Linköping Studies in Science and Technology

Dissertation No. 1120

Management of Real-Time Data Consistency and Transient Overloads in Embedded Systems

by

Thomas Gustafsson

National Graduate School of Computer Science

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

Linköping 2007

ISBN 978-91-85831-33-3
ISSN 0345-7524
Printed in Linköping, Sweden, by LiU-Tryck

Abstract

This thesis addresses the issues of data management in embedded systems' software. The complexity of developing and maintaining software has increased over the years due to increased availability of resources, e.g., more powerful CPUs and larger memories, as more functionality can be accommodated using these resources.

In this thesis, it is proposed that part of the increasing complexity can be addressed by using a real-time database since data management is one constituent of software in embedded systems. This thesis investigates which functionality a real-time database should have in order to be suitable for embedded software that control an external environment. We use an engine control software as a case study of an embedded system.

The findings are that a real-time database should have support for keeping data items up-to-date, providing snapshots of values, i.e., the values are derived from the same system state, and overload handling. Algorithms are developed for each one of these functionalities and implemented in a real-time database for embedded systems. Performance evaluations are conducted using the database implementation. The evaluations show that the real-time performance is improved by utilizing the added functionality.

Moreover, two algorithms for examining whether the system may become overloaded are also outlined; one algorithm for off-line use and the second algorithm for on-line use. Evaluations show the algorithms are accurate and fast and can be used for embedded systems.

Acknowledgements

I have learned many things during my time as a doctoral student. One thing, that is related to this very page, is that if people is reading something of a thesis it is very probably the acknowledgements. Why is that? I think it is normal human curiosity and I better start meeting the expectations... The cover does not reflect anything in my research nor has it any other deeper meaning; it is just a nice picture of a sand dune in Alger close to the West Saharan border taken by Elinor Sundén in February 2006.

Has life as a doctoral student been as I expected? Of course not! Have I learned as much as I thought I would? I have indeed. Probably a lot more. Most importantly, have I taken steps towards becoming a researcher? People must judge by themselves, but I certainly hope so. My supervisor Jörgen Hansson has believed in me and created an environment and an atmosphere where I have had the opportunity to develop skills necessary for my studies. I thank him, Mehdi Amirijoo, Aleksandra Tešanović and Anne Moe, and the rest of RTSLAB, ESLAB, TCSLAB, and late Emma Larsdotter Nilsson for providing this "world of research" where I have been living the past 5 years. It has been a great pleasure to get to know and work together with you.

I would like to thank the master thesis students Martin Jinnerlöv, Marcus Eriksson, Hugo Hallqvist, and Ying Du who have contributed to implementations for the research project. I would also like to thank Anders Göras, Gunnar Jansson and Thomas Lindberg from Mecel AB (now HOERBIGER Control Systems AB) for valuable input and comments to my research. Further, I thank Sven-Anders Melin, Jouko Gäddevik, and David Holmgren at GM Powertrain Europe.

My studies has been funded by Information Systems for Industrial Control and Supervision (ISIS) and I have been enrolled in the National Graduate School in Computer Science (CUGS).

Since September 2006 I am working in a new group at School of Engineering, Jönköping University. I thank especially Kurt Sandkuhl and his group for giving me this opportunity of conducting research in a, to me, new compelling area.

Last I would like to send my love to my family and in particular Elinor who makes life so much more fun.

Thomas Gustafsson
Jönköping, July 2007

CONTENTS

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", and was done in collaboration with Mecel AB and General Motors Powertrain Sweden; both companies are working with engine control software for cars. This thesis addresses data management issues that have been identified as challenges during the course of maintaining and developing embedded systems' software.

Section 1.1 gives a short summary of this thesis. Section 1.2 presents data management problems of embedded systems' software. Section 1.3 states the research goals of the thesis. Section 1.4 summarizes the research contributions achieved in this thesis. Section 1.5 lists published papers by the author, and, finally, Section 1.6 outlines the thesis.

1.1 Summary

This section gives a short summary of the problem we have studied and our achieved results.

Real-time systems are systems where correct functionality depends on derived results of algorithms and at which time a result was derived [24]. An embedded system is part of a larger system in which the embedded system has a specific purpose, e.g., controlling the system in a well-defined way [32]. Embedded systems have usually resource constraints, e.g., limited memory, limited CPU and limited energy. Further, some embedded systems are controlling systems that control the environment they are installed in. These embedded systems perform control actions by monitoring the environment and then calculating responses. Thus, it is important that data values the calculations use are up-to-date and correct. This thesis focuses on maintaining consistency of data values and at the same time utilizing the CPU efficiently. The performance of soft real-time embedded systems is measured with respect to deadline miss ratio. Algorithms are proposed that utilizes the CPU better, compared to existing algorithms, by enlarging the time between updates. Further, analytical formulae are proposed to estimate the workload imposed by updates of data items when using our algorithms. In addition, the proposed analytical formulae can be used both on-line and off-line to estimate the schedulability of a task set.

1.2 Motivation

This section gives an introduction to software development and database systems that highlight difficulties in developing software, which motivates the work we have conducted.

1.2.1 Software Development and Embedded Systems

Embedded systems are nowadays commonplace and can be found in many different application domains, from domestic appliances to engine control. Embedded systems are typically resource-constrained, dependable, and have real-time constraints [106]. A large part of all CPUs that are sold are used in embedded systems, thus, software that runs in embedded systems constitutes the main part of all software that is developed [23], which stresses the importance of finding adequate methods for developing software for embedded systems.

The software in embedded systems is becoming increasingly complex because of more functional requirements being put on them [32]. Verum Consultants analyzed embedded software development in the European and U.S. automotive, telecommunications, medical systems, consumer electronics, and manufacturing sections [31], and they found that currently used software development methods are unable to meet the demands of successfully developing software on time that fulfills specified requirements. They also observed that some embedded software roughly follows Moore's law and doubles in size every two years. Figure 1.1 gives a schematic view of areas that contribute to the complexity of developing and maintaining a software [34]. As we can see in Figure 1.1, software complexity does not only depend on the software and hardware related issues—e.g., which CPU is used—but also on the human factor, e.g., how hard/easy it is to read and understand the code. Many techniques have been proposed over the years that address one or several of the boxes in Figure 1.1. Recently, the most dominant technique has been component-based software development [36]. An example of this is a new initiative in the automotive industry called AUTomotive Open System ARchitecture (AUTOSAR) where the members, e.g., the BMW Group and the Toyota Motor Corporation, have started a standardization of interfaces for software in cars [2, 67]. The ideas of AUTOSAR are to support:

- management of software complexity;

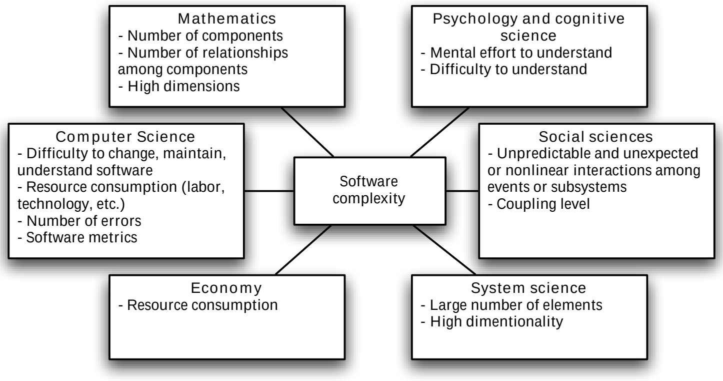

Mathematics
- Number of components
- Number of relationships among components
- High dimensions
Psychology and cognitive science
- Mental effort to understand
- Difficulty to understand
Computer Science
- Difficulty to change, maintain, understand software
- Resource consumption (labor, technology, etc.)
- Number of errors
- Software metrics
Software complexity
Social sciences
- Unpredictable and unexpected or nonlinear interactions among events or subsystems
- Coupling level
Economy
- Resource consumption
System science
- Large number of elements
- High dimentionality

Figure 1.1: Software complexity [34].

- flexibility for product modification, upgrade and update;
- scalability of solutions within and across product lines; and
- improved quality and reliability of embedded systems in cars.

The embedded systems we focus on in this thesis are composed of a controlled system and a controlling system [110], which is typical of a large class of embedded systems. Moreover, we have access to an engine control software that constitutes our real-life system where proof of concepts can be implemented and evaluated. This embedded system adheres to the control and controlling system approach.

The controlling system monitors the external environment by reading sensors, and it controls the controlled system by sending actuator values to actuators. Normally, the timing of the arrival of an actuator signal at an actuator is important. Thus, most embedded systems are real-time systems where the completion of a task must be within a specified time-frame from its start (for a further discussion of real-time systems see Section 2.1, Real-Time System). It is critical that values used by calculations correctly reflect the external environment. Otherwise actuator signals might form inaccurate values and therefore the controlling system does not control the controlled system in a precise way. This may lead to degraded performance of the system, or even have catastrophic consequences where the system breaks down, e.g., lengthy and repeating knocking of an engine.

1.2.2 Software Development and Engine Management Systems

Now we introduce a specific embedded system that is used throughout this thesis. It is an engine management system where we have concentrated on the engine control software.

Computing units are used to control several functional parts of cars, e.g., engine, brakes, and climate control. Every such unit is denoted an electronic control unit (ECU). Development and maintenance costs of software is increasing and one large part of this cost is data handling [17, 89]. The industrial partners have recognized that also the ECU software is becoming more complex due to increasing functionality (this is also acknowledged elsewhere [2, 22, 23, 45, 67]). The software in the engine electronic control unit (EECU) is complex and consists of approximately 100,000 lines of C and C++ code. One reason for this complexity is law regulations put on the car industry to extensively diagnose the ECUs; the detection of a malfunctioning component needs to be done within a certain time after the component breaks [99]. The diagnosis is a large part of the software, up to half of it, and many data items are introduced in the diagnosis [99]. Moreover, the software has a long life cycle, as long as several car lines, and several programmers are involved in maintaining the software. In addition to this, calculations in the 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 hard or soft real-time requirements. Further, the specifications of the engine management system we have access to are a 32-bit 16.7 MHz CPU with 64 kB RAM and it started to be used circa 15 years ago. Our intentions are to learn the properties of embedded systems' software, and in particular how data is managed in embedded systems.

The industrial partners have identified problems with their current approach of developing embedded software. These include:

- Efficiently 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. There is work done on determining fixed updating frequencies on data items to fulfill freshness requirements [68, 76, 88, 136] (see Chapter 6). 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

[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.

before. Hence, the recalculation is essentially unnecessary and resources are not utilized efficiently.

1.2.3 Databases and Software Development

Databases are used in many different applications to organize and store data [15, 19, 104]. 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. 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 orders of Gb of data and advanced query languages such as SQL. One feature of a database is the addition and deletion of data items, i.e., the data set can be dynamic and change when the system is running. 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.

Olson describes different criteria for choosing a database for an embedded system [102]. He classifies databases into client-server relational databases, client-server object-oriented databases, and, finally, embedded library databases. The embedded library databases are explicitly designed for embedded systems. The embedded database links directly into the software, and there is no need for a query language such as SQL. Existing client-server databases are not appropriate for real-time systems, because transactions cannot be prioritized. Nyström et al. identify that there currently are no viable commercial alternatives of embedded databases suited for embedded real-time systems [125]. The referenced technical report was published 2002 and to check current development of embedded databases a search in Google for the keyword 'embedded database' was conducted. The search yields the following top results that are not covered in [125]: eXtremeDB [96] that is a main-memory database that is linked with the application, DeviceSQL [46] that also is a main-memory database, Microsoft's SQL Server 2005 Compact Edition [98] that is an in-process relational database. These database systems do not, with respect to embedded real-time systems, improve upon the listed databases in [125] since they require relatively high memory foot-print (e.g., it is 100 Kb for eXtremeDB [96]) or operating systems not suitable for embedded systems (Microsoft's SQL Server 2005 Compact Edition requires at least Windows 2000).

Olson also points out that most database systems use two-phase locking to ensure concurrent transactions do not interfere with each other [102]. Two-phase locking is an approach to concurrency control that guarantees the consistency of the data [19]. However, for some applications the consistency can be traded off for better performance [90] (see Chapter 4). This trade-off is not possible if only conventional two-phase locking is available.

1.3 Goals

As discussed above, embedded systems' software becomes more and more complex, which increases the development times and costs. Further, data plays an important role in embedded systems especially in control and controlling systems, because monitored data is refined and then used to control the system. Thus, there is an identified need to find *efficient* methods to handle *application-specific requirements* on data consistency. These methods must also consider the non-functional requirement, that is usually found in embedded systems, of *timeliness*.

Databases have been successfully used in large systems to maintain data during several decades now. The hypothesis in this thesis is that databases can be used in embedded systems as means to efficiently handle data consistency and timeliness and at the same time reduce development time and costs. Thus, our goals are

G1: to find means—focusing on data management—to reduce development complexity;

G2: to meet the non-functional requirement of timeliness; and

G3: to utilize available computer resources efficiently.

Our approach is to assume the concept of databases is usable in embedded system. This assumption is based on the success of using databases in a wide range of applications over the years. We intend to investigate what the specific requirements on a database for an embedded real-time system are. We aim at using an EECU software as a case study to derive a data model that the database should support. EECU systems constitute a typical embedded system with a mix of hard and soft real-time tasks that use data values with consistency requirements. Thus, results presented in this thesis can be generalized to other types of computer systems that have deadlines associated with calculations and the calculations need to use consistent values of data items.

1.4 Contributions

The contributions of the research project "Real-Time Databases for Engine Control in Automobiles" and this thesis are:

- A database system platform for embedded real-time systems denoted **D**ata **I**n **E**mbedded **S**ystems ma**I**ntenance **S**ervice (DIESIS). DIESIS features:

  - A new updating scheme, simply denoted AUS, for marking changed data items and scheduling algorithms that schedule data items that need to be updated. The combination of marking data items and scheduling data items has shown to give good performance. Resources are used efficiently since scheduled data items reflect changes in the external state, i.e., the number of scheduled data items is adapted to the speed of changes in the current state of the external environment.
  - A new algorithm, MVTO-S, that ensures data items' values used in calculations are from the same system state. Such an algorithm is said to provide a snapshot of the data items' values at a time t. Moreover, updates of data items are scheduled using AUS and a scheduling algorithm. Using MVTO-S is shown to give good performance because historical values on data items remain in the database, and these data values do not need be updated if used by a calculation, i.e., less number of calculations need to be done using a snapshot algorithm. However, more memory is needed.
  - Overload handling by focusing CPU resources on calculating the most important data items during overloads. Performance results show that overloads are immediately suppressed using such an approach.

- Two new algorithms for analyzing embedded real-time systems with conditioned precedence constrained calculations:
  - An off-line algorithm denoted MTBIOfflineAnalysis that analyzes the mean time between invocations of calculations in a system where the execution of a calculation depends on values of data items.
  - An on-line algorithm denoted MTBIAlgorithm that estimates the CPU utilization of the system by using a model that is fitted to data using multiple regression.

1.5 Papers

The results in this thesis have been published and presented in the following peer-reviewed conferences:

[56] Thomas Gustafsson and Jörgen Hansson. Dynamic on-demand updating of data in real-time database systems. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 846–853. ACM Press, 2004.

[55] Thomas Gustafsson and Jörgen Hansson. Data management in real-time systems: a case of on-demand updates in vehicle control systems. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, pages 182–191. IEEE Computer Society Press, 2004.

[53] Thomas Gustafsson, Hugo Hallqvist, and Jörgen Hansson. A similarity-aware multiversion concurrency control and updating algorithm for up-to-date snapshots of data. In *ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 229–238, Washington, DC, USA, 2005. IEEE Computer Society.

[60] Thomas Gustafsson, Jörgen Hansson, Anders Göras, Jouko Gäddevik, and David Holmberg. 2006-01-0305: Database functionality in engine management system. *SAE 2006 Transactions Journal of Passenger Cars: Electronic and Electrical Systems*, 2006.

[57] Thomas Gustafsson and Jörgen Hansson. Data freshness and overload handling in embedded systems. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA06)*, 2006.

[59] Thomas Gustafsson and Jörgen Hansson. Performance evaluations and estimations of workload of on-demand updates in soft real-time systems. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA07). To appear*, 2007.

The following papers have been co-authored by the author but these are not part of this thesis:

[124] Aleksandra Tešanović, Thomas Gustafsson, and Jörgen Hansson. Separating active and on-demand behavior of embedded systems into aspects. In *Proceedings of the International Workshop on Non-functional Properties of Embedded Systems (NFPES'06)*, 2006.

[61] Thomas Gustafsson, Aleksandra Tešanović, Ying Du, and Jörgen Hansson. Engineering active behavior of embedded software to improve evolution and performance: an aspect-oriented approach. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 673–679. ACM Press, 2007.

The following technical reports have been produced:

[54] Thomas Gustafsson and Jörgen Hansson. Scheduling of updates of base and derived data items in real-time databases. Technical report, Department of computer and information science, Linköping University, Sweden, 2003.

[58] Thomas Gustafsson and Jörgen Hansson. On the estimation of cpu utilization of real-time systems. Technical report, Department of Computer and Information Science, Linköping University, Sweden, 2006.

1.6 Thesis Outline

The outline of this thesis is given below. Figure 1.2 summarizes the developed algorithms and where they are presented and evaluated.

Chapter 2, Preliminaries, introduces real-time systems and scheduling of real-time tasks, database systems and their modules, concurrency control algorithms, serializability and similarity as correctness criterion, overload handling in real-time systems, and analysis of event-based systems.

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

Chapter 4, Data Freshness, introduces data freshness in the value domain. This kind of data freshness is then used in updating algorithms whose purpose is to make sure the value of data items is up-to-date when they are used. The updating algorithms are evaluated and their performance results are reported in this chapter.[2]

Chapter 5, Multiversion Concurrency Control With Similarity, describes an algorithm that presents snapshots of data items to transactions. Implementations of the snapshot algorithm are evaluated and the performance results are reported in this chapter.[3]

Chapter 6, Analysis of CPU Utilization of On-Demand Updating, compares CPU utilization of updating on-demand to well-established algorithms for assigning deadlines and period times to dedicated updating tasks. Further, Chapter 6 develops analytical formulae that estimate mean interarrival times of on-demand updates, which can be used to estimate the workload of updates.

Chapter 7, Overload Control, describes how DIESIS handles overloads and how a developed off-line algorithm, MTBIOfflineAnalysis, can analyze the total CPU utilization of the system and investigate whether the system may become overloaded. Performance results of overload handling are also reported in the chapter.

Chapter 8, On-line Estimation of CPU Utilization, describes an on-line algorithm, MTBIAlgorithm, for analysis of CPU utilization of real-time systems.

Chapter 9, Related Work, gives related work in the areas of data freshness and updating algorithms, concurrency control, and admission control.

Chapter 10, Conclusions and Future Work, concludes this thesis and gives directions for future work.

[2]In order to ease the reading of the main results some detailed explanations of algorithms and some experiments are presented in Appendix B.

[3]Some experiments are moved to Appendix C in order to ease the reading.

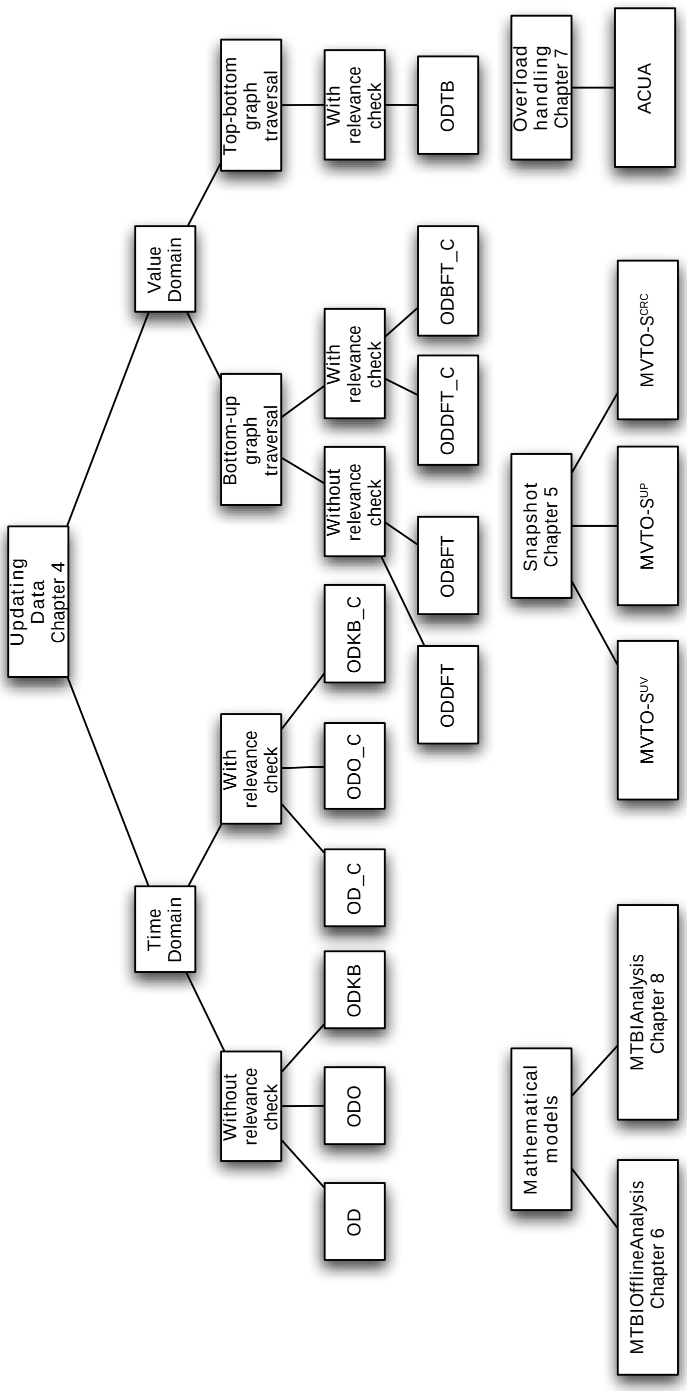


Figure 1.2: Algorithms presented in the thesis.

CHAPTER 2

Preliminaries

The purpose of this chapter is to prepare for the material in coming chapters. Real-time scheduling and admission control are introduced in Section 2.1. Databases and consistency are introduced in Section 2.2. Algorithms to update data items are discussed in Section 2.3. Linear regression is introduced in Section 2.4. Section 2.5 describes the engine management system we have investigated. Section 2.6 describes concurrency control algorithms, and, finally, Section 2.7 describes checksums and cyclic redundancy checks.

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.[1] 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 [79]. A periodic task is periodically made active, e.g., to monitor a sensor at regular intervals. Every activation of the task is called a task instance or a job. Sporadic tasks have a minimum interarrival time between their activations. An example of a sporadic task in the EECU software is the ignition 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 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

[1]The term task and real-time task are used interchangeably in this thesis.

are invoked occasionally and for sporadic tasks we know the smallest possible amount of time between two invocations. The real-time tasks constitute the workload of the system.

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 [24]. 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 *relative deadline* that is relative to the start time of the task. Note that a task has an *arrival time* (or *release 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.

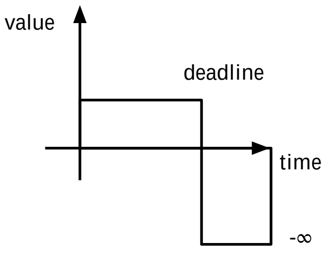
- *Firm real-time tasks*. The deadline is soft, i.e., if the deadline is missed it does not result in any damages to the environment, but the value the task produces has no meaning after the deadline of the task. Thus, tasks that do not complete in time should be aborted as late results are of no use.

