications are air, fuel, ignition, and diagnosis of the system, e.g., check if sensors are working. Tasks communicate results by storing them either in an application-wide data area (denoted \(ad\), application data in figure 2.4) or in a global data area (denoted \(gd\), global

\(^6\)This data is taken from an EECU in a SAAB 9-5.

\(^7\)An engine is knocking when a combustion occurs before the piston reaches its top position. Then the piston has a force in one direction and the combustion creates a force in the opposite direction. This results in high pressure inside the cylinder [49].
Figure 2.4: The software in the EECU is layered. Black boxes represent tasks, labeled boxes represent data items, and arrows indicate inter-task communication.

data in figure 2.4). In the original EECU software, when the system is overloaded, only some values needed by a calculation have to be fresh in order to reduce the execution time and still produce a reasonably fresh value. By definition, since all calculations are done in a best effort way, the system is a soft real-time system but with different significance on tasks, e.g., tasks based on crank angle are more important than time-based tasks, and, thus, tasks based on crank angle are more critical to execute than time-based tasks. The total number of data items in the EECU software is in the order of thousands.

Data items have freshness requirements and these are guaranteed by invoking the task that derives the data item as often as the absolute validity interval indicates. This way of maintaining data results in unnecessary updates of data items, thus leading to reduced performance of the overall system. This problem is addressed in chapter 4.

The diagnosis is running with the lowest priority, i.e., it is executed when there is time available but not more often than given by two periods (every 100 ms and 1 s). The diagnosis is divided into 60 subtasks that are executed in sequence and results are correlated using a Manager. Now, since the diagnosis has the lowest priority, this means that the calculations might be interrupted often by other parts of the system and if we measure the time from arrival to finishing one diagnosis, the elapsed time can be long [25]. Apart from delaying the completion of diagnosis, the low priority of the diagnosis can also lead to, as indicated in chapter 1, that diagnosis functions use relatively inconsistent values.
2.3.1 Data and Transaction Model

In the EECU software, calculations derive either actuator values or intermediate values used by other calculations. A calculation uses one or several data items to derive a new value of one data item, i.e., every data item is associated with a calculation, which produces a result constituting the value of the data item. Every calculation becomes a transaction in the database system and transactions are invoked by tasks. Hence, a data item is associated with one value, the most recently stored, and a transaction that produces a value of the data item. The data items in the EECU software can be classified as base items ($B$) or derived items ($D$). The base items are sensor values, e.g., engine temperature, and the derived data items are actuator values or intermediate values used by several calculations, e.g., a fuel compensation factor based on temperature. The relationship between data items can be described in a directed acyclic graph $G = (V, E)$, where nodes ($V$) are the data items, and an edge from node $x$ to $y$ shows that $x$ is used by the transaction that derives values of data item $y$. In this thesis we refer to $G$ as the data dependency graph. Figure 2.5 shows the data dependency graph on a subset of data items in an EECU software. The data dependency graph in figure 2.5 is used throughout the thesis as an example. All data items read by a transaction to derive a data item $d$ are denoted the read set $R(d)$ of $d$. The value of a data item $x$, stored in the database at time $t$, is given by $v^t_x$.

2.4 Concurrency Control

Concurrency control is in general used for ensuring the atomicity, consistency, and isolation properties of a transaction. The following subsections contain a background on different concurrency control algorithms.

2.4.1 Serializability

As described in the section Transactions (section 2.2.1), a transaction consists of operations: read, write, abort, and commit.

The task of the database system is to execute operations of concurrent transactions such that the following anomalies cannot occur [8]:

---

[8] In general, other operations are possible, see [9] for more details.
• *Lost update*, where a transaction overwrites the result of another transaction, and, hence, the result from the overwritten transaction is lost.

• *Dirty read*, where a transaction reads and uses a result of a transaction that is aborted later on, i.e., the transaction should not have used the results.

• *Inconsistent read*, a transaction reading the same data item several times gets, because of the effects of concurrent transactions, different values.

• *Ghost update*, where a transaction only sees some of the effects of another transaction, and, thus, consistency constraints do not hold any longer. For example [8], consider two transactions, $\tau_1$ and $\tau_2$, and the constraint $s = x + y + z = 1000$. The operations are executed in the order given in figure 2.6. The value of $s$ in $\tau_1$ at commit time is $1100$ since $\tau_1$ has seen intermediate results from $\tau_2$.

A transaction operation scheduler (TO scheduler) is used to schedule incoming operations from transactions such that lost updates, dirty reads, inconsistent reads, and ghost updates cannot occur. The task scheduler
schedules tasks that invoke transactions, and the TO scheduler schedules the operations from these transactions. The TO scheduler produces a history of the operations of active transactions. A transaction is active if its BOT operation has been executed and it has not yet aborted or committed. Thus, a history is a recording of all operations, and their relative order, that have executed and completed. Two histories are said to be equivalent if they are over the same set of transactions and have the same operations, and conflicting operations of non-aborted transactions have the same relative order.

In a serial history, for every pair of transactions all operations of one transaction execute before any operation from the other transaction. The anomalies described above cannot occur in a serial history. However, from a performance perspective, it is not efficient to execute transactions non-preemptibly in sequence since one transaction can wait for I/O operations to finish and in the meantime other transactions could have been executed. From a real-time perspective, important transactions should always have priority over less important transactions. This means that executing transactions non-preemptibly gives bad performance and does not obey priorities. Hence, the TO scheduler needs to schedule operations preemptibly and consider priorities on transactions.

Figure 2.6: Example of ghost update.
The committed projection of a history $H$ contains only the operations from transactions that commit. We say $H$ is serializable if the effect of executing operations from a committed projection of history $H$ generated by a TO scheduler is the same as the effect of executing operations from the committed projection of a serial history [9].

A history $H$ is said to be view-equivalent to a history $H'$ if (i) the results of read operations of transactions in $H$ are the same as the results of read operations the same transactions see in $H'$, and (ii) the final write operations write the same values in $H$ as in $H'$ [8, 52]. A history is view-serializable if the history is view-equivalent to a serial history. An operation $o_i$ from transaction $\tau_i$ is in conflict with an operation $o_j$ from transaction $\tau_j$ ($i \neq j$), if both operate on the same data item and at least one of the operations is a write. A history is conflict-equivalent to another history if they contain the same operations and the same operations under conflict appear in the same order in both histories. A history $H$ that is conflict-equivalent to a serial history is conflict-serializable. It can be shown, the set of all view-serializable histories contains all conflict-serializable histories, thus, view-serializability allows more histories than conflict-serializability.

However, the computational complexity of deciding if a history is view-serializable is NP-hard, since there exists a polynomial time algorithm to check if a history is view-equivalent to a given serial history [9, 52]. The problem is that this algorithm has to be used on every possible serial history. The number of serial histories is given by the permutations of all transactions [9, 52]. It can be shown that all conflict-serializable histories are also view-serializable but the converse is not always true, thus, conflict-serializability is a sufficient but not necessary condition for view-serializability [9, 52]. It is computationally easy to decide if a history is conflict-serial. It can be done by constructing a directed graph where nodes are transactions and an edge is a conflict between two transactions. If the graph is acyclic it is conflict-serializable, this can be checked in polynomial time in the size of the graph [9, 52].

Kuo and Mok define two histories, $H$ and $H'$, to be view-similar if [39, 40]:

1. They are over the same set of transactions.

2. Given two similar databases, i.e., the data items are similar in the databases, and that a set of transactions is executed in the databases such that $H$ and $H'$ are recorded, then the resulting databases are
also similar.

3. For every transaction in one of the histories and every value it reads, 
the corresponding transaction in the other history reads a similar 
value.

**Recovery**

The recovery module (see figure 2.2) is designed to make the database system resilient to failures. The recovery module must ensure that when the database system is recovered from a system failure only effects from committed transactions are seen. The database system clears the effect of transactions that need to be aborted by restoring the values of write operations. When a transaction aborts, possibly other transactions also need to abort. This is called cascading aborts.

A history is recoverable if a transaction \( \tau \) commits after the commitment of all transactions producing results that are read by \( \tau \), i.e., those transactions have written values to data items that \( \tau \) has read and the write operation occurred before the read operation of \( \tau \). Cascading aborts are avoided if transactions only read values written by already committed transactions.

Further, when clearing the effect of write operations when transactions are aborted, the so called before images of the data items need to be stored. These images are needed because the history the DBS has produced after transactions are aborted is the history where all operations of the aborted transactions are removed from the history. The value of a data item might need to be altered when a write operation is undone. This gives some problems, which is illustrated by the following two examples [9]:

**Example 2.1:** Consider the following history: Write\(_1\)(\(x,1\)), Write\(_1\)(\(y,3\)), Write\(_2\)(\(y,1\)), Commit\(_1\), Read\(_2\)(\(x\)), Abort\(_2\). The operation Write\(_2\)(\(y,1\)) should be undone, which it is by writing its before image of 3 into \( y \).

However, it is not always the case that the before image of a write operation in the history is the correct value to write into a data item.

**Example 2.2:** Consider the following history: Write\(_1\)(\(x,2\)), Write\(_2\)(\(x,3\)), Abort\(_1\). The initial value of \( x \) is 1. The before image of Write\(_1\)(\(x,2\)) is 1, but the value the write operation should be restored with is 3, i.e., the write operation of transaction \( \tau_1 \) does not have any effect because it is overwritten by Write\(_2\)(\(x,3\)).

In example 2.2, the miss in before images and values that should be
written to data items arises when several, not yet terminated, transactions have written to the same data item. This problem can be avoided by requiring that write operations are delayed until all transactions previously writing into the same data items have either committed or aborted. An execution sequence of operations that satisfies the discussed delays for both read and write operations is called strict.

2.4.2 Concurrency Control Algorithms

The objective of a concurrency control algorithm is to make sure operations issued by transactions are executed in an order such that the results produced by the involved transactions are consistent. The correctness criterion in non-real-time settings is normally serializability, i.e., the effect of the execution of the transactions is equivalent to a serial schedule. A TO scheduler implementing a concurrency control algorithm can either delay, accept, or reject an incoming operation. A concurrency control algorithm can be conservative, meaning that operations are delayed to still have the possibility to reorder operations in the future, or aggressive where incoming operations are immediately accepted [9].

There are three general ways to implement a concurrency control algorithm. The algorithms can either be based on (i) locks, (ii) conflict graph, or (iii) timestamps. Lock and timestamp ordering algorithms are presented in the remainder of this section. Note that for locking-based concurrency control algorithms, conservative TO schedulers are denoted pessimistic concurrency control algorithms, and aggressive TO schedulers are denoted optimistic concurrency control algorithms. Papadimitriou gives a good overview of concurrency control algorithms [52]. Another good book on the subject is Bernstein et al. [9].

Pessimistic

This section on pessimistic concurrency control algorithms covers the basic two-phase locking algorithm (2PL) and the enhanced high-priority two-phase locking algorithm, which is more suited for real-time system than the former algorithm.

Locking is a well-known and well-explored technique to synchronize access to shared data. It is used in operating systems for the same purpose by using semaphores. In a database, before a transaction may access a data item it has to acquire a lock. When the database system grants a lock to a
transaction, the transaction can continue its execution. Since a transaction accesses data items via the operations read and write, two types of locks are used, one for each operation. A read-lock on data item \( x \) is denoted \( rl[x] \) and, correspondingly a write-lock \( wl[x] \). Hence, a way to order conflicting operations is needed, and therefore the write-lock is stronger than the read-lock since a conflict always involves at least one write. The effects of this are that several transactions can read-lock the same data item, but only one transaction can hold a write-lock on a data item. The two-phase locking algorithm produces a conflict-serial history. The rules for the two-phase locking algorithm are [9]:

1. An incoming operation issues a lock and a test is done to see if a conflicting lock is already held by another transaction on the data item. If the data item is already locked and the new requested lock conflicts with it, then the operation is delayed until the conflicting lock is released. Otherwise, the data item is locked and the operation is accepted.

2. When the TO scheduler has set a lock for a transaction, the TO scheduler may not release the lock until the database module acknowledges that it has processed the corresponding operation.

3. When the TO scheduler has released one lock it may not acquire anymore locks for this transaction.

Rule three is called the two-phase rule, since it divides the locking into two phases, a growing phase where all locks are acquired and a shrinking phase where the locks are released.

The three rules above order operations such that a recording of them is serializable [9, 52]. Unfortunately, this algorithm can be subject to deadlocks, which means that two or more transactions cannot continue their execution because they are waiting for locks to be released, but the locks are never being released since they are held by transactions involved in the waiting. An example clarifies the reasoning.

**Example 2.3:** Transaction \( \tau_1 \) holds a write-lock on \( x \) and requests a read-lock (for instance) on \( y \). Transaction \( \tau_2 \) already holds a write-lock on \( y \) and requests a read-lock on \( x \). Now, both \( \tau_1 \) and \( \tau_2 \) wait infinitely for the locks on \( x \) and \( y \) to be released. Of course, deadlocks give unbounded blocking times that are unwanted in a real-time system. □
Strict (or conservative) 2PL avoids deadlocks by requiring every transaction to acquire all its locks before the execution of operations start. Thus, the read and write sets need to be predeclared. When the TO scheduler is given the read and write sets of a transaction, it is investigated if any of the locks are held by another transaction. If that is the case, the transaction is put on a waiting list together with the locks. When a transaction reaches its commit operation and its locks are released, the TO scheduler checks if a transaction on the waiting list can acquire all locks. When all locks can be acquired, a transaction starts to send its operations to the data manager.

Strict 2PL ensures strict histories. This is accomplished by releasing read-locks when the transaction is terminated and releasing write-locks after the data manager has processed the commit operation. The details of strict 2PL are outlined in [9].

High-Priority Two-Phase Locking

The high-priority two-phase locking (HP2PL) algorithm improves upon the two-phase locking algorithm in that priorities on transactions are taken into account in the scheduling of transaction operations [4]. Conflicts are resolved in favor for higher prioritized transactions. When a transaction issues a write operation and the TO scheduler tries to acquire a lock for the transaction, but the data item is already locked, then either the transaction waits if it does not have a higher priority than any of the transactions holding a lock on the data item, or the transaction has the highest priority and then all lock holders are aborted and the transaction acquires the write-lock. If the operation is a read instead, then if a conflicting lock is already given to another transaction—i.e., a write-lock—then that transaction is aborted if it has a lower priority. Otherwise, the issuing transaction waits. The HP2PL concurrency control algorithm is well suited for real-time systems since the TO scheduler preempts transactions and priority on transactions are considered. Furthermore, this algorithm is free of deadlocks.

Optimistic

As mentioned above, operations can be delayed or accepted immediately by the TO scheduler. The two-phase locking algorithm presented above is a conservative TO scheduler. An aggressive approach would be to immediately accept incoming operations. This is an optimistic approach since the TO scheduler accepts operations and hopes that they do not conflict.
The only means to resolve conflicts now are to restart transactions that are involved in a conflict. The TO scheduler checks the status of accepted operations when a commit operation arrives from a transaction. The process of checking if a commit operation should be accepted or rejected is called a certification, and TO schedulers that make such decisions are called certifiers. There exist certifiers for the three main concurrency control algorithms: locking, serializability graphs, and timestamp ordering. A certifier based on locking is presented here since that is the most well-known version of an optimistic concurrency control algorithm.

A transaction is divided into a read phase, a validation phase, and a write phase. In the read phase, write operations write to local memory, in the validation phase it is investigated if the transaction conflicts with other transactions and if it can continue to its write phase where writes are made global. One of the following three conditions must hold ($\tau_i$ is the validating transaction and $\tau_j$ is any other active transaction, and $rs(\tau)$ and $ws(\tau)$ are the read set and write set of $\tau$ respectively) [38]: (i) the write phase of $\tau_i$ completes before the read phase of $\tau_j$ starts, (ii) $ws(\tau_i) \cap rs(\tau_j) = \emptyset$ and the write phase of $\tau_i$ completes before the write phase of $\tau_j$, and (iii) $ws(\tau_i) \cap ws(\tau_j) = \emptyset$ and $ws(\tau_i) \cap rs(\tau_j) = \emptyset$ and the read phase of $\tau_i$ completes before the read phase of $\tau_j$. Kung and Robinson present a validation phase ensuring conditions (i) and (ii) and a validation phase ensuring all three conditions [38]. Ensuring the two first conditions, at the arrival of the commit operation of transaction $\tau$, the TO scheduler checks whether the read set of $\tau$ has any common element with the write set of all other active transactions. If no common element appears in these checks, the TO scheduler accepts the commit operation, otherwise, $\tau$ is aborted. This algorithm can be shown to be serializable [9, 52]. For details on a validation phase fulfilling conditions (i)–(iii) see [32,38].

Every transaction execute to the verification phase before the decision of aborting or committing the transaction is taken. The fact that a transaction needs to be restarted can be investigated in the write operation of some other transaction [4]. Concurrent readers of a data item that is being written by another transaction need to be aborted. This decision can be broadcast to these transactions immediately at the write operation. Upon the arrival of such a message, a transaction is aborted. Hence, there is no need to execute transactions to their commit operation, and, thus, CPU resources can be saved. This enhanced algorithm is denoted optimistic concurrency control with broadcast commit (OPT-BC) [57].
The optimistic concurrency control algorithm is deadlock-free and automatically uses priorities since transactions reach the commit operation based on how the operating system schedules the tasks that execute transactions.

**Timestamp**

In this concurrency control algorithm, the TO scheduler orders operations in strictly timestamp order. Each transaction is assigned a unique timestamp from a function \( ts \) and every operation of a transaction inherits the timestamp of the transaction issuing the operation. It is important that function \( ts \) is strictly monotonic, because then transactions get unique timestamps.

The basic timestamp ordering works as follows [9]: operations are accepted immediately and are output to the database module in a first-come-first served order. An operation is considered too late if the TO scheduler has already accepted a conflicting operation on the same data item, i.e., an operation \( o_i \) on data item \( x \) conflicts with operation \( o_j \) and \( ts(\tau_i) > ts(\tau_j) \).

The TO scheduler can only reject the operation from \( \tau_i \) and, hence, aborts \( \tau_i \). When \( \tau_i \) restarts it is assigned a higher timestamp from \( ts \) and has a higher chance of executing its operations.

It has been shown that the basic timestamp ordering algorithm generates a serializable history. **Thomas’ write rule** says that if the TO scheduler receives a write operation from transaction \( \tau_i \), but has already scheduled a write operation on the same data item from transaction \( \tau_j \) and \( ts(\tau_i) < ts(\tau_j) \), the late write, i.e., \( \text{Write}_i(x) \), can be skipped [9, 52]. According to the timestamp ordering the operation from \( \tau_i \) would be rejected, and, thus, \( \tau_i \) is aborted. However, if the TO scheduler only does write-write conflict resolution, then only the write operation with the latest timestamp is important. Consider the contrary case when \( \text{Write}_i(x) \) from \( \tau_i \) arrives before \( \text{Write}_j(x) \) from \( \tau_j \). Then, \( \text{Write}_j(x) \) overwrites the value from \( \text{Write}_i(x) \) which is the same effect as skipping \( \text{Write}_i(x) \) when it is too late.

Strictness and recoverability of timestamp ordering concurrency control are discussed in [9]. TO schedulers can be combined, e.g., two-phase locking for read-write conflicts and timestamp ordering for write-write conflicts. Such TO schedulers are denoted integrated TO schedulers.
Multiversion Concurrency Control

Another way to consider concurrency control is to use several versions on data items. The correctness criterion for execution of concurrent transactions is serializability, and conflicting operations lead to transactions being aborted or blocked. Now, if a write operation does not overwrite the value a concurrent transaction has read, but instead creates a new version of the data item, then late read operations can read an old version instead of being rejected resulting in the abortion of the transaction. Schedulers based on multiversion concurrency control can be based on two-phase locking, timestamp ordering, and serialization graph algorithm. Multiversion based on timestamp ordering and two-phase locking is described next, starting with the multiversion timestamp ordering (MVTO) algorithm [9,52]. In MVTO, operations are processed in a first-come-first-served manner and read operations, Read\(_i(x)\), are transformed into Read\(_i(x_k)\) —where \(x_k\) is a version produced by transaction \(\tau_k\)—with \(x_k\) having the largest timestamp less than \(ts(\tau_i)\). It is said that the read operation is reading the proper version of the data item. A write operation Write\(_i(x_i)\) of transaction \(\tau_i\) has to be rejected if a transaction \(\tau_j\) has read version \(x_h\) and \(ts(x_h) < ts(\tau_i)\), i.e., a later transaction reads a too early version which breaks the timestamp ordering of the transactions. Otherwise the write operation is translated into Write\(_i(x_i)\), i.e., a version is created with the timestamp of the transaction.

The two version two-phase locking (2V2PL) is a multiversion two-phase locking concurrency control algorithm that uses two versions of a data item to ensure read-locks and write-locks never conflict. The algorithm works as follows [9]. When a transaction \(\tau_i\) writes to a data item a new version \(x_i\) is created. A lock is set on data item \(x\) such that it is not accessible by other transactions. However, read operations can read the previous version of the data item. Hence, two-phase locking is used for write-write synchronization and version selection for read-write synchronization. Three types of locks are used: read, write, and certify. Read- and write-locks are set as usual for the two-phase locking algorithm, but when the commit operation is sent to the TO scheduler, all write-locks are converted into certify locks. The compatibility matrix of these types of locks is given in table 2.1.

A write operation is delayed until no other write-lock or certify lock is assigned to another transaction. Then a new version \(x_i\) (from transaction \(\tau_i\)) is created. A read operation can be executed as soon as there is no certify lock on the data item. There are two cases—either the transaction has a write-lock due to that it has written the data item itself, then the
Table 2.1: Compatibility matrix for 2V2PL.

<table>
<thead>
<tr>
<th></th>
<th>Read</th>
<th>Write</th>
<th>Certify</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>y</td>
<td>y</td>
<td>n</td>
</tr>
<tr>
<td>Write</td>
<td>y</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>Certify</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
</tbody>
</table>

read operation is $\text{Read}_i(x_i)$, otherwise $\text{Read}_i(x)$ is translated into $\text{Read}_i(x_j)$ where $x_j$ is the most recently committed version. Certify locks conflict with read-locks. At the reception of a commit operation the commit is delayed until there are no readers on the data items with write-locks that shall be converted into certify locks. Only committed versions can be read which results in that cascading aborts cannot occur, and the resulting histories are recoverable. Nyström et al. have used a similar concurrency control algorithm in a mixed hard soft real-time system [51]. Their approach is to divide transactions into hard and soft transactions and relax the serialization criterion. The concurrency control algorithm, denoted 2V-DBP, is shown to be deadlock free and hard transactions can never be delayed.

It has been shown that MVTO produces view-equivalent histories [9,52], thus, MVTO is serializable. Further, Bernstein et al. have shown that 2V2PL is view-serializable [9].

Relaxation of Correctness Criteria

The correctness criterion discussed so far is the well-known serializability, i.e., the effect of execution of transactions is as an execution of the transactions in sequence. One benefit or using serializability as a correctness criterion, as pointed out by Graham [21], is that it is easy to reason about the execution of transactions in sequence. However, as mentioned above, serializability punishes the performance of the database system, i.e., serializability might exclude execution orderings of operations that make sense for the application. Thus, the performance of the system can be increased by relaxing the serialization as correctness criterion. New concurrency control algorithms can be developed to support application-specific optimizations that extend the set of valid histories, i.e., the concurrency can be increased and the number of aborts can be decreased. An example of such concurrency control algorithms is the one developed by Kuo and Mok using similarity [39,40].
Concurrent Control and Consistency

A concurrency control algorithm can affect the relative and absolute consistency in the following ways:

- Relative consistency can be affected due to interruptions and blocking from transactions. An example is given.