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.

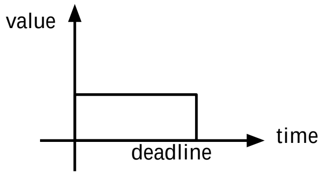
A task can be in one of the following states [84]: ready, running, and waiting. The operating system moves the tasks between the states. When several tasks are ready simultaneously, the operating system picks one of them, i.e., schedules the tasks. The next section covers scheduling of tasks in real-time systems. A task is in the waiting state when it has requested a resource that cannot immediately be serviced.

2.1.1 Scheduling

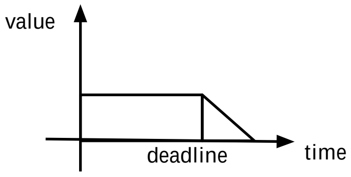
A real-time system consists of a set of tasks, possibly with precedence constraints that specify if a task needs to precede any other tasks. A subset of the tasks may be ready for execution at the same time, i.e., a choice has to be made which task should be granted access to the CPU. A *scheduling algorithm* determines the order the tasks are executed on the CPU. The process of allocating a selected task



(a) Hard real-time task.



(b) Firm real-time task.



(c) Soft real-time task.

Figure 2.1: Hard, soft and firm real-time tasks.

to the CPU 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 and constraints are met [24].

The computational complexity of algorithms constructing a schedule taking job characteristics, e.g., deadlines and the critically of them, into consideration depends on a number of things. First, the number of resources, i.e., the number of CPUs plays an important role. Further, type of conditions influences the complexity as well. Below we give a condensed overview of the computational complexity of algorithms with at least the condition that tasks have deadlines [24, 48, 114, 119]. A schedule can be preemptive, i.e., a task can interrupt an executing task, or non-preemptive, i.e., a started task runs to completion or until it becomes blocked on a resource before a new task can start to execute.

- *Non-preemptive scheduling* on uniprocessor. We will not use this further in this thesis. The problem Sequencing with Release Times and Deadlines in [48] shows that the general problem is NP-complete but can be solvable in polynomial time given specific constraints, e.g., all release times are zero.

- *Preemptive scheduling* on uniprocessor. This problem can be solved in polynomial time, e.g., rate-monotonic (RM) and earliest deadline first (EDF) (see section below for a description) run in polynomial time. A polynomial time algorithm even exists for precedence constrained tasks [28].
- *Multiprocessor scheduling.* We do not use multiprocessor systems in this thesis. An overview of results of computational complexity is given in [119].

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 [79].

2.1.2 Scheduling and Feasibility Tests

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) [91] and earliest deadline first (EDF) [69], 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 [79]:

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.

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

$$U \leq n(2^{1/n} - 1), \tag{2.1}$$

where U is the total utilization of a set of tasks and n is the number of tasks [24, 79]. The total utilization U, i.e., the workload of the system, is calculated as the sum of fractions of task computation times and task period times, i.e., $U = \sum_{\forall \tau \in T} \frac{wcet(\tau)}{period(\tau)}$, where T is the set of tasks, $wcet(\tau)$ the worst-case execution time of task τ, and $period(\tau)$ the period time of task τ. 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 [24,79]. 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

$$U \leq 1, \tag{2.2}$$

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 [24,119]. 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 [66].

Feasibility Tests and Admission Control

Baruah says that exact analysis of the schedulability of a task set is coNP-complete in the strong sense, thus, no polynomial time algorithm exists unless $P = NP$.

A test that checks whether the current state of the CPU (assuming uniprocessor) and the schedule of tasks lead to a feasible execution of the tasks is called a *feasibility test*. The test can be used in a system as depicted in Figure 2.2. Remember that exact algorithms do not exist for certain cases, e.g., when the deadline is less than period time [24]. Thus, feasibility tests probably take too long time to execute in on-line scenarios because they might not run in polynomial time. Polynomial algorithms do exist but they do not give exact answers, i.e., they might report a feasible schedule as infeasible, e.g., the RM CPU utilization test. However, all schedules they report as feasible are indeed feasible schedules. We say these algorithms are not as tight as the exact tests.

The tests in Equation (2.1) and Equation (2.2) can be implemented to run in polynomial time [24], e.g., an EDF feasibility test takes time $O(n^2)$, n is the number of tasks. Tighter algorithms than Equation (2.1) are presented in [11,43,87,105]

Two well-established scheduling algorithms with inherent support for handling overloads are (m, k)-firm and Skip-over scheduling. The (m, k)-firm scheduling algorithm says that m invocations out of k consecutive invocations must meet their deadlines [63]. A distance calculated is based on the history of the k latest invocations. The distance is transformed into a priority and the task with the highest priority gets to execute. The priority is calculated in the

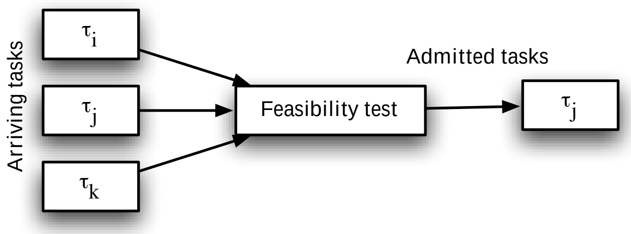


Figure 2.2: Admission control of tasks by utilizing a feasibility test.

following way: $p = k - l(m, s) + 1$, where s contains the history of the latest k invocations and $l(m, s)$ returns how many invocations since the m^{th} invocation meeting its deadline. The lower the p the higher is the priority.

In Skip-over scheduling, task invocations are divided into blue and red (note the resemblance to red, blue and green kernel in Rubus, Section 4.1.1) where red invocations must finish before their deadlines and blue invocations may be skipped, and, thus, miss their deadlines [77]. Feasibility tests are provided in [77] and also some scheduling algorithms, e.g., *Red Tasks Only* which means that only the task invocations being red are executed.

2.1.3 Precedence Constraints

Precedence constraints can be taken care of by manipulating start and deadlines of tasks according to the precedence graph—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—and ready tasks. One example is EDF* where start times and deadlines are adjusted and the adjusted tasks are sent to an EDF scheduler. 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 [24].

Another method to take care of precedence constraints is the PREC1 algorithm described in [79]. The precedence graph is traversed bottom-up from the task that is started, τ, and tasks are put in a schedule as close to the deadline of τ as possible. When the precedence graph has been traversed, tasks are executed from the beginning of the constructed schedule.

2.1.4 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 also 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 is allocated to aperiodic soft real-time 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 [24].

An interesting idea presented by Chetto and Chetto is the earliest deadline last server [27]. 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 [126].

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[2] [15], whereas a database is defined as a set of named data items where each data item has a value in [19]. Furthermore, a database management system (DBMS) is a software system able to manage collections of data, which have the following properties [15].

- *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.
- *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.

[2]In [15] 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.

- *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 [19]. A database system and its modules are depicted in Figure 2.3. 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.

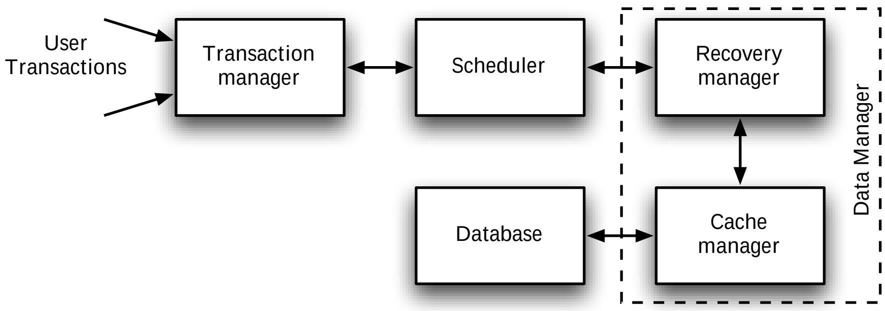


Figure 2.3: A database system.

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.

Different aspects of databases for real-time systems, so called real-time databases, have been extensively investigated in research work. In the case of a real-time database, the scheduler must be aware of the deadlines associated with the transactions in the system. Commercial databases, e.g., Berkeley DB [71], do not have support for transactions which makes them unsuitable for real-time systems [125].

2.2.1 Transactions

A transaction is a function that carries out database operations in isolation [15, 19]. 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 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 [15]. *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 [15, 19]. 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 [81].

- *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 [108] and Kao et al. discuss the consistency for discrete systems [75]. 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 [108] and [75], 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 [7, 9, 39, 55, 56, 75, 76, 88, 101, 108, 127, 130].

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 [108].

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

$$current\_time - timestamp(x) \leq avi(x). \tag{2.3}$$

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 [75].

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

$$\bigcap\{VI(x_i)|\forall x_i \in RS\} \neq \emptyset. \tag{2.4}$$

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 2.4 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 [39]. 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 2.4.

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 [81]. Similarity is a relation defined as: $similarity : D \times D \rightarrow \{true, false\}$, where D is the domain of data item d. The data items can have several versions. The versions are indicated by superscripting d_i, e.g., d_i^j means version j of d_i. If there is no superscript, the latest version is referred to. The value of a version is denoted $v_{d_i^j}$.

The value of a data item is always similar to itself, i.e., the similarity relation is reflexive. Furthermore, if a value of data item d_i, v'_{d_i}, is similar to another value of data item d_i, v''_{d_i}, then v''_{d_i} is assumed to be similar to v'_{d_i}. 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, the relation $similarity$ is symmetric. The relation in Figure 2.4 is reflexive, symmetric and transitive, but a similarity relation does not need to be transitive. The similarity relation $|v'_{d_i} - v''_{d_i}| \leq bound$ is reflexive since $v'_{d_i} = v''_{d_i} \iff |v'_{d_i} - v'_{d_i}| \leq bound$, and symmetric since $|v'_{d_i} - v''_{d_i}| \leq bound \iff |v''_{d_i} - v'_{d_i}| \leq bound$, 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 timestamps of data items being close enough in time, i.e., the values of the data items originate from the same system state [108]. 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 [108] the values need to be created at roughly the same time. Algorithms presented in this thesis use data freshness in the value domain by using similarity relations which have 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(t1, t2):
if t1 < 50 and t2 < 50
  return true
else if t1 >= 50 and t1 < 65 and t2 >= 50 and t2 < 65
  return true
else if t1 >= 65 and t1 < 95 and t2 >= 65 and t2 < 95
  return true
else if t1 >= 95 and t1 < 100 and t2 >= 95 and t2 < 100
  return true
else if t1 = 100 and t2 = 100
  return 100
else
  return false
```

Figure 2.4: An example of a similarity relation for temperature measurements.

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 [7,9,39, 51,54–56]. 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 and it executes before the triggering transaction[3] 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

[3]A triggering transaction is the transaction that caused the action of starting a new transaction.

absolute validity intervals or in the value domain by using a similarity relation. Formally, we define the triggering criterion as follows.

Definition 2.3.1 (On-Demand Triggering)**.** *Let O be operations of a transaction τ, A an action, and p a predicate over O. On-demand triggering is defined as checking p whenever τ issues an operation in O and taking A if and only if p is evaluated to true.*

An *active database* reacts to events, e.g., when a value in the database changes. The events can be described as ECA rules, where ECA stands for Event-Condition-Action [40]. An ECA rule should be interpreted as: when a specific event occurs and some conditions are fulfilled then execute the action. The action can, e.g., be a triggering of a transaction. Thus, definition 2.3.1 is in this respect an active behavior.

2.4 Linear Regression

Linear regression regards the problem of building a model of a system that has been studied. The model takes the following form [44, 113]:

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}, \tag{2.5}$$

where $\boldsymbol{Y}$ is an $(n \times 1)$ vector of observations, $\boldsymbol{X}$ is an $(n \times p)$ matrix of known form, $\boldsymbol{\beta}$ is a $(p \times 1)$ vector of parameters, and $\boldsymbol{\epsilon}$ is an $(n \times 1)$ vector of errors, and where the expectation $E(\boldsymbol{\epsilon}) = \mathbf{0}$, the variance $var(\boldsymbol{\epsilon}) = \boldsymbol{I}\sigma^2$ meaning that the elements of $\boldsymbol{\epsilon}$ are uncorrelated.

In statistics, an observation of a random variable X is the value x. The values of X occur with certain probabilities according to a distribution $F(x)$. The random sample $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ represents observations of the random variables $\boldsymbol{X} = (X_1, X_2, \ldots, X_n)$. The distribution depends on an unknown parameter θ with the parameter space A. The point estimate of θ is denoted $\hat{\theta}$ or $\hat{\theta}(\boldsymbol{x})$ to indicate that the estimate is based on the observations in $\boldsymbol{x}$. The estimate $\hat{\theta}(\boldsymbol{x})$ is an observation of the random variable $\hat{\theta}(\boldsymbol{X})$ and is denoted a statistic. This random variable has a distribution. A point estimate $\hat{\theta}(\boldsymbol{x})$ is unbiased if the expectation of the random variable is θ, i.e., $E(\hat{\theta}(\boldsymbol{X})) = \theta$. Further, the point estimate is consistent if $\forall\theta \in A$ and $\forall\epsilon > 0$ then $P(|\hat{\theta}(\boldsymbol{X}) - \theta| > \epsilon) \to 0$ when $n \to \infty$, i.e., as the sample size increases the better becomes the point estimate. Further information of an estimate $\hat{\theta}$ is given by a confidence interval. A $100(1-\alpha)$ % confidence interval says that $100(1-\alpha)$ % of intervals based on $\hat{\theta}(\boldsymbol{x})$ cover θ. If the distribution $F(x)$ is assumed to be a normal distribution $N(m, \sigma)$, where σ is unknown, then a confidence interval is derived using the t-distribution in the following way [21]: $(\bar{\boldsymbol{x}} - t_{\alpha/2}(n-1)d, \bar{\boldsymbol{x}} + t_{\alpha/2}(n-1)d)$, where $\bar{\boldsymbol{x}}$ is the mean of the n values in $\boldsymbol{x}$ and d is the standard error $\sqrt{\frac{1}{n-1}\sum_1^n (x_j - \bar{\boldsymbol{x}})^2}/\sqrt{n}$.

A common method to derive estimates of the values of $\boldsymbol{\beta}$ is to use the least square method [44, 113]. The estimates of $\boldsymbol{\beta}$ are denoted $\boldsymbol{b}$. By squaring the errors and differentiating them the so called normal equations are [44, 113]

$$(\boldsymbol{X}'\boldsymbol{X})\boldsymbol{b} = \boldsymbol{X}'\boldsymbol{Y}. \tag{2.6}$$

Thus, the least square estimates $\boldsymbol{b}$ of $\boldsymbol{\beta}$ are

$$\boldsymbol{b} = (\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}'\boldsymbol{Y}. \tag{2.7}$$

The solution $\boldsymbol{b}$ has the following properties [44, 113]:

1. It minimizes the squared sum of errors irrespective of any distribution properties of the errors.
2. It provides unbiased estimates of $\boldsymbol{\beta}$ which have the minimum variance irrespective of distribution properties of the errors.

If the following holds, which are denoted as the Gauss-Markov conditions, the estimates $\boldsymbol{b}$ of $\boldsymbol{\beta}$ have desirable statistical properties:

$$E(\boldsymbol{\epsilon}) = \boldsymbol{0} \tag{2.8}$$

$$var(\boldsymbol{\epsilon}) = \boldsymbol{I}\sigma^2 \tag{2.9}$$

$$E(\epsilon_i\epsilon_j) = 0 \text{ when } i \neq j \tag{2.10}$$

The conditions (2.8)–(2.10) give that [44, 113]:

1. The fitted values are $\hat{\boldsymbol{Y}} = \boldsymbol{X}\boldsymbol{b}$, i.e., the model predicts values based on the values set on the parameters of the model, $\boldsymbol{b}$, and on the readings used as inputs, $\boldsymbol{X}$.
2. The vector of residuals is given by $\boldsymbol{e} = \boldsymbol{Y} - \hat{\boldsymbol{Y}}$.
3. It is possible to calculate confidence intervals of values in $\boldsymbol{b}$ based on the t-distribution.
4. The F-distribution can be used to perform hypothesis testing, e.g., check whether the hypothesis that $\boldsymbol{b} = \boldsymbol{0}$ can be rejected.

There are different metrics that measure how well $\hat{\boldsymbol{Y}}$ estimates $\boldsymbol{Y}$. One such metric is the R^2 value which is calculated as

$$1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{\boldsymbol{y}})^2}. \tag{2.11}$$

A test of normality can be conducted using the Kolmogorov-Smirnov test [5] by comparing the distribution of the residuals to a normal distribution using s^2 as σ^2. The Kolmogorov-Smirnov test examines the greatest absolute distance between cumulative distribution functions. A value

$D = \max_{i \leq i \leq n} \left(F(Y_i) - \frac{i-1}{n}, \frac{i}{n} - F(Y_i) \right)$ is compared to a critical value and if D is greater then the hypothesis that the data has a normal distribution must be rejected. In Matlab Statistical toolbox [4], there is a command called `kstest` that can be used.

2.5 Electronic Engine Control Unit

Now we give a more detailed description of the engine management system that was introduced in Section 1.2.2.

A vehicle control system consists of several electronic control units (ECUs) connected through a communication link normally based on CAN [125]. 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.[4]

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,[5] 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 2.5. 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, i.e., based on the speed of the engine. 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

[4]This data is taken from an EECU in a SAAB 9-5.

[5]An engine is knocking when a combustion occurs before the piston has reached, close enough, 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 [99].

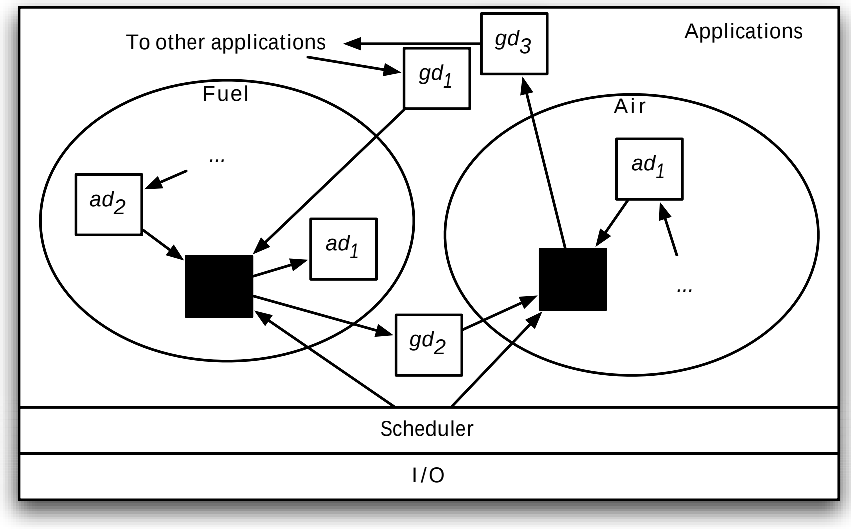


Figure 2.5: The software in the EECU is layered. Black boxes represent tasks, labeled boxes represent data items, and arrows indicate inter-task communication.

area (denoted *ad*, application data in Figure 2.5) or in a global data area (denoted *gd*, global data in Figure 2.5). There are many connections between the applications in the software and this means that applications use data that is also used, read or written, by other applications. Thus, the *coupling* [32] is high. In the 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.

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 of the system is important because, e.g., law regulations force the software to identify malfunctioning hardware within a certain time limit [99]. 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 [62]. 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.

In summary, embedded systems may become overloaded and the software must be designed to cope with it, e.g., at high revolutions per minute of an engine the engine control software cannot perform all calculations. Usually a few data items are compulsory in a calculation to derive a result. For instance the calculation of fuel to inject into a cylinder consists of several variables, e.g., temperature compensation factor, and a sufficiently good result can be achieved by only calculating a result based on few of these variables.

2.5.1 Data Model

In the EECU software, calculations derive either actuator values or intermediate values. 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. The data dependency graph in Figure 2.6 is used throughout the thesis as an example.

We note above that a calculation has a set of input values. This set can be divided into a set of *required* data items that are crucial to keep up-to-date. The other set of data items contains *not required* data items. If the optional data items are up-to-date, the result of the calculation is refined. We assume it is sufficient that mandatory data items are based only on up-to-date mandatory data items.

2.6 Concurrency Control

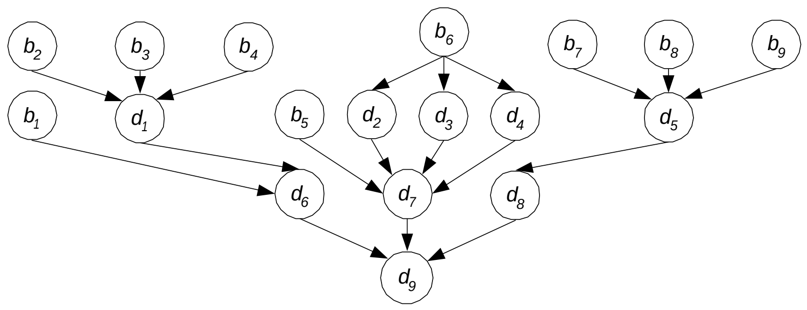
This section describes different concurrency control algorithms that usually are used in databases.

2.6.1 Serializability

As described in the section Transactions (Section 2.2.1), a transaction consists of operations: read, write, abort, and commit.[6] The task of the database system is to execute operations of concurrent transactions such that the following anomalies cannot occur [15]:

- *Lost update*, where a transaction overwrites the result of another transaction, and, hence, the result from the overwritten transaction is lost.

[6] In general, other operations are possible, see [19] for more details.



b_1	Basic fuel factor		d_1	Lambda factor
b_2	Lambda status variable		d_2	Hot engine enr. factor
b_3	Lambda status for lambda ramp		d_3	Enr.* factor one started engine
b_4	Enable lambda calculations		d_4	Enr. factor two started engine
b_5	Fuel adaptation		d_5	Temp. compensation factor
b_6	Number of combustions		d_6	Basic fuel and lambda factor
b_7	Airinlet pressure		d_7	Start enrichment factor
b_8	Engine speed		d_8	Temp. compensation factor
b_9	Engine temperature		d_9	Tot. mult. factor TOTALMULFAC

* Enr. = enrichment

Figure 2.6: Data dependency graph in the EECU.

- *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, consider two transactions, τ_1 and τ_2, and the constraint $s = x + y + z = 1000$ [15]. The operations are executed in the order given in Figure 2.7. The value of s in τ_1 at commit time is 1100 since τ_1 has seen intermediate results from τ_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

τ_1	τ_2
BOT(τ_1)	
Read$_1(x)$	
	BOT(τ_2)
	Read$_2(y)$
Read$_1(y)$	
	$\tau_2 : y = y - 100$
	Read$_2(z)$
	$\tau_2 : z = z + 100$
	Write$_2(y)$
	Write$_2(z)$
	Commit(τ_2)
Read$_1(z)$	
$s = x + y + z$	
Commit(τ_1)	

Figure 2.7: Example of ghost update.

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.

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 [19].

The computational complexity of deciding if operations in a history can be executed in an order such that the four anomalies above do not occur is NP-hard [19, 104].

Recovery

The *recovery module* (see Figure 2.3) 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 τ commits after the commitment of all transactions producing results that are read by τ, i.e., those transactions have written values to data items that τ has read and the write operation occurred before the read operation of τ. 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 [19]:

Example 2.1. *Consider the following history of two transactions: $Write_1(x,1)$, $Write_1(y,3)$, $Write_2(y,1)$, $Commit_1$, $Read_2(x)$, $Abort_2$. 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 τ_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*.

2.6.2 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 [19].

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 [104]. Another good book on the subject is Bernstein et al. [19].

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 systems 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 rules for the two-phase locking algorithm are [19]:

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 [19, 104]. 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 τ_1 holds a write-lock on x and requests a read-lock (for instance) on y. Transaction τ_2 already holds a write-lock on y and requests a read-lock on x. Now, both τ_1 and τ_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 2PL (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.

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 [6]. 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 (τ_i is the validating transaction and τ_j is any other active transaction, and $rs(\tau)$ and $ws(\tau)$ are the read set and write set of τ respectively) [80]: (i) the write phase of τ_i completes before the read phase of τ_j starts, (ii) $ws(\tau_i) \cap rs(\tau_j) = \emptyset$ and the write phase of τ_i completes before the write phase of τ_j, and (iii) $ws(\tau_i) \cap ws(\tau_j) = \emptyset \wedge ws(\tau_i) \cap rs(\tau_j) = \emptyset$ and the read phase of τ_i completes before the read phase of τ_j. Kung and Robinson present a validation phase ensuring conditions (i) and (ii) and a validation phase ensuring all three conditions [80]. Ensuring the two first conditions, at the arrival of the commit operation of transaction τ, the TO scheduler checks whether the read set of τ 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, τ is aborted. This algorithm can be shown to be serializable [19,104]. For details on a validation phase fulfilling conditions (i)–(iii) see [70,80].

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 [6]. 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) [112].

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 [19]: 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 τ_i and, hence, aborts τ_i. When τ_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. Strictness and recoverability of timestamp ordering concurrency control are discussed in [19]. 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 [19, 104]. In MVTO, operations are processed in a first-come-first-served manner and read operations, $\text{Read}_i(x)$, are transformed into $\text{Read}_i(x_k)$—where x_k is a version produced by transaction τ_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 $\text{Write}_i(x_i)$ of transaction τ_i has to be rejected if a transaction τ_j has read version x_h and $ts(x_h) < ts(\tau_i) < ts(\tau_j)$, 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 $\text{Write}_i(x_i)$, i.e., a version is created with the timestamp of the transaction.

Snapshots

A *snapshot* of a set of data items at a time t contains the values of the data items at time t [122].

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 [52], 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 [81, 82].

Concurrency 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 τ_3 reads data items a, b, and c, and writes d. Further, τ_1 reads e and writes c, and τ_2 reads f and writes g. Now, τ_3 arrives to the database system and it starts to execute. At this time data items a, b, and c are relatively consistent. Transaction τ_3 reads a, but then τ_2 arrives. After a while, τ_1 also arrives to the system. Transaction τ_1 updates data item c. Transactions τ_1 and τ_2 finish and τ_3 continues with reading b and c. Due to transaction τ_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 τ_3 having read relatively consistent values. Using multiversion concurrency control, the old value of c would have been read by τ_3. Another approach to solve this is to use snapshot data structures [123].*

- 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.

2.7 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 errors in the data, the checksum should consider the order of the data elements (e.g., the bytes) rather than only adding them together, adding 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 [138] and Adler32 [42]. 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 [138]. 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 [133]. 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 2.8 shows a pseudo-code of a CRC implementation [133]. 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 [3].