  Example 2.4: Assume transaction $\tau_3$ reads data items $a$, $b$, and $c$, and writes $d$. Further, $\tau_1$ reads $e$ and writes $c$, and $\tau_2$ reads $f$ and writes $g$. Now, $\tau_3$ arrives to the database system and it starts to execute. At this time data items $a$, $b$, and $c$ are relatively consistent. Transaction $\tau_3$ reads $a$, but then $\tau_2$ arrives. After a while, $\tau_1$ also arrives to the system. Transaction $\tau_1$ updates data item $c$. Transactions $\tau_1$ and $\tau_2$ finish and $\tau_3$ continues with reading $b$ and $c$. Due to transaction $\tau_1$ data item $c$ is not relatively consistent with $a$ and $b$ any longer. If the old value of $c$ would have been saved, it could later be read by $\tau_3$ having read relatively consistent values. Using multiversion concurrency control, the old value of $c$ would have been read by $\tau_3$. Another approach to solve this is to use snapshot data structures [63].

- Absolute consistency can be affected due to restarts of transactions, interruptions and blocking from transactions. The value of a data item is valid from a given time, but it takes time to store the value in the database. Conflicts and concurrent transactions can delay the writing of the value to the database. Hence, fewer restarts of transactions could speed up the writing of the value to the database.

2.5 Checksums and Cyclic Redundancy Checks

Checksums and cyclic redundancy check (CRC) are used to verify that some data is correct.

A checksum is constructed by adding up the basic elements of the data. Checksums are of a fixed length, typically 8, 16, or 32 bits. The longer the checksum is the more errors can be detected. The fixed length also means that there is a many to one mapping from data to a checksum. To detect error in the data, the checksum should consider the order of the data elements (e.g., the bytes) rather than only adding them together, adding
Background

zero-valued data elements should be detected, i.e., altering the checksum, and multiple errors that cancel should preferably not be able to occur.

Well-known checksum algorithms are the Fletcher’s checksum [72] and Adler32 [17]. Note that a CRC is not considered a checksum since binary divisions are used in their algorithmic steps. However, CRCs are considered stronger, i.e., better at detecting errors than checksums, but the CRC algorithms use heavier instructions in terms of CPU cycles, and, thus, it takes a longer time to calculate a CRC than a checksum.

The 8-bit Fletcher’s checksum algorithm is now described [72]. Two unsigned 8-bit 1’s-complement accumulators are used, denoted $A$ and $B$. They are initially set to zero and are calculated over the range of all data elements. The accumulators are calculated in a loop ranging from 1 to $N$, where $N$ is the number of data elements, by doing the following in each iteration: $A = A + D[i]$ and $B = B + A$. When all octets $D[i]$ have been added $A$ holds the 1’s-complement of the sum of all octets, i.e., $\sum_{i=1}^{N} D[i]$, and $B$ contains $nD[1] + (n - 1)D[2] + \cdots + D[N]$.

A CRC is the remainder of a division [69]. The data is a string of bits, and every bit represents a coefficient in a polynomial. The divisor is a polynomial, e.g., $x^{16} + x^{15} + x^2 + 1$, and the dividend polynomial is divided with the divisor polynomial using binary arithmetic with no carries. The remainder is interpreted as binary data and constitutes the CRC. CRCs are considered stronger than checksums since the remainder is affected by every bit in the dividend. Figure 2.7 shows a pseudo-code of a CRC implementation [69]. The algorithm can be table-driven, which reduces the time it takes to calculate the CRC. A C implementation of a table-driven CRC can be found in [2].

Load the register with zero bits.
Augment the message by appending $W$ zero bits to the end of it.
While (more message bits)
    Begin
    Shift the register left by one bit, reading the next bit of
    the augmented message into register bit position 0.
    If (a 1 bit popped out of the register during step 3)
    Register = Register XOR Poly.
    End
The register now contains the remainder.

Figure 2.7: The pseudo-code of an algorithm producing a CRC.
2.5. Checksums and Cyclic Redundancy Checks
Chapter 3

Problem Formulation

This chapter presents, in section 3.1, a description of the problems the industrial partners have faced when it comes to software development and data management. Notations used throughout the thesis are given in section 3.2, and section 3.3 gives the formulation of the problems this research project addresses.

3.1 Software Development and Data Management

Current ad hoc solutions in developing an EECU software, such as application-specific data areas and global data areas that use traditional data structures as described in chapter 2, make software maintenance an expensive and complicated task. Furthermore, it is important, from the perspective of memory and CPU consumption that intermediate results are stored only once and calculated only when necessary. In addition, the data freshness on data items has to be maintained for computed results to be relevant. Deadlines on transactions can be seen as landmarks; for some transactions the produced values must be ready before the deadline, i.e., such deadlines are hard; for some transactions the produced values have no value to the system after the deadline, i.e., such deadlines are firm; and for some transactions, the produced values still give a value to the system, i.e., such deadlines are soft. Furthermore, for some transactions it is important to use data values originating from exactly the same time and state of the system, whereas for other transactions this is less significant.

Given the large amount of data used in an EECU software, the time constraints and data freshness requirements discussed above, the industrial
partners have identified the following problems with their current approach of developing embedded software. These include:

P1 Data items are partitioned into several different data areas, i.e., global and application-specific. This makes it difficult to keep track of which data items exist. Also, a data item can accidentally exist in several data areas. This increases both CPU and memory usage.

P2 The current approach does not necessarily obey consistency constraints on data.

P3 CPU and memory resources are not efficiently used in the current system, which increases the resource-demand and thereby the costs due to more expensive hardware.

In this thesis we investigate the feasibility of using DBMS technology in a real-time setting. Hence, in addition to a database addressing problem P1, a data management system is needed that addresses problem P2, and concurrency control algorithms working in conjunction with the data management system are needed to address problem P3.

3.2 Notations and Assumptions

Based on the description of an EECU software in the section Electronic Engine Control Unit (section 2.3), we use the following notations in the rest of the thesis. Data items belong either to base items $B$ or derived items $D$. Every data item has one transaction updating it and writing the new value to the database. In addition, every data item $x$ has a weight, $\text{weight}(x)$, that indicates how important the data item is in calculations. The weights make it possible to prioritize updates in the case of insufficient time to complete all necessary calculations. The worst-case execution time, excluding blocking times and updates, of a transaction $\tau$ is denoted $\text{wcet}(\tau)$ and is assumed to be known. A transaction has an arrival time $at(\tau)$, a release time $rt(\tau)$, a logical timestamp $ts(\tau)$, and a relative deadline $dl(\tau)$. The priority of every task is determined either by EDF or RM, and tasks start transactions that execute with the same priority as the task.

\[\text{It is important that the logical timestamp assigned to transactions is monotone. However, it is easy to achieve this in a central database system by atomically assigning a timestamp to the transaction in its BOT operation.}\]
Table 3.1: Transaction types that a real-time embedded database system should have support for.

<table>
<thead>
<tr>
<th>Deadline</th>
<th>Issued by</th>
<th>Denoted</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>hard</td>
<td>DBS</td>
<td>Sensor transaction (ST)</td>
<td>VUT</td>
</tr>
<tr>
<td>soft/firm</td>
<td>Application</td>
<td>User transaction (UT)</td>
<td>AUT/VUT</td>
</tr>
<tr>
<td>soft/firm</td>
<td>DBS</td>
<td>Triggered update (TU)</td>
<td>VUT</td>
</tr>
</tbody>
</table>

The relations between data items are described in a directed acyclic graph $G = (V, E)$, denoted data dependency graph (see figure 2.5 in chapter 2). The relation is denoted $<_G$. The nodes of data dependency graph $G$ reside in specific levels given by the following definition:

**Definition 3.2.1 (Level of a Derived Data Item).** Each base item $b$ has a fixed level of 1. The level of a derived data item $d$ is determined by the longest path in a data dependency graph $G$ from a base item to $d$. Hence, the level of $d$ is

$$level(d) = \max_{x \in R(d)} (level(x)) + 1,$$

where $R(d)$ is the read set of data item $d$.

A user transaction is an incoming transaction, and it derives data item $d_{UT}$, a sensor transaction derives $b_{ST}$, and a triggered update derives $d_{TU}$. A triggered update is a transaction that is the result of a triggering criterion being fulfilled in an executing user transaction. Section Updating Algorithms (section 2.2.3) contains a description of triggering of transactions. User transactions are further divided into value user transactions (VUT) and actuator user transactions (AUT). A value user transaction derives one data item and writes the result to the database, whereas an actuator user transaction derives a data value, stores it in the database, and then sends the value to an actuator. If a transaction is skipped, because the value it derives is already fresh, the actuator part cannot be skipped. Thus, a transaction can look as depicted in figure 3.1. This work is based on that the database can provide support for three different types of transactions; these are listed in table 3.1.

Throughout the thesis, the following assumptions are made about the system:

SA1 Once a transaction has started its priority is not changed.
3.3 Problem Formulation

The overall objective of this thesis is to provide efficient data management for real-time embedded systems, i.e., provide efficient resource utilization and increased maintainability of the software.

Efficient data management can be achieved in several ways. The focus of this particular research project is to investigate how a real-time database can give advantages in maintainability and resource utilization in real-time embedded applications. The industrial partners in the project have stated requirements that a data management in an EECU needs to have [20]:

![User Transactions Diagram]

Figure 3.1: User transactions. The user transaction at the top derives a value, and the user transaction at the bottom sends a derived value to an actuator.

SA2 Hardware settings put the requirement that all data needs to be stored in RAM and/or in flash memory. Moreover, at a system failure, e.g., the triggering of a watch-dog timer, the system is restarted which means that all transactions also start over.

Assumption SA1 states that the priority of a user transaction cannot be lowered during its execution letting another user transaction to interrupt it. Priority inversion can occur if locks are used (see Buttazzo for a description of priority inversion [11]). The transaction that is active and has the highest priority is the one executing.

Assumption SA2 states that only main-memory databases are considered and that transactions need not be recovered since they all restart and therefore recoverability is not an issue.
R1 A way to organize data to ease maintenance of the software is needed, because of the large amount of data items (in the order of thousands) and the long life-cycle of the software.

R2 Support for monitoring data is needed. The data management software should activate the correct task when some data fulfill the specified conditions.

R3 The system should:

R3a protect a data item from being written by several concurrent tasks;
R3b avoid duplicate storage of data items;
R3c guarantee correct age on data items used in a task; and
R3d give low overhead, i.e., efficient utilization of resources.

A database, per definition, has the properties to address R1, R3a, and R3b. The database modules need to consider and maintain the freshness of data items (R3c). Moreover, dynamically changing systems can enter different states, e.g., when a driver presses the gas pedal to accelerate (transient state) or when the driver drives at constant speed (steady state). Ideally, the database system should detect the different states and adjust the frequencies of recalculations accordingly, i.e., update frequencies on data items are dynamically adjusted based on how rapidly sensor values change in the environment. Assigning adequate update frequencies is expected to have a positive effect on the overall available CPU (as opposed to approaches where sensor values are always updated when data is assumed to become obsolete at the end of a constant time period, representing the worst case). In resource-limited systems, like embedded systems, it becomes increasingly important that the algorithms maintaining data freshness consider adapting the update frequencies to reduce the imposed workload.

Also, as pointed out in chapter 1, the consistency of the produced result is important and can be affected by the concurrency of the system. This problem is stated in requirement R3c.
3.3. Problem Formulation
Chapter 4

Updating and Concurrency Control Algorithms

In the problem formulation, requirement R3 states the requirements on the functionality of the database system as seen from the perspective of the industrial partners in this research project. Specifically, requirement R3a is about the TO scheduler, requirement R3b about avoiding duplicate storage of data, requirement R3c about data freshness, and R3d about the problem of effective utilization of available resources. These requirements are solved by using a database.

Section 4.1 describes a proposal of a database system that can address R3a–d. Section 4.2 describes data freshness definitions. The data freshness is defined in the value domain of data items using similarity relations. Section 4.2 also describes two updating schema. One updating scheme gives a mechanism to decide if a data item is potentially affected by a change in other values. The other provides a mechanism to make a relevance check whether a calculation results in a non-similar value compared to the previous value, which is stored in the database. Sections 4.3–4.7 describe the updating algorithms on-demand (OD), on-demand depth-first traversal (ODDFT), on-demand breadth-first traversal (ODBFT), on-demand top-bottom with relevance check (ODTB), and on-demand depth-first traversal with relevance check (ODDFT_C). Section 4.9 describes supporting mechanisms and algorithms in the proposed database system. Section 4.10 describes multiversion concurrency control algorithms using similarity (MVTO-S), and section 4.11 describes three implementations of MVTO-S: MVTO-UV, MVTO-UP, and MVTO-CHECKSUM. Section 4.12 also de-
scribes an OCC algorithm using similarity, and section 4.13 discusses implementation details of HP2PL and OCC in the real-time operating systems Rubus and µC/OS-II.

4.1 Database System Prototype

A database system, as described in the section Databases (section 2.2), should be able to hold a large amount of data items, and due to this demand a secondary memory is needed. In vehicular systems, it is reasonable that data is stored in main-memory (SA2). If the system restarts, all data values are reset to initial values also stored in the database, i.e., no before images are needed for recovery (see section 2.4.1).

The depicted database system in figure 2.2 has a transaction manager that receives transactions from tasks in the application. However, as mentioned earlier, transactions calculating a value might result in deriving the same value again. To effectively use available resources, requirement R3d, an admission controller, rejecting unnecessary calculations, is added to the database system. Also, the values that transactions use need to be relatively consistent. Hence, a data management module that has the functionality of an admission controller and maintainer of data freshness is needed. The database system developed in this project is depicted in figure 4.1.

![Database System Diagram](image)

Figure 4.1: Database system that addresses requirement R3.

The data management module, dashed module in figure 4.1, interacts with the central repository module and decides if incoming transactions should be sent to the transaction manager and if additional transactions need to be triggered due to data freshness requirements. The central repository module is responsible for storing data and its meta-information. In this database system, a triggered transaction is updating a data item, and
is denoted a triggered update. The TU is generated by the database system by creating a new transaction instance with the same priority and time-stamp as the triggering transaction. A TU can only be triggered by user transactions, and TUs are generated when a UT arrives to the database system. Furthermore, the condition for triggering updates is implemented in the data management module meaning that the database system has no general support for triggering of transactions as opposed to active databases.

Transactions are executed either by RM or EDF. The TO scheduler, together with the concurrency control module, orders operations from transactions such that the priority of the transaction is considered, i.e., operations from a higher prioritized transaction have precedence over operations from lower prioritized transactions. The central repository stores data items (and possibly several versions of them) and meta-information in main-memory.

4.2 Data Freshness

One approach to achieve an adaptive tuning of the updating frequencies of data items, as used in this research project, is to define data freshness in the value domain. The rationale behind this decision is that time usually is a bad approximation of how much data values change between succeeding calculations since the allowed age of a data value needs to be set to correspond to the worst case change of that data value. Depending on the application and system state, it is not certain that the value changes that much all the time, and, hence, an age does not reflect the true freshness of a data item. Similarity defines the relation between values of a data item by the relation $f$ (described in the section Consistency (section 2.2.2)).

In this thesis two different similarity relations are used. One considers fixed validity intervals in the value domain of data items, i.e., the value domain is divided into fixed intervals and values falling within the same interval are similar. The other relation uses a flexible validity interval that is based on one value of a data item as the origin of the interval, i.e., all values that are within a given distance to the origin are similar to it. Figure 4.2 shows the distinction between the two similarity functions.

The flexible interval is defined as follows:

**Definition 4.2.1 (Flexible Data Validity Interval).** Each pair $(d, x)$, where $d$ is a derived data item and $x$ is an item from $R(d)$, has a data validity interval, denoted $\delta_{d,x}$, that states how much the value of $x$ can change before the value of $d$ is affected.
4.2. Data Freshness

Hence, the similarity relation $f$ is equal to $f : \mathbb{x}_{\text{old}} \times \mathbb{x}_{\text{new}} \rightarrow |v(\mathbb{x}_{\text{old}}) - v(\mathbb{x}_{\text{new}})| \leq \delta_{d,\mathbb{x}}$, where $v(\mathbb{x}_{\text{old}})$ is the value of $\mathbb{x}$ used when $d$ was previously derived and $v(\mathbb{x}_{\text{new}})$ is the current value of $\mathbb{x}$ stored in the repository. The similarity relation $f$ is applied to the parents of a data item. For instance, in figure 4.2(b) a new interval is created if the new value is outside the old validity interval.

The fixed interval is defined as follows:

Definition 4.2.2 (Fixed Data Validity Interval). Let $\text{fixedint}_x$ be a function mapping values of a data item $\mathbb{x}$ to natural integers, i.e., $\text{fixedint}_x : \mathbb{D} \rightarrow \mathbb{N}$, where $\mathbb{D}$ is the domain of data item $\mathbb{x}$. All values of $\mathbb{x}$ mapping to the same interval are similar.

The similarity relation is, thus, $f : \mathbb{x}_{\text{old}} \times \mathbb{x}_{\text{new}} \rightarrow \text{fixedint}_x(v(\mathbb{x}_{\text{old}})) = \text{fixedint}_x(v(\mathbb{x}_{\text{new}}))$. One example of the function $\text{fixedint}$ is:

$$\text{fixedint}_x(v(\mathbb{x})) = \left\lfloor \frac{v(\mathbb{x})}{64} \right\rfloor.$$

A value of a data item is fresh when all the values it is derived from are fresh. The freshness of a data item with respect to one of its read set members is defined as follows using flexible validity intervals.

Definition 4.2.3 (Data Freshness Based on One Parent Using Flexible Validity Intervals). Let $d$ be a derived data item and $\mathbb{x}$ a data item from $R(d)$, and $v^t_x$, $v^{t'}_x$ be values of $\mathbb{x}$ at times $t$ and $t'$ respectively. Then $d$ is fresh with respect to $\mathbb{x}$ when $|v^t_x - v^{t'}_x| \leq \delta_{d,\mathbb{x}}$.

Hence, the value of $\mathbb{x}$ used when deriving $d$ is the origin of a data validity interval, and as long as values of $\mathbb{x}$ lies within this interval, the value of $\mathbb{x}$
does not affect the value \( d \). Using fixed validity intervals, the freshness of a data item with respect to one of its read set members is as follows.

**Definition 4.2.4 (Data Freshness Based on One Parent Using Fixed Validity Intervals).** Let \( \text{fixedint}_x \) be a function mapping values of a data item \( x \) to natural integers, i.e., \( \text{fixedint}_x : D \rightarrow \mathbb{N} \), where \( D \) is the domain of data item \( x \). Let \( d \) be a derived data item and \( x \) a data item from \( R(d) \), and \( v^t_x \) and \( v^{t'}_x \) be values of \( x \) at times \( t \) and \( t' \) respectively. Then \( d \) is fresh with respect to \( x \) when \( \text{fixedint}_x(v^t_x) = \text{fixedint}_x(v^{t'}_x) \), i.e., as long as \( x \) is mapped to the same interval its value is considered to be unchanged.

Note that previous definitions (definitions 4.2.3 and 4.2.4) define data freshness with respect to one parent. We need to define data freshness for a data item with respect to all its parents. An observation is that as long as the values used when deriving data item \( d \) are similar to values that would be read during a rederivation of \( d \), we say that \( d \) is fresh. Using the definition of freshness with respect to one parent using flexible validity intervals (definition 4.2.3) on every data item in \( R(d) \), data freshness is defined as follows.

**Definition 4.2.5 (Data Freshness of a Data Item Using Flexible Validity Intervals).** Let \( d \) be a derived data item derived at time \( t \) using values of data items in \( R(d) \). Then \( d \) is fresh at time \( t' \) if it is fresh with respect to all data items from \( R(d) \), i.e.,

\[
\land_{x \in R(d)} \left\{ |v^t_x - v^{t'}_x| \leq \delta_{d,x} \right\}
\]  
(4.1)

evaluates to true. Thus, recomputation of \( d' \) is not necessary.

Using definition of data freshness with respect to one parent using fixed validity intervals (definition 4.2.4) on every data item in \( R(d) \), data freshness is defined as follows.

**Definition 4.2.6 (Data Freshness of a Data Item Using Fixed Validity Intervals).** Let \( d \) be a derived data item at time \( t \) using values of data items in \( R(d) \). Then \( d \) is fresh at time \( t' \) if it is fresh with respect to all data items from \( R(d) \), i.e.,

\[
\land_{x \in R(d)} \left\{ \text{fixedint}_x(v^t_x) = \text{fixedint}_x(v^{t'}_x) \right\}
\]  
(4.2)

evaluates to true.
There are slight differences in handling validity intervals as fixed or flexible. One can be seen in figures 4.2(a) and 4.2(b). At the highest peak of the plot, sampling number two is close to reading this value, sampling number two and three map to different intervals in figure 4.2(a), but in figure 4.2(b) sampling two and three map to the same interval. On the other hand, sampling six and seven map to the same interval in figure 4.2(a) but not so in figure 4.2(b).

The two different ways of defining data validity intervals are intuitive in different ways. Mapping to fixed intervals has better support for being implemented as entries into tables since an entry into a table is a fixed interval. The flexible interval is intuitive in that it is easy to reason about changes in values relative to an already stored value. However, a new value of a data item and the distance from it might cover several entries in a table. Hence, there are applications where one way to define data freshness fits better than the other. The following example reflects this.

**Example 4.1:** Flexible validity intervals are used on water temperature, and the application considers all changes within 5 degrees to be similar. At the temperature 97°C, this means that we can accept changes to 102°C. Such a temperature on water does not even exist, and the flexible validity interval does not reflect this. Therefore a division of the possible temperatures into fixed intervals might be a better similarity relation to use.

### 4.2.1 Example of Data Freshness in Value Domain

Now we give an example on how changes in the value of a data item affect other data items.

**Example 4.2:** An update of a data item \( d \) is only needed if the data item is stale, i.e., at least one of its parents has changed such that the update might result in a different value compared to \( d \) that is stored in the database. A data item can have several ancestors on a path to a base item in a data dependency graph \( G \). For instance, one possible path from \( d_9 \) to \( b_6 \), denoted \( P_{d_9-b_6} \), in figure 2.5 is: \( d_9, d_7, d_2, b_6 \). When a data item is updated it may make its neighbors in \( G \) stale (this can be checked using a definition of data freshness with respect to one parent). If the update makes a data item \( d \) stale, then all descendants of \( d \) are possibly stale since a recalculation of \( d \) may result in a new value of \( d \) that does not affect its descendants. Using the path \( P_{d_9-b_6} \), consider an update of \( b_6 \) making \( d_2 \) stale. Data items \( d_7 \) and \( d_9 \) are potentially affected and a recalculation of \( d_2 \) is needed and when it has finished it is possible to determine if \( d_7 \) is stale, i.e., affected by the
4.2.2 Marking of Changed Data Items

Example 4.2 shows that a data item is potentially affected by a change in a predecessor of it. The validity of a potentially affected data item \( d \) can only be determined by recomputing of all potentially affected data items on paths leading to \( d \). Every derived data item \( d \) has a timestamp, \( pa(d) \), indicating the latest logical time the data item was found to be (potentially) affected by a change in another data item.

An updating scheme can be used to determine which data items that are (potentially) affected by a change of the value of a data item. Two updating schema are presented below and both consist of three distinct and generalized steps. The steps are:

- **S1** Update base items to always keep them up-to-date.
- **S2** Mark data items as (potentially) affected by a change in a data item.
- **S3** Determine which data items should be updated before a UT starts to execute. This step is an on-demand step as a response to the arrival of a UT. A schedule of updates is generated and the scheduled updates are executed before the arrived UT starts to execute.

The first updating scheme is constructed to handle potentially affected data items. It is denoted potentially affected updating scheme (PAUS) and contains of the steps PAUS_S1, PAUS_S2, and PAUS_S3.

In the first step (PAUS_S1) all base items are updated with fixed frequencies such that the base items are always fresh. When a base item \( b \) is updated, the freshness according to the definitions of data freshness with respect to one parent (definitions 4.2.3 and 4.2.4) is checked for each child of \( b \) in data dependency graph \( G \). Thus, in our example, base items \( b_1 \sim b_9 \) from figure 2.5 are updated with fixed frequencies, e.g., base item \( b_3 \) is updated, then \( d_1 \) is checked if it is still fresh.

The second step (PAUS_S2) is performed when a data item \( d \) is found to be stale due to the new value of parent \( x \), where \( x \) can be either a base item or a derived item. Data item \( d \) and all its children are marked as potentially affected by the change in \( x \). The \( pa \) timestamp is set to \( \max(pa(x), ts(\tau)) \), where \( x \) is \( d \) or one of its children, and \( ts(\tau) \) is the logical timestamp of the
4.2. Data Freshness

transaction updating \( x \). This means that every data item is marked with the latest timestamp of a transaction that makes the data item potentially affected by the written value produced by the transaction.

The third step (PAUS_S3) is an on-demand step and occurs every time a UT starts to execute. The data items the UT reads must be valid. Hence, all potentially affected data items on paths to \( d_{UT} \) need to be considered for being updated. Also in this step, when an update has updated a data item \( d \), the timestamp \( pa(d) \) is set to zero if \( d \) is not potentially affected by changes in any other data item, i.e.,

\[
pa(x) = \begin{cases} 
0 & \text{if } ts(\tau) \geq pa(x) \\
pa(x) & \text{otherwise.}
\end{cases}
\]

(4.3)

Every executed TU ends with marking any potentially affected data items, i.e., after an update of data item \( d \) all potentially affected children have their \( pa \) timestamps set to values reflecting that they are affected by a change in \( d \). Thus, step PAUS_S2 is used for every single transaction including updates.

Data dependency graphs can be traversed bottom-up or top-bottom. Using a bottom-up traversal it is impossible to determine if potentially affected data items are stale or not. The reason is that parents of a potentially affected data item need to be recalculated. However, staleness can be decided using the \( pa \) timestamp and a function that returns the maximum possible deviation under a given time duration. If \( pa(d) > 0 \), the data item \( d \) is potentially affected, and if the deviations, at a given time point, of its parents affect \( d \) then \( d \) is assumed to be stale. The maximum deviation of data item \( d \) at time \( t \) is given by a function \( \text{error}(d, t) \).

In every branch from \( d_{UT} \), scheduled updates execute top-bottom to obey precedence constraints. If scheduling is done top-bottom, then S2 can mark affected data items since they are found in top-bottom scheduling step. Using a bottom-up traversal, S2 must mark potentially affected data items so they are scheduled in S3 and the affected data item is finally found. The marking of affected data items works as a relevance check. Only the data items marked as affected need to be updated. Formally we have the following definition.

**Definition 4.2.7 (Relevance Check).** Let \( \tau_{UT} \) be an update for data item \( d_{UT} \). Assuming transactions are deterministic and time invariant, the relevance of executing \( \tau_{UT} \) is determined by checking whether \( d_{UT} \) is affected by a change in a parent, i.e., marked as affected.
The PAUS updating scheme needs to be slightly changed to support top-bottom scheduling of updates and relevance checks. The new scheme is denoted affected updating scheme (AUS). Step AUS_S1 is exactly the same as PAUS_S1. Step AUS_S2 assigns \( pa(x) = \max(pa(x), ts(\tau)) \) of the data item \( x \) that is found to be affected by an update made by transaction \( \tau \).

Thus, only children in one level below the changed data item is marked as opposed to PAUS_S2 where all children are marked as potentially affected. An affected data item is marked with the timestamp of the latest transaction producing a value that affects the data item. Step AUS_S3 is an on-demand step as PAUS_S3, and since \( G \) is traversed top-bottom only those data items with \( pa > 0 \) are affected and need to be updated. In AUS_S3, when a data item is updated, \( pa \) of that data item is set according to equation 4.3, and affected data items are marked according to step AUS_S2. Hence, step AUS_S3 consists of building a top-bottom schedule and only execute those data items with \( pa > 0 \).

In summary, the AUS updating scheme supports a relevance check.

**Correctness of Determining Potentially Affected Data Items**

Steps PAUS_S1–PAUS_S3, discussed above, give a mechanism to determine if a data item is potentially affected by a change in any of its parents.

Using updating scheme PAUS, the timestamp \( pa \) guarantees the possibility to determine if a data item is potentially affected by a change in any of its parents. A data item is potentially affected if its \( pa \) timestamp is greater than zero. This can be shown using the following reasoning. If a child is stale then \( pa \) is set to the maximum of \( pa \) of the data item and timestamp of the transaction (PAUS_S2). When a transaction commits (user or triggered) the \( pa \) timestamp of the installed data item is set to zero only if the update has a higher timestamp than \( pa(x) \), i.e., the update is issued after the item was found to be stale (PAUS_S3) since the timestamps are increasing monotonically. Hence, timestamp \( pa \) then holds the timestamp of the transaction that last made the data item stale or zero if the item is fresh. Thus, if \( pa \) is zero then the data item is up-to-date, and if \( pa > 0 \) then the data item is potentially affected by another data item.

Using updating scheme AUS, timestamp \( pa \) of a data item \( x \) guarantees the possibility to determine if \( x \) is affected by a change in a parent. This can be shown in the following way. The timestamp \( pa(x) \) can only have a

\[ \text{We choose to keep the notion potentially affected on the timestamp because it is easier to have only one timestamp to reason about.} \]
value greater than zero if step AUS_S2 found the data item to be affected by a change in a parent. The only way the timestamp can be set to zero is if \(ts(\tau) \geq pa(x)\), i.e., a data item is considered fresh if an update is issued after the data item was found to be stale. In step AUS_S3, an update of data item \(x\) in a top-bottom schedule is executed if \(pa(x) > 0\). Thus, a stale data item is guaranteed to have \(pa > 0\).

It should be noted that using binary flags indicating whether data items are potentially affected cannot guarantee a correct mapping from a potentially affected data item to true on the binary flag [22–24]. Step PAUS_S2 was merely to set the change flag of \(d\) and its children to true. Step PAUS_S3 was to set the change flag of the updated data item to false. The following example shows that it is not always possible to correctly map from a potentially affected data item to true on the binary flag.

**Example 4.3:** Consider data item \(d_7\) in figure 2.5 and \(\text{change}(d_7)\) equals true. A triggered update \(\tau_{d_7}\) has started to update \(d_7\) because \(\text{change}(d_7)\) equals true (and \(pa(d_7) > 0\)). Assume a UT starts for another data item in the system and the UT has a higher priority than \(\tau_{d_7}\). Meanwhile, \(d_7\) is again marked as potentially affected by a change in one of the base items \(b_5\) and \(b_6\). Using the \(pa\) timestamp means that \(pa\) is set to a new higher value since the transaction updating the base item has a higher timestamp, and this timestamp is propagated to \(d_7\). Using the \(\text{change}\) flag, when \(\tau_{d_7}\) commits and writes \(d_7\), the change flag is reset to false. Hence, the system now believes that \(d_7\) is not potentially affected by changes in any of its parents, which is not a correct mapping.

**Complexity of Constructing a Schedule of Updates**

Before we continue with a description of algorithms that implement step three of the PAUS or the AUS updating schema, we discuss computational complexity of building schedules of updates. To determine which data items that need to be updated, the data dependency graph \(G\) can be traversed either top-bottom from base items to \(d_{UT}\) or bottom-up from \(d_{UT}\) to base items. If we assume that a node in \(G\) knows about its immediate parents and children, then during a top-bottom traversal of \(G\) a search in \(G\) is needed to decide if a data item is a parent of \(d_{UT}\). In a bottom-up traversal from \(d_{UT}\), every reached node is a parent of \(d_{UT}\) and no searching is needed. Thus, the computational cost of determining parents of a data item is cheaper for bottom-up traversal than for top-bottom traversal.

Another possibility to searching is to store relation \(<_G\) in memory with
Updating and Concurrency Control Algorithms

O(1) time for memory accesses to check $x <_G y$, $x, y \in G$. However, the memory consumption can be too great. For instance, for the graph in figure 4.3 the number of elements needed to be stored in the lookup-table is $m^2(k-1) + m^2(k-2) + \cdots + m^2 = m^2k(k-1)/2$. This grow quadratically with the size of the graph, which might be too much for some systems. Remember that the focus of this work is on real-time embedded systems.

In summary, since the performance of algorithms that are executed often is of utmost importance, a bottom-up traversal or a top-bottom traversal in a pregenerated schedule must be used to reduce searching.

4.3 On-Demand Updating Algorithms in Time Domain

This section describes on-demand updating algorithms presented by Ahmed and Vrbsky [7]. These algorithms use different predicates $pred$ in the definition of on-demand triggering (definition 2.2.3) to divide updating algorithms into consistency-centric and throughput-centric. Every time a data item is requested by a read operation, condition 2.1 is checked. If the condition is evaluated to true, the database system starts triggering a transaction\(^2\) that updates the data item the read operation is about to read. The on-demand algorithm using condition 2.1 is denoted OD. The triggering criterion can be changed to increase the throughput of UTs. Ahmed and Vrbsky present three options of triggering criteria [7]. These are (i) no option, which represents OD, (ii) optimistic option, where an update is only triggered if it can fit in the slack time of the transaction that does the read operation (denoted ODO), and (iii) knowledge-based option, where an update is triggered if it can fit in the slack time when the remaining response time of the transaction has been accounted for (denoted ODKB).

Formally, the triggering criteria for (i)–(iii) above are [7]:

(i): $current\_time - timestamp(x) \leq avi(x)$

(ii): $(current\_time - timestamp(x) \leq avi(x))$
     $\land (dl(\tau) - at(\tau) - wcet(\tau) \geq 0)$

(iii): $(current\_time - timestamp(x) \leq avi(x))$
      $\land (dl(\tau) - at(\tau) - wcet(\tau) - rr(\tau) \geq 0)$, where $rr(\tau)$ is the remaining

\(^2\)A transaction is triggered by the database system by creating a new transaction instance having the same priority as the triggering transaction.
response time of the transaction $\tau$, and is calculated in the following way:

\[
\text{wait_factor} = \frac{\text{wait_time}}{(\# \text{ executed operations})}
\]

\[
\text{rr} = \text{wait_factor} \times (\# \text{ remaining operations in UT + } \\
\# \text{ operations in TU})
\]

and \text{wait_time} is the time the UT has been waiting so far for resources, e.g., the CPU.

Data freshness defined in the value domain can be used together with these triggering criteria. The current time $- \text{timestamp}(x) \leq \text{avi}(x)$ part is the definition of data freshness (definitions 4.2.5 and 4.2.6). Using these criteria the algorithms are denoted OD$_V$, ODO$_V$, and ODKB$_V$.

Next we start describing the proposed updating algorithms. Table 4.1 gives an overview of all updating algorithms presented in this thesis.

**Computational Complexity**

The computational time complexity of OD, ODO, and ODKB grows polynomially with the size of the graph. ODO and ODKB do not generate triggered updates when they cannot fit in the remaining slack time. Thus, since execution times are finite and have approximately the same size, then ODO and ODKB schedule a polynomial number of updates. Checking \text{pred}, which takes polynomial time, precedes every scheduled update.

If we assume updated data items do not need to be updated again during the execution of a UT, then \text{pred} is checked at maximum once for every data item. Since there is a finite fixed finite number of data items the computational complexity of OD is polynomial. However, if we assume that every read operation needs to be preceded by an update of the data item, the computational complexity of OD grows exponentially with the size of the graph. Consider the graph in figure 4.3. The number of paths from $d$ at level $k$ to a node at level $k - n$, $0 \leq n < k$, is $m^{(k-n-1)}$. Making updates for data items then involve $m \times m^{(k-1)} = m^k$ checks of \text{pred} which takes exponential time.

**4.4 ODDFT Updating Algorithm**

This section describes the on-demand depth-first traversal (ODDFT) algorithm that implements the on-demand scheduling of updates in step
Table 4.1: A summary of updating algorithms.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Long name</th>
<th>Relevance check</th>
<th>Time domain</th>
<th>Value domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>OD</td>
<td>On-demand</td>
<td></td>
<td>supported</td>
<td></td>
</tr>
<tr>
<td>ODO</td>
<td>On-demand with optimistic option</td>
<td></td>
<td>supported</td>
<td></td>
</tr>
<tr>
<td>ODKB</td>
<td>On-demand with knowledge-based option</td>
<td></td>
<td>supported</td>
<td></td>
</tr>
<tr>
<td>OD_V</td>
<td>OD with value domain</td>
<td></td>
<td>supported</td>
<td></td>
</tr>
<tr>
<td>ODO_V</td>
<td>ODO with value domain</td>
<td></td>
<td>supported</td>
<td></td>
</tr>
<tr>
<td>ODKB_V</td>
<td>ODKB with value domain</td>
<td></td>
<td>supported</td>
<td></td>
</tr>
<tr>
<td>ODKB_C</td>
<td>On-demand with knowledge-based option and relevance check</td>
<td>supported</td>
<td>supported</td>
<td></td>
</tr>
<tr>
<td>ODDFT</td>
<td>On-demand depth-first traversal</td>
<td></td>
<td>supported</td>
<td></td>
</tr>
<tr>
<td>ODBFT</td>
<td>On-demand breadth-first traversal</td>
<td></td>
<td>supported</td>
<td></td>
</tr>
<tr>
<td>ODTB</td>
<td>On-demand top-bottom traversal with relevance check</td>
<td>supported</td>
<td>supported</td>
<td></td>
</tr>
<tr>
<td>ODDFT_C</td>
<td>On-demand depth-first traversal with relevance check</td>
<td>supported</td>
<td>supported</td>
<td></td>
</tr>
</tbody>
</table>
PAUS\_S3. The goal of step PAUS\_S3 is to create a schedule of updates that, when executed, make data items fresh before the UT continues to execute. The algorithmic steps of ODDFT for achieving this are: (i) traverse $G$ bottom-up using depth-first order, (ii) in each reached node determine if the corresponding data item needs to be updated, and (iii) put needed updates in a schedule.

Algorithmic step (i) of ODDFT is realized by recursively visiting every parent of a node. In this way, $G$ is traversed bottom-up in depth-first order.\footnote{The algorithm PREC1 on page 9 is implemented in this way.} In algorithmic step (ii) of ODDFT, every reached node in step (i) needs to be considered for being updated. The PAUS updating scheme makes the $pa$ timestamps available for determining potentially affected data items. Ideally, only stale data items should be put in the schedule of updates in algorithmic step (iii). However, as mentioned, that is impossible in a bottom-up traversal. Algorithmic step (iii) can be realized by using a worst-case value change of data items together with the $pa$ timestamp to determine if potentially affected data items are stale.

The ODDFT algorithm is given in figure 4.4. The input parameters
ODDFT\( (d, t, UTrl. \text{freshness\_deadline}) \)

1: for all \( x \in R(d) \) in prioritized order do
2: \hspace{1em} if \( pa(d) > 0 \land error(x, \text{freshness\_deadline}) > \delta_{d,x} \) then
3: \hspace{2em} if Skip late updates and \( t - \text{wcet}(\tau_x) \times \text{blockingf} < at(\tau_d) \) then
4: \hspace{3em} break
5: \hspace{2em} else
6: \hspace{3em} Put \( \tau_x \) in schedule \( S_{d_{UT}} \). Check for duplicates and remove any.
7: \hspace{3em} rl(\tau_x) = t - \text{wcet}(\tau_x) \times \text{blockingf}
8: \hspace{3em} dl(\tau_x) = UTrl
9: \hspace{3em} ODDFT(\( x, rl(\tau_x), UTrl, \text{freshness\_deadline} \))
10: \hspace{2em} end if
11: end if
12: end for

Figure 4.4: The ODDFT algorithm.

are \( d \), the data item that might be included in the schedule, \( t \), the release time of an update, \( UTrl \), which is the release time of the UT, and \( \text{freshness\_deadline} \), which is the earliest time a data item should be valid needing no update. \( \text{freshness\_deadline} \) is set to the deadline of the arrived UT. The variable \( \text{blockingf} \) on line 7 is used to regard interruptions from higher prioritized transactions and their updates.

Algorithmic step (i) is implemented by the for-loop on line 1 and the recursive call on line 9. Algorithmic step (ii) is implemented with the if-statement on line 2, where \( error(x, \text{freshness\_deadline}) \) is a worst-case value change of \( x \) at time \( t \) from the value previously used when deriving \( d \). If \( d \) is stale, then algorithmic step (iii) is implemented with lines 6–9. Line 7 calculates the latest possible release time of the update updating \( d \), and line 8 sets the deadline of the update.

The ODDFT algorithm can be adjusted to skip scheduling updates whose calculated release times are earlier than the release time of the UT executing ODDFT. The if-statement on line 3 implements this check.

ODDFT is now described by an example using \( G \) in figure 2.5.

**Example 4.4:** Assume a UT, deriving the total multiplicative fuel factor \( d_9 \), arrives and that the temperature compensation factor and the start enrichment factors \( (d_8, d_5, d_7, d_2, d_3, d_4) \) are marked as potentially affected. Now, the two parents of \( d_9, d_7 \) and \( d_8 \), have \( pa \) set to values greater than zero. Moreover, if \( error(x, t) > \delta_{d_7,x} \) evaluates to true for
4.4. ODDFT Updating Algorithm

some \( x \in \{d_2, d_3, d_4\} \), then \( d_7 \) needs to be updated. Assume both \( d_7 \) and \( d_8 \) need to be updated. The algorithm then chooses the one with highest error by evaluating \( error(d_7, t) \) and \( error(d_8, t) \), and continues with the chosen branch. If \( d_7 \) has the highest error, then an update \( \tau_{d_7} \) is put in the schedule followed by updates for \( d_2, d_3, \) and \( d_4 \) according to a prioritization. Finally, the algorithm continues with the \( d_8 \) branch and \( \tau_{d_8} \) is put in the schedule followed by \( \tau_{d_5} \). The total schedule is \([\tau_{d_5}, \tau_{d_8}, \tau_{d_4}, \tau_{d_3}, \tau_{d_2}, \tau_{d_7}]\) and is shown in figure 4.5. Every update is tagged with the latest possible release time and deadline by accounting for WCETs of added updates in the schedule. When the release time of an update is earlier than the arrival time of UT \( \tau_{d_9} \) the algorithm is terminated since no more updates can be executed (assuming WCETs on updates).

\[\begin{align*}
T(n) &= mT(n+1) + O(m \log m) \\
T(k) &= 1
\end{align*}\]

where \( O(m \log m) \) is the running time of an algorithm that prioritizes nodes and \( m \) is the maximum out-degree of a node. The total running time of algorithm ODDFT is \( O(m^n m \log m) \), where \( m \) is the maximum in-degree of a node in graph \( G \), and \( n \) is the number of levels in the graph. However, in reality data items do not have the relationship described in figure 4.3,
and, thus, the running time of the algorithm is probably polynomial with the size of the graph in realistic examples.

Note that if the if-statement on line 3 in the ODDFT algorithm is being used, then the execution time of ODDFT grows polynomially with the size of the slack of the user transaction. This is because by using line 3, ODDFT is stopped when there is no slack time left for updates. There can only be a polynomial amount of updates in the slack time since the execution time of updates and the slack time are of the same order of magnitude.

### 4.5 ODBFT Updating Algorithm

This section describes the ODBFT algorithm that implements the on-demand scheduling of updates in the PAUS_S3 step. Algorithmic step (i) is traversing $G$ bottom-up using breadth-first order. Algorithmic step (ii) is the same as for ODDFT, i.e., in each reached node determine if the corresponding data item needs to be updated. Step (iii) is also the same as for ODDFT, i.e., put needed updates in a schedule. Algorithmic step (i) is described now. The breadth-first algorithm (see [14]) is implemented by using a FIFO queue denoted $Q$ for determining from which node to continue to search for data items in $G$. This is not sufficient, instead the nodes should be picked in both level and priority order. Level order is used to obey the precedence constraints, and priority order is used to pick the most important update first. The relation $\sqsupseteq$ is introduced, and $x \sqsupseteq y$, where $x$ and $y$ are data items in the database, is defined as:

$$x \sqsupseteq y \text{ iff } level(x) > level(y) \vee (level(x) = level(y) \land prio(x) > prio(y)) \vee (level(x) = level(y) \land prio(x) = prio(y) \land id(x) > id(y)),$$

where $prio$ is the product of the priority of the data item and the weight, $level$ is the level the data item resides at, and $id$ is a unique identifier associated with the data item. If data item $d_4$ has the integer 4 as an identifier and data item $d_5$ the integer 5, then $d_5 \sqsupseteq d_4$ if they reside in the same level and is assigned the same priority.

In algorithmic step (ii), all nodes are initially colored white to represent unscheduled data items. Nodes that are inserted into $Q$ must be white and represent data items that need to be updated. When a data item is inserted into $Q$ it is considered scheduled and is colored gray. Every time a node of the data dependency graph is inserted into $Q$, the node is inserted in the
right position based on relation $\Box$. The head of $Q$ is used by ODBFT to start a new search for undiscovered nodes in the graph. Since $\Box$ orders the data items according to level, ODBFT behaves as a breadth-first search.

In algorithmic step (iii), the head of $Q$ is inserted into the schedule of updates. This is iterated as long as there are elements in $Q$. In this way only data items that are stale according to $pa > 0$ and the function $error$ are scheduled.

ODBFT($\tau$, $t$, $freshness\_deadline$)

1: Assign color WHITE to all nodes in the data dependency graph
2: Let $d$ be the data item updated by $\tau$
3: Put $d$ in queue $Q$
4: while $Q \neq \emptyset$ do
5: Let $u$ be the top element from $Q$, remove $u$ from the queue
6: Let $\tau_u$ be the transaction associated with $u$
7: $dl(\tau_u) = t$
8: $rt(\tau_u) = t - wcet(\tau_u) \times blockingf$
9: if Skip late updates and $rt(\tau_u) < at(\tau_{UT})$ then
10: break
11: end if
12: priority_queue = AssignPriority($u$, $t$)
13: $t = t - wcet(\tau_u) \times blockingf$
14: for all $v \in priority\_queue$ in priority order do
15: if $color(v) = WHITE$ then
16: $color(v) = GRAY$
17: if $pa(v) > 0 \land error(v, freshness\_deadline) > \delta_{d,v}$ then
18: Put $v$ in $Q$ sorted by relation $\Box$
19: end if
20: end if
21: end for
22: $color(u) = BLACK$
23: Put $\tau_u$ in the scheduling queue $S_{d_{UT}}$
24: end while

Figure 4.6: The ODBFT algorithm.

The ODBFT algorithm is described in figure 4.6. The input parameters are the arrived UT $\tau$, $t$ initially set to the $dl(\tau) = wcet(\tau)$, and $freshness\_deadline$ which is the earliest time a data item should be valid needing no update. $freshness\_deadline$ is set to the deadline of the ar-
rived UT. Algorithmic step (i) is implemented by lines 4, 5, 15, 16, and 18. The algorithm cycles through nodes put in $Q$ by using a while-loop (lines 4 and 5), inserted nodes are colored gray so they cannot be found again (lines 15 and 16), and nodes are inserted in $Q$ on line 18.