```
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 2.8: The pseudo-code of an algorithm producing a CRC.

CHAPTER 3
Problem Formulation

This chapter presents, in Section 3.1, a description of software development and data management problems of embedded systems' software. Notations used throughout the thesis are given in Section 3.2, and Section 3.3 gives the formulation of the problems this thesis addresses. Section 3.4 wraps up the chapter.

3.1 Software Development and Data Management

When developing software for embedded systems there are development issues in several areas that need to be considered. Below we review three areas that are pivotal to data-intensive embedded systems that control a system.

- **Data management**. As we saw in Chapter 1 software maintenance is an expensive and complicated task [2, 17, 22, 23, 31, 45, 67, 89]. The reason is that the complexity of software for embedded systems has increased over the years as a response to more available resources such as CPU and memory, and increased functional requirements on the systems. Below we list two areas of data management that we have found challenging [56].
  - **Organizing data**. We have found that (see Section 2.5) a commonly adopted and ad hoc solution is to store data in modules, which is supported by programming languages such as C and C++. However, the coupling between the modules is high which increases the software complexity [32, 37]. For instance, partitioning data into data structures accessible from different modules makes it hard to keep track of which data items exist, and what their time and freshness constraints are, and also their location in more complex architectures.

  - **Data freshness**. Data items need to be fresh, i.e., they need to reflect the current state of the external environment [108]. The problem of keeping data fresh relates to the problem of scheduling of tasks that updates data items' values, which is NP-hard in the strong sense (see Section 3.3.1). Yet the scheduling and execution of tasks should be efficient, i.e., tasks keeping data fresh should not be executed too often.
  - **Act on events**. Embedded systems need to react to events in the external environment, and conventional databases outside the area of embedded systems have had such functionality for a long time, e.g., in the form of ECA rules.

- **Concurrency control**. Even though tasks, performing calculations, in embedded systems might use task-specific data, our experience is that tasks share data to a high degree. When a system has concurrent tasks sharing data, there might be concurrent reading and writing to the same data items (see Section 2.6 for further discussions of concurrency control). Thus, there must be a way to handle such situations. Also, sometimes it is important that tasks read values that originate from exactly the same system state. Examples include diagnosis of a system, e.g., model-based diagnosis [100] where a mathematical model is used to calculate an expected value of a sensor and then it is compared with a reading of the sensor. Thus, the software needs functionality to handle concurrent accesses to data and to be able to ensure that data values origin from the same system state.

- **Overload handling**. Many embedded systems consist of a mix of hard and soft real-time tasks where the interarrival times of the tasks, particularly soft real-time tasks, are not fixed. An example is the tasks being triggered by crank angles in the engine control software (see Section 2.5). This leads to a system whose workload changes over time and the workload of the tasks can occasionally be so high that the system starts missing deadlines. We say that the system has a transient overload, in which the system needs to gracefully degrade the performance of the system.

3.2 Notations and Assumptions

A summary of notations used in this thesis is given in Appendix A. This section introduces a basic data and transaction model that is extended in chapters 4 and 6. Based on the description of an embedded system's software in Section 2.5, we use the following notations in the rest of the thesis. Relationships of data items are described in a *directed acyclic graph* (DAG) denoted $G = (N, E)$, where N is the set of nodes where each node corresponds to a data item and to a calculation that updates the value of the data item. The set E contains directed edges, $(i, j) \in E$ where i is the *tail* and j the *head*, describing the relationships of the data items. The partially ordered set (poset) described by G is denoted $<_G$. The *in-degree* of a node is the sum of edges having the node as their head. The *out-degree* of a node is the sum of edges having the node as their tail. Data items belong either to the set of base items B, which are those nodes in G having zero in-degree (these nodes may be referred to as source nodes), or to the set of derived items D, which are the nodes in G having an in-degree larger than zero. Nodes with zero out-degree are referred to as leaf nodes. A path is a sequence of nodes and directed edges $n_1e_1n_2e_2 \dots e_mn_m$, where every $e_i \in E$ and every $n_i \in N$, $1 \leq i \leq m$. The *ancestors* of $n \in N$ are all nodes belonging to paths with n as their final node. The *descendants* of $n \in N$ are all nodes on paths where n is the starting node in the sequence of nodes and edges. The *immediate parents* of a node n constitute the read set of the data items that a calculation representing the node reads. The read set of a data item $d \in D$ is denoted $R(d)$. The *immediate children* of $n \in N$ are all nodes c such that $(n, c) \in E$. Nodes with positive out-degree and in-degree are called *intermediate nodes*. Next we define the level of a data item.

Definition 3.2.1 (Level of a 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*

$$level(d) = \max_{\forall x \in R(d)} (level(x)) + 1, \tag{3.1}$$

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

A data item has one or several versions and the set of versions of data item d_i is denoted $V(d_i)$. Each version d_i^j of data item d_i has a write timestamp $wt(d_i^j)$. A version is said to be valid during a time interval starting from its timestamp until the timestamp of the following version, i.e., $[wt(d_i^j), wt(d_i^{j+1})]$. If d_i^j is the newest version it is assumed to be valid until a newer version is installed. Hence, the time interval is $[wt(d_i^j), \infty]$. A proper version with respect to a timestamp t is the latest version with a write timestamp less than or equal to t, i.e., a proper version of d_i at t has the following timestamp: $\max\{wt(d_i^j) | \forall d_i^j \in V(d_i), wt(d_i^j) \leq t\}$. The proper version with respect to the current time is denoted the current version.

The worst-case execution time, excluding blocking times and updates, of a transaction/calculation τ is denoted $wcet(\tau)$ and is assumed to be known. A transaction has an arrival time $at(\tau)$, a release time $rt(\tau)$, a logical timestamp $ts(\tau)$,[1] a relative deadline $dt(\tau)$, and a priority $prio(\tau)$.

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

- **SA1**. The embedded system has a single CPU.

[1]It is important that the logical timestamp assigned to transactions is monotonically increasing. However, it is easy to achieve this in a central database system by atomically assigning a timestamp to the transaction in its BOT operation.

- **SA2**. The embedded system is a controlling system that controls an environment.
- **SA3**. A task calls a transaction that performs the calculation of a data item represented by one of the nodes in G and the transaction executes with the same priority as the task and once a transaction has started its priority cannot be changed by itself or other tasks. Tasks are scheduled according to RM or EDF.
- **SA4**. Hardware settings put the requirement that all data needs to be stored in RAM and/or in flash memory. Moreover, at a detected but irrecoverable system failure the system is restarted which means that all transactions also start over.
- **SA5**. The system consists of tasks with interarrival times that may change dynamically.

Assumption SA1 states that the embedded system has one CPU, because nowadays many of embedded systems use one core and the engine control system we studied has one core.

Assumption SA2 states that the embedded system needs up-to-date data in order to correctly control the environment.

Assumption SA3 states that the priority of a user transaction cannot be lowered during its execution, letting another user transaction to interrupt it and start to execute. This is the normal case for real-time operating systems where the ready task with the highest priority is always executing. In desktop operating systems, e.g., Windows 2000, tasks get higher priority as they wait in the waiting queue. Hence, assumption SA3 removes the possibility to use a general purpose operating system. Priority inversion can occur if locks are used [93].

Assumption SA4 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.

Assumption SA5 states that the workload of the system can change dynamically, which also indicates that the system also might enter into transient overloads.

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 thesis is to investigate how a real-time database can give advantages in maintainability and resource utilization in real-time embedded applications.

The initial requirements were that an embedded system—and in particular an EECU—needs to have [50]:

- **R1**. A way to organize data to ease maintenance of the software, 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. The data management software should activate the correct task when some data fulfill specified conditions.
- **R3**. Functionality to:
  - **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, because it is the task of a database to store data in a structured way and perform user initiated actions on the data and to maintain the consistency of data. Since data is stored in a structured way, duplicate storage can be detected. In addition to these requirements, we have also identified the need to have support for the following requirements:

- **R4** Gracefully degrade performance in the case of a transient overload.
- **R5** Determine whether the system is in a transient overload.

We believe requirements R1–R5 are reasonable requirements on data management functionality in data-intensive applications, because they are derived from problems identified by industry and by our own experience of working with the thesis, and by taking part in conferences on real-time systems. We also believe that many real-life systems can be mapped to the data and transaction model (see Section 3.2) we use. Therefore, we are confident that the algorithms we describe in chapters 4, 5, 6, 7, and 8 are also applicable to systems other than EECUs, that are our main target platform to test algorithms on.

The requirements, R1–R5, on data management for embedded systems are translated into the three goals that were presented in Section 1.3. We repeat them here. The goals are

G1: to find means—focusing on data management—to reduce development complexity;

G2: to meet the non-functional requirement of timeliness; and

G3: to utilize available computer resources efficiently.

3.3.1 Computational Complexity of Maintaining Data Freshness

The requirements listed above are concerned with using resources in an efficient way. This section ends this chapter with a discussion of how difficult it is with respect to computation time to decide which data items that should, at a given time, be updated in order to keep data items up-to-date. We say that updates of data items are *scheduled* and put into a schedule.

Assume we have the following general problem of scheduling updates of data items. The relationships among data items are described in a data dependency graph G and one of the data items, d_i, in G is requested for being used in a user transaction. We say a data item is *potentially affected* if it lies on a path where a data item whose value has changed is the starting node (see Figure 3.1). Potentially affected data items need to be considered for being recalculated before d_i is recalculated. All recalculations, including the one of d_i, must be executed within a specified deadline. The immediate parents of d_i should be fully updated, i.e., all ancestors of $d_j \in R(d_i)$ that are potentially affected have to be scheduled for recalculation (d_j and d_k in Figure 3.1). We have two sets: the set of potentially affected ancestors, $PAA(d_i)$, which is $\{d_j, d_k\}$ in Figure 3.1, and the set of potentially affected read set, $PAR(d_i)$, which is $\{d_j\}$ in Figure 3.1. The affected data items that are ancestors of d_i are denoted $A(d_i)$, which is $\{d_j, d_k\}$ in Figure 3.1. The problem is mathematically formulated below as an integer programming optimization problem. We refer to the problem as General Scheduling of Updates Concerning Data Freshness (GSUCDF).

$$\begin{aligned}
\text{maximize} \quad & \sum_{\forall d_j \in R(d_i)} r_j \\
\text{subject to} \quad & \sum_{\forall d_j \in R(d_i)} r_j wcet(d_j) + \sum_{\forall d_j \in A(d_i) \setminus R(d_i)} x_j wcet(d_j) \leq dt(\tau), \\
& r_j \leq_G x_k, \quad d_k \in PAA(d_j), d_j \in PAR(d_i), \\
& r_j = x_k, \quad d_k \in A(d_j), d_j \in PAR(d_i), j = k, \\
& r_j \in \{0, 1\}, \\
& x_j \in \{0, 1\}.
\end{aligned} \tag{3.2}$$

GSUCDF maximizes the number of immediate parents that are up-to-date, i.e., the ancestors of the immediate parents are also up-to-date, such that all recalculations can be executed within the given time frame. The variable r_j is one if there is a scheduled update of d_j and d_j is an immediate parent of d_i. The variable x_k is one if d_k is an ancestor of an immediate parent that is scheduled for being updated.

The set-union knapsack problem (SUKP) considers N items and a set $P := \{1, \dots, m\}$ of elements where each item j corresponds to a subset of elements $P_j \subseteq P$ (see Figure 3.2). Every item has a nonnegative profit p_j and every element i of P has a weight w_i. The weight of an item j is the sum of

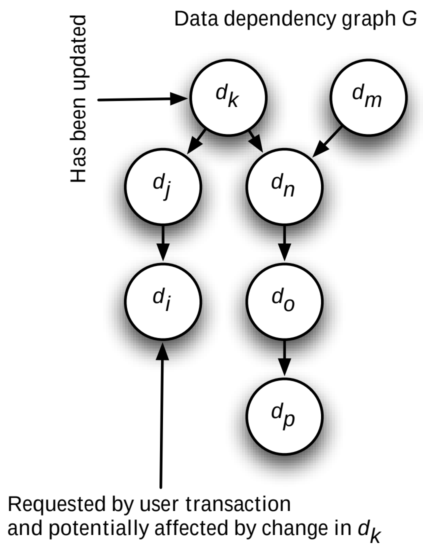

Data dependency graph G
Has been updated
d_k
d_m
d_j
d_n
d_i
d_o
d_p
Requested by user transaction
and potentially affected by change in d_k

Figure 3.1: Data dependency graph.

weights of the elements in P_j. The objective is to find a set $Q \subseteq N$ that fits in a knapsack of capacity c and has maximal possible profit. SUKP can be formulated as a linear programming problem as follows.

$$\begin{aligned} \text{maximize} \quad & \sum_{j \in Q} p_j \\ \text{subject to} \quad & \sum_{i \in P_Q} w_i \leq c. \end{aligned} \tag{3.3}$$

The computational complexity of SUKP is NP-hard in the strong sense even when $p_j = w_i = 1$ and $|P_j| = 2$ [49]. To determine the computational complexity of GSUCDF the following steps should be taken [48].

1. Show GSUCDF is in NP.
2. Choose an NP-complete problem Π.
3. Construct a transformation from Π to GSUCDF. A lemma (page 34 in [48]) says that if there exists a polynomial transformation of a problem into another problem which can be solved in polynomial time, then the first problem can also be solved in polynomial time. However, if there exists a polynomial transformation of a problem, Π_1, which is not in P, into another problem, Π_2, then Π_2 is also not in P.

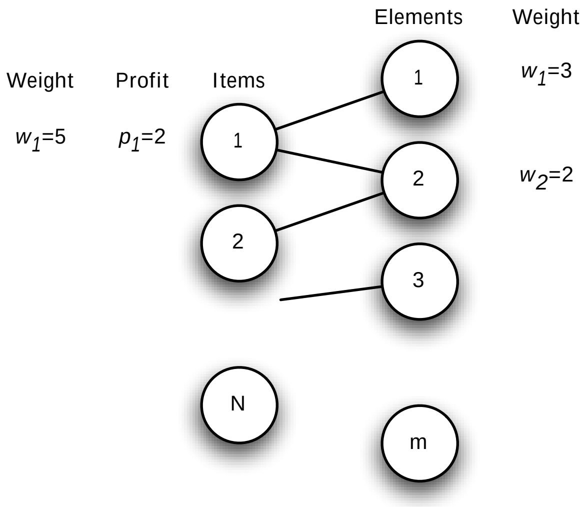


Figure 3.2: Set-union knapsack problem (SUKP).

4. Show that the transformation takes polynomial time.

Step 1 is carried out by showing that a verification of a schedule of calculations to perform to get up-to-date data items given by a guessing of a non-deterministic computer takes polynomial time in the size of the schedule. Lets assume a data item d_i is about to be used by a transaction, and thus a schedule of updates needs to be created. A non-deterministic algorithm would guess such a schedule. It should be checked that the order of updates in the schedule follows the relationships in G and that all ancestors of scheduled immediate parents of d_i are also scheduled. It is easy to see that these checks take polynomial time in the size of the schedule. Hence, GSUCDF is in NP.

In step 2 we choose the problem SUKP, which is similar to the GSUCDF problem.

In step 3 we construct the algorithm presented in Figure 3.3. This algorithm constructs a data dependency graph G of ancestors of a data item x from an instance of SUKP where $p_j = 1, \forall j \in N$.

The algorithm in Figure 3.3 adds a node for each item in N and they correspond to immediate parents of x. A node is created and added to G for each element in each set P_j if the element has not been already created. For each element k in a set P_j, a directed edge (k, j) is added to G.

The last step is to prove that the algorithm described in Figure 3.3 is a transformation. Assume SUKP has found Q, then each item $i \in Q$ corresponds to an immediate parent and all elements in P_i are also scheduled. Since the profit

```
1: Start with empty graph $G$
2: Add $|N|$ to each element of each set $P_j$ in order to make elements distinct
     from items
3: for all $i \in N$ do
4:     add node $i$ to $G$
5:     for all $j \in P_i$ do
6:         if $j \in G$ then
7:             add edge $(j, i)$ to $G$
8:         else
9:             add node $j$ to $G$
10:            add edge $(j, i)$ to $G$
11:        end if
12:    end for
13: end for
14: Let every data item in $G$ corresponding to an item have execution time 0.
15: Let every data item in $G$ corresponding to an element have execution time
     $w_i$
16: Let $c$ be available time until deadline of the transaction using data item $x$
```

Figure 3.3: Transformation from SUKP to GSUCDF.

$p_i = 1$, the value $\sum_{j \in Q} p_j$ counts the number of items in Q. This corresponds to $\sum_{j \in Q} r_j$ in GSUCDF, i.e., the items in Q corresponds to a schedule found by GSUCDF. Now, assume GSUCDF finds a schedule with a certain number of immediate parents $\sum_{j \in Q} r_j$. Then SUKP would put these data items in Q.

3.4 Wrap-Up

This chapter introduces three areas that are important to data-intensive embedded systems, namely, data management, concurrency control, and overload handling. This chapter also summarizes requirements on functionality of data management for software for an EECU. This chapter also shows a reduction from the set-union knapsack problem to choosing data items to update such that the number of data items being chosen is maximized with respect to the available time to execute updates.

CHAPTER 4

Data Freshness

The requirements presented in the previous chapter state requirements on the data handling part of a software. As mentioned in Chapter 3 our data management software is a database with functionality aimed at being useful for embedded real-time systems. In this chapter, we discuss the aspect of database functionality for embedded systems that is concerned with data freshness and updating algorithms. Note that some sections related to updating algorithms are presented in Appendix B in order to ease the reading of this chapter.

The objectives of introducing updating algorithms are (i) to use CPU resources more efficiently (addresses requirement R3d), and (ii) ease the development efforts of developing the software (addresses requirement R1). We show in this chapter that the updating algorithms we propose use the CPU resource more efficient than updating data items using algorithms proposed by other research groups. Furthermore, the development efforts can be eased due to functionality being encapsulated in the database. Programmers do not have to write code to ensure data items have up-to-date values, because it is ensured by the updating algorithms in the database.

The performance of introducing on-demand updating algorithms is evaluated in several experiments. These experiments are:

- **Experiment 1:** The purpose of this experiment is to evaluate performance of on-demand updating algorithms that do not use relevance checks (see Definition 4.3.2 for a definition of relevance check). Their performance is compared to well-established updating techniques. The concurrency control algorithm is fixed and different updating algorithms are evaluated. A discrete event simulator denoted RADEx++ is used in these experiments. The simulator setup is described in Section 4.5.4. Performance evaluations are described in Section 4.5.5.

- **Experiment 2:** The purpose of this experiment is to evaluate performance of on-demand updating algorithms that use relevance checks. The same simulator settings as for Experiment 1 are used. The evaluations are presented in Section B.8.
- **Experiment 3:** The purpose of this experiment is to evaluate DIESIS in a real-life setting. DIESIS is configured for HP2PL and ODTB and it is used in an engine control software, where a subset of the data in the engine control software is stored in DIESIS. This experiment is used as proof of concept and investigates if using similarity and on-demand updating algorithms behave as intended in a real-life setting.

The outline of this chapter is as follows. Section 4.1 gives an overview of DIESIS that is being used in the performance evaluations. DIESIS has support for the algorithms that are described in this chapter. A key part of updating algorithms is the way data freshness is measured and this is formally described in Section 4.2. Section 4.3 describes a mechanism to mark data items such that an updating algorithm can find data items needing to be updated. Sections 4.4 and 4.5 describe algorithms that determine, in a best-effort way, which data items to update to keep them up-to-date. Section 4.5 also includes performance evaluations that compare our algorithms to well-established algorithms. Section 4.6 wraps up this chapter.

4.1 Database System: DIESIS

In this section, we introduce our database system that we denote **D**ata **I**n **E**mbedded **S**ystems ma**I**ntenance **S**ervice (DIESIS), and in Section 4.1.1 we discuss implementation details. DIESIS is used in the performance evaluations in sections 4.5.6 and 5.5 in order to compare the updating and snapshot algorithms we propose to well-established updating and concurrency control algorithms. DIESIS is designed for resource-constrained real-time embedded systems and based on our experiences from studying an engine control software we use the design outlined in Figure 4.1. The following design decisions have been taken.