Algorithmic step (ii) is implemented by lines 12, 14, and 17. The AssignPrio algorithm used on line 12 is described in the Supporting Mechanisms and Algorithms section (section 4.9). The for-loop on line 14 cycles through all parents of a data item. The if-statement on line 17 checks if a found data item should be put in the schedule of updates. Algorithmic step (iii) is implemented on line 23.

ODBFT is described by an example using $G$ in figure 2.5.

Example 4.5: A UT that derives data item $d_9$ arrives, and $d_9$ is put into queue $Q$. Assume $d_2$–$d_9$ are marked as potentially affected by changes in base items. Parents $d_6$, $d_7$, and $d_8$ are put into $Q$ and the one with highest priority is picked first in the next iteration of the algorithm. Assume that $d_6$ is picked. Its parent $d_1$ has $pa$ set to zero and $d_1$ is not inserted into $Q$. Next, $d_7$ is picked from $Q$. Assume its whole read set is inserted into $Q$. The relation $\sqsubseteq$ sorts these items to be placed after $d_8$, that still resides in $Q$, because $\sqsubseteq$ orders items first by level and $d_8$ has a higher level than $d_2$–$d_4$. The next iteration picks $d_8$, which has the parent $d_5$ that is placed in $Q$. Since $d_5$ has the same level as $d_2$–$d_4$, they are already placed in $Q$ by an earlier iteration, the priority of the data items determine their order. None of $d_2$–$d_5$ has derived data items as parents so the algorithm finishes by taking the data items one by one and putting an update into the schedule. Thus, the resulting schedule is $[\tau_{d_2}, \tau_{d_3}, \tau_{d_4}, \tau_{d_5}, \tau_{d_6}, \tau_{d_7}, \tau_{d_8}]$. □

Computational Complexity

The total running time of algorithm ODBFT is $O(V + E)$ if the operations for enqueuing and dequeuing $Q$ take $O(1)$ time [14]. In algorithm ODBFT, the enqueuing takes in the worst case $O(\log V)$ since the queue can be kept sorted and elements are inserted in the sorted queue. The total running time of algorithm AssignPriority called by ODBFT is the same as the for-loop adding elements to a sorted queue, i.e., $O(E \log p)$, where $p$ is the maximum size of the read set of a data item. Thus, the algorithm has a total running time of $O(V \log V + E \log p)$. 
4.6 ODTB Updating Algorithm With Relevance Check

ODDFT and ODBFT use the PAUS updating scheme and use no relevance checks on scheduled updates. In this section the on-demand top-bottom traversal with relevance checks (ODTB) updating algorithm is presented. It is built upon the AUS updating scheme and ODTB has the following algorithmic steps. Algorithmic step (i) of ODTB is a top-bottom traversal of $G$ to find affected data items, and step (ii) of ODTB is a traversal from affected data items down to $d_{UT}$ and updates of the traversed data items are inserted in a schedule of updates.

In algorithmic step (i) of ODTB, the top-bottom traversal is done using a pregenerated schedule, because on-demand searching in the representation of data dependency graph $G$ is too time consuming. We discuss how a schedule can be pregenerated and used in ODTB. One schedule is stored, and it is generated by BuildPreGenSchedule (4.7). Corollary 4.6.2 shows that all top-bottom schedules that might be needed can be found inside this schedule. Furthermore, in our target application, the set of data items is fixed making it feasible to use a pregenerated schedule. Before corollary 4.6.2 is shown, we give some notations and a theorem.

Data dependency graph $G = (V, E)$ describes the relation $<_G$. To obtain a pregenerated schedule that can be used by ODTB, a bottom node is added, denoted bottom, to $V$ and all leaf nodes are connected to it by adding edges to $E$. A schedule is generated using BuildPreGenSchedule for the added bottom node\(^4\) and denote it $S$.

**Theorem 4.6.1.** It is always possible to find a sub-schedule of $S$ that is identical, with respect to elements and order of the elements, to a schedule $S_d$ starting in node $d$ and $S_d$ is generated by BuildPreGenSchedule.

**Proof.** Assume the generation of $S$ by BuildPreGenSchedule has reached node $d$. Start a generation of a schedule at $d$ and denote it $S_d$. BuildPreGenSchedule only considers outgoing edges from a node. Assume two invocations of BuildPreGenSchedule, which origin from the same node, always pick branches in the same order. BuildPreGenSchedule has no memory

\(^4\)The order of choosing branches can be arbitrary. In this thesis the order branches is chosen is from low data item numbers to high numbers, i.e., $b_1 < b_i < d_1 < d_j$ ($i > 1, j > 1$). If the order is important then weights can be assigned to each edge and a branch is chosen in increasing weight order.
BuildPreGenSchedule($d$)
for all $x \in R(d)$ in prioritized order do
  Put $\tau_x$ in schedule $S$
  $etime = etime + wcet(\tau_x)$ // etime is the cumulative execution time of scheduled updates.
  Annotate $\tau_x$ with $etime$
  BuildPreGenSchedule($x$)
end for

Figure 4.7: The BuildPreGenSchedule algorithm.

of which nodes that have already been visited. Hence, the outgoing edge that is picked by BuildPreGenSchedule generating $S$ is the same as BuildPreGenSchedule generating $S_d$ and, thus, there exists a sub-schedule $S$ that has the same elements and the same order as $S_d$.

Corollary 4.6.2. A schedule $S_d$ generated by BuildPreGenSchedule for data item $d$ with $l$ number of updates can be found in $S$ from index $start_d$ to index $stop_d$ where $l = |start_d - stop_d|$.

Proof. Follows immediately from theorem 4.6.1.

By corollary 4.6.2 it is always possible to get, from $S$, a sub-schedule of all possibly needed updates for data item $d_{UT}$ that a UT derives. Every data item has start and stop indexes indicating where its BuildPreGenSchedule schedule starts and stops within $S$. Every data item also knows about its neighbors (parents and children) in $G$. Every element in the schedule $S$, which is a data item, can be annotated with an execution time (line 4). The execution time is the cumulative execution time of all data items currently traversed by the algorithm (line 3). The execution time of a sub-schedule of $S$ is calculated by taking the annotated execution time of the start element minus the execution time of the stop index. The cumulative execution time of elements $bottom \ldots start_d$ is cancelled.

In algorithmic step (ii) of ODTB, the schedule of updates is created. In algorithmic step (i) a set of sub-schedules is found. Every sub-schedule has a start index which is $d_{UT}$ and a stop index that is the found affected data. The schedule of updates is constructed by determining which sub-schedules to include in the final schedule of updates.

The ODTB algorithm is shown in figure 4.8. Algorithmic step (i) of ODTB is implemented on lines 2, 3, and 4. The for-loop on line 2 cycles
4.6. ODTB Updating Algorithm With Relevance Check

ODTB\((d_{UT})\)
1: \(at = \text{deadline}(\tau_{UT}) - release\_time(\tau_{UT}) - wcet(\tau_{d_{UT}}) \times \text{blockingf}\)
2: \textbf{for all } \(x \in R(d_{UT})\) \textbf{do}
3: \hspace{1em} Get schedule for \(x, S_x,\) from \(S\)
4: \hspace{1em} \textbf{for all } \(u \in S_x\) \textbf{do}
5: \hspace{2em} \textbf{if } \text{pa}(u) > 0 \textbf{ then}
6: \hspace{3em} wcet\_from\_u\_to\_x = (\text{WCET of path from } u \text{ to } x) \times \text{blockingf}
7: \hspace{2em} \textbf{if } wcet\_from\_u\_to\_x \leq at \textbf{ then}
8: \hspace{3em} \text{Add data items } u \text{ to } x \text{ to schedule } S_{d_{UT}}. \text{ Calculate release times and deadlines.}
9: \hspace{3em} at = at - wcet\_from\_u\_to\_x
10: \hspace{2em} \textbf{else}
11: \hspace{3em} Break
12: \hspace{2em} \textbf{end if}
13: \hspace{1em} \textbf{end if}
14: \hspace{1em} \textbf{end for}
15: \hspace{1em} \textbf{end for}

Figure 4.8: Top- Bottom relevance check algorithm (pseudo-code).

\begin{table}[h]
\centering
\begin{tabular}{c}
\hline
\textbf{Schedule} & \textbf{Index} \\
\hline
s_1 & 0 \\
\hline
s_2 & 1 \\
\hline
s_3 & 2 \\
\hline
s_4 & 3 \\
\hline
s_5 & 4 \\
\hline
s_6 & 5 \\
\hline
s_7 & 6 \\
\hline
s_8 & 7 \\
\hline
\end{tabular}
\end{table}

Figure 4.9: Pregenerated schedule by BuildPreGenSchedule for \(G\) in figure 2.5.

through all parents of \(d_{UT},\) and for every parent a sub-schedule is fetched from \(S\) on line 3. The fetched sub-schedule is traversed top-bottom with the for-loop on line 4. Algorithmic step (ii) of ODTB is implemented on lines 5, 7, and 8. The \text{pa} timestamp of a data item in the sub-schedule is checked on line 5. If it is stale, then it is determined on line 7 if the remainder of the sub-schedule should be inserted in the schedule of updates. The remainder of the sub-schedule is copied on line 8.

Note that using line 7 in the ODTB algorithm makes it impossible to guarantee fresh values on data items since needed updates might not be put in the schedule of updates. Using ODTB together with multiversion concurrency control algorithms, this line is changed into if(1).

Next we give an example of using ODTB.

\textbf{Example 4.6:} A UT \(\tau_{d_7}\) arrives to a system that has a data dependency
Updating and Concurrency Control Algorithms

graph as given in figure 2.5. The fixed schedule \( S \) is given in figure 4.9, indexes for starts and stops within schedule \( S \) for \( d_7 \) are 2 and 5, i.e., schedule \( S_{d_7} \) is the sub-schedule that spans the indexes 2 through 5 in \( S \). For every parent \( x \) of \( d_7 \) (\( d_2, d_3, \) and \( d_4 \)) the schedule \( S_{d_x} \) is investigated from the top for a data item with \( pa > 0 \) (see figure 4.8). If such a data item is found, WCET for the data item \( u \) and the remaining data items in \( S_x \), denoted \( \text{wcet\_from\_u\_to\_x} \), has to fit in the available time \( \text{availt} \) of \( \tau_{d_7} \). The execution time of the updates can be stored in the pregenerated schedule by storing, for each update, the cumulative execution time of all updates up to and including itself. By taking the difference between two updates from \( S_{d} \) the cumulative part of the update for \( d \) is cancelled and the result is the execution time between the updates. When ODTB is finished the schedule \( S_{d_{UT}} \) contains updates that can be executed in the interval between the current time until the deadline of UT.

\[ \square \]

Computational Complexity

This algorithm is built on the same traversal of \( G \) as ODDFT, i.e., a depth-first order. A BuildPreGenSchedule pregenerated schedule is traversed for every parent of \( d_{d_{UT}} \). There are a polynomial number of parents to a data item, but, as described for ODDFT, the schedule can contain exponentially, in the size of the graph, many elements. Thus, the pregenerated schedule can also contain exponentially, in \( |V| \), number of updates. In the worst case, all of these updates need to be checked and, thus, ODTB has exponential complexity in the size of the graph. However, every step of ODTB is cheaper than for both ODDFT and ODBFT, since the only thing the algorithm is doing is reading values from arrays and copying values between arrays.

The algorithm ODTB traverses a pregenerated schedule top-bottom and if a stale data item is found the remaining part of the schedule is put in a schedule of updates. Some of these items might be fresh and unrelated to the found stale data item, i.e., they are unnecessary updates. Duplicates of a data item can be placed in the schedule. Checks for detecting these two issues can be added to the algorithm but this is not done in this thesis, because the target platform is an EECU and, thus, the overhead of CPU usage should be kept small.

ODTB takes the longest time to execute when none of the data items in the schedule is stale. One way to address this is to have two \( pa \) timestamps, one that indicates a stale data item and one that indicates that none of the parents are changed. These two timestamps are a combination of the second
4.7 ODDFT_C Updating Algorithm With Relevance Check

The bottom-up and top-bottom approaches can be combined into an algorithm denoted on-demand depth-first traversal with relevance check (ODDFT_C). A relevance check is added to ODDFT that checks if the parents of $d$ that an update will recalculate, make $d$ stale or not. If they do not make $d$ stale, then the update can be skipped. Thus, the ODDFT_C algorithm builds a schedule of updates by traversing the data dependency graph bottom-up. When the schedule is constructed, a relevance check is done before an update starts to execute. The relevance check is done by checking either equation 4.1 or equation 4.2 in the data freshness definitions.

Computational Complexity

This algorithm have the same complexity as ODDFT since the schedule of updates is generated by ODDFT.

4.8 ODKB_C Updating Algorithm With Relevance Check

This algorithm is the on-demand with knowledge-based option using data freshness defined in the value domain, i.e., ODKB_V, and to every scheduled update is a check done if the current value being updated is affected by any changes to data items in $G$. If the data item is unaffected by any changes, the update is not triggered. Hence, a relevance check is added to the triggering criterion.

Computational Complexity

ODKB_C has polynomial time complexity since the data freshness check has polynomial complexity and the check is applied to every scheduled up-
date generated by ODKB_V, and ODKB_V has polynomial time complexity.

4.9 Supporting Mechanisms and Algorithms

This section covers algorithms that describe how transactions are started, how updates are started, and how updates can be prioritized in the ODDFT, ODBFT, and ODDFT_C algorithms.

4.9.1 BeginTrans

The database system is notified by a UT when it starts to execute the BeginTrans algorithm. In this chapter, we only discuss the generation of timestamps and execution of updating algorithms. The algorithm is presented in figure 4.10 and \( gts \) is the global timestamp. As can be seen on line 2, the \( gts \) variable is monotonically increasing implying that UTs get unique timestamps.

\[
\begin{align*}
1: & \text{Begin critical section} \\
2: & gts = gts + 1 \\
3: & \text{End critical section} \\
4: & ts(\tau) = gts \\
5: & \text{Execute updating algorithm} \\
6: & \text{ExecTrans}(\tau_{UT})
\end{align*}
\]

Figure 4.10: Pseudo-code of the BeginTrans algorithm.

4.9.2 ExecTrans

This subsection describes the ExecTrans algorithm that implements PAUS_S2 and AUS_S2, triggers updates, and has the relevance check if an update is needed in ODDFT_C and ODTB.

There are two choices how to handle late updates. Remember that latest possible release time of an update is calculated in the updating algorithms. Either all updates are executed before the UT continues to execute, or late updates are skipped. Executing all updates means that the derived data item, \( d_{UT} \), is based on relatively consistent data. There is a risk that the UT can be finished too late if all updates are executed including
those that are late, i.e., the UT might miss its deadline. However, skipping late transactions, the derived data item might instead be based on stale data. The designer of the database system needs to choose one of these two approaches.

Line 2 implements the ability to skip late updates. If this functionality is unwanted, lines 2–4 are removed from ExecTrans. Lines 6–11 implement the relevance check, i.e., the current update is skipped if a value in the database is unaffected by changes in its parents. If the update \( \tau_x \) cannot be skipped, then the transaction is generated and started in line 12. Lines 16–25 implement the steps PAUS_S2 and AUS_S2.

4.9.3 AssignPrio

When the updating algorithms ODDFT, ODBFT, and ODDFT_C can choose from several nodes in \( G \), the AssignPriority algorithm (see figure 4.12), prioritizes the nodes, and the updating algorithm chooses branches in priority order. A function \( \text{error}(d, t) \) is used in AssignPriority to approximate the error in the stored value of \( d \) at time \( t \). This function is application-specific and can look as in figure 4.13 where the error is approximated by how much the value can possibly change during the duration until \( t \). Time \( t \) in AssignPriority is the future time at which data items should be fresh for the transaction to derive a fresh value [22]. The most natural value to assign to \( t \) is the commit time of the UT. However, when the schedule of updates is constructed, it is impossible to know the commit time of the transaction since it depends on the actual execution of the updates and other transactions. In this thesis, \( t \) is set to the deadline of the UT, i.e., the same as in an absolute system described by Kao et al. [35].

4.10 Multiversion Concurrency Control With Similarity

As mentioned earlier, it is important that the priority of transactions are obeyed, e.g., the HP2PL algorithm is a priority-aware extension of the 2PL algorithm. Also, requirement R3c states the importance to guarantee correct age on data items in a calculation. In this research project we investigate the approach to use multiple versions of each data item to address requirement R3c. Multiple versions have been used by other research groups for concurrency control in real-time systems, e.g., Nyström et al. use two
ExecTrans(τ)
1: for all $x \in S_{dUT}$ do
2:   if current_time > rt(x) then
3:     break
4:   end if
5:   $ts(\tau_x) = ts(\tau_d)$

-- Relevance Check --
6:   if ODDFT_C and $\forall y \in R(x)$, previously used value of $y$ is valid compared to new value of $y$ using a definition of data freshness with respect to one parent (definitions 4.2.3 or 4.2.4) then
7:     continue
8:   end if
9:   if ODTB and $pa(x) = 0$ then
10:     continue
11:   end if
12: Execute $\tau_x$
13: if $pa(x) < ts(\tau_x)$ then
14:   Set $pa$ of $x$ to 0
15: end if

-- Step PAUS_2 and AUS_S2 --
16: for all children of $x$ that have $x$ as parent do
17:   if child $c$ affected by change in $x$ then
18:     $pa(c) = \max(ts(\tau_x), pa(c))$
19:     if PAUS updating scheme then
20:       for all children $c$ of $x$ do
21:         $pa(c) = \max(ts(\tau_x), pa(c))$
22:       end for
23:     end if
24:   end if
25: end for
26: end for

Figure 4.11: Pseudo-code of the ExecTrans algorithm.
AssignPriority\( (d, t) \)
1: \textbf{for all } \( x \in R(d) \) \textbf{do}  
2: \hspace{1em} \textbf{if } error(x, t) \geq \delta_{d,x} \textbf{ then}  
3: \hspace{2em} total_error = total_error + error(x, t)  
4: \hspace{2em} \text{Put } x \text{ in queue } Q_1  
5: \hspace{1em} \textbf{end if}  
6: \textbf{end for}  
7: \textbf{for all } x \in Q_1 \textbf{ do}  
8: \hspace{1em} prio(x) = error(x, t)  
9: \hspace{2em} \text{Multiply } prio(x) \text{ with } weight(x)  
10: \hspace{2em} \text{Put } x \text{ in queue } Q_2 \text{ sorted by priority}  
11: \hspace{1em} \textbf{end for}  
12: \text{Return } Q_2

Figure 4.12: The AssignPriority algorithm.

\[ error(d, t) = (t - timestamp(d)) \times max\_change\_per\_time\_unit \]

Figure 4.13: Example of \( error \) function.

versions of each data item to divide data accesses made by hard real-time transactions from data accesses made by soft real-time transactions [51]. Their approach ensures that hard real-time transactions always can execute, but no guarantees on relative consistency can be made. Sundell and Tsigas develop a wait-free snapshot mechanism using multiple versions for sporadic real-time calculations [63]. Their approach guarantees correct age on data. However, similarity is not taken into consideration. In this section are multiversion timestamp ordering concurrency control with similarity algorithms proposed. They are denoted MVTO-S, MVTO-UV, MVTO-UP, and MVTO-CHECKSUM. The algorithms are view-similar to serial schedules.

4.10.1 Relative Consistency

It is important that calculations produce consistent results. In a database system this is achieved by introducing transactions as discussed in section Databases (section 2.2). Moreover, data items used by a transaction need to be relative consistent, which is described in section Consistency (section 2.2.2). Concurrency control algorithms, which are part of the TO scheduler, affect the relative consistency of a read-set of a transaction. The definition
of relative consistency used in this thesis is presented in definition 2.2.2 on page 14, and the set
\[ I = \bigcap_{\forall x \in RS} \{VI(x)\} \]
represents the time interval at which the data items in RS are valid. The set RS contains versions of read set members of the data item being updated. It is necessary that interval I lies within \( VI(\tau_{UT}) = (rt(\tau_{UT}), dl(\tau_{UT})) \) to have values that are consistent when the transaction is executing. The definition of relative consistency does not say anything about where in \( VI(\tau_{UT}) \) interval I has to be positioned, and, thus, it is sufficient with any I such that \( I \cap VI(\tau_{UT}) \neq \emptyset \). If versions of data items from different time intervals are available, then RS can contain versions such that \( I \cap VI(\tau_{UT}) \neq \emptyset \), i.e., the versions in RS are relatively consistent. The versions have logical timestamps equal to the logical timestamp of the transaction writing the version. The problem, as pointed out in the section Consistency (section 2.2.2), is how to construct RS, i.e., which version of a data item to choose. One intuitive way, that is also easy to implement and reason about, is to let \( \tau_{UT} \) read those versions that were valid when \( \tau_{UT} \) started. Furthermore, given that an updating algorithm schedules and executes all needed updates, and RS consists of proper versions, i.e., versions that were valid when the transaction started (see page 30), of the data items of \( R(d_{UT}) \), then interval I includes rt(\( \tau_{UT} \)). Thus, \( I \cap VI(\tau_{UT}) \) is always a non-empty set. Hence, transaction \( \tau_{UT} \) reads relatively consistent data (a snapshot of the database).

That \( \tau_{UT} \) reads relatively consistent data can be shown in the following way. A read operation is reading a proper version by choosing the version with the latest timestamp less than the timestamp of the transaction, i.e., the timestamp of the version is \( \max\{ts(x_k) | \forall x_k \in V(d_{UT}), ts(x_k) < ts(\tau_{UT})\} \), where \( V(d_{UT}) \) is the set of versions of \( d_{UT} \). Assume rt(\( \tau_{UT} \)) is not part of I = \( \bigcap_{\forall x \in RS} \{VI(x)\} \). This can happen in the following ways:

- The version \( x_k \) that the read operation of \( \tau_{UT} \) reads is not the latest version relative to \( \tau_{UT} \). This can only happen if another transaction installs a version \( x_{k+1} \) that \( \tau_{UT} \) should read instead. This is impossible since all necessary updates to make data items fresh are executed, and \( x_{k+1} \) would have been created before \( \tau_{UT} \) continues by an update.

- Some of the read operations could not read a proper version because such a version has been purged from the pool of versions. As a consequence, transaction \( \tau_{UT} \) must be restarted, getting a new timestamp,
and the updating algorithm is rescheduled. However, for successfully committed UTs this case cannot occur.

In summary, when $\tau_{UT}$ has done all read operations it is guaranteed to have read proper versions and they are all valid when $\tau_{UT}$ starts, i.e., $rt(\tau_{UT})$ is included in $I$ meaning that $I \cap VI(\tau_{UT}) \neq \emptyset$.

### 4.10.2 MVTO with Similarity

Requirement R3c, guaranteeing correct age on data items, can be resolved by using several versions of data items. Versions should be chosen such that they are valid at the same time. From a performance perspective it is important to have restart-free concurrency control algorithms as transactions being restarted have produced results that are not useful due to conflicts in the execution, i.e., resources are not utilized efficiently. Furthermore, transactions have to produce consistent results, i.e., be view-similar to a serial schedule.

The MVTO concurrency control algorithm, described in the section Multiversion Concurrency Control (section 2.4.2), transforms read operations of a data item into reading the version of a data item that has the largest timestamp less than the timestamp of the transaction. However, since data freshness is defined in the value domain and the concurrency control algorithm should work in conjunction with the updating algorithm some optimizations can be done to MVTO. They are:

- Since the concurrency control algorithm should work together with a bottom-up or a top-bottom updating algorithm, it must be possible to check if a scheduled update is needed.