- DIESIS is a main-memory database meaning that data is stored in main-memory because of assumption SA4 (page 41). Furthermore, this assumption states that data values are reset at a system restart which means that no before images of data items' values are needed. This assumption holds true for the engine control software we have studied.
- DIESIS has no query language, because in embedded systems data records are simple and need not be represented as tables as in relational databases. Thus, one data value usually corresponds to one data item, i.e., instead of using a query and an indexing mechanism to fetch a value of a data item it is more efficient to access it directly. Furthermore, the set of data items

may be fixed in embedded systems. The data set is fixed in the engine control software. When the set is fixed there is no need for database operations to insert and delete data items. Furthermore, no indexing is needed since the values of data items can be stored in an array.

- DIESIS is designed for simplistic real-time operating systems, e.g., Rubus and μC/OS-II, because they, or similar RTOSes, are likely to be used in an embedded system. Limitations of these RTOSes are: started transactions cannot restart in the middle of its execution and started transaction cannot be removed without finishing its execution.

The depicted database system in Figure 2.3 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 updating algorithm rejecting unnecessary calculations is added to the database system. Also, the values that transactions use need to be relatively consistent. Hence, a snapshot algorithm is also added to the data management module. The functionality of the depicted admission control is discussed in Chapter 7. The database system developed in this project is depicted in Figure 4.1.

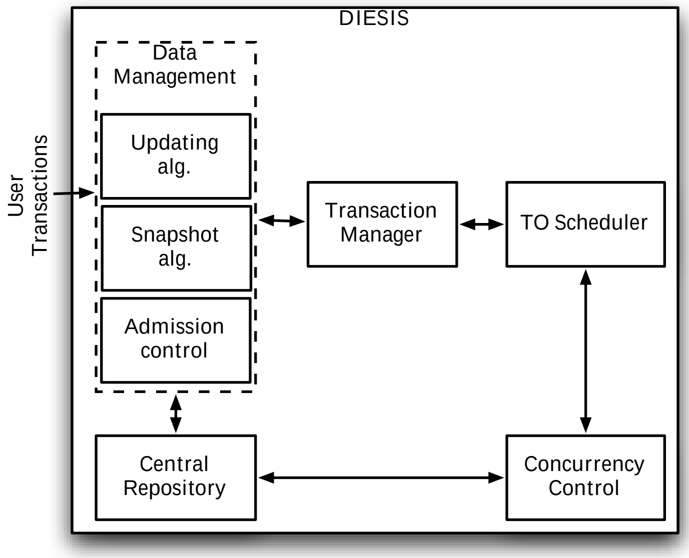


Figure 4.1: Modules in DIESIS.

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.

DIESIS uses the following transactions. A *user transaction* (UT) is an incoming transaction, and it derives data item d_{UT}, a *sensor transaction* (ST) derives data item b_{ST}, and a *triggered update* (TU) derives data item 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 2.3) contains a description of triggering of transactions. A user transaction deriving a data item corresponding to a leaf node is referred to as an actuator transaction.

In DIESIS, a triggered update is updating a data item. The TU executes within the same task starting the UT but before the UT starts. 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.

In DIESIS, transactions are prioritized either according to RM or EDF, and the ready transaction with the highest priority is executing. Transactions may be preempted. 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.

4.1.1 Implementation of Database System

In the implementation of the database system, two real-time operating systems are used: Rubus version 3.02 and μC/OS-II v2.53. The reasons for using two real-time operating systems are:

- Rubus was the only operating system available for the EECU system.
- Rubus exists for Windows, but we experienced problems with real-time tasks' execution and measuring of time under Windows. Since μC/OS-II has the same functionality as Rubus and runs under DOS, where the real-time task execution and time measurements are stable even in a DOS box in Windows, we choose to also use μC/OS-II. It is used in situations where it is more convenient to use a PC than an EECU system to execute simulations.

Rubus, from Arcticus Systems AB [1], is used in the EECU for scheduling and communication between tasks (see Section 4.5.6 for performance evaluations). Rubus consists of basic services and three kernels: *red*, *green*, and *blue*. The basic services supply 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.

The DIESIS configuration used in the EECU uses HP2PL and the EDF scheduling algorithm. We are currently using the periodic tasks of the EECU software, 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 necessary to properly map crank angle interrupts to blue tasks. The reason is that crank angle interrupts have a higher priority than time-based interrupts. 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.

An example of a transaction in DIESIS is given in Figure 4.2. BeginTransaction starts a transaction with a relative deadline of 10000 μs that derives the data item TOTALMULFAC, d_9 in Figure 2.6. 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 [47].

4.2 Data Freshness

We show in this thesis that one key part of using the CPU resource efficient is how data freshness is measured. In this section, we first recollect data freshness measured in the time domain followed by an introduction of our usage of

```
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 4.2: Example of a transaction in the EECU software (C-code).

similarity that we use to contrast data freshness measured in the time domain and data freshness measured in the value domain throughout the remainder of the thesis.

4.2.1 Data Freshness in Time Domain

A value of a data item is assumed to live for an amount of time, i.e., a value is valid as long as it is younger than a certain age. We have definition 2.2.1 that we repeat below.

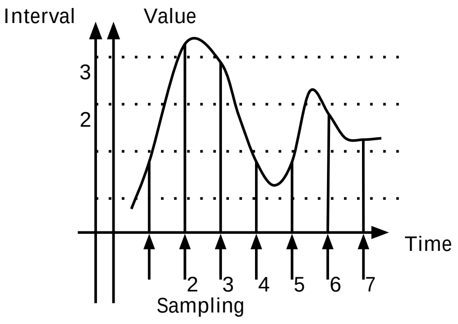
Definition 2.2.1 (Absolute Consistency). *Let x be a data item. Let $timestamp(x)$ be the time when x was created and $avi(x)$, the absolute validity interval (AVI), be the allowed age of x. Data item x is absolutely consistent when:*

$$current\_time - timestamp(x) \leq avi(x).$$

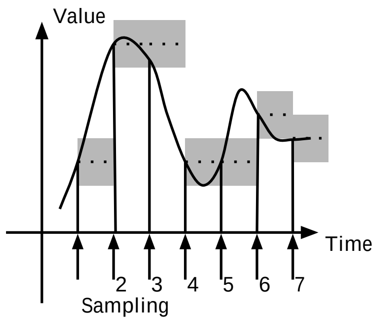
4.2.2 Data Freshness in Value Domain

A fixed time might be a bad approximation of how much data values change between succeeding calculations of them since the allowed age of a data value needs to be set to correspond to the worst-case change of that data item. Depending on the application and system state, it is not certain that the value changes that much all the time, and, hence, a fixed time does not reflect the true freshness of a data item. Similarity defines the relation between values of a data item by the relation $f : D \times D \rightarrow \{true, false\}$ (described in Section Consistency (Section 2.2.2)), where D is the value domain of two values of a data item that are checked for similarity.

In this thesis two different similarity relations are used. One considers 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 is based on one value of a data item as the origin of an allowed distance, i.e., all values that are within a given distance to the origin are similar to the value of the origin. Figure 4.3 shows the distinction between the two similarity functions.



(a) Interval-based similarity



(b) Distance-based similarity

Figure 4.3: Two similarity functions mapping to intervals and to distances.

The similarity relation based on distances is defined as follows:

Definition 4.2.1 (Distance-based similarity). *Each pair (d_i, d_k), where d_i is a derived data item and d_k is an item from $R(d_i)$, has a data validity interval, denoted δ_{d_i,d_k}, that states how much the value of d_k can change before the value of d_i is affected. Let $v^t_{d_k}$ and $v^{t'}_{d_k}$ be values of d_k at times t and t' respectively. A version j of d_i reading $v^t_{d_k}$ is fresh, with respect to the version of d_k valid at t, for all t' fulfilling $|v^t_{d_k} - v^{t'}_{d_k}| \leq \delta_{d_i,d_k}$.*

Hence, the similarity relation f is equal to

$$f : v^t_x \times v^{t'}_x \to |v^t_x - v^{t'}_x| \leq \delta_{d,x}, \tag{4.1}$$

where v^t_x and $v^{t'}_x$ are two values of data item x and d is a child of x. We refer to Equation (4.1) as distance-based similarity.

Using intervals, the freshness of a data item is as follows:

Definition 4.2.2 (Interval-based similarity). *Let $fixedint_{d_k}$ be a function mapping values of a data item d_k to natural integers, i.e., $fixedint_{d_k} : D \to \mathbf{N}$, where D is the domain of values of data item d_k. All values of d_k mapping to the same interval are similar. Let $v^t_{d_k}$ and $v^{t'}_{d_k}$ be values of d_k at times t and t'*

respectively. A version j of d_i reading $v^t_{d_k}$ is fresh, with respect to the version of d_k valid at t, for all t' fulfilling

$$fixedint_{d_k}(v^t_{d_k}) = fixedint_{d_k}(v^{t'}_{d_k}). \tag{4.2}$$

One example of the function $fixedint$ is: $fixedint_{d_k}(v^t_{d_k}) = \left\lfloor \frac{v^t_{d_k}}{64} \right\rfloor$, where the value domain of data item d_k is divided into intervals of size 64. As long as the value of d_k maps to the same number as the value of d_k being used to derive d^j_i, the value changes of d_k do not affect the value of d^j_i. We refer to equation (4.2) as interval-based similarity.

There are slight differences in handling validity intervals as an interval or as a distance. One can be seen in Figures 4.3(a) and 4.3(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 4.3(a), but in Figure 4.3(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 4.3(a) but not so in Figure 4.3(b).

The two different ways of defining similarity are intuitive in different ways, which is elaborated in Example 4.1. Mapping to intervals has better support for being implemented as entries into tables since an entry into a table is a fixed interval. A distance is intuitive in that it is easy to reason about changes in values relative 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. *A distance is used as the similarity relation of 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 water temperature does not exist, and the similarity relation does not reflect this. Therefore a division of the possible temperatures into intervals might be a better similarity relation to use.*

The staleness of a data item is described in the following definition.

Definition 4.2.3 (Staleness of a version of a data item). *Let a version j of data item d_i be derived at time t using values of data items in $R(d_i)$. The value of d^j_i is denoted $v^t_{d_i}$. The value $v^t_{d_i}$ is stale at time t' if there exists at least one element d_k in $R(d_i)$ such that $|v^t_{d_k} - v^{t'}_{d_k}| > \delta_{d_i,d_k}$ or $fixedint_{d_k}(v^t_{d_k}) \neq fixedint_{d_k}(v^{t'}_{d_k})$ depending on which definition of freshness is used. The value of d_i is valid if it is not stale.*

The validity of a value of a data item can easily be derived in the following way.

Definition 4.2.4 (Validity of a version of a data item). *A version j of data item d_i derived at time t is valid at all times t' when $\forall x \in R(d_i), |v_x^t - v_x^{t'}| \leq \delta_{d_i,x}$ or $\forall x \in R(d_i), fixedint_x(v_x^t) = fixedint_x(v_x^{t'})$ depending on which data freshness is used.*

4.2.3 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., when at least one of its ancestors has changed such that the update might result in a different value compared to the value of 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 $Path_{d_9-b_6}$, in Figure 2.6 is: d_9, d_7, d_2, b_6. When a data item is updated it may make its immediate children in G stale (this can be checked using definition 4.2.3). If an update of data item makes 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 $Path_{d_9-b_6}$, 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 change in d_2. Figure 4.4 shows this example.*

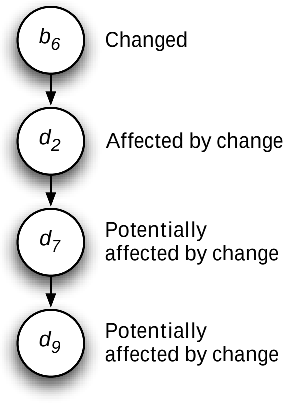

b_6
Changed
d_2
Affected by change
d_7
Potentially affected by change
d_9
Potentially affected by change

Figure 4.4: Depicting example 4.2.

4.3 Marking of Changed Data Items

Example 4.2 shows that a data item might be potentially affected by a change in a predecessor of it. The validity (definition 4.2.4) of a potentially affected data item d can only be determined by recomputing all potentially affected data items on paths leading to d. In order to find potentially affected data items, every derived data item d has a timestamp that is denoted potentially affected timestamp of d, $pa(d)$, and is defined below.[1]

Definition 4.3.1 (Potentially Affected Timestamp)**.** *Let $pa(d)$ be the latest logical timestamp when data item d was directly or indirectly affected by a change in another data item.*

An updating scheme can be used to determine which data items are (potentially) affected by a change of the value of a data item. The AUS updating scheme is presented below. The steps of AUS are:

- **AUS_S1:** Update base items to always keep them up-to-date.
- **AUS_S2:** Mark data items as (potentially) affected by a change in a data item.
- **AUS_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.

In the first step (AUS_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 (definition 4.2.4) is checked for each immediate child of b in data dependency graph G. Thus, in our example, base items b_1–b_9 from Figure 2.6 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 (AUS_S2) is performed when a data item d is found to be stale due to the new value of ancestor x, where x can be either a base item or a derived item. Data item d is marked as potentially affected by the change in x. The pa timestamp is set to $\max(pa(d), ts(\tau))$, where $ts(\tau)$ is the logical timestamp of the transaction updating x. This means that a data item is marked with the timestamp of the latest transaction that makes the data item potentially affected by the written value produced by the transaction.

The third step (AUS_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

[1]Section B.2 discusses why a timestamp is needed to indicate stale data items.

timestamp $pa(d)$ is set to zero if d is not potentially affected by changes in any other data item, i.e.,

$$pa(x) = \begin{cases} 0 & \text{if } ts(\tau) \geq pa(x) \\ pa(x) & \text{otherwise.} \end{cases} \tag{4.3}$$

Every executed TU ends with marking any potentially affected data items. Thus, step S2 is used for every single transaction including updates.

Formally we have the following definition of a relevance check.

Definition 4.3.2 (Relevance Check). *Let τ_{UT} be an update for data item d_{UT}. Assuming transactions are deterministic and time invariant, the relevance of executing τ_{UT} is determined by checking whether d_{UT} is affected by a change in an immediate parent, i.e., checking whether d_{UT} is marked as affected.*

As can be understood from the definition above, a relevance check is intended to be used to skip, if possible, updates of data items. The definition is applicable to triggered updates and should be applied before the update starts to execute. On-demand scheduling of updates of data items, which is discussed in Sections 4.4 and 4.5, can use relevance checks on scheduled updates to determine whether they can be skipped or not.

Another similar scheme, denoted potentially affected updating scheme (PAUS), used by some updating algorithms is described in Section B.1.

4.3.1 Correctness of Determining Potentially Affected Data Items

Steps AUS_S1–AUS_S3, discussed above, give a mechanism to determine if a data item is potentially affected by a change in any of its ancestors. Below we show that a potentially affected timestamp greater than zero on a data item means that the data item is stale.

Proposition 4.3.1. *Let d_i be a data item and $pa(d_i)$ the timestamp of the current version set in steps AUS_S2 and AUS_S3. If data item d_i is stale according to definition 4.2.3 then its timestamp is larger than zero, i.e., $pa(d_i) > 0$.*

Proof. Proof by contradiction. Assume a data item d_i is stale. The $pa(d_i)$ timestamp has been set by AUS_S2 otherwise d_i is not stale. The $pa(d_i)$ timestamp is determined by taking $pa(d_i) = \max(pa(d_i), ts(\tau_1))$; further, assume τ_1 is the latest update affecting d_i, thus, $pa(d_i) = ts(\tau_1)$ since timestamps are monotonically increasing. If $pa(d_i) = 0$, then d_i has been updated by a transaction τ_2, implying $ts(\tau_2) \geq pa(d_i)$ and $ts(\tau_2) > ts(\tau_1)$. Hence, τ_2 arrived after τ_1 since timestamps on transactions increase monotonically, and d_i is up-to-date which is a contradiction. Thus, a stale data item d_i implies $pa(d_i) > 0$. □

4.4 On-Demand Updating Algorithms in Time Domain

This section describes on-demand updating algorithms presented by Ahmed and Vrbsky [9]. These algorithms use different predicates p in the definition of on-demand triggering (definition 2.3.1) to divide updating algorithms into consistency- and throughput-centric algorithms. In an overloaded system, consistency-centric algorithms prioritize data freshness before timeliness, and throughput-centric algorithms prioritize timeliness before data freshness. In an underloaded system, the consistency- and throughput-centric algorithms trigger the same updates.

Every time a data item is requested by a read operation in a transaction, condition (2.3) in Definition 2.2.1 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 (2.3) is denoted OD. The triggering criterion can be changed to increase the throughput of UTs in the case of an overload. Ahmed and Vrbsky present three options of triggering criteria [9]. 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 options (i)–(iii) above are [9]:

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

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

(iii): $(current\_time - timestamp(x) \leq avi(x))$
$\wedge\ (dt(\tau) - at(\tau) - wcet(\tau) - rr(\tau) \geq 0)$, where $rr(\tau)$ is the remaining response time of the transaction τ, and is calculated in the following way:

```
wait_factor=wait_time/(# executed operations)
rr=wait_factor*(# remaining operations in UT +
                # operations in TU)
```

and `wait_time` is the time the UT has been waiting so far for resources, e.g., the CPU.

Computational Complexity

The computational time complexity of OD, ODO, and ODKB grows polynomially with the size of the data dependency graph G. ODO and ODKB do not generate

[2]A transaction is triggered by the database system by creating a new transaction instance having the same priority as the triggering transaction.

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, which, in the worst case, is $|N|$. Checking p (a predicate, see page 23), which takes polynomial time, precedes every scheduled update. Thus, the computational complexity is $O(|N| \times poly(p))$, where $poly(p)$ is a polynomial of p describing its running time.

If we assume updated data items do not need to be updated again during the execution of a UT, then p is checked at maximum once for every data item. Since there is a fixed finite number of data items in the system 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 4.5. 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 p which takes exponential time.

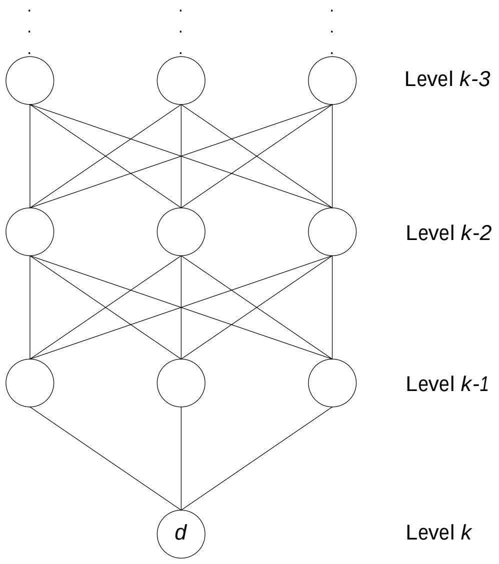


Figure 4.5: A graph that gives the worst-case running time of algorithms OD, ODO, and ODKB.

4.5 On-Demand Updating Algorithms in Value-Domain

In this section the on-demand updating algorithms OD, ODO, and ODKB are extended with functionality to use data freshness in the value domain. Furthermore, the on-demand updating algorithm, On-Demand Depth-First

Table 4.1: A summary of updating algorithms.

			Data Freshness	
Abbreviation	Long name	Relevance check	Time domain	Value domain
OD	On-demand		√	
ODO	On-demand with optimistic option		√	
ODKB	On-demand with knowledge-based option		√	
OD_V	OD with value domain			√
ODO_V	ODO with value domain			√
ODKB_V	ODKB with value domain			√
ODKB_C	On-demand with knowledge-based option and relevance check	√		√
ODDFT	On-demand depth-first traversal			√
ODBFT	On-demand breadth-first traversal			√
ODTB	On-demand top-bottom traversal with relevance check	√		√
ODDFT_C	On-demand depth-first traversal with relevance check	√		√

Traversal (ODDFT), is also described. The algorithms On-Demand Knowledge-Based option with relevance check (ODKB_C) and On-Demand Breadth-First Traversal (ODBFT) are presented in Appendix B. Simulation results show that using data freshness defined in the value domain gives better performance compared to using data freshness defined in the time domain. Two experiments are presented in this section. Three complementary experiments are described in B.8.

We saw in Section 4.4 that the triggering criteria involve checking whether the data item about to be read by a transaction is too old. This check can be changed to either distance-based data freshness (definition 4.2.1) or interval-based data freshness (definition 4.2.2). In this thesis we use distance-based data freshness. The triggering criteria are changed as indicated below:

- For OD, the triggering criterion for a read operation, Read(x), at time t is $|v_x^t - v_x^{t'}| < \delta_{d_i,x}$, where $x \in R(d_i)$, v_x^t is the current value of data item x, and $v_x^{t'}$ is the value used at time t' when d_i was last derived. This OD algorithm is denoted OD_V.
- For ODO, the triggering criterion is changed to $|v_x^t - v_x^{t'}| < \delta_{d_i,x} \wedge (dt(\tau) - at(\tau) - wcet(\tau)) \geq 0$, and the new algorithm is denoted ODO_V.
- For ODKB, the triggering criterion is changed to $|v_x^t - v_x^{t'}| < \delta_{d_i,x} \wedge dt(\tau) - at(\tau) - wcet(\tau) - rr(\tau) \geq 0$, and the new algorithm is denoted ODKB_V.

4.5.1 On-Demand Depth-First Traversal

Note that On-Demand Depth-First Traversal (ODDFT) uses the PAUS updating scheme. The goal of step PAUS_S3, as also for AUS_S3, is to create a schedule of updates that, when executed, make data items fresh before the UT continues to execute. With ODDFT we try to achieve this with a simple algorithm. The algorithmic steps of ODDFT are: (i) traverse data dependency graph G bottom-up using depth-first order, (ii) in each reached node determine if the corresponding data item needs to be updated, (iii) put needed updates in a schedule, and (iv) execute the updates in the schedule.

The marking of potentially affected data items in step AUS_S2 includes all descendants of the changed data item. The reason changes need to be propagated down in the data dependency graph G is because ODDFT traverses G bottom-up. In order to know whether a data item has stale ancestors, the markings need to propagate down G.

Algorithmic step (i) of ODDFT is realized by recursively visiting every ancestor of a node corresponding to the data item. In this way, G is traversed bottom-up in depth-first order.[3] 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

[3]The algorithm PREC1 on page 16 is implemented in this way.

of updates in algorithmic step (iii). 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.

Algorithmic step (ii) is implemented by the following if-statement (see Section B.3 for the pseudo-code of ODDFT).