- A version should not be created if it is similar to an already existing version.

Also the following requirement comes from an assumption (SA2 in chapter 3) that the available memory is limited, and, thus, all versions should not be kept in order to reduce memory consumption. Thus, occasionally versions need to be purged when the memory pool becomes full.

The algorithmic steps of a multiversion timestamp ordering with similarity (MVTO-S) are now described.

1. A global virtual timestamp is assigned $gvt = \min_{\forall \tau \in activeUT(\tau)} \{ts(\tau)\}$, where $activeUT(\tau)$ is the set of all active user transactions.
2. A schedule of needed updates is constructed atomically, i.e., uninterrupted by other transactions. Those data items with \(pa > 0\) are put in the schedule. The schedule of updates gives a snapshot-view of which data items need be updated at the time the transaction was started.

3. When a transaction enters its BOT operation, an investigation is done whether the version of the data item the transaction derives already exists; if it does, the transaction is skipped. The timestamp of version \(z\) is calculated as \(z = \max\{\text{ts}(x) | x \in R(d_{UT})\}\). Let \(y\) be the version of \(d_{UT}\) satisfying \(\max\{\text{ts}(x_k) | x_k \in V(d_{UT}), \text{ts}(x_k) < \text{ts}(z)\}\), i.e., \(y\) is the proper version of \(z\). A transaction can be skipped if

\[
\forall p \in R(d_{UT}), f(v(p, y), v(p, z)) = \text{true},
\]

where \(v(p, y)\) is the value of \(p\) when \(y\) was derived, and \(v(p, z)\), the value of \(p\) that \(\tau_{UT}\) will read if the transaction continues to execute. The similarity relation \(f\) returns true if the values are similar.

4. A new version of a data item \(x\) is created when a transaction writes a data item \(x\), \(\text{ts}(\tau) > g\text{vts}\) and the timestamp the version of \(x\) would get is different from all stored versions of \(x\). The timestamp of the new version is \(\max\{\text{ts}(x) | \forall x \in RS\}\).

5. When the available memory for holding versions is full the transaction with timestamp equal to \(g\text{vts}\) is restarted and \(g\text{vts}\) is recalculated. Versions older than the new \(g\text{vts}\) are purged to free memory. In this way the oldest active transaction gets restarted, and this is also the transaction with lowest priority if transactions are executed according to assumption SA1. Thus, MVTO-S is aware of priority on transactions and restarts low prioritized transactions before transactions with high priority.

Data items have only a write timestamp, and versions of a data item are valid from their write timestamp until the write timestamp of the following version. If such a version does not exist, the most recent version is valid at all future times until a newer version is installed. A version gets its write timestamp from its parents rather than from the transaction as in MVTO. The reason is that the version created by the transaction with \(\text{ts}(\tau)\) would also be created by a transaction with an earlier timestamp down to and including timestamp \(\max_{x \in R(d)} \{\max_{y \in V(x)} \{\text{ts}(y) | \text{ts}(y) \leq \text{ts}(\tau)\}\}\). This is explained in the following example.
Example 4.7: Consider the data items in figure 4.14. A UT reading data items \(d_{9}, d_{15}, d_{18}, \) and \(d_{34}\) starts at time 20. The proper versions have timestamps 18, 11, 15, and 5. These versions would also be read if the transactions started at times 18 and 19, but at time 17 the version of \(d_{9}\) with timestamp 12 would be read. The timestamp of the produced data item \(d_{UT}\) can be set to time 18 since the value on \(d_{UT}\) is the same in the interval \([18, 20]\). This is depicted by the gray area in figure 4.14. □

Figure 4.14: Example of the write timestamp of versions using MVTO-S.

Under the assumption that an updating algorithm schedules and executes all needed updates and supports assumption SA1, the MVTO-S concurrency control algorithm is restart-free if the memory pool does not become full. Hence, in MVTO-S, the conflict resolution of conflicting concurrent transactions does not involve restarting transactions, and the CPU resource is utilized efficiently using MVTO-S.

This can be shown in the following way. It is assumed that an updating algorithm with relevance check schedules all needed updates and that the scheduling algorithm consults the MVTO-S algorithm during the scheduling to determine validity of data items. Furthermore, it is assumed that all scheduled updates are executed and that the system supports assumption SA1. In the MVTO algorithm, described in section Multiversion Concurrency Control (section 2.4.2), a transaction \(\tau\) is restarted only if a data item has a read timestamp larger than the timestamp of \(\tau\) issuing a write operation on the data item. We prove by contradiction that this restart cannot occur using MVTO-S and the updating algorithm. Under assumption SA1, started transactions have a fixed priority ordering, the MVTO restart can only be due to that \(\tau_{1}\) writing \(x\) starts, becomes preempted by \(\tau_{2}\) reading \(x\) and \(ts(\tau_{1}) < ts(\tau_{2})\), and then \(\tau_{1}\) continues and writes \(x\), but the write
operation is rejected because the $rts(x) > ts(\tau_1)$, where $rts(x)$ is the read timestamp of data item $x$.

When $\tau_2$ starts, all necessary updates are scheduled and executed. This means that $x$ would be updated if $x$ is stale. There is no update scheduled for $x$ by $\tau_2$, and, thus, $x$ must be valid. However, since $\tau_1$ is active, data item $x$ is stale otherwise $\tau_1$ would have been skipped by the updating algorithm. This is a contradiction and a reader of a concurrent writer cannot consequently occur, because the read would be preceded by an update. Thus, the update and the concurrent writer write the same version to the database.

Step 4 in MVTO-S says that the write timestamp of a data item $x$ is written as early as possible. With the same reasoning as above, if $x$ has a read timestamp higher than the write timestamp of a new version of $x$, this means that the transaction doing the read missed an update of $x$. This is impossible since an updating algorithm schedules and executes all needed updates. Thus, assigning an earlier timestamp to a version does not affect that MVTO-S can be restart-free.

Hence, MVTO-S with the given assumptions is restart-free if there is enough memory to store versions.

An important aspect of concurrency control is serializability. MVTO-S is view-similar to a serial schedule under the assumptions that an updating algorithm with a relevance check is used, that the system supports assumption SA1, and that there are enough versions in the memory-pool making MVTO-S restart-free. This can be shown in the following way. We want to show MVTO-S, together with an updating algorithm using a relevance check, produces histories that are view-similar to serial histories. We record in a history, denoted $H_C$, all transactions reaching their EOT operation and commit has been performed. Also, we have a history $H_B$ containing a recording of all transactions, even those that later are skipped. Since MVTO-S is restart-free $H_B$ contains only one occurrence of every started transaction. $H_C$ is the history MVTO-S produces and $H_B$ is a serial history that is a superset of $H_C$. We now show that $H_C$ is view-similar to $H_B$. The transactions that are not in $H_C$ but in $H_B$ are skipped because the updating algorithm uses step 3 in MVTO-S. The only way a transaction can be skipped is because it is unaffected of changes in its parents. Let $\tau_s$ be a skipped transaction, i.e., $\tau_s \in H_B \setminus H_C$. Let $T_B$ be the set of transactions in $H_B$ that read the value produced by $\tau_s$, and let $T_C \subseteq T_B$ be the set of transactions in $H_C$ reading the data item $\tau_s$ updates. Every
transaction $\tau_{Ci} \in T_C$ reads similar values compared to corresponding transaction $\tau_{Bi} \in T_B$ since the value written by $\tau_s$ must be similar to the value already stored in the database. This is because if the updating algorithm skips $\tau_s$ then its parents are assumed to not affect the value $\tau_s$ updates, and then $\tau_s$ produces a similar value of $d_s$ compared to one already stored in the database. Hence, transactions $\tau_{Ci}$ and $\tau_{Bi}$ read similar values of $d_s$. This reasoning can be applied to all skipped transactions. Hence, all transactions in $H_C$ see the same view as the transactions in $H_B$. History $H_B$ is view-serializable because MVTO-S is restart-free and translates read operations in the same way as MVTO, which is view-serializable. Thus, $H_C$ is view-similar to a serial schedule.

4.11 Implementation of MVTO-S

We now present three different implementations of MVTO-S. They are denoted MVTO-UV, MVTO-UP, and MVTO-CHECKSUM. These algorithms implement algorithmic step 3 in different ways.

4.11.1 MVTO-UV

One solution to determine similar versions is to store the value of read data items together with the version. Similarity is then only a matter of comparing every read value of an existing version to the corresponding read value of the new version. The concurrency control algorithm is denoted MVTO-UV, UV is an abbreviation for Use Versions. One of the definitions of data freshness with respect to one parent is used (definition 4.2.3 or definition 4.2.4). An example is depicted in figure 4.15. In this example, the old version is not similar to the new version since parent $d_{27}$ has changed too much between the derivation of the two versions.

4.11.2 MVTO-UP

This section describes an implementation of MVTO-S denoted MVTO-UP, where UP is an abbreviation for use memory-pool.

The overhead of storing all used values in the versions might be too heavy for some memory-constrained systems. Since the versions are purged only when the memory pool is full, the versions needed for checking similarity can be found in the memory pool storing versions and, thus, no
Figure 4.15: Determine if two versions are similar by using function $f$ on every pair of parents.

additional memory is needed for storing values of parents inside the versions. An example of this idea is depicted in figure 4.16. UT $\tau$ derives $d_{36}$, and data item $d_4$, $d_7$, and $d_{23}$ are parents of $d_{36}$. The dashed area of data item $d_{36}$ is the potentially new version that UT $\tau$ would write. UT $\tau$ need to execute if values used to derive the existing version are not similar to values $\tau$ would read. In this example only parent $d_4$ has changed since the version was derived. If the two versions of $d_4$ are similar, then $\tau$ can be skipped.

Figure 4.16: UT $\tau$ derives $d_{36}$. Algorithm CheckSimilarity investigates if the existing version of $d_{36}$ is similar to the one $\tau$ derives.

The algorithmic steps for doing the similarity check are as in figure 4.17. The parameters are $\tau$, the UT deriving data item $d_{UT}$. Line 1 derives the timestamp that the new version would have if being written to the database in the algorithmic step 4 of MVTO-S. If a version with the timestamp
already exists, this means that $\tau$ was preempted by another UT updating $d_{UT}$. CheckSimilarity can return true indicating that there already exists a similar version. This is implemented with the if-statement on line 2. Similarity is checked against the most recent version relative the UT, denote the version $z$. Thus, we need to find the timestamp of $z$ (line 5). If $z$ is unavailable then the new version of $d$ must be installed and CheckSimilarity returns false to indicate that the new version is not similar to any existing version. If $z$ exists then values read by $\tau$ need to be checked against values used deriving $z$. If the versions are similar, then the new version of $d$ is similar to $z$ and UT $\tau$ can be skipped. This is implemented in the for-loop on line 6, and the if-statement on line 10.

CheckSimilarity($\tau,d_{UT}$)
1: Derive timestamp version of $d$: $ts(d) = \max\{ts(x) | \forall x \in RS\}$, where $RS$ contains proper versions of data items in $R(d_{UT})$.
2: if version with the timestamp already exists then
3: return true
4: end if
5: Find version of $d$ with timestamp less than $ts(d)$. Denote this version $z$ and $ts(z) = \max\{\forall v \in V(d) | ts(v) < ts(d)\}$. Return false if such a version cannot be found.
6: for all $x \in R(d)$ do
7: Let $value(x_\tau)$ be the value stored in the version of parent $x$ read by $\tau$.
8: Let $value(x_z)$ be the value stored in the version of parent $x$ read by $\tau_z$.
9: Break algorithm if a version cannot be found.
10: if $f(value(x_\tau), value(x_z)) \neq true$ then
11: return false
12: end if
13: end for
14: return true

Figure 4.17: CheckSimilarity algorithm.

In MVTO-UP, when a transaction would derive a similar version, because the values of the parents are similar to the values of an already installed version, the new version is installed. The CheckSimilarity algorithm would always work if all versions were always available. However, they are not, and this is because of practical reasons. There are not unlimited
memory available in real-time embedded systems.

The possibility to determine if two versions are similar is dependent on finding values on read set members. Since versions can be purged from the memory pool a similarity check can fail because versions of parents have been removed from the memory pool. Storing the read values in the version as in MVTO-UV has the benefit that values on parents always can be found. The disadvantage is that every version has a high memory overhead. In MVTO-UP, this memory overhead is removed, and replaced with searching in the memory-pool. Thus, every version becomes smaller than in MVTO-UV, but there is a possibility that similar versions are interpreted as not being similar because values of parents could not be found.

4.11.3 MVTO-CHECKSUM

This section describes an implementation of MVTO-S that is denoted MVTO-CHECKSUM. One way to reduce the memory overhead of MVTO-UV is to assign an indicator to each version that uniquely identifies which values on parents that have been used. If the indicator takes less memory to store than the read values of parents as in MVTO-UV, then the memory overhead is reduced. Checksums/CRCs (see section Checksums and Cyclic Redundancy Checks (section 2.5)) and fixed validity intervals (see section Data Freshness (section 4.2)) are convenient to use. An example using MVTO-UV is depicted in figure 4.18(a), and a version using an indicator is depicted in figure 4.18(b).

A value can be uniquely identified by one fixed validity interval. If not more than 256 validity intervals are needed on each data item, then each fixed validity interval can be represented by an 8-bit integer. Hence, the value of a data item can be accurately represented by the 8-bit Fletcher’s checksum or a CRC of the fixed validity intervals of the read values. Hence, when a transaction is deriving a new version and it is installed, then the checksum, or the CRC of the used values, is calculated and stored together with the version. A similarity check is then only a matter of comparing the checksums or CRC.

The robustness of checksums for the application of similarity checks has been tested by calculating checksums for a fixed small number of octets and all combinations of a distribution of a change among the octets. The distribution of a change works as follows

(i) Tuples consisting of 6 elements are constructed. The elements are
4.11. Implementation of MVTO-S

(a) Size of a version for MVTO-UV.

(b) Size of a version for MVTO-UP.

Figure 4.18: Sizes of versions for MVTO-UV and MVTO-UP.

octets that can take the values 0–255.

(ii) A change to a tuple is applied. The change represents how many unit steps in positive direction from a base tuple that can be taken among arbitrary axis in the 6 dimensional space. For instance, the change 2 to the tuple (100, 100, 100) results in the following possible tuples:

(102, 100, 100)  (100, 102, 100)  (100, 100, 102)
(101, 101, 100)  (101, 100, 101)  (100, 101, 101)

(iii) A Fletcher’s 8-bit checksum and a CRC-32 are calculated for all tuples resulting from the changes 1, 2, 3, 4, 5, and 6 to a base tuple (100, 100, 100, 100, 100, 100) or a base tuple with random elements.

Table 4.2 shows statistical data on how the Fletcher’s checksum algorithm behaves on this small set of data. Out of the 738 possible tuples 712 of them produce a checksum that is equal to the checksum from at least one other tuple.

The results in table 4.2 are disappointing, but they are also to some extent documented in [61]. The reason of this behavior is the small changes in the octets. In the Fletcher’s checksum algorithm every octet is multiplied with a coefficient which is equal to the order the octets are used in the checksum algorithm. For instance, if octet number 2 is increased with 2, the checksum is increased with 4. If at the same time octet number 4 is
Table 4.2: Investigation of the robustness of Fletcher’s checksum algorithm.

<table>
<thead>
<tr>
<th># of octets</th>
<th>Range of changes</th>
<th>Random</th>
<th># of equal checksums</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1–6</td>
<td>No</td>
<td>712 out of 738</td>
</tr>
<tr>
<td>6</td>
<td>1–6</td>
<td>Yes</td>
<td>712 out of 738</td>
</tr>
</tbody>
</table>

decreased with 1, the checksum is decreased with 4. As can be seen, these two small changes in close octets cancel, and the calculated checksum is unaltered.

The CRC-32 algorithm is run on the same set of octets and the results can be found in table 4.3, where 0 of the 738 possible combinations of octets are duplicates. The divisor polynomial is: \( x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1 \). CRC-32 can be efficiently implemented using a table consuming 256 bytes (an implementation is described in [2]). Hence, even though a CRC-32 might take a bit longer to calculate than using Fletcher’s checksum algorithm, the tests suggest using CRC-32 anyway due to its better performance in producing unique indicators for the kind of octets that are used in the application of similarity checks.

Table 4.3: Investigation of the robustness of CRC-32.

<table>
<thead>
<tr>
<th># of octets</th>
<th>Range of changes</th>
<th>Random</th>
<th># of equal checksums</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1–6</td>
<td>No</td>
<td>0 out of 738</td>
</tr>
<tr>
<td>6</td>
<td>1–6</td>
<td>Yes</td>
<td>0 out of 738</td>
</tr>
</tbody>
</table>

The concurrency control algorithm using a checksum or a CRC-32 to check similarities is denoted MVTO-CHECKSUM.

4.12 Single-version Concurrency Control With Similarity

Single-version concurrency control algorithms, i.e., those that only use one version of each data item, can also be extended to use similarity in order to reduce number of conflicts. Lam and Yau added similarity to HP2PL [43]. In this thesis, the OCC algorithm is enhanced with a similarity-aware vali-
4.12. Single-version Concurrency Control With Similarity

dation phase. The algorithm is denoted OCC-S. The optimistic concurrency control algorithm described in the section Optimistic (section 2.4.2) has a validation phase looking as follows [38]:

```plaintext
1: Begin critical section
2: valid = true
3: for all other active transactions \( \tau_j \) other than \( \tau_i \) do
4:   if \( ws(\tau_i) \cap rs(\tau_j) \neq \emptyset \) then
5:     valid = false
6:   end if
7: end for
8: if valid then
9:   write phase
10: end if
11: End critical section
12: if valid then
13:   cleanup
14: else
15:   restart
16: end if
```

The if-statement on line 4 checks if the committing transaction can be serialized with respect to other active transactions. If the committing transaction tries to make a change permanent to a data item that is currently used by other transactions, these transactions would not be serialized. Line 8 checks if any conflicts have been found, and if not, the transaction copies changes to data items from local storage to the database (line 9). If the transaction cannot write changes to the database the database system decides if the transaction should be restarted (line 15).

If conflicting operations involve similar values, then there is no conflict since the written value is similar to the value already read by another transaction. Hence, the number of restarts can be reduced if some conflicts can be relaxed to non-conflicts by a similarity relation. Line 4–6 is instead as in figure 4.19. Line 1 checks if all read-write conflicts involves similar values according to similarity relation \( f \). If that is the case, then the committing transaction can proceed to its write phase.

Updating and Concurrency Control Algorithms

1: **if** \((w_s(\tau_i) \cap r_s(\tau_j) \neq \emptyset) \land (\forall d \in w_s(\tau_i) \land d \in t_s(\tau_j)), f(read(d), written(d)) \neq t)\) **then**

2: \(valid = false\)

3: **end if**

Figure 4.19: OCC-S validation phase.

### 4.13 Implementation of Database System

This section contains a description of an implementation of the proposed database system [18, 25].

In the implementation of the database system, two real-time operating systems are used: Rubus and \(\mu C/OS-II\). Rubus, from Arcticus Systems AB [1], is used in the EECU for scheduling and communication between tasks. Rubus consists of basic services and three kernels: red, green, and blue. The basic services supplies primitives for intertask communication through signals and mail boxes, mechanisms for locking critical regions, and memory pools. The red kernel is used for static off-line generated schedules that can guarantee successful execution of hard real-time tasks, i.e., they finish within deadlines. A tool called Rubus Visual Studio is used to define a set of schedules and then it is possible to switch between the schedules in Rubus on-line. The green kernel maps interrupt handlers to the operating system. By doing this it is possible to send signals from an interrupt handler to a blue task. The blue kernel schedules tasks which are soft real-time tasks and denoted blue tasks. The blue kernel executes in idle time, i.e., no guarantees can be given on the successful execution of a blue task. Blue kernel supports 14 priority levels and several blue tasks can have the same priority. Tasks cannot be dynamically created and the priority of a task cannot be changed during run-time.

\(\mu C/OS-II\) has the same functionality as Rubus, and the database system is implemented on top of \(\mu C/OS-II\) for being able to execute in a DOS command window on Windows 2000. The functionality to measure time durations is, based on our experiences, better for \(\mu C/OS-II\) in a DOS command window than for Rubus running in Windows 2000. Rubus is not available under DOS, but \(\mu C/OS-II\) is, therefore we also use the database implementation on \(\mu C/OS-II\).

The database system implemented on the EECU include concurrency control algorithms and the earliest deadline first (EDF) scheduling algorithm. We are currently using the periodic tasks of the EECU software,
void TotalMulFac(s8 mode)
{
    s8 transNr = TRANSACTION__START;
    while(BeginTransaction(&transNr,
                          10000, 10, HIGH_PRIORITY__QUEUE, 
                          mode, TOTALMULFAC))
    {
        ReadDB(&transNr, FAC12_5, &fac12_5);
        /* Do calculations */
        WriteDB(&transNr, TOTALMULFAC, 
                local_fac, &TotalMulFac);
        CommitTransaction(&transNr);
    }
}

Figure 4.20: Example of a transaction in the EECU software (C-code).

i.e., not the crank angle based tasks, because Rubus has no support for
dynamic priorities on tasks and dynamic creation of tasks, which is neces-
sary to properly map crank angle interrupts to blue tasks. The reason is
that crank angle interrupts have a higher priority than time-based inter-
rupts. The priority of the interrupt dispatcher is lowered one level during
the execution of some code parts meaning that a new crank interrupt can
interrupt the handler of the previous crank interrupt. The execution time of
the interrupts handlers are quite long and the response time of an interrupt
needs to be short, therefore the priority is lowered.

All time-based tasks are mapped to blue tasks in Rubus. One red task
is implemented as the scheduler of the blue tasks by measuring the time
since a blue task was last invoked and sending a signal if the time is longer
than the period of the task. Blue tasks have the following period times: 5
ms (which is the period time of the red scheduling task), 10 ms, 25 ms, 50
ms, 100 ms, 250 ms, and 1000 ms. The database system is added to the
EECU software and it runs in parallel to the tasks of the original EECU
software. Hence, it is possible to compare the number of needed updates
of data items between the original EECU software and the added database
system. All data items are stored in one data area and access to the data
items is possible through a well-defined interface.

An example of a transaction in this system is given in figure 4.20. Be-
ginTransaction starts a transaction with a relative deadline of 10000 μs
that derives the data item TOTALMULFAC, \( d_9 \) in figure 2.5. Read and write operations are handled by ReadDB and WriteDB, and CommitTransaction notifies the database system that the transaction commits. The next invocation of BeginTransaction either breaks the loop due to a successful commit or a deadline miss, or restarts the transaction due to a lock-conflict. Detailed elaboration of the interface is presented in [18].

### 4.13.1 Implementation Details of Concurrency Control

Rubus has no support for (i) dynamic priorities, (ii) dynamic creation of tasks, (iii) restart of tasks, i.e., longjmp in a UNIX environment, and (iv) no knowledge of deadlines. \( \mu C/OS-II \) has support for (i) and (ii), but since the database implementation should be able to execute on top of both operating systems, Rubus sets the limits. No restart of tasks means that transactions need to execute until CommitTransaction before they can be restarted, i.e., all computation work done by the transaction from the point it is marked for being restarted until it reaches CommitTransaction is unnecessary. There is no straightforward way to resolve this in Rubus. A longjmp could be simulated by polling the restart flag in the calculation part in a simulation. Moreover, assumption SA1 is realized by Rubus. HP2PL, OCC, and OCC-S are described in figure 4.21.

Every concurrent task has a unique priority which means that a conflict always results in a restart for HP2PL. This indicates that HP2PL and OCC should have almost the same performance since all conflicts except write-read conflicts result in restarts in OCC.

Due to the inability to change the priority of a task in Rubus, HP2PL suffers from priority inversion in a write-read conflict. When the read-lock is requested the lower prioritized transaction, \( \tau_1 \), holding the write-lock rollbacks and is marked for restart. The read-locker, \( \tau_2 \), has to wait for the write-locker to rollback and the rollback is done with the priority of \( \tau_1 \), i.e., a transaction \( \tau_3 \) with priority \( \text{prio}(\tau_1) < \text{prio}(\tau_3) < \text{prio}(\tau_2) \) can preempt and execute before \( \tau_1 \) continues to rollback.

In systems with many conflicts it should be a clear difference in performance between HP2PL and OCC compared to MVTO-S algorithms, and using no concurrency control since the latter two are restart free and do not suffer from unnecessary computation of restarted transactions.
r_lock: if write-locked then wait for transaction to rollback
w_lock: mark item as write-locked and mark readers for restart
(a) HP2PL

r_lock: add event to transaction log
w_lock: add event to transaction log
verify: check if a transaction has accessed the data item the verifying transaction writes, if so, mark that event as a clash. If the verifying transaction has any clashes in its log, then restart the transaction.
(b) OCC

r_lock: add event to transaction log
w_lock: add event to transaction log
verify: check if an active transaction has accessed the data item the verifying transaction writes and the accessed value is not similar to the value that is about to be written, if so, mark that event as a clash. If the verifying transaction has any clashes in its log, then restart the transaction.
(c) OCC-S

Figure 4.21: Implementation details of concurrency control algorithms.
Chapter 5

Performance Evaluations

This chapter presents evaluations of the updating algorithms and concurrency control algorithms presented in the Updating and Concurrency Control Algorithms chapter.

5.1 Methodology of Experiments

A database system to address requirements R1 and R3 has been proposed in this research project. Two families of algorithms have been developed: updating algorithms and concurrency control algorithms. These families of algorithms need to be evaluated individually and together. Due to requirement R3d, use available resources efficiently, we have taken the following approach: first investigate which updating algorithm has the best performance, then investigate which concurrency control algorithm has the best performance using this updating algorithm. Three simulator environments have been used. These are:

- RADEEx++ which is a discrete event simulator [27]. A benefit of using a discrete event simulator is that code taking long time to execute but are not part of the algorithms do not affect the performance of the algorithms. The experiments performed in the RADEEx++ simulator conforms to an absolute system [35] which has the following definition:

Definition 5.1.1 (Absolute Consistent System [35]). In an absolute consistent system, a UT, with a commit time $t$ and a read set
is given the values of all the data items in $R$ such that this set of values can be found in an instantaneous system at time $t$.

An instantaneous system applies base item updates and necessary recomputations in zero time. Hence, in an absolutely consistent system, a UT is valid if changes to data items during the execution of the transaction do not affect the derived value of the transaction. The process of gathering statistical results is time-consuming, e.g., checking if an updated data item is unaffected by concurrent writes to other data items requires storing all versions of data items. However, using RADEx++ we can collect the statistical data without affecting the performance results (this is also known as the probe effect). All paths from base items to the updated data item need to be checked if they affect the new value of the data item. This is a time-consuming process and can only be carried out in a discrete event simulator without affecting the performance results.

- An engine simulator which is a device connected to an EECU. The engine simulator sends sensor values to the EECU such that the environment resembles a true engine environment. We have implemented a database implementation using the ODTB updating algorithm is running in the EECU software. This simulation setup can be used to verify the results from the RADEx++ simulations. In this case, the engine simulator is utilized to confirm that calculations can be skipped. In this application we have a subset of all data items transformed into transactions using a database implementation. The concurrency control algorithms cannot be tested in the engine control settings because the implemented user transactions do not conflict and therefore there is no concurrency.

- A PC using the same database implementation as in the engine simulator simulations. The user transactions are generated by application software running as tasks. In this way it is possible to achieve concurrency and test the concurrency control algorithms. However, the value of data items needs also to be simulated since the updating and concurrency control algorithms use similarity.

RADEx++ contains more code than the database implementation for an EECU and therefore it is easier to implement the proposed multiversion concurrency control algorithms proposed with similarity in the database implementation. Furthermore, since the database
Performance Evaluations

system executes on a real-time operating system, performance evaluations give a good view of the behavior of the data base in a real-life system. For these reasons, we have focused our efforts to implement and evaluate concurrency control algorithms using our database implementation running on a soft target of a real-time operating system in a DOS command window in Windows 2000.

The experiments are reported in the following order. First, we describe all experiments conducted using the RADEEx++ simulator. The aim is to test the performance of updating algorithms. The performance metric is number of valid committed UTs using the definition of an absolute system (definition 5.1.1), i.e., the UT commits within its deadline and it used values that are still valid. The experiments are presented in section 5.2. Second, an experiment conducted using the engine simulator to confirm that the similarity concept in updating algorithms works in a real-life setting. The engine simulator experiment is presented in section 5.3. Third, experiments conducted using the database implementation running in a real-time operating system on a PC showing the performance of concurrency control are presented in section 5.4. Finally, this chapter ends with a wrap-up of our observations in section 5.5. Some of the graphs presented in this chapter are also presented in appendix A with 95% confidence intervals.

5.2 RADEEx++ Experiments

The experiments conducted using the RADEEx++ simulator test the performance of different on-demand updating algorithms: OD, ODKB, OD_V, ODKB_V, ODDFT, ODBFT, ODDFT_C, ODKB_C, and ODTB. The experiments are divided into experiment 1 and 2. Experiment 1 tests performance of updating algorithms lacking a relevance check; experiment 2 tests performance of updating algorithms having relevance checks. Both experiment 1 and 2 test the throughput of valid UTs in systems where any data item can be requested by a UT, and in systems where only actuator transactions are requested.

5.2.1 Simulator Setup

RADEEx++ is setup to function as a firm real-time main-memory database. Two queues for transactions are used: STs in the high priority queue, and
5.2. RADEx++ Experiments

UTs in the low priority queue. HP2PL is used as concurrency control protocol and transactions are scheduled based on EDF. The updating frequency of base items is determined by their absolute validity intervals, \( avi(b) \). An \( avi(d) \) is also assigned to each derived data item to determine the freshness for on-demand algorithms OD, ODO, and ODKB. UTs are aperiodic and the arrival times of UTs are exponentially distributed. The user transactions are seen as value user transactions (VUT) in all experiments except in one experiment (1b) where user transactions derive only leaf nodes. The data item a VUT derives is randomly chosen from the set of all derived data items. A leaf node is the result of a chain of derivations and the resulting value is sent to an actuator, i.e., such UTs are actuator user transactions (AUT). The data item a AUT derives is randomly determined from the set of all leaf nodes in the data dependency graph \( G \). The triggered updates are not executed if the calculated release time is earlier than the current time, i.e., line 2 in algorithm ExecTrans is activated in the experiments conducted using RADEx++. The number of read operations is the cardinality of read set \( R(d_{UT}) \). The WCET of a transaction is determined by the number of operations and the maximum execution time of these. The single write operation for STs always takes \( STProcCPU \) time. The maximum execution time for one operation in a UT is \( UTProcCPU \). During simulation each operation in a UT takes a uniform time to execute, which has an average determined during initialization of the database. This randomness models caches, pipelines, but also the usage of different branches of an algorithm. The deadline of a transaction is its WCET times a uniformly chosen value in the interval \((1,7)\).

Values of the data items are simulated with the parameter \( \text{max}_\text{change} \), which is individual for each data item, and it expresses the upper bound of how much a value may change during its \( avi \) in the simulation. When a new value for a data item is written to the database, the stored value is increased with an amount that is taken from a standard distribution, \( N(\text{max}_\text{change}/2,\text{max}_\text{change}/4) \), limited to the interval \((0,\text{max}_\text{change})\). The value of \( \text{max}_\text{change} \) and \( avi \) are derived from the same uniform distribution \( U(200,800) \). Data validity interval \( \delta_{i,j} \), where \( j \) is a parent of \( i \), is given by \( avi(j) \) times \( \text{factor} \). A \( \text{factor} \) equal to one implies that the \( avi \) give a good reflection of the value changes if \( \text{factor} \) is greater than one, the absolute validity intervals are pessimistic, i.e., the values of data items are generally fresh for a longer time than the absolute validity intervals indicate. The blocking factor \( \text{blockingf} \) is set to one if not stated otherwise.
The database parameters and the settings are given in table 5.1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Explanation</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>avi</td>
<td>absolute validity interval</td>
<td>U(200,800) msec</td>
</tr>
<tr>
<td>δ_i,j</td>
<td>data validity interval for (i)</td>
<td>(factor \times avi(j))</td>
</tr>
<tr>
<td>max_change</td>
<td>max change of a data item during its avi</td>
<td>U(200,800)</td>
</tr>
<tr>
<td>STProcCPU</td>
<td>max execution time of a ST operation</td>
<td>1 msec</td>
</tr>
<tr>
<td>UTProcCPU</td>
<td>max execution time of a UT operation</td>
<td>10 msec</td>
</tr>
<tr>
<td>factor</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>blockingf</td>
<td>Blocking factor</td>
<td>1 (default)</td>
</tr>
</tbody>
</table>

A database is given by \(|B| \times |D|\). The directed acyclic graph giving the relationships among data items is randomly generated once for each database, i.e., the same relationships are used during all simulations. In the experiments a \(45 \times 105\) database are used, implying that there are 150 data items in the database, and the ratio of base items and derived items is 0.3. Moreover, the maximum cardinality of a read set \(R(d)\) is 6, and the likelihood that a member of \(R(d)\) is a base item is 0.6. This creates a broad database since it is more likely that a parent is a base item than a derived item. We believe that data dependency graphs in real-time systems are normally broad (as opposed to deep) since intermediate nodes in the graphs are shared among transactions and there are probably only a few derivations of a sensor value when the final result of an AUT is sent to an actuator. The examples we have seen, e.g., the data dependency graph in figure 2.5, contain a few number of elements in the read sets. Therefore we have chosen the cardinality of a read set to be maximum 6. The error function is defined as: \(error(x,t) = t - timestamp(x)\), since \(max\_change\) and \(avi\) are taken from the same distribution.
5.2.2 Experiment 1a: Consistency and Throughput With No Relevance Check

The objective of this experiment is to determine the performance of updating algorithms without relevance checks, i.e., the algorithms are OD, ODO, ODKB, ODO_V, ODKB_V, ODDFT, and ODBFT. The main performance metric is the number of successfully committed valid UTs according to the definition of an absolute system (definition 5.1.1), i.e., a UT is valid if, at commit time, the deadline is met and the derived value is unaffected by changes in other data items.

Figure 5.1 shows total number of committed UTs in figure 5.1(a) and number of committed UTs that are valid based on an absolute system in figure 5.1(b). The ratio of number of valid committed UTs and number of generated UTs over all simulations runs, i.e.,

\[
\frac{\sum_{i=1}^{5} \# \text{ valid committed UTs in run } i}{\sum_{i=1}^{5} \# \text{ generated UTs in run } i},
\]

is plotted in figure 5.2. First, the distinction between the time domain on-demand updating algorithms can easily be seen. OD is a consistency-centric updating algorithm, i.e., the freshness of data items is more important than the throughput of UTs, whereas ODO and ODKB are throughput-centric since these updating algorithms can reject updates if it seems there are not enough time for a UT to meet its deadline. The throughput of UTs for ODKB is higher than the throughput for OD. In figure 5.1(b), it can be seen that ODKB produces more valid committed UTs than OD. The reason is that albeit some updates are rejected by ODKB, values of data items can anyway be valid when they are used and since more UTs commit under ODKB compared to OD, this keeps the total number of valid data items higher.

A counterintuitive result can be seen in figure 5.1(b). OD_V and ODKB_V let fewer valid UTs commit compared to corresponding time domain algorithms, i.e., OD and ODKB. The reason is that when a read operation is issued by a transaction using the _V version of the algorithms, it is checked whether the data item is valid or not by investigating the validity of the parents. If the parents are valid, the data item is not updated. However, the data item might have been potentially affected by a change in a parent, and this goes unnoticed by OD_V and ODKB_V. Since values that should have been updated never were updated it affects the validity of the produced result. The time domain algorithms update all values that
Figure 5.1: Experiment 1a: Consistency and throughput of UTs (confidence intervals are presented in figure A.1).
are too old, and in the experiment value changes match absolute validity intervals. Thus, a data item that needs to be updated is probably too old implying that it gets updated.

ODDFT and ODBFT are consistency-centric, as OD, and figure 5.1(a) shows that their number of valid committed UTs are less than for ODKB but a bit higher than for OD. Figure 5.1(b) shows that ODDFT and ODBFT let an equal number of valid UTs to commit. Both these algorithms take a pessimistic approach and assume every data item having a pa timestamp greater than zero to be stale (except for recently updated data items that still are considered to be fresh, this is because of the usage of the error function on line 2 in AssignPriority). This approach pays off, since the number of committed valid UTs is higher than for any other algorithm.

Figure 5.3(a) shows the difference between an on-demand algorithm triggering updates based on pa > 0 and ODDFT. As can be seen, ODDFT lets more valid UTs commit compared to OD_with_pa. In figure 5.3(b) it can be seen that at arrival rates 0-40, ODDFT generates fewer updates than OD_with_pa and OD. This experiment shows that the pa timestamp alone cannot increase the performance compared to OD. The ODDFT scheduling algorithm combines the pa timestamp and the error function using the deadline of the transaction as t. The usage of the error function enhances the performance.
Figure 5.3: Experiment 1a: Effects of measuring staleness of data items at deadline of UT (confidence intervals are presented in figure A.2).
5.2.3 Experiment 1b: Deriving Only Actuator User Transactions

The objective of this experiment is to investigate the performance of a system that only derives actuator user transactions. The performance metric is valid committed user transactions.

One question is how to interpret the nodes in the data dependency graph $G$ and how these nodes map to transactions. The nodes can be divided into base nodes, i.e., those that correspond to sensor data, and the nodes corresponding to derived data: intermediate nodes and leaf nodes. A leaf node has no children, and intermediate nodes have both parents and children. Intermediate nodes can be seen as containing data that are shared among the leaf nodes, and the leaf nodes represent data at the end of a refinement process involving data on paths from base nodes to the leaf node. The data in leaf nodes is then likely data that is sent to actuators.

The results for when UTs derive only leaf nodes can be seen in figure 5.4. Figure 5.4(a) shows the number of valid committed UTs deriving data items that have at least one derived data item as parent. ODDFT and ODBFT are performing much better than the other updating algorithms due to their ability to update data that is judged to be stale at the deadline of the UT. Note that using no updating scheme always gives stale data. In the previous showed simulation, a UT is deriving a data item that is randomly chosen among all derived data items. This way of choosing data items helps keeping them valid, because there is a probability that a data item and its parents are updated often enough to keep them valid. In this experiment, however, intermediate nodes are not kept valid by these random transactions. All updating algorithms are suffering from this which can be seen in figure 5.5 showing the number of triggered updates. Comparing these numbers to those in figure 5.3(b) shows more updates are needed.

Figure 5.4(b) shows the results for UTs deriving data items depending only on sensor values. Remember that using no updating algorithm gives fresh data items since the values read by the UTs are always valid. The other updating algorithms also produce valid UTs, but their performance drop due to triggering updates taking time to execute, and, thus, delaying UTs such that they miss their deadlines.
(a) Number of valid committed UTs where the UTs are deriving leaf nodes having at least one derived data item as a parent.

(b) Number of valid committed UTs for UTs deriving data items having only base nodes as parents.

Figure 5.4: Experiment 1b: Consistency and throughput of UTs that only derive leaf nodes.
5.2. RADEx++ Experiments

5.2.4 Experiment 1c: Comparison of Using Binary Change Flag or $pa$ Timestamp

The objective with this experiment is to investigate how the usage of the $pa$ timestamp changes the behavior of the updating algorithms compared to using a boolean change flag as in our previous work [22–24], where the $pa$ timestamp is a boolean flag denoted $change$. Unfortunately, there can be no guaranteed mapping from $change(d)$ equals true to $pa(d) > 0$ and from $change(d)$ equals false to $pa(d) = 0$. Example 4.3 shows this.

The reason the $pa$ timestamp is used is to correctly determine if an update is needed. This is important when relative consistency of data is considered. The updating algorithms that are implemented in the RADEx++ simulator are implemented with the choice of skipping late updates. This destroys the possibilities to measure relative consistency, and, thus, it might be sufficient to use the change flag which takes less memory to store. What happens if the change flag is used instead of the $pa$ timestamp? Figure 5.6 shows that the performance of the system is not dependent upon using either $pa$ or $change$, i.e., the potentially missed updates from using the change flag do not affect validity of committed transactions. This shows that the $change$ flag approach can be used, and such an implementation uses less memory than using the $pa$ timestamp.

Figure 5.5: Experiment 1b: Number of generated triggered updates where UTs are AUT (confidence intervals are presented in figure A.3).
5.2.5 Experiment 1d: Transient and Steady States

The objective of this experiment is to investigate how state changes in the external environment affect the workload of updates scheduled by updating algorithms.

One interesting aspect of using data freshness in the value domain is that the number of generated updates should be affected by the current state of the system. If the system is in a steady state, i.e., the external environment does not change much implying that the sensor values are not changing much, then the number of updates should be less than in a transient state where sensor values are changing rapidly. This subsection presents a simulation with the state changes: from transient to steady, and then back to transient again.

The number of generated triggered updates during a simulation is counted. The simulation is conducted as follows: the arrival rate is 30 UTs/second, the size of the database is $45 \times 10^5$, and 100 sec is simulated. Two parameters are introduced: \texttt{change\_speed\_of\_sensors} and \texttt{change\_speed\_of\_user\_trans}. Data items change with the following speed: $N(max\_change/\texttt{change\_speed\_of\_X}, max\_change/(4 \times \texttt{change\_speed\_of\_X}))$, where \texttt{X} is substituted with \texttt{sensors} or \texttt{user\_trans}. For the first 15 sec, \texttt{change\_speed\_of\_sensors} is
Table 5.2: Experiment 1d: Statistical data from transient and steady state simulation.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># committed UTs</th>
<th># valid committed UTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>OD</td>
<td>2035</td>
<td>1742</td>
</tr>
<tr>
<td>ODDFT</td>
<td>2445</td>
<td>2207</td>
</tr>
<tr>
<td>ODKB</td>
<td>2698</td>
<td>2138</td>
</tr>
<tr>
<td>ODKB_V</td>
<td>2748</td>
<td>2121</td>
</tr>
</tbody>
</table>

set to 1.2 which gives rapid changes (transient state), from 15 sec to 75 sec change_speed_of_sensors is set to 50 (steady state), and from 75 sec the system enters again a transient state where change_speed_of_sensors is set to 2.0. During the simulation change_speed_of_user_trans is set to 2.0.

Figure 5.7 contains the simulation results. The horizontal lines represent the average number of generated triggered updates during the indicated interval. ODDFT clearly generates fewer triggered updates during the interval 15–75 sec than OD, which is unaware of that base items live longer in this interval. ODKB_V, which uses a value-aware triggering criterion, also has less generated triggered updates in steady state. Hence, the load on the CPU is lower for ODDFT during a steady state than OD, and the extra load for OD consists of unnecessary triggered updates. Table 5.2 shows the number of committed UTs and the number of valid committed UTs from the four simulations shown in figure 5.7. Comparing the consistency-centric algorithms, ODDFT gets better results. The number of committed UTs is higher than for OD, and the number of generated triggered updates could be reduced considerably during the steady state. Comparing ODKB and ODKB_V, they let the same number of UTs commit, but ODKB_V also can adapt the number of triggered updates to the state of the system.

5.2.6 Experiment 1e: Effects of Blocking Factor

The objective of this experiment is to investigate how the blocking, modeled with blockingf, affects the performance of updating algorithms. Late updates are not executed since line 2 in ExecTrans rejects late updates. A blocking factor greater than zero effectively blocks more updates, because the calculated latest release time of an update is earlier for blockingf >
Figure 5.7: Experiment 1d: Simulation of transient and steady states of a system.
In essence, the workload from updates can be reduced by increasing blocking factor, but the risk is that more user transactions become invalid.

The blocking factors are part of the algorithms ODDFT, ODBFT, and ODTB, and figure 5.8(a) shows the performance of ODDFT using different blocking factors where UTs derive random data items. The figure shows the ratio of valid committed UTs and the generated UTs. The blocking factor can give a positive effect on the number of valid committed UTs. The reason is that fewer triggered updates are started which can be seen in figure 5.8(b), and, thus, the workload on the system is reduced letting more triggered updates and UTs to commit. However, if the blocking factor becomes too large, the risk of not updating enough data items is higher. This can be seen in figure 5.8(a) where ODDFT with blocking factor 3.0 lets fewer valid UTs to commit compared to when a blocking factor of 2.0 is used.

Note that, as indicated by figure 5.9, the number of committed UTs increases but also the number of invalid committed UTs. Hence, increasing a blocking factor above 1 makes ODDFT more of a throughput-centric updating algorithm than a consistency-centric. It is a design decision for the designer of the system what to set blocking factor to.

Figure 5.10 shows the effect of blocking factors where UTs derive only leaf nodes, i.e., the system only executes actuator user transactions. In this setting, only triggered updates can make intermediate nodes valid while in the case of UTs deriving random data items an intermediate node can be made valid by a UT deriving the data item. Thus, more triggered updates are needed to keep data items valid. Also note that out of the 105 derived data items, 43 data items are leaves in the data dependency graph, and out of these 15 data items have only base items as parents. The UTs derive only one of the 43 leaf nodes meaning that transactions deriving a specific node arrive more often to the system in this experiment compared to when UTs derive any derived data item. This implies that there are also more generated triggered updates for the intermediate nodes compared to the random case. Thus, in the only leaves experiment, data items on paths from base items to leaf nodes get updated more often compared to randomly choosing data items. Figure 5.10 shows the ratio of valid committed UTs and generated UTs in an experiment deriving only leaf nodes and changing blocking factor. Comparing the results in figure 5.10 to the results in figure 5.8(a), we see that a larger value of blocking factor is needed to drop the performance for the only leaves experiment. This confirms the fact that more triggered updates
Figure 5.8: Experiment 1e: The effect of $blocking_f$ on the number of valid committed UTs where a UT randomly derives a data item.
Figure 5.9: Experiment 1e: Statistical data for ODDFT using two different blocking factors.
are generated for data items in the only leaves experiment since the larger the blockingf the more updates are not scheduled.

![Figure 5.10: Experiment 1e: The effect of blockingf on the number of valid committed UTs where a UT derives leaf nodes.](image)

5.2.7 Experiment 1f: Varying Weights

The objective of this experiment is to examine how updating algorithms prioritize data items. The idea is that some data items are more important to keep valid than others. The data items are assigned weights, and where a weight reflects the importance of a data item.

In this experiment the weights of data items are set randomly by taking a number $n$ from a normal distribution, $N(2.0, 3.0)$, where 2.0 is the expected value and 3.0 is the standard deviation. If the number $n$ is less than 1, then the weight is set to 1, otherwise $n$ is used.

ODDFT and ODBFT take weights into consideration when prioritizing updates. The AssignPriority algorithm used by ODDFT and ODBFT multiplies the priority returned by the error function with the weight on the data item. The on-demand algorithms with any option (OD, ODO, ODKB) do not use weights on the data items.