```
if $pa(d) > 0 \wedge error(x, freshness\_deadline) > \delta_{d,x}$ then
    code...
end if
```

Proposition 4.3.1 is used (the test $pa(d) > 0$) to check whether data item d needs to be updated. The function, $error$, estimates how much the value of data item x will change until the time $freshness\_deadline$. Thus, $freshness\_deadline$ is the latest time the values of used data items in a transaction should still be fresh. $freshness\_deadline$ is set to the deadline of the UT that triggers the updating of a data item. The computational complexity of ODDFT is given in Section B.3.

4.5.2 Relevance Check: ODDFT_C

ODDFT can be enhanced with a relevance check (definition 4.3.2) that checks if the immediate 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. The algorithm is denoted ODDFT_C, and it builds a schedule of updates by traversing the data dependency graph bottom-up. When the schedule has been constructed and the updates should start executing, a relevance check is done before an update starts to execute. If the relevance check determines an update to be irrelevant to execute because the data item's value will not change, then the update is skipped.

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

4.5.3 On-Demand Top-Bottom with relevance check

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. Step (ii) of ODTB is a traversal from affected data items down to d_{UT}—the data item a user transaction will use—and updates of the traversed data items are inserted in a schedule of updates. Step (iii) is to execute updates of the data items that are put in the schedule.

In algorithmic step (i) of ODTB, the top-bottom traversal is done using a pregenerated schedule, because on-line searching in the representation of data dependency graph G for paths from base items to a specific derived data item is too time consuming. We discuss how a schedule can be pregenerated and used in ODTB in Appendix B (Section B.6) not to clutter the text.

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 item. The schedule of updates is constructed by determining which sub-schedules to include in the final schedule of updates.

The computational complexity of ODTB can in the worst-case be exponential and the derivation of this result is presented in Section B.6.

4.5.4 RADEx++ Settings

This section describes simulator settings in experiments using the RADEx++ discrete-event simulator. RADEx++ uses the auxiliary functions: BeginTrans, ExecTrans, and AssignPrio that are described in Appendix B (Section B.7).

The experiments conducted using the RADEx++ 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 experiments 1a, 1b, 1c, 1d, 1e, 2a, and 2b. They evaluate different aspects of updating algorithms.

RADEx++ is set up to function as a firm real-time main-memory database. Two queues for transactions are used: STs in the high priority queue, and 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 can use any data item in all experiments except in one experiment (1b) where user transactions derive only data items that reside in leaf nodes in G. The data item a UT derives is randomly chosen from the set of all derived data items. In experiment 1b, the leaf node 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, because scheduled updates are scheduled as late as possible and, thus, the latest start time has been missed. 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 `max_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(`max_change/2`,`max_change/4`), limited to the interval $(0,$ `max_change`$)$. The value of `max_change` and avi are derived from the same uniform distribution U(200,800). Data validity interval δ_{d_i,d_j}, where d_j is an immediate parent of d_i, is given by $avi(d_j)$ times $factor$. A $factor$ equal to one implies that the avis give a good reflection of the value changes if $factor$ 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 $blockingf$ is set to one if not stated otherwise. The database parameters and the settings are given in Table 4.2.

Table 4.2: Parameter settings for database simulator.

Parameter	Explanation	Setting
avi	absolute validity interval	U(200,800) ms
$\delta_{i,j}$	data validity interval for i	$factor \times avi(j)$
`max_change`	max change of a data item during its avi	U(200,800)
`STProcCPU`	max execution time of a ST operation	1 ms
`UTProcCPU`	max execution time of a UT operation	10 ms
$factor$	1	
$blockingf$	Blocking factor	1 (default)

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 45×105 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 an ancestor 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 a UT is sent to an actuator (see Section 8.1.2 for a discussion). The examples we have seen, e.g., the data dependency graph in Figure 2.6, 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: $error(x,t) = t - timestamp(x)$.

4.5.5 Performance Results

In this section we present Experiment 1a and Experiment 1d. Experiments 1b, 1c, and 1e are presented in Appendix B. The experiments show how updating algorithms using time domain for defining data freshness, e.g., OD, perform compared to updating algorithms using value domain for defining data freshness, e.g., OD_V. A discrete-event simulator called RADEx++ has been used to evaluate the presented updating algorithms. The simulator settings are described in Section 4.5.4. The algorithms that are evaluated are listed in Table 4.1. Note that Chapter 6 shows comparisons between on-demand updating and well-established updating using dedicated tasks. Further, Chapter 6 also shows analytical formulae for estimating workload of on-demand updates.

Experiment 1a: Consistency and Throughput With No Relevance Check

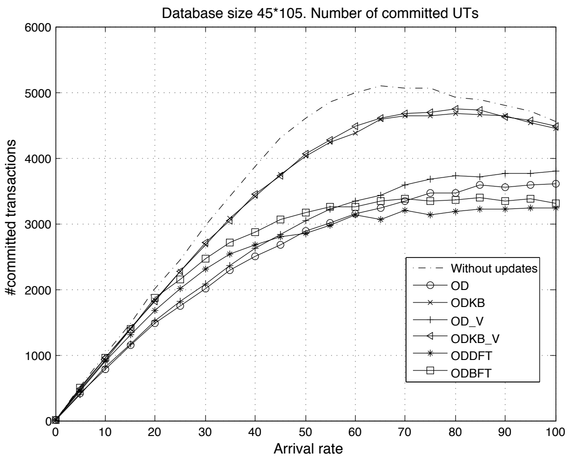
The objective of this experiment is to determine the performance of updating algorithms without relevance checks, i.e., the algorithms are OD, ODO, ODKB, ODO_V, ODKB_V, ODDFT, and ODBFT (described in Section B.4). The main performance metric is the number of successfully committed valid UTs according to the definition of an absolute system, definition follows below, i.e., a UT is valid if, at commit time, the deadline is met and the derived value is unaffected by concurrent changes in other data items.

Definition 4.5.1 (Absolute Consistent System [75]). *In an absolute consistent system, a UT, with a commit time t and a read set R, 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.*

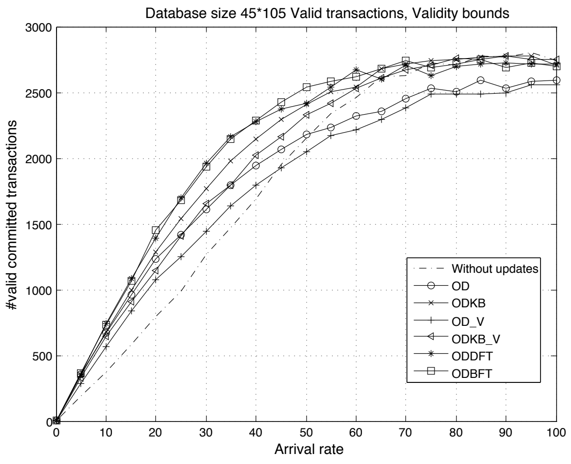
Figure 4.6 shows total number of committed UTs in Figure 4.6(a) and number of committed UTs that are valid based on an absolute system in Figure 4.6(b). The ratio of number of valid committed UTs and number of generated UTs over all simulation runs, i.e.,

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

is plotted in Figure 4.7. 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 is 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 4.6(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



(a) Number of committed UTs.



(b) Number of valid committed UTs.

Figure 4.6: Experiment 1a: Consistency and throughput of UTs (confidence intervals are presented in Figure D.1).

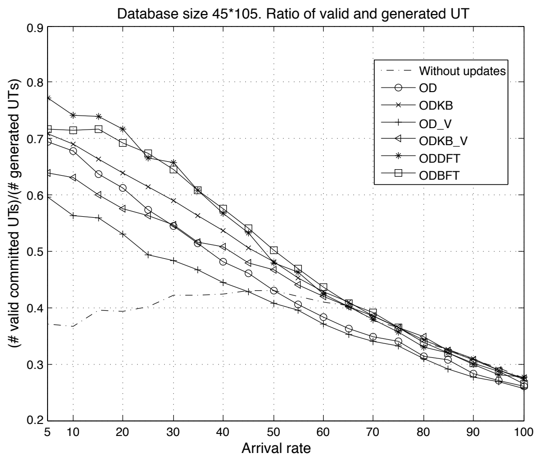


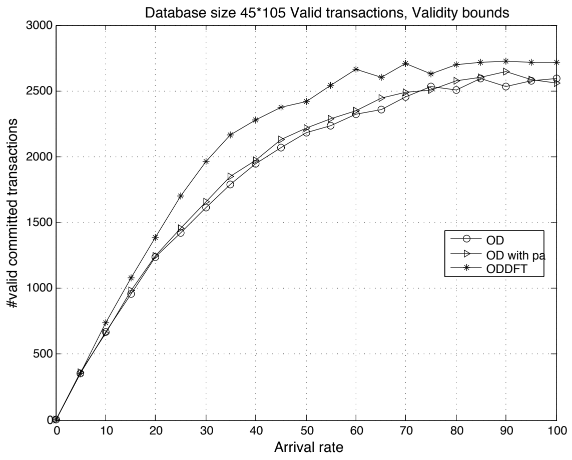
Figure 4.7: Experiment 1a: Ratio of valid committed UTs and total number of

and since more UTs commit under ODKB compared to OD, this keeps the total number of valid data items higher.

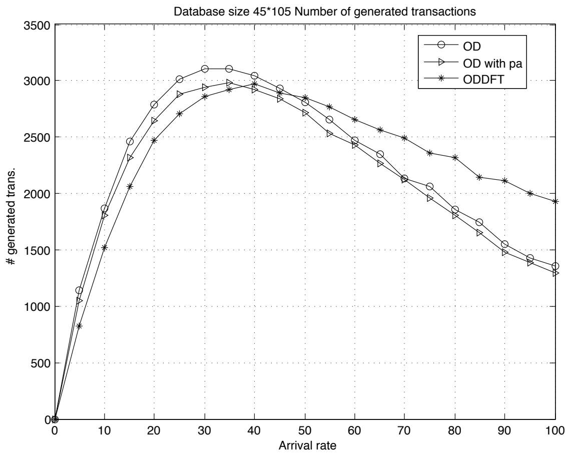
A counterintuitive result can be seen in Figure 4.6(b). OD_V and ODKB_V let fewer valid UTs commit compared to corresponding time domain algorithms, i.e., OD and ODKB. The reason is that when a read operation is issued by a transaction using the _V version of the algorithms, it is checked whether the data item is valid or not by investigating the validity of its ancestors. If the ancestors are valid, the data item is not updated. However, the data item might have been potentially affected by a change in an ancestor, 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 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 4.6(a) shows that their number of valid committed UTs is less than for ODKB but a bit higher than for OD. Figure 4.6(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 AssignPrio). This approach pays off, since the number of committed valid UTs is higher than for any other algorithm.

Figure 4.8(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 4.8(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.



(a) Number of valid committed UTs.



(b) Number of generated triggered updates.

Figure 4.8: Experiment 1a: Effects of measuring staleness of data items at deadline of UT (confidence intervals are presented in Figure D.2).

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.

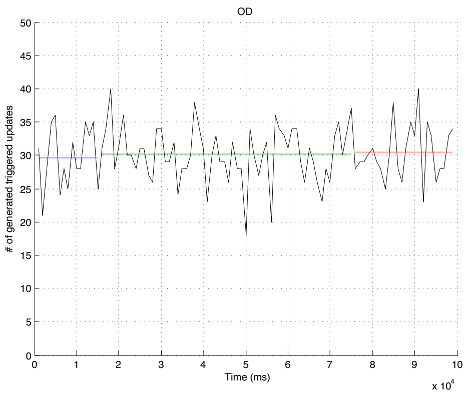
Table 4.3: Experiment 1d: Statistical data from transient and steady state simulation.

Algorithm	# committed UTs	# valid committed UTs
OD	2035	1742
ODDFT	2445	2207
ODKB	2698	2138
ODKB_V	2748	2121

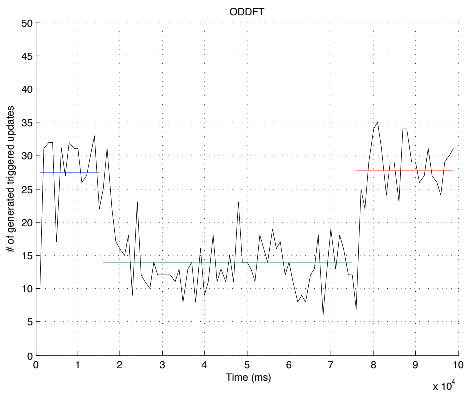
One interesting aspect of using data freshness in the value domain is that the number of generated updates should be affected by the current state of the system. If the system is in a steady state, i.e., the external environment does not change much implying that the sensor values are not changing much, 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 45×105, and 100 s is simulated. Two parameters are introduced: `change_speed_of_sensors` and `change_speed_of_user_trans`. Data items change with the following speed: N(`max_change/change_speed_of_X`, `max_change/`($4\times$`change_speed_of_X`), where `X` is substituted with `sensors` or `user_trans`. For the first 15 s, `change_speed_of_sensors` is set to 1.2, which gives rapid changes (transient state), from 15 s to 75 s `change_speed_of_sensors` is set to 50 (steady state), and from 75 s the system again enters 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.

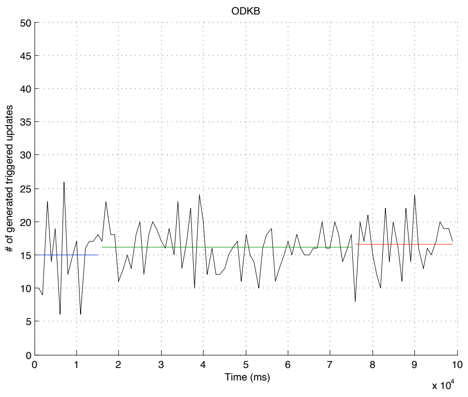
Figure 4.9 contains the simulation results. The horizontal lines represent the average number of generated triggered updates during the indicated interval. ODDFT clearly generates fewer triggered updates during the interval 15–75 s than OD, which is unaware of that base items live longer in this interval. 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 4.3 shows the number of committed UTs and the number of valid committed UTs from the four simulations shown in Figure 4.9. 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.



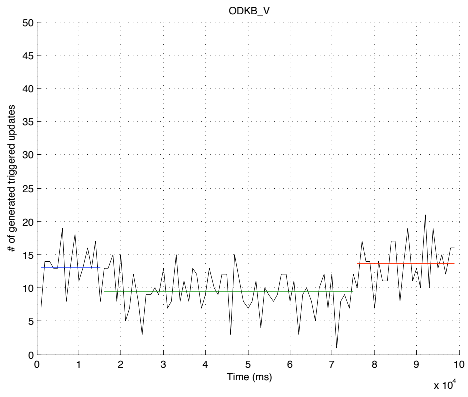
(a) OD



(b) ODDFT



(c) ODKB



(d) ODKB_V

Figure 4.9: Experiment 1d: Simulation of transient and steady states of a system.

4.5.6 DIESIS 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 an external environment affect the workload of updates scheduled by updating algorithms.

Simulator Setup

The ODTB updating algorithm is evaluated using DIESIS integrated in an EECU software. The simulator setup is depicted in Figure 4.10. 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 change the external environment and in particular 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.

DIESIS 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 2.6 are implemented. There is one UT that is requested periodically by the original EECU software. The UT is deriving TOTALMULFAC. The implementation is further described in the section Database System: DIESIS (Section 4.1).

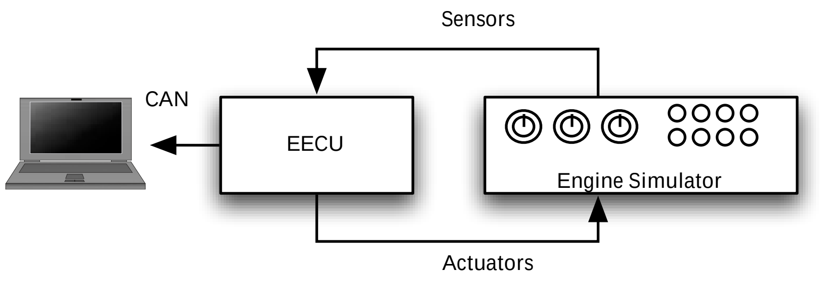


Figure 4.10: Overview of the EECU and engine simulator.

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.

Recalculations of TOTALMULFAC are needed when the engine speed changes. Figure 4.11 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 facilities.

4.6 Wrap-Up

In the introduction of this chapter, we stated that the objectives of introducing functionality in a real-time database for keeping data items up-to-date are to address requirements R1 (organize data) and R3 (protect data, avoid duplicate storage, guarantee correct age of data and low overhead of data maintenance). In this chapter, we showed that maintaining data freshness on-line and measuring data freshness in the value domain can use the CPU resource more efficient compared to if data freshness is measured in the time domain. However, the family of on-line updating algorithms presented in this chapter cannot guarantee that data items are updated before a transaction starts in such a way that the transaction can produce an acceptable result. The reason is that the general problem of choosing updates and considering data relationships is NP-hard in the strong sense (see Section 3.3.1) and the on-line algorithms described in this chapter are simplified to reduce computational complexity in such a way that they reject updates when the updates cannot be fitted within the available time. Let us give an example of this behavior. Let us assume d_9 in Figure 2.6 is about to be read in a user transaction and d_3 and d_5 are marked as potentially affected by changes in the external environment. ODDFT is used

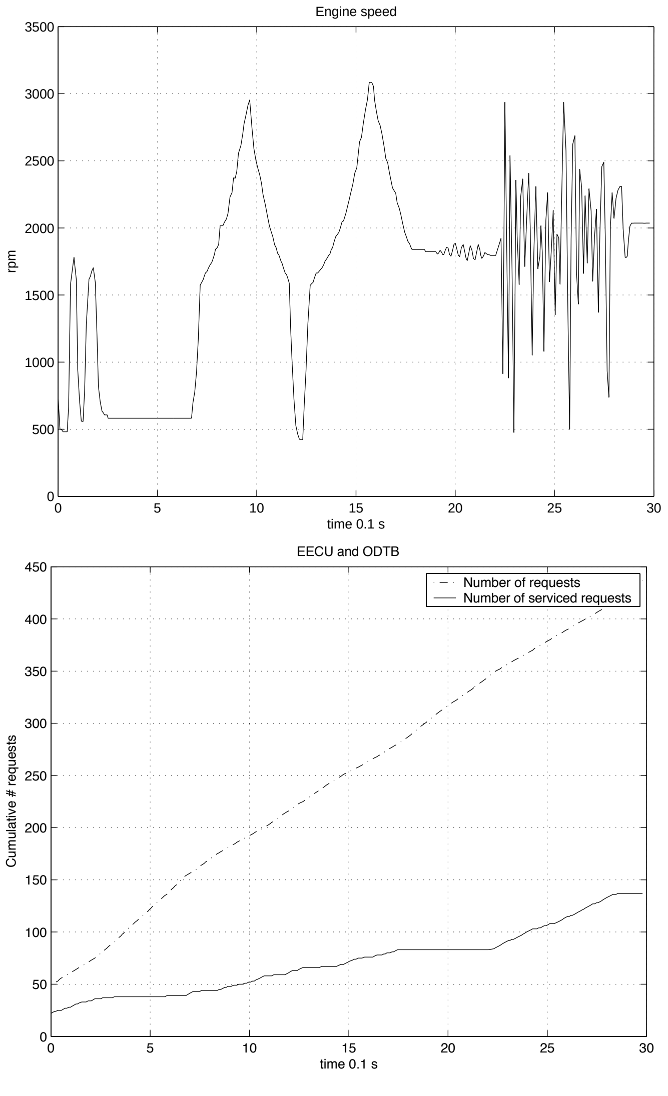


Figure 4.11: Experiment 3: Performance results of a database implementation in an EECU. The performance metric is the number of cumulative recalculations.

as the updating algorithm with the if-statement on line 3 in Figure B.1 enabled. ODDFT traverses d_7 and d_3 followed by d_8 and d_5. Let us assume there is time available for updating d_7 but not d_3 and for d_8 but not d_5. Thus, updates are executed for d_7 and d_8 and they will read old values on d_3 and d_5 since they were not updated. This issue is further discussed in Chapter 7.

Performance of on-demand updating measuring data freshness in the value domain has been compared to on-demand updating in time domain in this chapter. The findings are that measuring data freshness in value domain gives better utilization of the CPU resource compared to using time domain. In addition, in Chapter 6 we compare on-demand updating to updating using dedicated tasks [134–136].

CHAPTER 5

Multiversion Concurrency Control With Similarity

This chapter describes the issues of data consistency and snapshots. Note that some sections related to multiversion concurrency control with similarity are presented in Appendix C in order to ease the reading of this chapter. The objective of introducing algorithms providing snapshots are (i) to make it possible to use consistent and up-to-date data that is correlated in time (addresses requirement R3c in Chapter 3), (ii) concurrency control (addresses requirement R3a), (iii) use resources more efficient (addresses requirement R3d), and (iv) to ease development efforts (addresses requirement R1). We show in this chapter that by combining an updating algorithm with a mechanism to store several versions of data items and then choosing versions of different data items such that they form a snapshot at the correct time, resources can be used more efficient than using single versions of data items, which is the normal way of implementing embedded systems. Furthermore, the snapshot algorithm takes care of concurrency control. The development efforts can be eased since the programmers do not need to handle snapshots explicitly, because the snapshot functionality is part of the database.

The performance of snapshot algorithms is evaluated in Experiment 4 where DIESIS is used on a PC to evaluate performance of different concurrency control and snapshot algorithms. The updating algorithm is ODTB in all evaluations.

The outline of this chapter is as follows. Section 5.1 describes the outline of the MVTO-S algorithm. Section 5.2 describes three implementations of the algorithm: MVTO-S^{UV}, MVTO-S^{UP}, and MVTO-S^{CRC}. Section 5.3 describes an extension of the OCC concurrency control algorithm that uses similarity. Section 5.4 gives implementation details of MVTO-S on Rubus and μC/OS-II. Section 5.5 shows performance evaluations. Finally, Section 5.6 wraps up this chapter.

5.1 Multiversion Concurrency Control With Similarity

The requirements stated in Chapter 3 put a requirement on the data management functionality of an embedded system to provide snapshots to transactions. In this section are multiversion timestamp ordering concurrency control with similarity algorithms (MVTO-S) proposed. Three implementations of MVTO-S are presented namely MVTO-S^{UV}, MVTO-S^{UP}, and MVTO-S^{CRC}. The algorithms combine updating algorithms using similarity with storing multiple versions of data items. The performance results are presented in Section 5.5 and in Appendix C.

5.1.1 MVTO with Similarity

Requirement R3c presented in Section 3.3, 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 and when the transaction using values has started. 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 which they should according to requirement R3d. Furthermore, transactions have to produce consistent results, i.e., be view-similar to a serial schedule, which is in agreement with requirement R3a.

The MVTO concurrency control algorithm, described in the section Multiversion Concurrency Control (Section 2.6.2), 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 (SA4 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.

We first discuss the outline of the MVTO-S algorithm in the context of one UT. Assume one transaction, τ, is about to start, and its read operations should perceive values as originating from the same system state. The read operations must then read correct versions of data items, and these versions must be up-to-date. Hence, there should be a way of mapping the readings by read operations in τ to updated versions.

The mapping from transaction to versions is done via logical time. It is sufficient to read versions that were valid when τ started, because τ then perceives versions from the same state that also are sufficiently close in time to the calculation the transaction performs. A proper version of a data item is the version with latest timestamp less than or equal to $ts(\tau)$. If the updating algorithm atomically generates a schedule of updates when τ starts, then we know which updates are needed to make data items up-to-date. Due to similarities some of the updates might be possible to skip. MVTO-S is divided into two sub-algorithms: arriving transaction (AT) that creates a schedule, and executing transaction (ET) that checks similarities and writes new versions.

The AT sub-algorithm executes when a transaction τ arrives. The steps are:

AT1: A global virtual timestamp $gvts$ is assigned the timestamp of the oldest active transaction, i.e., $gvts = \min_{\forall i, \tau_i \in activeT}\{ts(\tau_i)\}$, where $activeT$ is the set of all active transactions.

AT2: If τ is a UT then a schedule of needed updates is constructed atomically, i.e., uninterrupted by other transactions, by an updating algorithm, e.g., ODTB.

The steps of the ET sub-algorithm are:

ET1: When a transaction τ enters its BOT operation the following steps are taken:

ET1.1: Calculate the write timestamp of version j of data item d_i that τ derives:

$$wt(d_i^j) = \max\left\{\max\{wt(d_k^l)|\forall d_k^l \in V(d_m)\}|\forall d_m \in R(d_i)\right\} \quad (5.1)$$

ET1.2: Find a proper version at time $wt(d_i^j)$ and denote it d_i^n. If $wt(d_i^j) = wt(d_i^n)$, then the update can be skipped since the version already exists. Otherwise continue with ET1.3.

ET1.3: Check the relevance of executing transaction τ by using similarity. The value of read set members of d_i^j is compared to values of read set members of d_i^n. A read set member is denoted d_m. The check is done as follows using distance-based similarity:

$$\forall d_m \in R(d_i), |v_{d_m}^{wt(d_i^j)} - v_{d_m}^{wt(d_i^n)}| \leq \delta_{d_i,d_m}, \quad (5.2)$$

and as follows using interval-based similarity, $\forall d_m \in R(d_i)$:

$$fixedint_{d_m}\left(v_{d_m}^{wt(d_i^j)}\right) = fixedint_{d_m}\left(v_{d_m}^{wt(d_i^n)}\right). \qquad (5.3)$$

If all checks in equations (5.2) or (5.3) evaluate to true this means that τ can be skipped. Otherwise start executing τ.

ET2: Every read operation of τ reading a data item d_i reads a proper version n of d_i.

ET3: Handling of write operations of τ.

ET3.1: If $ts(\tau) > gvts$, then an operation writing data item d_i creates a new version if enough space can be accommodated for such a version (otherwise go to step ET3.2). If $ts(\tau) = gvts$ then no transaction is interrupted and might need the old version, and, thus, τ overwrites the current version of the data item. The timestamp of the new version is the maximum of the write timestamp of read values, i.e., $wt(d_i^j) = \max\{wt(d_k^n) | \forall d_k^n \in RS\}$. Also in this step, all versions older than $gvts$ are pruned from the memory pool to free memory.

ET3.2: If there is not enough space for a new version, the transaction with timestamp equal to $gvts$ is restarted and $gvts$ is recalculated. Versions with a write timestamp less than the new $gvts$ are purged to free memory. In this way the oldest active transaction gets restarted, and this is also the transaction with the lowest priority (note that transactions are executed according to priority). Thus, MVTO-S is aware of transaction priorities and restarts low priority transactions before high priority transactions.

Next an example is given on how the MVTO-S algorithm works.

Example 5.1. *Consider that an arriving UT, τ_1, using data item d_5 is assigned timestamp 8. Step AT1 assigns 8 to $gvts$. Step AT2 creates a schedule of needed updates, e.g., $[\tau_{d_1}, \tau_{d_3}, \tau_{d_2}, \tau_{d_4}]$, where d_5 directly depends on d_3 and d_4 and indirectly on d_1 and d_2. Assume two STs arrive updating base items d_8 (that d_1 reads) with timestamp 9 and d_9 (that d_2 reads) with timestamp 10. Step ET3.1 creates new versions of d_8 and d_9 since both STs had larger timestamps than $gvts$.*

Next arrives τ_2 with timestamp 11 using data item d_6. It has higher priority than τ_1 since it is not yet finished. Thus, $gvts$ is 8, and step AT2 creates the following schedule $[\tau_{d_2}, \tau_{d_4}]$. The TUs τ_{d_2} and τ_{d_4} are executed with timestamp 11. In step ET1.1 of τ_{d_2}, the write timestamp of a possibly new version of d_2 is calculated by looking at read set members of d_2. In this case it is 10 since a ST with timestamp 10 updated d_9. Step ET1.2 finds a proper version of d_2, say with timestamp 5. In step ET1.3 a similarity check is done for each read set member. Hence, a similarity check is done between a version of d_9 with timestamp 5 and the version with timestamp 10. If these two versions are *similar, then transaction τ_{d_2} can be skipped, and transaction τ_{d_4} would read the version of d_2 with timestamp 5.*

Next we give theorems and proofs on the behavior of MVTO-S.

Lemma 5.1.1. *Using MVTO-S, a proper version of a data item d_i at time $t = ts(\tau)$ represents an up-to-date value.*

Proof. Assume d_i^n is a proper version but it is stale. Now assume step ET3.1 of a TU installs a version since an update was scheduled in step AT2 which schedules all needed updates. Denote the new version d_i^{n+1}. The timestamps are ordered as follows $wt(d_i^n) < wt(d_i^{n+1}) \le t$ since by step ET1.1 and ET2 the write timestamp of d_i^{n+1} is the maximum of all accessed read set members but limited by t, i.e., $\forall d_m \in R(d_i)$, $wt(d_i^{n+1}) = \max\left\{\max\{wt(d_k^j)|\forall d_k^j \in V(d_m), wt(d_k^j) \le t\}\right\} \le t$, and $wt(d_i^n) < wt(d_i^{n+1})$ since d_i^n was taken for a proper version and is stale. Version d_i^{n+1} is an up-to-date proper version, and it is valid at time t.

When versions are removed from the pool by step ET3.2, they are removed according to earliest timestamp first. Thus, if version d_i^{n+1} is removed, version d_i^n has been removed before d_i^{n+1} and therefore a proper version d_i^{n+2} of data item d_i at time t is up-to-date. □

Theorem 5.1.2. *MVTO-S ensures that a transaction τ reads up-to-date versions of read set members (step ET2) such that the start time of τ is in the time interval $I = \bigcap\{VI(d_i^j)|\forall d_i^j \in R(d_i)\}$, where $VI(d_i^j)$ is the interval when version j of d_i is valid.*

Proof. We only consider transactions that commit. An interval I is built iteratively for each read operation. We have that for any version j of data item d_i, $VI(d_i^j) = [wt(d_i^j), wt(d_i^{j+1})]$. A proper version n has by definition $wt(d_i^n) \le ts(\tau)$. For every read version d_i^n (ET2) it holds that $wt(d_i^n) \le ts(\tau)$ since by lemma 5.1.1 there cannot exist a not yet updated version in the interval $[wt(d_i^n), ts(\tau)]$.

We must show that $ts(\tau) < wt(d_i^{n+1})$ for all read versions d_i^n, i.e., an up-to-date proper version is alway chosen. Since a version to read is chosen such that $wt(d_i^n) \le ts(\tau)$ and $wt(d_i^{n+1}) > wt(d_i^n)$ as step ET1.2 forces unique timestamps on versions, then $wt(d_i^{n+1}) > ts(\tau)$ otherwise d_i^{n+1} would have been chosen in step ET2. Thus, we have shown that read operations executed by τ choose versions such that they are relative consistent (definition 2.2.2) and $ts(\tau)$ is included in the interval where these versions are valid. □

The effect of theorem 5.1.2 is that MVTO-S guarantees that transactions read an up-to-date snapshot of the database that was valid when the transaction started. This is an important property of the algorithm. Some transactions need to read values of data items that are correlated in time, e.g., diagnosis transactions. Next we describe three versions of MVTO-S that differ in the amount of meta-data every version has.

5.2 Implementation of MVTO-S

Step ET1.3 needs information of values of data items in a read set. We now present three different implementations that differ in the amount of meta-data every version has and how ET1.3 can check similarity. The implementations are denoted MVTO-S^{UV}, MVTO-S^{UP}, and MVTO-S^{CRC}.

5.2.1 MVTO-S^{UV}

One solution to determine similar versions (step ET1.3) 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-S^{UV}, UV is an abbreviation for Use Versions. The similarity check is performed using one of either definition 4.2.1 or definition 4.2.2. An example is depicted in Figure 5.1. In this example, the old version is not similar to the new version since immediate parent d_{27} has changed too much between the derivation of the two versions.

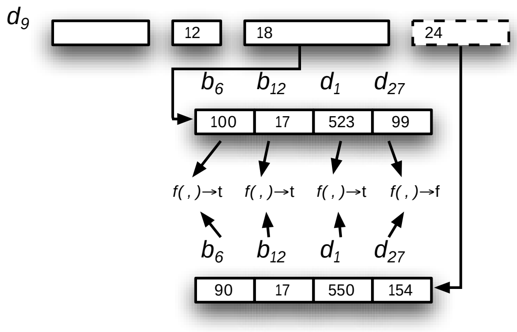

d_9
12
18
24
b_6 b_{12} d_1 d_{27}
100 17 523 99
f(,)→t f(,)→t f(,)→t f(,)→f
b_6 b_{12} d_1 d_{27}
90 17 550 154

Figure 5.1: Determine if two versions are similar by using function f on every pair of immediate parents.

5.2.2 MVTO-S^{UP}

This section describes an implementation of MVTO-S denoted MVTO-S^{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 additional memory is needed for storing values of immediate parents inside the versions. An example of this idea is depicted in Figure 5.2. UT τ derives d_{36}, and data item d_4, d_7, and d_{23} are immediate parents of d_{36}. The dashed area of data item d_{36} is the potentially new version that τ would write. UT τ needs to execute if values used to derive the existing version are not similar to values τ would read. In this example only immediate parent d_4 has changed since the version was derived. If the two versions of d_4 are similar, then τ can be skipped.

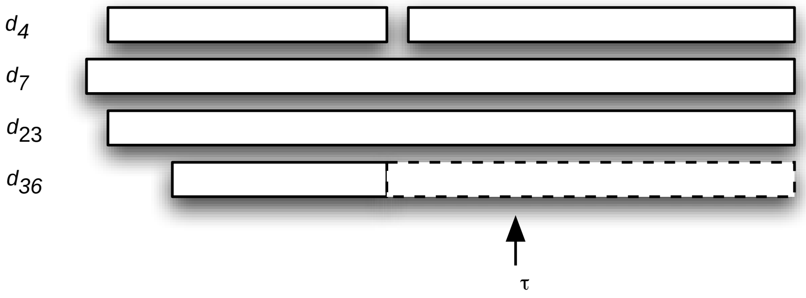


Figure 5.2: UT τ derives d_{36}. Algorithm CheckSimilarity investigates if the existing version of d_{36} is similar to the one τ derives.

The algorithmic steps for doing the similarity check are as in Figure 5.3. The parameters are UT τ and 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 ET3.1 of MVTO-S. If a version with the timestamp already exists, this means that τ 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 τ 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 τ can be skipped. This is implemented in the for-loop on line 6, and the if-statement on line 10.

In MVTO-S$^{\text{UP}}$, when a transaction would derive a similar version, because the values of the immediate 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 immediate parents have been removed from the memory pool. Storing the read values in the version