The metric used in this experiment is the weight ratio:

$$WR(d) = \frac{\sum_{x: \forall x \in R(d), |x - x'| \leq \delta_{d,x}} \text{weight}(x)}{\sum_{\forall x \in R(d)} \text{weight}(x)}.$$
Figure 5.11: Experiment 1f: Varying weights.

(a) Ratio of sum of weights of fresh parents and sum of all parents weights.

(b) Ratio of valid committed UTs and generated UTs.
where $x'$ is the value used in the previous derivation of $d$, i.e., the sum of the weights for all valid parents $x \in R(d)$ divided by the sum of the weights for all parents. A high weight ratio means that strongly weighted parents are prioritized in scheduling of updates.

Figure 5.11 shows that as the arrival rate increases, ODDFT and ODBFT have the ability to keep the strongly weighted data items valid due to that AssignPriority takes the weights into account during scheduling of updates. At low arrival rates (5–15 in figure 5.11(a)), the difference between ODDFT and ODBFT compared to OD, ODKB, OD_V, and ODKB_V is because almost all data items are stale at low arrival rates. The value of a data item can then have a large error because it has not been updated for a long time, and then the large weight on other data items cannot prioritize these data items higher than the data item with a high error. At higher arrival rates, data items are more often valid because of the higher chance of being updated by a triggered update of UT. This also means that the errors in values of data items are almost the same, and then the weights of the data items tell the order in which triggered updates should be put in the schedule of updates.

5.2.8 Experiment 2a: Consistency and Throughput With Relevance Check

The objective of this experiment is to investigate the performance of updating algorithms using relevance checks. The rationale of using relevance checks is to skip unnecessary recalculations of data items, and, thus, decrease the workload on the system whenever possible.

Figure 5.12(a) shows the total number of committed UTs within their deadlines for value domain based updating algorithms (ODKB_V, ODDFT, and ODTB), time domain based updating algorithms (OD and ODKB), and using no updates. In this experiment, the algorithms OD, ODKB, ODKB_V, and ODDFT have no relevance check and, thus, they try to execute as many of the updates as possible even though some of them might be unnecessary. They are plotted to represent base lines to compare ODTB to. A skipped transaction is considered to be successful if the skip happened before the deadline. At around 45 UTs per second the system becomes overloaded since the number of committed UTs stagnates using no updates. ODTB has the best performance and at high arrival rates more UTs can commit than using no updates. This is because of the skipping of transactions reduces the concurrency thereby giving more time to transactions that
need to be executed. The load on the system can be decreased by using ODTB since it lets unnecessary updates and UTs to be skipped. The value of a data item stored in the database can be used without recalculating it. Thus, this enables resources to be reallocated to other tasks, e.g., the diagnosis application of an EECU. Figure 5.12(b) shows the number of valid and committed UTs. ODTB lets the most valid UTs to commit and during overload (above 45 UTs per second) the difference is in the order of thousands UTs or more than a 50% increase compared to updating algorithms not skipping transactions.

The results of comparing ODTB to ODDFT\_C and ODKB\_C are in figure 5.13. ODDFT\_C and ODKB\_C can now let more UTs to commit at high load as many updates can be skipped since executing them produces only the same result as the one already stored in the database, i.e., unnecessary updates are skipped. The total load on the system is, thus, decreased.

From figure 5.13 we see that ODKB\_C lets slightly more UTs commit than ODTB, but more UTs are valid for ODTB. ODTB also has more valid committed UTs than ODDFT\_C. A comparison between ODTB and ODDFT\_C using blocking factors greater than 1 is presented in section Experiment 2b.

5.2.9 Experiment 2b: Effects of Blocking Factors and Only Deriving Actuator Transactions

Figure 5.14 shows the behavior of updating algorithms that can skip updates and their behavior with different values on $\text{blocking}_f$. ODDFT\_C increases the performance with $\text{blocking}_f > 1$ while ODTB decreases the performance for all $\text{blocking}_f > 1$. The reason ODTB drops in performance is that a check is done for the execution time of the remainder of a path to the data item being updated (the if-statement on line 7). If the check fails, none of the updates are put in the schedule of updates. Since $\text{blocking}_f > 1$ increases the execution time of a path, it is more likely there is not enough time for these updates. Skipping too many updates results in a negative impact on the performance. ODDFT\_C could put some of these data items in the schedule, which makes it a more consistency-centric algorithm than ODTB.

Figure 5.15(a) shows the performance of ODTB, ODDFT\_C, and ODKB\_C when they perform the best. A blocking factor of 1.5 has been chosen for ODDFT\_C since all the blocking factors perform equally as
Figure 5.12: Experiment 2a: Consistency and throughput of UTs with no relevance check on ODDFT and ODKB_V.
5.2. RADEx++ Experiments

Figure 5.13: Experiment 2a: Consistency and throughput of UTs with a relevance check on triggered updates on ODDFT (confidence intervals are presented in figure A.5.)
Figure 5.14: Experiment 2b: Performance for updating algorithms that have the possibility to skip updates.
shown in figure 5.14(a). The blocking factor is not used in the ODKB\_C algorithm. ODDFT\_C has the best performance of the algorithms. However, a full ODDFT schedule is needed and a validity check of the parents is needed before a triggered update is started. The scheduling step and the validity check are cheaper in the ODTB algorithm.

Figure 5.16 shows the performance of ODTB and ODDFT\_C, and it shows that ODTB lets more transactions commit, but ODDFT\_C let more valid transactions to commit. Hence, ODDFT\_C is consistency-centric while ODTB is throughput-centric.

5.3 Database Implementation in EECU

Often, an embedded and real-time system is installed in a dynamically changing environment, where the system has to respond to these changes. Since tasks use data that should be fresh, state changes in the environment also affect the need to update data. One experiment is conducted using an engine simulator and an EECU. The experiment is designed to test if the result from Experiment 1d can be achieved in a real-life setting, i.e., we want to investigate how state changes in the external environment affect the workload of updates scheduled by updating algorithms.

5.3.1 Simulator Setup

The ODTB updating algorithm is evaluated using the implementation of a database system integrated in the EECU. The system is depicted in figure 5.17. The EECU is connected to an engine simulator. The engine simulator sends sensor values to the EECU that functions as in a real-life situation calculating and sending actuator values to the engine simulator. Values on statistical variables are collected by using a vendor-specific CAN-based protocol, and computer application called AppTool.

In the performance evaluations in this section, the engine simulator is used to adjust the engine speed. The EECU reacts upon the sensor signals as if it controlled a real engine. Thus, from the perspective of the EECU software, there is no distinction between an engine and an engine simulator.

The database implementation is executing on top of Rubus, and the original software is executing by being scheduled by Rubus. Transactions for the data dependency graph depicted in figure 2.5 are implemented. There is one UT that is requested periodically by the original EECU software. The
(a) Number of valid committed UTs for the ODDFT\(_C\), ODTB, and ODKB\(_C\) with parameters such that they perform the best.

(b) Number of generated triggered updates.

Figure 5.15: Experiment 2b: Performance for updating algorithms that has the possibility to skip updates (confidence intervals are presented in figure A.6).
5.3. DATABASE IMPLEMENTATION IN EECU

![Graph of experimental results](image)

(a) ODDFT_C with $blocking f = 1.5$.

(b) ODTB

Figure 5.16: Experiment 2b: Performance metrics for ODDFT_C with $blocking f = 1.5$ and ODTB.
UT is deriving TOTALMULFAC. The implementation is further described in the section Implementation of Database System (section 4.13).

![Diagram of EECU and engine simulator](image)

**Figure 5.17: Overview of the EECU and engine simulator.**

### 5.3.2 Experiment 3: Transient and Steady States in EECU

This experiment considers steady and transient states and the number of required updates in each state. The number of updates is contrasted between ODTB and periodic updates, i.e., time domain for data freshness.

The derived data item TOTALMULFAC is requested periodically by a task in the EECU software. The data dependency graph is shown in figure 2.5.

Recalculations of TOTALMULFAC are needed when the engine speed changes. Figure 5.18 shows how the requests for calculations are serviced only when the system is in a transient state, i.e., when the engine speed is changing. The plots in the bottom graph are cumulative numbers of requests. The number of requests is increasing linearly since the requests are periodic (remember that all time-based tasks are executed with fixed periodicity) and in the original EECU software each such request is processed. However, with the usage of ODTB only some of the requests need to be processed. The number of serviced requests shows how many of the requests need to be processed. In steady states, none of the requests need to be processed, and the stored value in the database can be used immediately, e.g., the steady state in the time interval 2–7. Hence, during a steady state a considerable amount of requests can be skipped. Notice also that the data validity intervals allow the database system to accept a stored value if changes to the engine speed are small (in this case ±50 rpm). This can be seen in the time interval 17-22, where the small changes in engine speed do
not result in recalculations of the TOTALMULFAC variable. The number
of serviced requests does not increase in this interval.

This experiment clearly shows that using a database system with ODTB
as the updating algorithm decreases the CPU load during a steady state
significantly compared to the original EECU software without database fa-
cilities.

5.4 Database Implementation in PC

The objective with the following experiments is to investigate how much
the choice of concurrency control algorithm affects the performance of the
system. The database implementation used in Experiment 3 is also used
in the experiments presented in this section. The operating system used in
these experiments is µC/OS-II, which is functionally equivalent to Rubus.
The real-time operating system and a database implementation execute in
a DOS command window on top of Windows 2000.¹ Hence, a simulation
environment has been created using the same software components as in
the EECU.

One important metric in the experiments in this section is the relative
consistency. In order to guarantee relative consistency in the multiversion
concurrency control algorithms, all updates scheduled by an updating al-
gorithm need to be executed. Thus, in the experiments in this subsection
all scheduled updates are executed. Hence, blockingf can be set to an
arbitrary value.

To compare single-version and multiversion concurrency control algo-
rithms, the OCC and OCC-S algorithms are altered to abort an update
and restart the UT if the UT, or one of its triggered updates, reads a data
item with a write timestamp with a higher value than the timestamp of
the UT. If such an event occurs this means that the UT would base its
result on values that were produced after the UT started. Instead, the UT
is restarted, gets a new timestamp, and reschedules the updates. The UTs
that commit are guaranteed to use values derived from the external envi-
ronment at the same time, i.e., the values are relatively consistent. The
algorithms are denoted RCR-OCC and RCR-OCC-S, where RCR denotes
relative consistency restarts.

¹µC/OS-II is used instead of Rubus because we have had problems doing precise time
measurements using the Windows version of Rubus.
Figure 5.18: Experiment 3: Performance results of a database implementation in an EECU. The performance metric is the number of cumulative recalculations.
5.4.1 Simulator Setup

A set of tasks is executing periodically, and they invoke UTs that execute with the same priority as the task. The tasks are prioritized according to RM, and the base period times are: 60 ms, 120 ms, 250 ms, 500 ms, and 1000 ms. These period times are multiplied with the ratio $32/\text{arrival\_rate}$, where 32 is the number of invoked tasks using the base period times, and $\text{arrival\_rate}$ is the arrival rate of UTs. The data item a UT derives is randomly determined by taking a number from the distribution $U(0,|D|)$. In the experiments a $45 \times 10^5$ database has been used. Every sensor transaction executes for 1 ms and every user transaction and triggered update executes for 10 ms. A simulation runs for 150 s with a specified arrival rate. The database system is running on $\mu$C/OS-II [3] in a DOS command window on an IBM T23 laptop running Windows 2000 servicepack 4. The PC has 512 Mb of RAM and a Pentium 3 running with 1.1 GHz. The user transactions are not started if they have passed their deadlines, but if a transaction gets started it executes until it is finished.

As in the discrete event simulator RADEx++, a value on every data item is simulated by adding a random value such that the value of a data item is always monotonically increasing. Every write operation creating the most recent version is adding, if not stated otherwise, a value from the distribution $U(0,350)$ to the previous most recent version. The data validity intervals are set to 400 for all data items. The creation of versions by multiversion concurrency control algorithms involves taking values of the two closest versions, one older and one newer and then randomly deriving a value that is not larger than the newer version. Base item updates are executing on average every 100 ms with a priority higher than UTs, i.e., the period time is 50 ms and for every base item $b_i$ there is a chance of 50% that $a\ b_i$ is updated. The memory pool used by the multiversion concurrency control algorithms is set to 300 data items and 150 of these are always used to store the latest version of every data item.

The updating algorithm in all conducted experiments presented in this section is the ODTB algorithm since it has shown (Experiment 2a) to give good performance. $\text{blockingf}$ is set to 1 and execution times on updates are not used since all scheduled updates are executed.
5.4.2 Experiment 4a: Committed User Transactions

The objective with this experiment is to investigate the throughput of single-version and multiversion concurrency control algorithms. The performance metric is successfully committed UTs, i.e., UTs that commit within the deadline.

The concurrency control algorithms that are evaluated are HP2PL, OCC, MVTO, MVTO-UV, MVTO-UP, and MVTO-CHECKSUM. As a baseline we also use the no concurrency control (NOCC) scheme. Figure 5.19(a) shows the number of user transactions committing before their deadlines for single-version algorithms without the restart facility. HP2PL and OCC perform the same. The OCC-S algorithm performs significantly better than similarity unaware single-version concurrency control algorithms.

In figure 5.19(b), the MVTO algorithm is performing bad, much worse compared to single-version concurrency control algorithms and the enhanced multiversion algorithms. The reason MVTO performs worse than MVTO-UV, MVTO-UP, and MVTO-CHECKSUM is the less number of transactions that can be skipped. MVTO cannot do the same accurate tests since similarity is not used as in the enhanced algorithms, and therefore more transactions are executed resulting in worse performance. The number of skips is plotted in figure 5.20. Comparing figures 5.19(a) and 5.19(b), the enhanced multiversion algorithms, MVTO-UV, MVTO-UP, and MVTO-CHECKSUM, perform better than HP2PL and OCC. The multiversion concurrency control algorithms can also guarantee relative consistency.

RCR-OCC and RCR-OCC-S are compared to MVTO and MVTO-UV in figure 5.21. The single-version algorithms with restarts are penalized by more restarts. Every restart is due to that values have changed, which increase the probability that TUs scheduled by the restarted UT cannot be skipped. Figure 5.4.2 shows that the number of restarts is higher for the restart algorithms. The difference in number of restarts between RCR-OCC and RCR-OCC-S is the number of restarts that can be saved by the similarity relation used in the verify phase.

Figure 5.21 shows that the MVTO-UV and MVTO-UP let considerably more UTs to commit compared to RCR-NOCC, RCR-OCC, and RCR-OCC-S.

MVTO-CHECKSUM is not up to par with MVTO-UV and MVTO-UP in figure 5.19(b). The reason is that MVTO-CHECKSUM must use fixed validity intervals and that values are monotonically increasing. Using a flexible validity interval, every new version creates a new interval and all
(a) Number of committed UTs for single-version concurrency control algorithms (confidence intervals are presented in figure A.7(a)).

(b) Number of committed UTs for multiversion concurrency control algorithms (confidence intervals are presented in figure A.7(b)).

Figure 5.19: Experiment 4a: Number of UTs committing before their deadlines using single- and multiversion concurrency control algorithms.
Figure 5.20: Experiment 4a: Number of transactions that can be skipped using ODTB in conjunction with the concurrency control algorithms.

Figure 5.21: Experiment 4a: A comparison of single-version concurrency control algorithms enforcing relative consistency and multiversion concurrency control algorithms (confidence intervals are presented in figure A.8).
values less than 400 are similar to the new version. However, using fixed validity intervals the value of a new version lies somewhere in the interval and the distance to the next interval is shorter compared to a flexible validity interval. In effect, fewer transactions can be skipped using fixed validity intervals since values are more often assigned different validity intervals. Figure 5.23(a) shows how the MVTO algorithms are behaving when fixed validity intervals are being used. The pool size is 300. In this setting, it is not possible to tell the difference from MVTO-CHECKSUM, MVTO-UV, and MVTO-UP. How pool sizes affect the performance is discussed in Experiment 4b, but here we only conclude from figure 5.23(b) that MVTO-CHECKSUM performs as good as MVTO-UV. However, the overhead of storing the values of elements in the read set is reduced to a 32-bit CRC in MVTO-CHECKSUM. Table 5.3 shows how many times MVTO-CHECKSUM makes the wrong decision in skipping a transaction, and there are no misses at all. As can be seen, using the CRC-32 is very robust. Hence, if fixed validity intervals are a reasonable design decision, then MVTO-CHECKSUM is a better choice than MVTO-UP since a smaller pool size can be used.
Figure 5.23: Experiment 4a: The similarity-aware multiversion concurrency control algorithms using fixed validity intervals.
Table 5.3: Experiment 4a: The number of times the checksum check misses to detect similar values compared to using values in MVTO-UV.

<table>
<thead>
<tr>
<th>Arrival rate</th>
<th>Missed similarities</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>60</td>
<td>0</td>
</tr>
</tbody>
</table>

5.4.3 Experiment 4b: Memory Pool Size

This experiment investigates how the memory pool size influences the performance of the system. The simulator use the settings described in the Simulator Setup section. The hypothesis is that MVTO-UP should note a larger decrease in performance when the pool size decrease. MVTO-UP uses more versions since a new version is stored even though it is similar to an already existing version.

Results in figure 5.24 support the hypothesis. MVTO-UP clearly has worse performance when the pool size is small. Note that at least 150 versions are used to store the current versions of all data items. The remaining versions in the pool are used by the concurrency control algorithm because of concurrent transactions.

Figure 5.23(b) shows the number of committed user transactions before their deadlines using multiversion concurrency control algorithms using fixed validity intervals. MVTO-CHECKSUM and MVTO-UV have the same performance, whereas MVTO-UP has worse performance for small pool sizes. If data items in a system are best modeled by fixed validity intervals, then MVTO-CHECKSUM is a good choice of a concurrency control algorithm.
Performance Evaluations

5.4.4 Experiment 4c: Priority Levels

This experiment investigates at which priority levels transactions are restarted. Restarts should preferably be based on priorities, restarting transactions with low priority before restarting those with higher priority. The purging and restart mechanisms of the MVTO concurrency control algorithms are designed for restarting low priority transactions first. Figure 5.25 shows how many transactions are restarted in each priority level.

The absolute number of restarts is highest for high prioritized transactions for HP2PL and OCC whereas the multiversion algorithms restart transaction in low levels first. This is indicated as the number of restarts drops for high prioritized transactions and this number is increasing for HP2PL and OCC for higher priority levels.

5.4.5 Experiment 4d: Overhead

The purpose of this experiment is to investigate how the overhead affect performance of concurrency control algorithms.

Experiment 4a has also been conducted on a slower computer (denoted computer two), and the results for NOCC and OCC on both computers are in figure 5.26(a), where NOCC-C2 and OCC-C2 are executed on computer two. Technical data of computer two are: Pentium 3 600 MHz, Windows 2000 Service Pack 4.
5.4. Database Implementation in PC

Figure 5.25: Experiment 4c: Number of restarts at different priority levels at an arrival rate of 40 UTs per second. Level one represents the highest priority.

All transactions have a fixed execution time, and the workload of the sensor transactions has an average utilization of $45/2 \times 1/100 = 0.225$. Thus, if the system had no overhead then the algorithms would give the same performance on both computers. Figure 5.26(b) shows that the database system introduces overhead that penalizes the performance on a slow computer. The scheduling of updates also takes time, and concurrency control algorithms add more overhead to the system and this can be seen in figure 5.26(b) where the OCC/OCC-C2 plot is above the NOCC/NOCC-C2 plot.

5.4.6 Experiment 4e: Low Priority

In this experiment an additional task with the lowest priority has been added, issuing one UT reading 25 data items. The period time of the new task is 1000 ms. The throughput of this transaction shows how the system can cope with, e.g., a diagnosis task executing with the lowest priority as in the EECU. The results in figure 5.27(a) show that the low prioritized task only gets time to execute at low arrival rates, because the system is overloaded at high arrival rates. No low prioritized transactions are successfully committed using the RCR algorithms. The reason can be seen in figure 5.27(b) plotting the restarts of the lowest prioritized transaction. This indicates that the values read by the transaction are not relative con-
(a) Number of committed UTs on fast and slow computers.

(b) Ratio of committed UTs of fast and slow computer.

Figure 5.26: Experiment 4d: An experiment executed on fast and slow computers.
sistent. This is expected since the read set has 25 members, and it is quite likely that at least one of them has changed making the set not relative consistent. In this case using a large read set, we see that multiversion concurrency control gives better performance since fewer transactions need to be restarted.

5.4.7 Experiment 4f: Transient State

In this experiment, the simulator is setup as in experiment 4a, but value changes are not random. Instead every write of a data item increase its value with 450 making every new version outside the data validity intervals since these are set to 400 for every data item. The ODTB algorithm cannot skip as many transactions as in experiment 4a. Comparing results in figure 5.28 to those in figure 5.19 show that the throughput of UTs has lessened when values change with 450 instead of randomly. The few restarts of multiversion concurrency control algorithms do not affect the performance, and they perform the same as using no concurrency control at all. In this experiment, OCC-S has the same performance as OCC, which is expected since no restarts can be skipped in OCC-S due to that values are never similar.

The RCR algorithms have considerably worse performance compared to results presented in figure 5.21. The reason is again that values are not similar. When values are not similar, updates cannot be skipped and changes in a base item are propagated to all its descendants. This means that it is likely that some values read by a transaction are derived after the transaction started which results in a restart of the transaction.

5.5 Observations

The performance evaluations show that ODTB and ODDFT_C give the best performance since they give the functionality to skip unnecessary updates. The performance evaluations of concurrency control algorithms show that when the system is in a state where the values on data items change moderately, almost all conflicts that occur in single-version concurrency control involve similar data, and, thus, using concurrency control can be questioned. This confirms the same questions Graham addressed in [21]. Graham discussed that inherent in the design of software for embedded systems, the access to data items might enforce no concurrency, and, thus,
Figure 5.27: Experiment 4e: Performance of an additional task with the lowest priority issuing a transaction reading 25 data items.

(a) Number of committed UTs executing with lowest priority.

(b) Number of restarts at low arrival rates of the transaction with the lowest priority.
5.5. Observations

Figure 5.28: Experiment 4f: Every write of a data item changes its value with 450 (confidence intervals are presented in figure A.9).

no concurrency control is needed. The experiments shown in this chapter show that even in a randomly generated data dependency graph and random data accesses, conflicts often involve similar values. Moreover, multiversion concurrency control algorithms can greatly improve the performance of the system since these algorithms keep a history of values and can accurately determine if a transaction can be skipped. These algorithms also have the benefit of guaranteeing relative consistency. In essence, if similarity can be used, without adding too much overhead, it should in many cases increase the performance of the system.

This section gives data on the storage overhead for using a real-time operating system, a database implementation, and pregenerated schedules. The schedule in figure 4.9 consists of 9 elements. The pregenerated schedule being used in the RADEEx++ simulations consists of 246 elements. Assuming the execution times can be stored in a byte the storage of a pregenerated schedule takes 18 and 492 bytes of memory. In the database implementation, 500 bytes of flash memory is used to represent the schedule and data relationships, and 1.5 Kb of RAM for data items, meta-information, queues for EDF and concurrency control, and statistics. The code size is 10 Kb and 8 Kb for the database system and Rubus respectively. Memory pools for database functions, mutexes, stacks for the periodic tasks take 19 Kb of RAM.
Chapter 6

Related Work

The main focus of the research in this project has been on similarity and applying similarity to real-time embedded systems. Similarity was first used in concurrency control algorithms [39], but has later on also been used when optimizing task period times [30], updating of data [68], and data dissemination on the web [16, 47, 56]. Similarity, as adopted in this thesis, can be divided into updating algorithms and concurrency control algorithms. The following sections relates the work achieved in this thesis to previous work done on data freshness (section 6.1), and to concurrency control algorithms (section 6.2).