```
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 immediate parent $x$
         read by $\tau$.
 8:    Let $value(x_z)$ be the value stored in the version of immediate 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
```

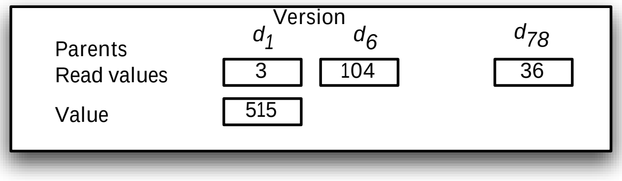
Figure 5.3: CheckSimilarity algorithm.

as in MVTO-S$^{\text{UV}}$ has the benefit that values on immediate parents always can be found. The disadvantage is that every version has a high memory overhead. In MVTO-S$^{\text{UP}}$, this memory overhead is removed, and replaced with searching in the memory-pool. Thus, every version becomes smaller than in MVTO-S$^{\text{UV}}$, but there is a possibility that similar versions are interpreted as not being similar because values of immediate parents could not be found.

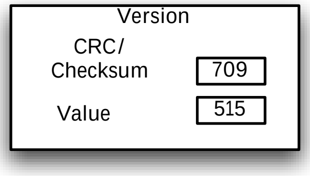
5.2.3 MVTO-S$^{\text{CRC}}$

This section describes an implementation of MVTO-S that is denoted MVTO-S$^{\text{CRC}}$. One way to reduce the memory overhead of MVTO-S$^{\text{UV}}$ is to assign an indicator to each version that uniquely identifies which values on immediate parents that have been used. If the indicator takes less memory to store than the read values of immediate parents as in MVTO-S$^{\text{UV}}$, then the memory overhead is reduced. Checksums/CRCs (see section Checksums and Cyclic Redundancy Checks (Section 2.7)) and interval-based similarity (see section Data Freshness (Section 4.2)) are convenient to use to calculate such an indicator. An example using MVTO-S$^{\text{UV}}$ is depicted in Figure 5.4(a), and a version using an indicator is depicted in Figure 5.4(b).

A value can be uniquely identified by one interval using an interval-based similarity. If not more than 256 validity intervals are needed on each data item, then each interval using interval-based similarity can be represented by



(a) Size of a version for MVTO-S$^{\text{UV}}$.



(b) Size of a version for MVTO-S$^{\text{UP}}$.

Figure 5.4: Sizes of versions for MVTO-S$^{\text{UV}}$ and MVTO-S$^{\text{UP}}$.

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 interval number (using an interval-based similarity) 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 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:

$$\begin{array}{lll} (102, 100, 100) & (100, 102, 100) & (100, 100, 102) \\ (101, 101, 100) & (101, 100, 101) & (100, 101, 101) \end{array}$$

(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 5.1 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.

Table 5.1: Investigation of the robustness of Fletcher's checksum algorithm.

# of octets	Range of changes	Random	# of equal checksums
6	1–6	No	712 out of 738
6	1–6	Yes	712 out of 738

The results in Table 5.1 are disappointing, but they are also to some extent documented in [120]. 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 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 5.2, 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 [3]). 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 5.2: Investigation of the robustness of CRC-32.

# of octets	Range of changes	Random	# of equal checksums
6	1–6	No	0 out of 738
6	1–6	Yes	0 out of 738

The concurrency control algorithm using a checksum or a CRC-32 to check similarities is denoted MVTO-S^{CRC}.

5.3 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 [86]. In this thesis, the OCC algorithm is enhanced with a similarity-aware validation phase. The algorithm is denoted OCC-S. The optimistic concurrency control algorithm described in section Optimistic (Section 2.6.2) has a validation phase looking as follows [80]:

```
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 5.5. 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.

```
1: **if** $(ws(\tau_i) \cap rs(\tau_j) \neq \emptyset) \wedge (\forall d(d \in ws(\tau_i) \wedge d \in ts(\tau_j)), f(read(d), written(d)) \neq t)$ **then**
2:     $valid = false$
3: **end if**
```

Figure 5.5: OCC-S validation phase.

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 5.6: Implementation details of concurrency control algorithms.

5.4 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. μC/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 has executed 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 straight-forward way to resolve this in Rubus. A longjmp could be simulated by polling the restart flag[1] in the calculation part in a simulation.

HP2PL, OCC, and OCC-S are described in Figure 5.6. 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, τ_1, holding the write-lock rollbacks and is marked for restart. The read-locker, τ_2, has to wait for the write-locker to rollback and the rollback is done with the priority of τ_1, i.e., a transaction τ_3

[1]The restart flag is set to true whenever the transaction is involved in a conflict and it needs to restart.

with priority $prio(\tau_1) < prio(\tau_3) < prio(\tau_2)$ can preempt and execute before τ_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.

5.5 Performance Results of Snapshot Algorithms

This section contains performance results of DIESIS configured for a range of snapshot and concurrency control algorithms. Different aspects of the performance of these algorithms are evaluated. The updating algorithm is ODTB in all experiments. In this section, experiment 4a is presented. Experiments 4b–4f are presented in Appendix C. The simulator setup is also described in Appendix C. The RCR versions of OCC and OCC-S, that are used in the evaluations, are restarting transactions until they are able to read a snapshot of the database.

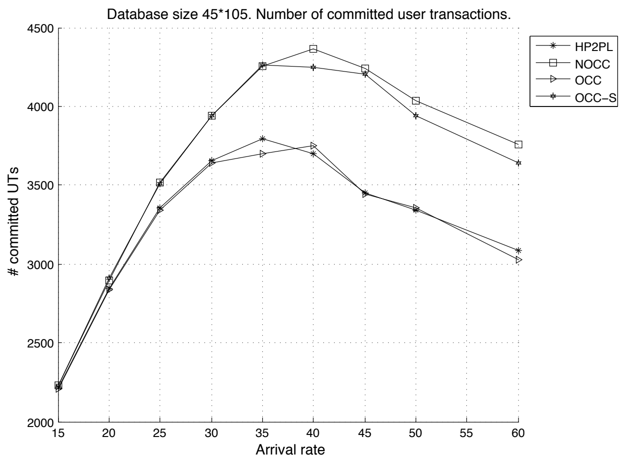
5.5.1 Experiment 4a: Committed User Transactions

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 its deadline.

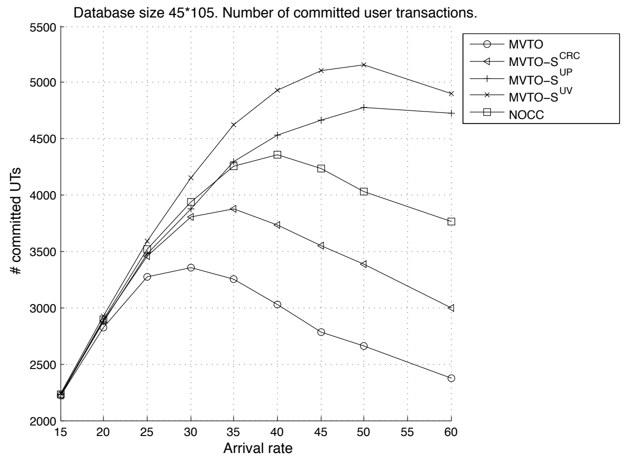
The concurrency control algorithms that are evaluated are HP2PL, OCC, MVTO (Section 2.6.2), MVTO-S$^{\text{UV}}$, MVTO-S$^{\text{UP}}$, and MVTO-S$^{\text{CRC}}$. As a baseline we also use the no concurrency control (NOCC) scheme. Figure 5.7(a) shows the number of user transactions committing before 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 5.7(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-S$^{\text{UV}}$, MVTO-S$^{\text{UP}}$, and MVTO-S$^{\text{CRC}}$ 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 5.8. Comparing Figures 5.7(a) and 5.7(b), the enhanced multiversion algorithms, MVTO-S$^{\text{UV}}$, MVTO-S$^{\text{UP}}$, and MVTO-S$^{\text{CRC}}$, 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-S$^{\text{UV}}$ in Figure 5.9. The single-version algorithms with restarts are penalized by more restarts. Every restart is due to that values have changed, which increase the probability



(a) Number of committed UTs for single-version concurrency control algorithms (confidence intervals are presented in Figure D.7(a)).



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

Figure 5.7: Experiment 4a: Number of UTs committing before their deadlines using single- and multiversion concurrency control algorithms.

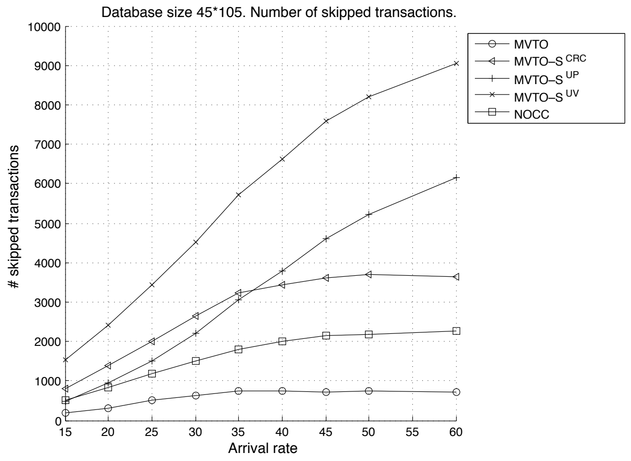


Figure 5.8: Experiment 4a: Number of transactions that can be skipped using ODTB in conjunction with the concurrency control algorithms.

that TUs scheduled by the restarted UT cannot be skipped. Figure 5.10 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 5.9 shows that the MVTO-S^{UV} and MVTO-S^{UP} let considerably more UTs to commit compared to RCR-NOCC, RCR-OCC, and RCR-OCC-S.

MVTO-S^{CRC} is not up to par with MVTO-S^{UV} and MVTO-S^{UP} in Figure 5.7(b). The reason is that MVTO-S^{CRC} must use interval-based similarity and that values are monotonically increasing. Using a distance-based similarity, every new version created is an origin of a distance where all values less than 400 are similar to the new version. However, using an interval-based similarity the value of a new version lies somewhere in an interval and the distance to the next interval tend to be shorter than 400. In effect, fewer transactions can be skipped using interval-based similarity since values are more often assigned to different intervals. Figure 5.11(a) shows how the MVTO algorithms are behaving when interval-based similarity is used. The pool size is 300. In this setting, it is not possible to tell the difference from MVTO-S^{CRC}, MVTO-S^{UV}, and MVTO-S^{UP}. How pool sizes affect the performance is discussed in Experiment 4b, but here we only conclude from Figure 5.11(b) that MVTO-S^{CRC} performs as good as MVTO-S^{UV}. Even though the overhead of storing the values of elements in the read set is reduced to a 32-bit CRC in MVTO-S^{CRC}. Table 5.3 shows how many times MVTO-S^{CRC} 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 interval-based similarity is a reasonable design decision, then MVTO-S^{CRC} is a better choice than MVTO-S^{UV} since a smaller pool size can be used.

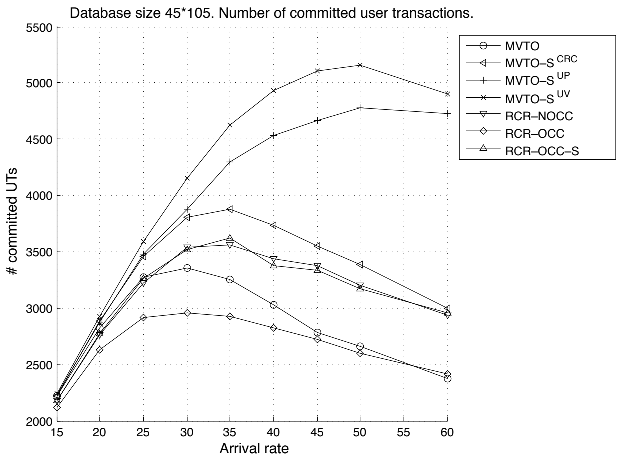


Figure 5.9: Experiment 4a: A comparison of single-version concurrency control algorithms enforcing relative consistency and multiversion concurrency control algorithms (confidence intervals are presented in Figure D.8).