6.1 Updating Algorithms and Data Freshness

In order to utilize the CPU resource as efficient as possible unnecessary updates must be avoided. Fixed updating schedules cannot achieve this [30, 36, 44, 71]. Hamdaoui and Ramanathan introduced \((m, k)\)-firm deadlines, where \(m\) deadlines out of \(k\) consecutive invocations of a task have to be met [26]. Hence, an invocation of a task can be skipped and it can be used to even out the load during an overload of the system and, thus, it increases the possibility of tasks to meet at least \(m\) deadlines. However, the \((m, k)\)-firm deadlines are unaware of data freshness in the value domain, and updates of data items are invoked even though values are unchanged. Thus, although skips are possible using \((m, k)\)-firm deadlines, resources are not efficiently used at steady states as they are using updating algorithms ODDFT_C, ODKB_C, or ODTB.

Kuo and Mok introduce a similarity bound saying that two writes to a
data item are similar if the time between them is less than the similarity bound \([39, 40]\). Hence, data freshness is in practice defined in the time domain. Wedde et al. define data freshness as \(|old – new| \leq bound\), i.e., data freshness is defined in the value domain of data items \([68]\). The updating of data items works as follows \([68]\). The system is distributed and tasks are executing at designated nodes. Tasks are non-preemptable and do not migrate to other nodes. Further, tasks are using either local or remote data and are executed until completion but are seen as failed if they miss their deadlines. Every node has an object manager and if an object has changed outside a given similarity bound, then the object manager notifies the object managers at the other nodes where tasks use this object. Tasks reading the object are marked and only marked tasks need to execute. A transaction instance starts to execute when the object manager marks any of its tasks as changed. A transaction instance can be seen as a schedule of updates as generated by ODDFT. ODDFT _C and ODTB are also updating algorithms that can skip transactions, but they are designed for a single-CPU system. ODDFT _C and ODTB differ in two ways from the updating algorithm of Wedde et al.: (i) ODDFT _C and ODTB can dynamically create the updating schedule, and (ii) ODDFT _C and ODTB generate and execute a schedule once for every UT. In contrast the algorithm presented by Wedde et al. re-executes the transaction instance as soon as a task is marked for execution, i.e., updates in the pregenerated schedule are re-executed. Thus, ODDFT _C and ODTB are aimed for being used on-demand.

For a single-CPU system, ODTB together with the RCR concurrency control algorithms have almost the same functionality as the updating algorithm presented by Wedde et al. The difference is that ODTB together with the RCR algorithms are used on-demand, but in the system presented by Wedde et al., data items need to be recalculated as soon as they might be changed (such an updating approach is denoted updates-first). Adelberg et al. investigate the difference in performance between on-demand and updates-first and find that on-demand updating of data performs better than updates-first \([5]\). The evaluations of similarity-aware multiversion concurrency control in chapter 5 show that ODTB with multiversion concurrency control lets more UTs commit within deadlines compared to using ODTB with single-version concurrency control with RCR as used by Wedde et al in \([68]\).

Imprecise computational tasks consist of a mandatory part and an optional part \([46]\). The optional part is normally finished, but during an
overload the optional part can be skipped, i.e., an approximate value produced by the mandatory part is used and freed CPU resources can be used to execute as many tasks as possible. In contrast, in the approach proposed in this thesis, an update is skipped only when the value is already up-to-date or when it is impossible to make the value up-to-date. Hence, no approximate results are produced. Furthermore, during a steady state when the system is not overloaded, the imprecise computation technique would produce fully accurate values and, thus, not skip any calculations at all which means that CPU resources are not freed at a steady state as it is using ODKB_C, ODDFT_C, and ODTB.

A view can be either a virtual relation derived each time it is requested or a materialized relation, i.e., stored in a database. Data objects (a data object is a relation of data values) are stale when there exists at least one unapplied update which the data object depends on [41]. This resembles the $pa$ timestamp in ODDFT and ODKB_V where data items are assumed to be changed when at least one parent has changed. In ODTB, on the other hand, a data item with $pa > 0$ is stale. Using ODTB, there is no need to mark data items as potentially affected. Thus, the number of marked data items decreases and it is therefore easier to make correct decisions on which data items that need to be updated. Data freshness in [41] is used for defining quality of data and to schedule updates to relations and views. In comparison, in our work data items have scalar values and therefore it is possible to decide if a change in a value of a data item affects the values of other data items. This is done by introducing validity intervals.

Kao et al. introduce definitions of data freshness for discrete data objects in [34] that are based on the time domain. A hybrid updating scheme is proposed that updates data items immediately during idle time and on-demand when a transaction arrives. The hybrid updating scheme is shown to maintain data freshness better than on-demand. However, the hybrid updating scheme cannot skip unnecessary updates and adapt the number of updates to the state of the system due to the adoption of time domain for data freshness. Adelberg et al. [6] investigated how recomputations of derived data affect transaction and data timeliness. They found that a forced delay can be used for delaying recomputations and thereby allow more updates of data items to arrive before a recomputation is started. The data validity intervals introduced in this thesis work like a forced delay since several small changes are skipped and when the change is large enough, i.e., outside the allowed data validity interval, an update is triggered.
Blakely et al. show how it is possible to decide which updates to base and derived data objects affect the views, i.e., derived data objects [10]. It is assumed that a data object is a relation, i.e., contains several columns of data. Modifications to data values might not affect a view and the test is complex and all changes to data values are considered for determining staleness of a view [10]. In our approach, however, a data item is a scalar value and the freshness of a data item can easily be tested using inequality tests for each read set member. By doing a top-bottom traversal of a graph it is possible to determine stale data items.

For hard real-time systems static period times for updates and calculations are derived to guarantee freshness and timeliness of produced results [30,36,44,71]. If unnecessary updates should be avoided, static period times are not feasible because an update is executed in every period, but the input to the update might not have changed since the previous period implying that updates are unnecessarily executed. In this research project, we are considering a soft real-time system and the objective is to achieve a highly efficient utilization of the CPU. Hence, static period times are unnecessarily restrictive to achieve an effective utilization of the CPU. Specific updating frequencies for each mode could be assigned to the data items. In this context, mode refers to that for each mode data items have a specific updating frequency, and switching between modes, e.g., because of a knocking engine, means that the updating frequencies change on the data items. An example of a mode change is start enrichment compensation factors in the EECU software. These compensation factors change much when the engine is started, then the values gradually stabilize, and finally these compensation factors are not used any more. Our proposed updating schema for freshness maintenance of data do not need mode changes since all modes are covered by the data validity intervals, and this gives varying updating frequencies in each mode.

Datta and Viguier describe transaction processing for rapidly changing systems, where base item updates are committed even though the updates do not affect other data items, i.e., unnecessary updates are executed [15]. Moreover, in their work calculations only depend on base items, i.e., intermediate results are not considered, whereas in this thesis arbitrary dependencies among data items are allowed.

Flexible period times are used to adjust period times between two limits by a feedback controller such that the workload of updates is lessened when the miss ratio of transactions is above a specific reference [33]. In their
work, all produced values are valid if the updating frequency lies within the bounds, but similarity of data values is not considered at all. Thus, unnecessary updates can still happen if the value is similar between updates.

Data-deadline and forced wait [70] are designed to achieve freshness at the deadline of a user transaction as in our work indicated by time \( t \) in the function \( \text{error} \) used in function AssignPrio. However, Xiong et al. only consider base data that is updated periodically. In contrast to our proposed algorithms, the data-deadline and forced wait cannot deal with derived data [70].

6.2 Concurrency Control

Concurrency control for computer systems in general has been studied for a long time and is a well-explored area. We observe that evaluations are primarily performed using simulators. Unfortunately, the lack of benchmarks for real-life settings can make it hard to decide which concurrency control algorithm is best suited for an application. The performance evaluations reported in chapter 5 are done using a real-time operating system and well-known concurrency control algorithms implemented to execute on the operating system. The results improve our understanding on how concurrency control algorithms affect the performance in a real-life system.

Two-phase locking and optimistic concurrency control have been evaluated for real-time systems [13, 28, 32]. In some of these experiments, it is found that optimistic concurrency control algorithms give better performance than two-phase locking algorithms, but in the experiments high parallelism—achieved by simulating, e.g., 20 CPUs and disks—is used in order to stress the concurrency control algorithms. Such a setting is not plausible for most real-time embedded systems. We have found that HP2PL and OCC give similar performance when they are executed and evaluated in a more realistic setting. This is due to that transactions are executing with fixed priorities and limitations given by the real-time operating system; it is impossible to restart a currently executing transaction, and dynamic priorities are not supported. To the best of our knowledge, no evaluation of the performance of HP2PL and OCC on such system has been documented elsewhere.

Lam et al. show evaluations of concurrency control algorithms for mixed soft real-time transactions and non-real-time transactions [42]. They found that an integrated TO scheduler using OCC for soft real-time transactions
and 2PL for non-real-time transactions perform best. However, for systems where relative consistency is important for the transactions, our evaluation of the RCR algorithms shows that single-version algorithms perform poorly, and, thus, indicates that the integrated TO scheduler is not suited for such systems.

Multiversion concurrency control algorithms have also been evaluated [58, 59, 62]. It has been found that 2PL performs better than MVTO and the single-version timestamp ordering concurrency control algorithm [62]. Song and Liu evaluate the 2PL and OCC multiversion algorithms in a hard real-time system [59]. In their work, a set of data items is said to be temporally consistent when they are absolute and relative consistent. The evaluation results show that temporal consistency is highly affected by the transaction conflict patterns and also, OCC is poor in maintaining temporal consistency in systems consisting of periodic activities. Our evaluations show that MVTO-based algorithms are free of restarts (except for when the memory pool becomes full) and, thus, the conflict pattern does not affect MVTO-based algorithms.

We extend the OCC algorithm to being similarity-aware in the verification phase. Similarity has been added to other single-version concurrency control algorithms: HP2PL by Lam et al. [43] and O2PL by Wedde et al. [68]. The proposed multiversion concurrency control algorithms, MVTO-UV, MVTO-UP, and MVTO-CHECKSUM use similarity. To the best of our knowledge, using multiversion concurrency control and similarity is a novel approach. The main reason to use multiversion concurrency control is to be able to guarantee relative consistency. This can also be guaranteed by using a snapshot technique using wait-free locks [63]. The multiversion concurrency control algorithms are also lock-free. The size of the memory pool can only be analyzed in an off-line step if worst-case period times are assumed on sporadic tasks, as in the approach of Sundell and Tsigas [63]. However, this can result in waste of resources when using similarity in a soft real-time system, because similarity can reduce the need to store versions, e.g., when the external environment is stable. Hence, in some systems it can be feasible to limit the needed memory and pay the prize by restarting transactions when the memory pool becomes full. We have taken this approach in our database since when the external environment starts changing rapidly the system becomes overloaded by necessary updates. Low priority transactions will miss their deadlines and they can therefore be restarted to free memory. Our performance evaluations indicate that it pays off in
terms of performance to use similarity in updating algorithms as well as in concurrency control.

Relative consistency can be important and there are different ways to achieve relative consistency among data items. Wedde et al. use similarity in updating algorithms and concurrency control, and they use a single-version concurrency control. In their approach, to guarantee relative consistency transactions are restarted until they use fresh values [68]. These restarts are the same as the RCR algorithms. The performance evaluations show that using a multiversion concurrency control algorithm aware of similarity significantly increases performance compared to well-established single-version concurrency control algorithms. The evaluations also show that multiversion concurrency control using a limited memory pool can be constructed to better obey priority on transactions than HP2PL and OCC. When the memory pool becomes full MVTO, MVTO-UV, MVTO-UP, and MVTO-CHECKSUM start restarting active transactions with lowest priority until there are enough memory for the current operation.
Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this research project, we investigate efficient data management in real-time embedded systems ensuring high data quality. Efficient utilization of available resources is achieved by using data similarity. The contributions of this research are:

- Updating algorithms using similarity: on-demand depth-first traversal (ODDFT), on-demand breadth-first traversal (ODBFT), on-demand top-bottom traversal with relevance check (ODTB), and on-demand depth-first traversal with relevance check (ODDFT_C).

- Multiversion concurrency control algorithms with similarity: multiversion timestamp ordering using versions (MVTO-UV), multiversion timestamp ordering using memory-pool (MVTO-UP), and multiversion timestamp ordering using checksum (MVTO-CHECKSUM).

- A database implementation with support for data similarity and absolute and relative consistency.

The objective of introducing updating algorithms is to let the database system, using an updating algorithm, ensure that transactions read valid data items. Using an updating algorithm eases the task of the programmer who does not need to consider updating of data items. There are two families of updating algorithms, those scheduling updates based on age of data items, which is the current approach often used in real-time systems,
and those scheduling updates based on value of data items. This work has focused on using similarity of data items, which has the potential to reduce the number of updates whenever possible, e.g., when the system changes state. Performance evaluations show that updating algorithms using similarity reduce the number of executed updates significantly compared to periodic updating of data. Moreover, similarity-aware updating algorithms have been extended with a relevance check making it possible to skip calculations. The correctness of this approach has been evaluated in a real-life setting consisting of engine control software using the database implementation. Performance evaluations show that using similarity and a relevance check considerably improve the performance of the system.

The objective with introducing multiversion concurrency control is to guarantee a transaction reads relatively consistent values of data items. The industrial partners have found that data needs to be correlated in time, e.g., in a diagnosis of a system. We have extended the multiversion timestamp ordering concurrency control algorithm with similarity (MVTO-S). Using updating algorithms with relevance check together with MVTO-S have shown to guarantee transactions are given a snapshot of the database that is valid when a transaction started. Also, the updating algorithm together with MVTO-S give restart-free transactions, which greatly improves performance compared to well-established concurrency control algorithms. Performance evaluations are conducted using an implementation of the database system on a real-time operating system. In the evaluations, the updating algorithm is fixed to ODTB and different concurrency control algorithms are evaluated. The evaluations show that implementations of MVTO-S perform significantly better than HP2PL and OCC, which are common concurrency control algorithms for real-time systems. MVTO-S guarantees relative consistency of data items without affecting the performance of the system, whereas HP2PL and OCC need to introduce more restarts to guarantee relative consistency. Furthermore, it is found in the performance evaluations that most conflicts in HP2PL and OCC involve similar values.

7.2 Future Work

The CPU overhead of the algorithms are not thoroughly measured in chapter 5 and this needs to be done. One problem is that the collection of statistical data is consuming CPU resources, i.e., the probe effect, and,
Conclusions and Future Work

thus, gives a negative effect on the performance of the algorithms. Hence, it is difficult to measure the true overhead of the algorithms. Furthermore, it is not clear to what the overhead costs should be compared. The original EECU software, for instance, stores data using registers using no concurrency control and no transactions. The CPU overhead of the proposed database system is probably noticeably higher than for the simpler system in the original software. The effect of overhead is only noted when the system is overloaded, and the system becomes overloaded at a lower incoming load when the overhead increases. However, our algorithms can skip calculations which compensates for the added overhead. More measurements need to be made to clearly establish the amount of CPU time saved.

Admission control of transactions is needed since the system can sporadically become overloaded. A policy for prioritizing calculations and rejecting low prioritized calculations is needed. An admission controller may have a negative impact on the snapshot capability of the multiversion concurrency control algorithms. It should be investigated how an admission controller can cooperate with a multiversion concurrency control algorithm to achieve both a good approximation of a snapshot and execution of the most important calculations.

Requirement R2 puts a requirement on active behavior on the data management of the EECU software to monitor data and activate a task when a condition is fulfilled. The proposed updating algorithms use a data dependency graph both in scheduling updates but also to determine staleness of data items by using the potentially affected timestamp \( p_a \). It is likely that there are conditions that have to be fulfilled before a calculation is performed. Thus, this affects the way \( p_a \) should be distributed by the second step in the updating schema (PAUS and AUS). These conditions could be modeled by removing and adding edges to the data dependency graph. Dynamically adding and removing edges may show that the accuracy in scheduling updates that are needed is improved, and that the database system gets an active behavior, but more investigation in this area is needed.

Real-time embedded systems can be nodes in a network. One example is the different ECUs in a car that are connected by a CAN network. Common data items could then be shared among the nodes using a distributed database. Hence, the proposed database system would be extended to handle distributed data and the updating and concurrency control algorithms need to be distributed. We have found that multiversion concurrency control algorithms using similarity can greatly enhance the performance on a
single-CPU system. It would be interesting to investigate if this holds for a distributed system.

The similarity-aware multiversion concurrency control algorithms proposed in this research project store a new version when the write timestamp is larger than the timestamp of the oldest active transaction. Further optimizations can be made if it is the case that many versions are not needed by the transactions. A more thrifty policy of storing versions can reduce the number of restarts, because restarts are due only to a full memory pool. Reducing the number of restarts gives better performance since resources are not wasted.

To be able to analyze the updating algorithms offline in order to make predictions on how the system will behave at runtime, a probabilistic approach can be taken. Manolache has developed a schedulability analysis for real-time systems with stochastic task execution times [48]. In addition to stochastic execution times, the value of data items might also be modeled by stochastic processes.
Bibliography


Appendix A

Graphs with Confidence Intervals

This appendix contains selected graphs from the Performance Evaluations chapter that are plotted with 95% confidence intervals.
Database size 45*10^5. Number of committed UTs

(a) Number of committed UTs.

Database size 45*10^5 Valid transactions, Validity bounds

(b) Number of valid committed UTs.

Figure A.1: Experiment 1a: Consistency and throughput of UTs.
Figure A.2: Experiment 1a: Effects of measuring staleness of data items at deadline of UT.
Figure A.3: Experiment 1b: Number of generated triggered updates where UTs are AUT.

Figure A.4: Experiment 1c: Number of valid committed UTs using either the pa timestamp or a change flag to indicate potentially affected data items.
Figure A.5: Experiment 2a: Consistency and throughput of UTs with a relevance check on triggered updates using ODDFT.
(a) Number of valid committed UTs for the ODDFT_C, ODTB, and ODKB_C with parameters such that they perform the best.

(b) Number of generated triggered updates.

Figure A.6: Experiment 2b: Performance for updating algorithms that has the possibility to skip updates.
(a) Number of committed UTs for single-version concurrency control algorithms.

(b) Number of committed UTs for multiversion concurrency control algorithms.

Figure A.7: Experiment 4a: Number of UTs committing before their deadlines using single- and multiversion concurrency control algorithms.
Figure A.8: Experiment 4a: A comparison of single-version concurrency control algorithms enforcing relative consistency and multiversion concurrency control algorithms.

Figure A.9: Experiment 4f: Every write of a data item changes its value with 450.
Maintaining Data Consistency in Embedded Databases for Vehicular Systems

The amount of data handled by real-time and embedded applications is increasing. This calls for data-centric approaches when designing embedded systems, where data and its meta-information (e.g., temporal correctness requirements) are stored centrally. The focus of this thesis is on efficient data management, especially maintaining data freshness and guaranteeing correct age on data.

The contributions of our research are updating algorithms and concurrency control algorithms using data similarity. The updating algorithms keep data items up-to-date and can adapt the number of updates of data items to state changes in the external environment. Further, the updating algorithms can be extended with a relevance check allowing for skipping of unnecessary calculations. The adaptability and skipping of updates have positive effects on the CPU utilization, and freed CPU resources can be reallocated to, e.g., more extensive diagnosis of the system. The proposed multiversion concurrency control algorithms guarantee calculations reading data that is correlated in time.

Performance evaluations show that updating algorithms with a relevance check give significantly better performance compared to well-established updating approaches, i.e., the applications use more fresh data and are able to complete more tasks in time. The proposed multiversion concurrency control algorithms perform better than HP2PL and OCC and can at the same time guarantee correct age on data items, which HP2PL and OCC cannot guarantee. Thus, from the perspective of the application, more precise data is used to achieve a higher data quality overall, while the number of updates is reduced.

Real-Time Systems, Data Management, Concurrency Control, Scheduling, Consistency models, Data similarity, Vehicular Systems
No 17
Vojin Plavsic: Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)

No 28

No 29

No 48

No 52

No 60

No 71

No 72

No 73
Ola Strömoffs: A Structure Editor for Documents and Programs, 1986.

No 74

No 104

No 108

No 111

No 113

No 118

No 126

No 127

No 139

No 140

No 146

No 150

No 165

No 166

No 174

No 177
Peter Åberg: Design of a Multiple View Presentation and Interaction Manager, 1989.

No 181
Henrik Eriksson: A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.

No 184

No 187

No 189

No 196

No 197

No 203

No 212

No 230

No 237

No 250

No 253

No 260

No 283

No 298

No 319

No 326

No 328

No 333

No 335

No 348

No 352

No 371

No 378
Johan Ringström: Compiler Generation for Parallel Languages from Denotational Specifications, 1993.

No 383

No 386

No 398


Eva Toller: Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.


Dan Fristedt: Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.


Staffan Fldin: Efficient Management of Object-Oriented Queries with Late Binding, 1996.


Magnus Werner: Multidatabase Integration using Polymorphic Queries and Views, 1996.

Mikael Lind: Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
FiFa-4  Carita Åbom: Videomötesteknik i olika affärsituationer - möjligheter och hinder, 1997.
FiFa-6  Tommy Wedlund: Att skapa en företagsanpassad systemutvecklingsmodell genom rekonsstruering, värdering och vidareutveckling i T30-bolag inom ABB, 1997.
No 627  Fredrik Eklund: Declarative Error Diagnosis of GAPLog Programs, 1997.
No 653  Mats Gustafsson: Bringing Role-Based Access Control to Distributed Systems, 1997.
No 668  Per-Ove Zetterlund: Normalization of Swedish redovisning - En studie av tillkomsten av Redovisningsrådets rekommendation om koncernredovisning (RR01:91), 1998.
FiFa-16  Marie-Therese Christiansson: Inter-organisatorisk verksamhetsutveckling - metoder som stöd vid utveckling av partnerskap och informationssystem, 1998.
FiFa-25  Bengt E W Andersson: Samverkande informationssystem mellan aktörer i offentliga åtaganden - En teori om aktörssamband i samverkan om utbyte av information, 1998.
No 742  Pawel Pietrzak: Static Incorrectness Diagnosis of CLP (FD), 1999.
No 766  Martin V. Howard: Designing dynamic visualizations of temporal data, 1999.
No 787  Charlotte Björkgren: Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999.
No 807  Svein Bergum: Managerial communication in telework, 2000.


Att komponentbasera informationssystem - Vad säger teori och praktik?, 2000.


Fuzzy Control of Unmanned Helicopter, 2002.

Automatic Parallelization of Simulation Code from Equation Based Simulation Languages, 2002.

IT-system och filtrering av verksamhetskunskap - kvalitetsproblem vid analyser och beslutsfattande som bygger på uppgifter hämtade från polisens IT-system, 2001.

IT-system och filtrering av verksamhetskunskap - kvalitetsproblem vid analyser och beslutsfattande som bygger på uppgifter hämtade från polisens IT-system, 2001.

| No 1084 | Charlotte Stoltz: Calling for Call Centres - A Study of Call Centre Locations in a Swedish Rural Region, 2004. |
| No 1095 | Ulf Johansson: Rule Extraction - the Key to Accurate and Comprehensible Data Mining Models, 2004. |
| Fif-a 77 | Ulf Larsson: Designarbete i dialog - karaktärisering av interaktionen mellan användare och utvecklare i en systemutvecklingsprocess, 2004. |
| No 1127 | Per-Ola Kristensson: Large Vocabulary Shorthand Writing on Stylus Keyboard, 2004. |