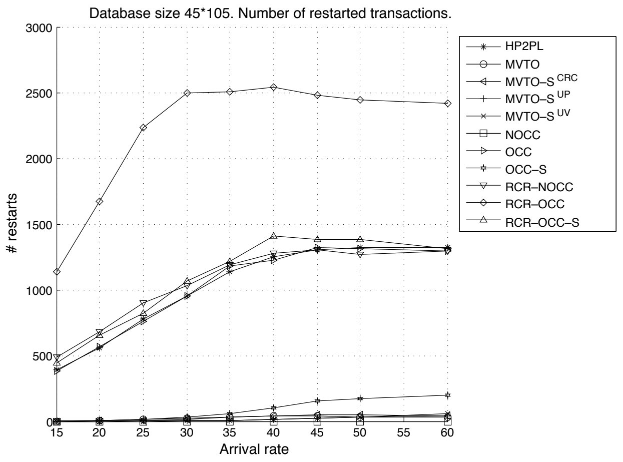
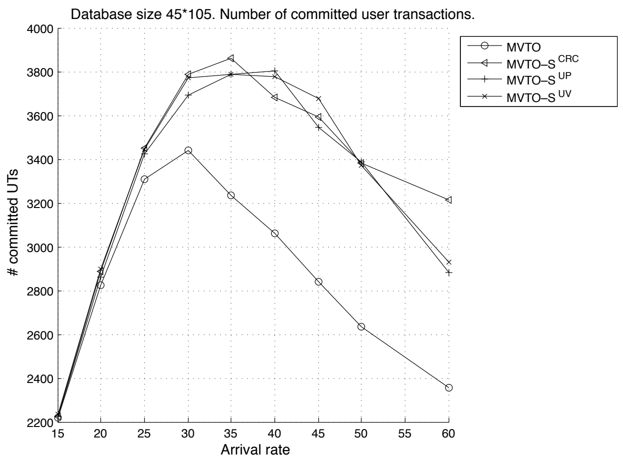
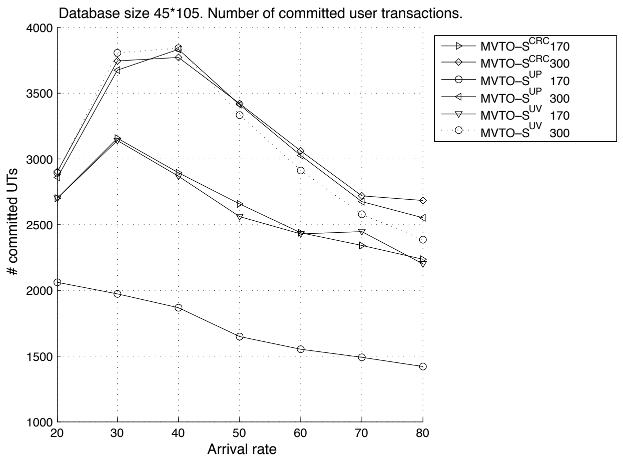


Figure 5.10: Experiment 4a: Number of restarts of transactions for the concurrency control algorithms.



(a) Number of committed user transactions.



(b) Pool sizes.

Figure 5.11: 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-S^{UV}.

Arrival rate	Missed similarities
15	0
20	0
25	0
30	0
35	0
40	0
45	0
50	0
60	0

5.6 Wrap-Up

This chapter describes a snapshot algorithm and three different implementations of it that provides transactions with up-to-date values of data items. The values are up-to-date at the start of the transaction. Performance evaluations show that, besides providing snapshots, the algorithm can also provide enhanced throughput of transactions since updates of historical versions can be skipped if they are already present. This is not possible if single-version concurrency control algorithms are used.

CHAPTER 6

Analysis of CPU Utilization of On-Demand Updating

We have seen in previous chapters that many applications, but not limited to real-time systems, need up-to-date data items. We have seen that it is possible to construct algorithms that can schedule data items to be updated on-demand at a given point in time. These algorithms have been empirically evaluated and discussed in the previous chapters and it is found that the proposed on-demand algorithms give better performance compared to existing algorithms. However, the proposed algorithms introduce a new problem, namely, how to off-line or on-line determine the total workload of the system[1]. This chapter aims at constructing analytical methods to calculate off-line or on-line the total workload of a system that uses on-demand updates. Calculating workload addresses requirement R5, determining if the system is in a transient overload.

The outline of this chapter is as follows. Section 6.1 describes the task model used in this chapter. Section 6.2 gives related work and theoretical results important for deriving the analytical formulae. Section 6.3 presents the analytical formulae. Section 6.4 presents evaluation results and Section 6.5 wraps up the chapter.

6.1 Specialized Task Model

This chapter is devoted to highlighting the performance differences of (i) on-demand updating within tasks and (ii) dedicated tasks update data items. In this

[1]With total workload we mean the workload that includes the work of keeping data items up-to-date.

```
if currentTime - lastUpdate > avi
    Update data item
    lastUpdate = currentTime
    local data item = data item
end if
Execute task
```

Figure 6.1: A τ^{time}-task measuring data freshness in the time domain.

```
if abs(local sensor value - new sensor value) > delta
    Update data item
    local sensor value = new sensor value
    local value = updated value
end if
Execute task
```

Figure 6.2: A τ^{value}-task measuring data freshness in the value domain.

chapter, simplifications are made to the transaction and data model in Section 3.2. There are two reasons for this. First, research on dedicated tasks [134–136] does not handle derived data items where a derived data item depends on another derived data item. Hence, in order to get comparable results with the body of research in the area of dedicated tasks, the data dependency graph G must only consist of two levels. One level with base items and one level with derived data items that are derived from base items only. The second reason is that using a data dependency graph with two levels is a simpler problem than using a graph with arbitrary number of levels. Thus, it may be possible to find analytical formulae for estimating updating workload using a simpler graph, which can then be a guidance in finding formulae for data dependency graphs with arbitrary number of levels.

The on-demand updating algorithms, e.g., ODDFT and ODTB, create a schedule of data items needing to be updated and the order the updates should be executed. The scheduling is separated from the tasks in that it is performed by the database, e.g., DIESIS. However, we can reason about on-demand updates as being part of a task. In doing so, the tasks become conditioned. In this chapter using the specialized task model, tasks using on-demand updating measuring data freshness in the time domain are described as in Figure 6.1. Tasks measuring data freshness in the value domain are described as in Figure 6.2. We refer to these tasks as τ^{time}- and τ^{value}-tasks, respectively.

The data items being used in the system consists thus of two sets. The set B contains base items $b_1, \ldots, b_m$, i.e., sensor values reflecting the state of physical entities. The set D contains derived items, $d_1, \ldots, d_n$, that are derived by functions having only $b \in B$ as inputs, e.g., $f(v_{b_2}^t, v_{b_3}^t) = v_{b_2}^t + v_{b_3}^t$.

6.2 Preliminaries

This section gives background in the area of updates of data items and gives some theoretical results that are used later in the chapter.

In the area of real-time systems, keeping data values up-to-date has previously been studied. As discussed in the previous chapters, there are two ways to determine when to update a data item: either by a dedicated task (DT) executed often enough, or on-demand (OD). Also, two ways to measure data freshness have been devised, namely (i) time domain (TD), e.g., AVIs, and (ii) value domain (VD), e.g., similarity. Thus, there are four ways to configure a system with respect to updating data items: DT+TD, DT+VD, OD+TD, and OD+VD. We now describe previous research in the different configurations.

In the area of AVI and dedicated tasks (DT+TD), techniques have been derived that decrease CPU utilization needed by the dedicated tasks, but yet allowing for sufficient schedulability tests [134–136]. Half-Half (HH) and Deferrable Scheduling (DS) are two examples of these techniques and DS is, to this date, the technique reducing workload imposed by updates the most [134, 135]. As the techniques increase in complexity in order to reduce the CPU workload further, the analysis of schedulability and CPU workload also become more complicated. For instance, CPU workload using DS can only be estimated using the algorithm in Figure 6.3 [135]. The estimate's accuracy has not been stated, but in [135] the measured workload and the estimated workload are very close.

```
Deadline $dt(\tau_0^{time}) = 1$
$period(\tau_0^{time})$ =period time
**for all** $d_i$ **do**
    $U$ =utilization of updates of data items with index less than $i$
    $dt(\tau_i^{time}) = \frac{1}{1-U}$
    $period(\tau_i^{time})$ =period time$-dt(\tau_i^{time})$
**end for**
$U = \sum_{i=0}^{m} \frac{wcet(\tau_i^{time})}{period(\tau_i^{time})}$
```

Figure 6.3: Workload estimation of DS [135]. Note that tasks are sorted in increasing order based on period time

In the area of AVI and on-demand updating (OD+TD), Ahmed and Vrbsky constructed algorithms for soft real-time systems [9]. Arbitrary data relationships are supported, i.e., derived items can depend on values of other derived items. Data relationships are normally described in terms of base items representing entities of the environment and derived items that only depend on base items. Examples of work considering such data models are [73, 135]. No schedulability analysis is given in [9] nor any estimates of how often updates execute, i.e., the system is not analyzable.

Kuo and Mok formalized the notion of similarity (similarity was introduced in Section 2.2.2). In the area of similarity and dedicated tasks (DT+VD), data validity bounds have been used to determine how long period times of tasks can be enlarged in order to reduce workload imposed by updates [68]. Thus, changes of values are translated into how long time a value can be considered up-to-date, which resembles AVIs, which is then used to enlarge the period times. Analyzability can be achieved by using RM or EDF and a CPU utilization test since tasks are periodic.

Chapter 4 was devoted to the OD+VD approach, and it was shown using simulations and real-life proof-of-concepts that OD+VD uses the CPU resource more efficient than periodic tasks using AVI, i.e., HH, and Ahmed and Vrbsky's work. However, since on-demand updating and arbitrary graphs are used it is hard to analyze the system since it is cannot be determined to know at which time a given data item will be updated because it depends on how other data items have changed. Thus, only estimates of workload of updates can be given. Therefore in this chapter we limit the data model to contain base items and derived items that are derived only from base items (the data model in Section 3.2 uses a data dependency graph of arbitrary depth). This is done in order to investigate whether there are any estimation techniques for this simpler model that then can be further extended. Chapter 8 shows a way to relax the assumption that G has only two levels in the case of OD+VD.

The remainder of this section gives a theorem on how the mean time between invocations of a set of periodic events can be calculated.Lemma 6.2.1 gives a formula for two events and theorem 6.2.2 generalizes the lemma to the case with n events.

Lemma 6.2.1. *Let two events* e_1 *and* e_2 *have the period times* $period(e_1)$ *and* $period(e_2)$*. The mean time between invocations of* e_1 *and* e_2 *is*

$$\frac{1}{\frac{1}{period(e_1)} + \frac{1}{period(e_2)}}. \tag{6.1}$$

Proof. We can derive the mean time between invocations by drawing a timeline of the occurrences of e_1, and on top of that draw the occurrences of e_2. Then we take the mean time between occurrences by taking the distance between every two consecutive occurrences and form the mean. The timeline repeats after $period(e_1)period(e_2)/\gcd(period(e_1), period(e_2))$ time units. During this time event e_2 occurs $\frac{period(e_1)}{\gcd(period(e_1),period(e_2))}$ times.[2] By also considering event e_1 an additional $\frac{period(e_2)}{\gcd(period(e_1),period(e_2))}$ occurrences need to be considered. The mean time between invocations is now

$$\frac{\frac{period(e_1)period(e_2)}{\gcd(period(e_1),period(e_2))}}{\frac{period(e_1)}{\gcd(period(e_1),period(e_2))} + \frac{period(e_2)}{\gcd(period(e_1),period(e_2))}},$$

[2]Because $period(e_2) \times x = \frac{period(e_1)period(e_2)}{\gcd(period(e_1),period(e_2))}$, where x denotes the number of times e_2 occurs in the time interval $\frac{period(e_1)period(e_2)}{\gcd(period(e_1),period(e_2))}$.

which can be written as

$$\frac{1}{\frac{1}{period(e_1)} + \frac{1}{period(e_2)}}. \tag{6.2}$$

$\square$

Lemma 6.2.1 is now generalized in the following theorem.

Theorem 6.2.2. *Let a set of n periodic events denoted e_i, $1 \leq i \leq n$, have the period times $period(e_i)$. The mean time between invocations of the events is*

$$\frac{1}{\frac{1}{period(e_1)} + \cdots + \frac{1}{period(e_n)}}. \tag{6.3}$$

Proof. We use lemma 6.2.1 on three events and then generalize the result to n events. Denote the mean time between invocations of the events e_1 and e_2 as $MTBI(e_1, e_2) = 1/(\frac{1}{period(e_1)} + \frac{1}{period(e_2)})$. Now use lemma 6.2.1 on $MTBI(e_1, e_2)$ and $period(e_3)$:

$$\frac{1}{\frac{1}{MTBI(e_1,e_2)} + \frac{1}{period(e_3)}}. \tag{6.4}$$

Substituting $MTBI(e_1, e_2)$ in Equation (6.4) gives the following

$$\frac{1}{\frac{1}{period(e_1)} + \frac{1}{period(e_2)} + \frac{1}{period(e_3)}}. \tag{6.5}$$

It is easy to see that the substitution $MTBI(e_4, e_5)$ of events e_4 and e_5 can be done in Equation (6.5). Hence, such substitutions can be done for all n events. The resulting formula for the mean time between invocations of n periodic events is thus

$$\frac{1}{\frac{1}{period(e_1)} + \cdots + \frac{1}{period(e_n)}}. \tag{6.6}$$

$\square$

We refer to the equation in Theorem 6.2.2 as

$$MTBI(\mathbf{P}) = \frac{1}{\sum_{\forall \tau_i \in \mathbf{P}} period(\tau_i)^{-1}}, \tag{6.7}$$

i.e., the equation states the expected mean time between arrivals of a set, $\mathbf{P}$, of periodic tasks. $MTBI(\mathbf{P})$ is applied to both time- and value-domain tasks therefore the superscript is left out in Equation (6.7).

The next section discusses workload of updates and CPU utilization based schedulability tests.

6.2.1 Workload and Schedulability Tests

For scheduling of periodic tasks, there exist scheduling algorithms that have schedulability tests, e.g., RM and EDF [24]. The CPU utilization based schedulability test works by calculating the CPU utilization of each individual task and then comparing the sum of utilizations to a bound that is scheduling algorithm specific (see also Chapter 2). Thus,

$$\sum_{i=1}^{n} \frac{wcet(\tau_i^{time})}{period(\tau_i^{time})} \leq Bound, \tag{6.8}$$

where $Bound$ depends on the scheduling algorithm being used, and n is the number of tasks in the system. The test in Equation (6.8) works if the deadline equals the period time and that each task is independent, i.e., no synchronization mechanism is used. When using on-demand updating we note that a part of $wcet(\tau_i^{time})$ constitutes the on-demand updating. Thus, Equation (6.8) can be rewritten as follows:

$$\sum_{i=1}^{n} \frac{wcet_i(b_1)wcet(update\_b_1) + \cdots + wcet_i(b_m)wcet(update\_b_m) + CWOD_i}{period(\tau_i^{time})} =$$
$$wcet(update\_b_1)\sum_{i=1}^{n} \frac{wcet_i(b_1)}{period(\tau_i^{time})} + \cdots + \sum_{i=1}^{n} \frac{CWOD_i}{period(\tau_i^{time})} \leq Bound, \tag{6.9}$$

where m is the number of base items in the system, and $wcet(update\_b_1)$ is the worst-case execution time of on-demand updating of b_1, $wcet_i(b_1) = 1$ if task i has an on-demand update of b_1, otherwise $wcet_i(b_1) = 0$, and $CWOD_i$ is the execution time without on-demand updating of task i. Hence, $CWOD_i$ is the worst-case execution time that is used when dedicated tasks are used.

Denoting the mean interarrival time of on-demand updates of b_i as $MTBI(b_i)$, we have that

$$wcet(update\_b_1)\sum_{i=1}^{n} \frac{wcet_i(b_1)}{period(\tau_i^{time})} + \cdots + \sum_{i=1}^{n} \frac{CWOD_i}{period(\tau_i^{time})} =$$
$$\frac{wcet(update\_b_i)}{MTBI(b_1)} + \cdots + \frac{wcet(update\_b_m)}{MTBI(b_m)} + \sum_{i=1}^{n} \frac{CWOD_i}{period(\tau_i^{time})} \leq Bound. \tag{6.10}$$

Equation (6.10) relates to Equation (6.8) in that the left hand side part of Equation (6.8) is equal to the left hand side of Equation (6.10), which is shown using equations (6.3) and (6.9). Equation (6.10) means that the mean interarrival times of on-demand updates contribute to the schedulability of the system. However, Equation (6.10) assumes the worst-case scenario where all on-demand updates are always executed. In the following sections we outline a formula that estimates the mean interarrival time of on-demand updates.

The formula can be used for task sets whose distances between tasks' arrivals can be described with probability density functions. We have found a result that describes such probability density functions of periodic events. This result makes it possible to use our formula for real-time systems with periodic tasks using time domain to measure data freshness.

6.3 Estimation of Mean Interarrival Times of On-Demand Updates

This section presents an analytical formula that answers the important question: 'What is the imposed workload of the updates?' of a system adhering to the task model in Section 6.1.

6.3.1 Time Domain using AVI

Equation (6.10) describes the CPU utilization under the assumption that every on-demand update needs to be executed. However, the if-statements in Figure 6.1 guard the on-demand updates so we know that an update is not needed during the following $avi(d_i)$ time units from the time the update wrote the data value. Thus, we want to calculate the mean interarrival time of an on-demand update when we know its AVI and the interarrival times of the tasks that use the on-demand update. We have devised a formula that derives the mean interarrival times of the execution of on-demand updates. The formula uses a probability density function (pdf) describing the distribution of distances between task arrival times of the tasks that use the on-demand update, because it is assumed the on-demand updates are at the start of the tasks:

$$MTBI(y, \mathbf{P}) = \frac{M \times MTBI(\mathbf{P}) - M \times Q(y, \mathbf{P}) \int_0^y q(x, \mathbf{P}) x dx}{M - M \times Q(y, \mathbf{P})}, \tag{6.11}$$

where M will cancel but it denotes the number of distances used to derive the pdf under study and $\mathbf{P}$ is the set of tasks having a particular on-demand update and thus are used to construct the pdf $q(x, \mathbf{P})$. $MTBI(\mathbf{P})$ is the expected value of the pdf given input $\mathbf{P}$ and $MTBI(\mathbf{P})$ can be calculated by Equation (6.7). Thus, $M \times MTBI(\mathbf{P})$ gives the total sum of the distances. $Q(y, \mathbf{P})$ is the cumulative density function and states the fraction of distances being less than y. Further, $M \times Q(y, \mathbf{P})$ is the number of distances less than y. The product $M \times Q(y, \mathbf{P}) \int_0^y q(x, \mathbf{P}) x dx$ is the sum of distances less than y. The denominator calculates the number of distances remaining if those less than y are removed. The number of distances M in Equation (6.11) cancels. Hence, Equation (6.11) calculates the mean of distances greater than y, which is an estimation of the expected mean interarrival time between executions of the on-demand update with AVI equal to y.

In order to use Equation (6.11) to remove pessimism in the schedulability test described by Equation (6.10), we must find a pdf that describes the distances between arrivals of periodic tasks. Cox and Smith have developed such a pdf. Thus, in Equation (6.11) the following substitutions can be done:

$$\begin{aligned}
q(y,\mathbf{P}) &= \sum_{\forall\tau_i^{time}\in\mathbf{P}} \frac{q_i(y)}{period(\tau_i^{time})} / \sum_{\forall\tau_i^{time}\in\mathbf{P}} period(\tau_i^{time})^{-1}, \\
q_i(y,\mathbf{P}) &= \sum_{\forall\tau_j^{time}\in\mathbf{P},j\neq i} period(\tau_i^{time})^{-1} \prod_{\forall\tau_k^{time}\in\mathbf{P},k\neq i,j} \frac{period(\tau_k^{time})-y}{period(\tau_k^{time})}, \\
Q(y,\mathbf{P}) &= \int_0^y q(x,\mathbf{P})dx,
\end{aligned}$$

where τ_i^{time} refers to one of the periodic tasks in $\mathbf{P}$ using the on-demand update that we want to know the mean interarrival time. The substitution yields $MTBI(avi(b_i),\mathbf{P})$ (Equation (6.11)) to produce an estimate of the mean interarrival time of the execution of the on-demand update. $MTBI(avi(b_i),\mathbf{P})$ can now substitute $MTBI(b_i)$ in Equation (6.10) to remove the pessimism of assuming every on-demand update always execute.

Cox and Smith also showed that the pdf $q(y,\mathbf{P})$ tends to go toward e^{-y} as the number of periodic tasks increases [35]. They showed that the following equation is a good approximation of the substitutions given above:

$$r(z,\mathbf{P}) = e^{-z}\left[1 - \frac{(1+C^2)(z^2-4z+2))}{2m}\right], \tag{6.12}$$

where $z = y\sum_{\forall\tau_i^{time}\in\mathbf{P}} period(\tau_i^{time})^{-1}$, $\mu = \sum_{\forall\tau_i^{time}\in\mathbf{P}} period(\tau_i^{time})^{-1}/m$, $\mu_2' = \sum(period(\tau_i^{time})^{-1})^2/m$, and $C^2 = \mu_2'/\mu^2 - 1$.

In order to calculate Equation (6.11), values of integrals must be estimated. This can efficiently be achieved using Simpson's rule where a segment of the integral can be estimated using $\int_a^b f(x)dx \approx \frac{b-a}{6}(f(a) + 4f((a+b)/2) + f(b))$. Thus, the computational complexity of Equation (6.11) is linear in the size of the value of y since the interval $[0,y]$ (or the interval $[0,z]$ if Equation (6.12) is used) is divided into a linear number of segments. To give an idea of the execution times of Equation (6.11) using $q(y,\mathbf{P})$ and $r(z,\mathbf{P})$, and the algorithm in Figure 6.4, which constructs a timeline of task arrivals and estimates $MTBI(y,\mathbf{P})$ from the timeline, the following performance evaluations have been performed. A task set of 5 periodic tasks with on-demand updates has been used. The max in the algorithm in Figure 6.4 is set to 100000. Equation (6.11) and the algorithm Figure 6.4 are executed 1000 times for 78 simulations and the average execution time is derived. Simpson's rule is used and the interval is divided into 10 segments. The computer is a Mac mini 1.25GHz PowerPC with 512MB memory and the Java runtime system is Apple's Java 1.5 implementation. The average execution time of Equation (6.11) using $q(y,\mathbf{P})$ is 0.72 ± 0.02 ms, using $r(z,\mathbf{P})$ is 0.016 ± 0.0002 ms, and the algorithm in Figure 6.4 is 1.15 ± 0.04 ms.

```
1: Let $earliestArrival$ =earliest arrival time of a task
2: Let earliest task be $\tau_e^{time}$
3: $previous = num = dist = 0$
4: **while** $earliestArrival < max$ **do**
5:     **if** $earliestArrival - previous > avi$ **then**
6:         $dist = dist + earliestArrival - previous$
7:         $previous = earliestArrival$
8:         $num = num + 1$
9:     **end if**
10:    $A_e = A_e + P_e$, where $A_e$ is arrival time of task $\tau_e^{time}$
11:    Let $earliestArrival$ =earliest arrival time of a task
12:    Let earliest task be $\tau_e^{time}$
13: **end while**
14: return $dist/num$
```

Figure 6.4: Timeline approach to estimation of mean interarrival time of an on-demand update using AVI.

Hence, this indicates that Equation (6.11) is 71.9 times faster than the timeline approach. Depending on the CPU being used, the execution time of Equation (6.11) using $q(y, \mathbf{P})$ or $r(z)$ may be low enough to be usable in on-line CPU utilization tests.

The running time of the algorithm in Figure 6.4 is pseudo-polynomial if a correct mean interarrival time is to be achieved since max must be set to the hyper-period of tasks' period times [24]. However, the algorithm can have a polynomial computational complexity if max is set to a fixed value. The while-loop on line 4 iterates $\sum_{\forall \tau_i^{time} \in \mathbf{P}} max/period(\tau_i^{time})$ times. The running time of lines 5–12 is constant. Thus the computational complexity is $O(max)$. Moreover, if the value is set to a value much larger than the period times of the tasks the algorithm will produce a result that is close to the one that would be achieved using the hyper-period.

To summarize this section, we have proposed a formula that estimates the mean interarrival time of executions of on-demand updates. The formula uses probability density functions that describe distances between arrivals of tasks. In this chapter we use a pdf proposed by Cox and Smith that describes interarrival times of periodic events. The formula can be used in a schedulability test in the following way:

$$\sum_{i=1}^{n} \frac{CWOD_i}{period(\tau_i^{time})} + \sum_{j=1}^{m} \frac{wcet(update\_b_j)}{MTBI(avi(b_j), \mathbf{P})} \leq Bound. \tag{6.13}$$

6.3.2 Value Domain

Kuo and Mok formalized the notion of similarity, which says that two values of a data item can be considered similar if and only if they differ not more than a predefined amount [81, 82] (see Section 2.2.2 and Section 4.2). A task using similarity can be seen as containing the following on-demand condition:

```
if abs(localDi - di) > deltaDi ||
   abs(localDj - dj) > deltaDj
  Update dk
  localDi = di
  localDj = dj
execute task
```

Here we see that d_k is derived by reading d_i and d_j, and d_k is recalculated if any of the data items it depends on has changed such that they are dissimilar, i.e., Equation (4.1) does not hold for at least one of the data items d_k depends on.

Equation (4.1) is generalized in this chapter to constitute a probability that d_i (and d_j) has changed such that it is dissimilar to the value it had last time it was used, i.e., $P_{update}(d_i)$ is the probability that d_i is dissimilar. This generalization enables the, soon to be derived (Equation (6.14)), mean interarrival time of updates to be applicable to other events than updates of data. The analytical formula is verified using non-linear regression. Thus, there must be data available to conduct the regression analysis. The simulator setup to get data is described next.

6.3.3 Estimation Formula Using Similarity

The simulator is set up to use RM scheduling, because it is a well-established and often used scheduling algorithm [25]. In the simulator there is one data item d_i that depends on a set of base items, b_j, and it is used by a set of periodic tasks using on-demand updating using similarity. The execution of an on-demand update is performed under disabled interrupts meaning that the execution cannot be preempted. Each data validity bound $\delta(b_j)$ is randomly selected by drawing a number from $U(0, 1)$. Each base item is updated by a dedicated periodic task that runs with period time equal to the shortest possible period time of the tasks using d_i, which is 20 time units in this simulator setup. Each update task assigns a new value to its corresponding base item from $U(0, 1)$, which means that the data validity bound $\delta(b_j)$ is equal to $P_{update}(b_j)$, i.e., it denotes the probability that b_j affects the value of d_i.

The estimation of the mean workload of the update of d_i is the following non-linear regression:

$$MTBI = (1 + (P_{update}(A))MTBI(\mathbf{P}) \tag{6.14}$$

where A is the event that all base items are similar, i.e., $P_{update}(A) = \prod_{\forall b_j} P_{update}(b_j)$, and $MTBI(\mathbf{P})$ is derived by using Equation (6.7). It is shown below in this section that this estimation has an R^2-value in the interval [0.43,0.95]. An R^2-value is always in the interval [0,1] and a value close to 1 indicates the non-linear regression gives values close to simulated values, i.e., closer to 1 the better is the estimation [132].

Figure 6.5 shows the ratio 'measured $MTBI$' over $MTBI$, where measured $MTBI$ is derived from simulation data for different configurations shown in Table 6.1. The minimum ratio is 0.93 meaning that the mean interarrival time of an update is at maximum 1/0.93=1.07 too large. Hence, a worst scenario is that the CPU utilization of the workload of one update is underestimated by 7%.

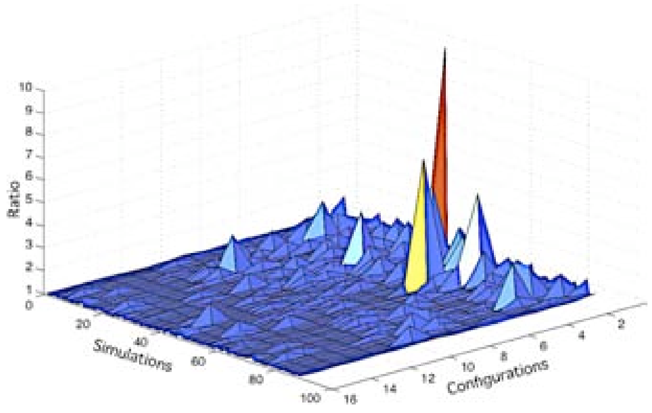


Figure 6.5: Ratio of estimated MTBI of on-demand updates using similarity.

The reason for the spikes in Figure 6.5 is that the estimate uses mean values to estimate the mean interarrival time. Configuration 1 has the following parameters in its largest spike: $\delta(b_0) = 0.98$, $\delta(b_1) = 0.96$, $period(\tau_0^{value}) = 96$ and $period(\tau_1^{value}) = 849$. The estimate given by $MTBI(\mathbf{P})$ is 166.0 but the true mean interarrival time is 1649.0 because the large values of $\delta(b_0)$ and $\delta(b_1)$ make most of the invocations of the task with period time 96 not trigger the update. Thus, the mean interarrival time depends instead on the task with period time 849. This behavior cannot be captured in the estimate.

Configuration 1 has the most spikes (see Figure 6.5) and an evaluation of Equation (6.14) using non-linear regression in SPSS [118] yields an R^2-value of 0.59 without the greatest spike and 0.43 with the greatest spike. This indicates that the estimate is not very accurate for this configuration. However, the spikes give underestimates of the mean interarrival time and, thus, an overestimate of the workload. The R^2-value of configuration 12 is 0.95. Hence, the goodness of the estimate increases with the number of tasks using a data value. This fact can also be seen in Figure 6.5 as the spikes become less frequent and lower as the configuration number increases.

Table 6.1: Configurations of simulations of on-demand updates using similarity.

Configuration	
1	2 base items 2 derived items
2	2 base items 3 derived items
3	2 base items 4 derived items
4	2 base items 5 derived items
5	2 base items 6 derived items
6	3 base items 2 derived items
7	3 base items 3 derived items
8	3 base items 4 derived items
9	3 base items 5 derived items
10	3 base items 6 derived items
11	4 base items 2 derived items
12	4 base items 3 derived items
13	4 base items 4 derived items
14	4 base items 5 derived items
15	4 base items 6 derived items

6.4 Evaluations using AVI

This section evaluates the performance of using on-demand updates or dedicated tasks as well as accuracy of the formula $MTBI(avi(b_i), \mathbf{P})$ presented in Section 6.3.1.

The evaluations show that:

- In a setting where systems are overloaded according to the baseline, which is Equation (6.9), i.e., it is assumed every on-demand update is always executed, we note that as the AVI increases the fraction of systems being unschedulable decreases. This is an expected behavior since the number of times on-demand updates need to execute decreases with increasing AVI since the stored data value lives a longer time.

- Over a varied number of tasks and varied AVI of a data item the workload imposed by updates of on-demand updating is always less than that of DS.

The remainder of this section is outlined as follows. Section 6.4.1 presents the simulator setup, Section 6.4.2 presents evaluations of workload of updates, and Section 6.4.3 gives a test of the accuracy of estimations in a practical setting.

6.4.1 Simulator Setup

The simulations are performed in a discrete-event simulator written in Java. The simulated system consists of a set of base items and a set of tasks that calculate derived items. Each base item is used by a specified number of tasks. The period times of the tasks are randomly chosen from a uniform distribution. The execution times of the tasks consist of two parts (i) on-demand updates and (ii) the computation. The execution time of an on-demand update is 1 time unit if it executes, otherwise it is 0. The computation always executes and its execution time is randomly chosen at simulation start-up time.

6.4.2 Performance Evaluations of Workload

In the first setting, the system has one base item and the number of tasks and the AVI of the base item is varied. The AVI is determined by taking a fraction of the lowest period time in the system. The period times are integers and chosen randomly from $U(50, 1000)$ and the execution times of the computations are set to 1. The scheduling algorithm used is RM. We are now interested in measuring the workload imposed by on-demand updates which is done by taking the worst-case execution time of the update divided by the mean interarrival time of its execution. The workload of the on-demand update is compared to the workload imposed if DS were used. In the case of one data item, DS sets a dedicated updating task's period time to the data item's AVI minus one (see Figure 6.3 for the pseudo-code of DS). Figure 6.6 shows the ratio (workload imposed by DS)/(workload imposed by OD), i.e., the z-axis shows how many times bigger the workload imposed of updates by DS is compared to OD. We make the following observations:

- The number of tasks influences the ratio in that it decreases as the number of tasks increases. The reason is that the higher the number of tasks the more likely it is that a task starts to execute at $avi(b_i)$ time units since b_i was last updated, which gives an imposed workload close to $wcet(b_i)/avi(b_i)$.
- The ratio is high, above 2, for small number of tasks. This is expected as described above. If the number of tasks is low it is very likely that the time between updates is larger than $avi(b_i)$ since there is no task starting close to $avi(b_i)$ time units since the last update. With this reasoning OD can be arbitrarily better than DS. If the system consists of one task and one data item and AVI is 10, then DS sets the period time of a dedicated updating task to 9. Thus, the workload of the update is WCET/9. However, using OD, assuming the period time of the task is 1000, the workload of the update is WCET/1000.
- As the AVI increases the likelihood that a task starts to execute after $avi(b_i)$ time units increases even for a small number of tasks. Thus, the ratio decreases as the AVI increases even for a small number of tasks. This can be seen in Figure 6.6.

Figure 6.8 shows the workload when each task has 10 on-demand updates, i.e., there are 10 base items in the system in this setting. The same parameters are used as above. If a deadline is missed, then that system's parameters are

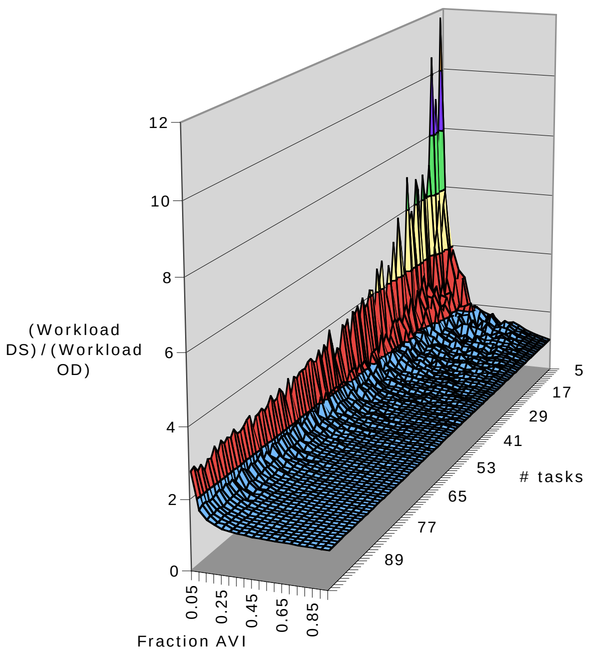


Figure 6.6: Comparison of imposed workload of updates of OD and DS.

chosen again and the system is re-simulated. As we can see, the general behavior is as observed in Figure 6.6. However, for some simulated systems—the AVI is set to a low fraction of the shortest period time in the system and low number of tasks—the ratio is low but always above 1. The reason is that the phasing of the execution of the on-demand updates is such that few of them can be skipped. We observed this behavior only on 12% of the systems where AVI is set to 0.05 of the shortest period time in the system. This constitutes 0.6% of all the simulated systems.

Figure 6.7 shows the minimal observed ratios for each AVI in Figure 6.6. We see that the ratio is always above 1 which suggests that the workload imposed by updating is lower for on-demand updating compared to DS.

In summary, we observe that OD imposes less workload than DS. Further, there exists settings when many on-demand updates are used in tasks that give unexpected high workload of updates. However, the workload is lower than DS, because the period times assigned by DS are lower than the AVIs (see line 6 in Figure 6.3 where the period time is reduced; the period time is the AVI of the data item being updated by the dedicated task whose settings are determined by the algorithm) but using on-demand updating the time between updates cannot be lower than the AVI. Thus, the workload algorithm for DS (Figure 6.3) could

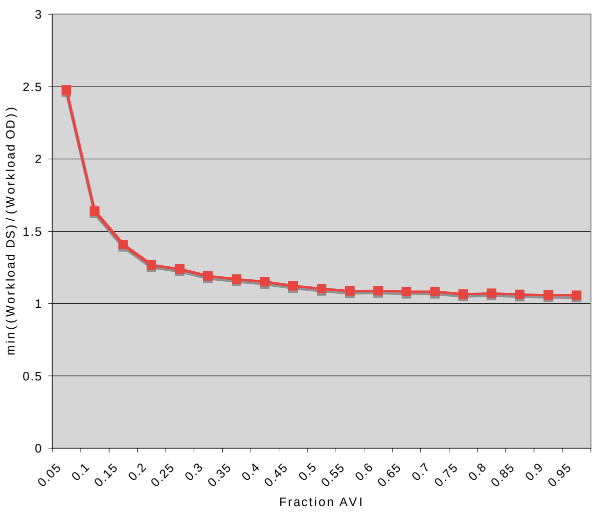


Figure 6.7: The minimal z-axis value for each AVI.

be used to describe an upper bound on the workload for OD. In the next section we evaluate the accuracy of the analytical formula and focus on tasks using one on-demand update.

6.4.3 Performance Evaluations of Estimations

In this experiment, we evaluate the accuracy of Equation (6.11) as used in the schedulability test described in Equation (6.13).

Simulations of systems that are overloaded according to Equation (6.9) are executed. These systems constitute our baseline since there exists no other known, to our best knowledge, schedulability test for on-demand updating than to assume every update always executes. The number of tasks are varied. Five tasks read each base item. The AVI of each base item is set to a fraction of the shortest period time in the simulation. The period times are integers and derived from $U(20, 2000)$. The execution time of each update is 1 time unit and the execution time of each tasks' computation is an integer and randomly chosen from $U(1, 15)$. Each setting of number of tasks and AVI is run 100 times with new period times and execution times each run. Each simulation run is started once the random values give a total utilization, according to Equation (6.9), that lies between 1 and 1.04, i.e., the system is lightly overloaded according to the baseline and the scheduling algorithm being used is EDF. The simulator runs for 500000 time units and checks if there were deadline misses.

Figure 6.9 shows the number of systems that the baseline correctly classifies as overloaded. Note that if the baseline were correct 100 systems would always

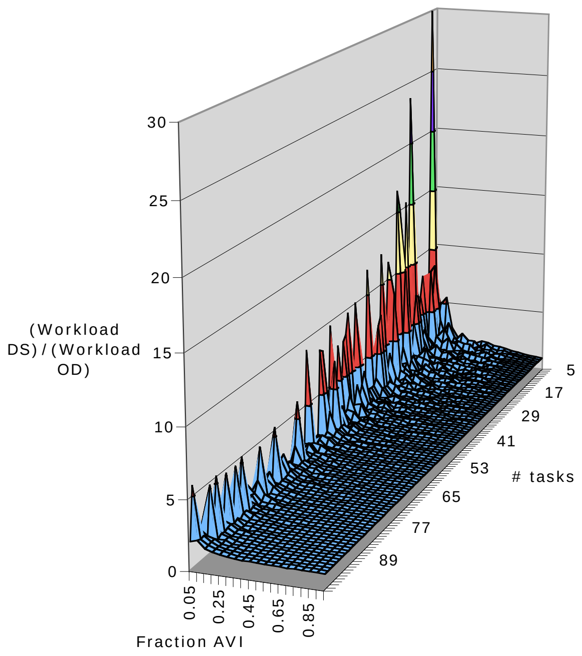


Figure 6.8: Ten base items in the system and each task has on-demand updates for each base item.

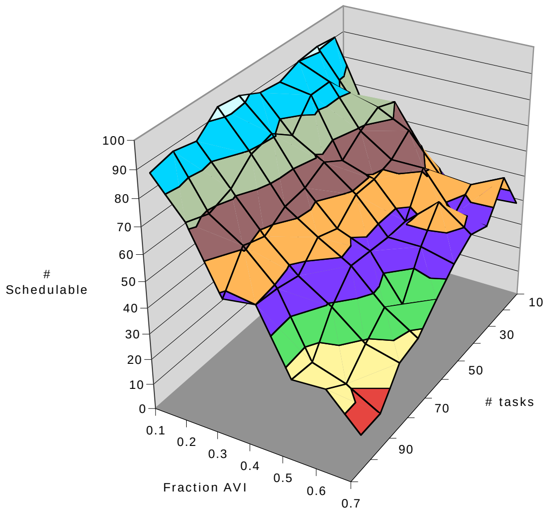


Figure 6.9: Number of systems that misses deadlines.

be unschedulable. We note the following:

- The pessimism in the baseline increases as the AVI increases. This is expected since the longer the AVI the less often on-demand updates need to execute.
- The pessimism also increases as the number of tasks increases. This is also an expected behavior since as the number of tasks increases the number of times on-demand updates is accounted for in the schedulability test increases.

Figure 6.10 shows the minimal number of correct classifications of systems as overloaded or underloaded. All AVI fractions for each number of tasks are considered. ‘MTBI’ is using Equation (6.11) with $q(y, \mathbf{P})$, ‘Timeline’ is an estimation based on drawing a timeline of task arrivals as described by the algorithm in Figure 6.4, and ‘Fast MTBI’ is using Equation (6.11) with $r(z, \mathbf{P})$. We see that in the worst observed cases, ‘MTBI’ gives an accuracy of 91%, ‘Baseline’ of only 14%, ‘Timeline’ of 89%, and ‘Fast MTBI’ of 60%. Figure 6.11 shows the mean number of correctly classified systems and it is 95% or above for ‘MTBI’. ‘Fast MTBI’ has a mean accuracy above 80%, which could be acceptable for some systems. However, the running time of ‘Fast MTBI’ is considerable shorter than ‘MTBI’ and ‘Timeline’ as shown in Section 6.3. The mean classification accuracy of the baseline is 57%.

In the evaluations presented in this chapter, all execution times and period times are integers and on-demand updates execute for one time unit if the

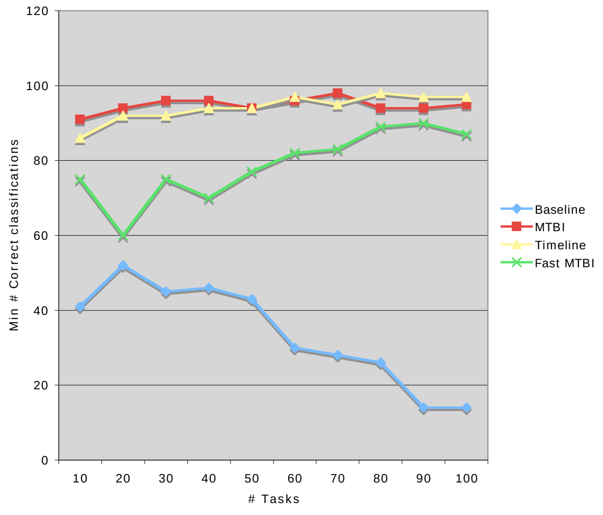


Figure 6.10: Minimum number of systems in Figure 6.9 that misses deadlines. All AVI fractions are considered for each number of tasks.

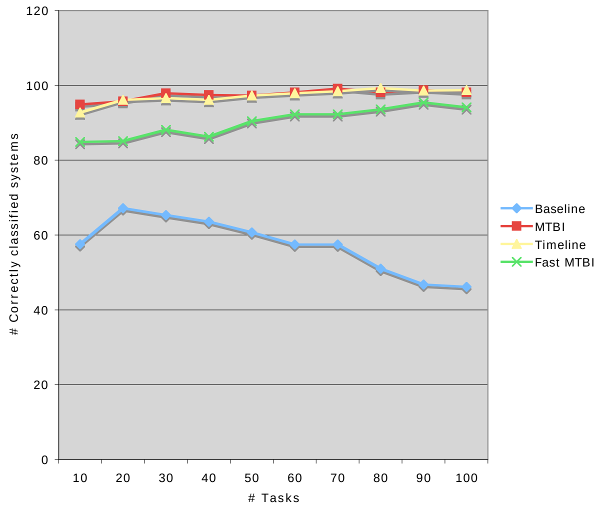


Figure 6.11: Mean number of correctly classified systems.

condition is fulfilled, otherwise they execute for zero time units. Thus, on-demand updates are never preempted. Preemptions of the updates can happen in real-life applications and could be resolved with, e.g., Priority Ceiling Protocol or turning off interrupts. If execution times of updates are short, which they usually are in real-time embedded systems, it is likely that an update is never preempted. Thus, Equation (6.13) resembles, to a high degree, real-life situations.

6.5 Wrap-Up

It has been recognized that data items need to be up-to-date when data values are used in applications [108], e.g., control of an external environment or decision making. In real-time systems it is particularly important to perform timely updates of data items, because the data values must be used before a deadline. The workload of updating data items should be as low as possible, because then the system can use as many data items as possible. It is known that updating data items on-demand may utilize the CPU better compared to updating data items periodically [7, 55, 56]. Also, to the best of our knowledge, there are no analytical formulae to express an exact or estimated workload imposed by on-demand updates. In this chapter we presented analytical formulae that estimates the mean interarrival time of on-demand updates both when data freshness is measured in the time domain and in the value domain. In addition to this our results are:

- Performance evaluations in this chapter show that on-demand updating imposes significantly less workload compared to DS, which is the, to date, algorithm reducing workload imposed by dedicated updating tasks the most [134, 135]. This finding suggests that on-demand updating of data items is a strategy suitable for resource constrained embedded systems, which has also been noticed in Chapter 4. However, using dedicated tasks it is possible to exactly analyze the workload imposed by updates. This is important in hard real-time systems. Using on-demand updates, we have shown in this chapter that it is possible to construct an analytical formula that can estimate the workload. The properties of the formula is given below.

- The estimates' properties are:
  - Fast to execute. It takes 0.016 ms to estimate mean interarrival time for each data item (1.15 ms using the timeline approach, the speed increase is 71.9 times). This estimate's accuracy is 60% in its worst observed case and above 80% in the mean case, which may be usable for some systems. Using a probability density function that is more accurate, but takes longer time to create gives an accuracy of 91% in the worst-case and above 95% in the mean case. The running time is 0.72 ms.

- Good accuracy in the case of one on-demand update in each task. As summarized above, the accuracy is above 91%, which means that at least 91 systems (overloaded according to the baseline which assumes all on-demand updates always execute) are correctly classified as normal load or overloaded. The baseline's mean performance is 57%.
- Good accuracy in the case of several on-demand updates in each task, but there can be situations were the phasing of tasks make fewer on-demand update skips possible than our proposed formula proposes. We observed 0.6% of all simulated systems to be of this kind. However, the algorithm that calculates workload imposed by updates when using DS can be used as an upper bound of workload imposed by on-demand updates. This algorithm introduces less pessimism than the baseline.
- The analytical estimate in the case of measuring data freshness in the value domain underestimates the workload by 7% in the worst found case.

CHAPTER 7

Overload Control

Chapter 4 was devoted to data freshness and data consistency functionality in a database aimed at being used in embedded real-time systems. In Chapter 6, the performance results in Chapter 4 were confirmed in a specialized setting that simplified comparisons of on-demand updating to well-established algorithms for updating using dedicated tasks. However, in Chapter 3, we noted that the problem of optimally choosing which data items to update in order to achieve up-to-date values is NP-hard in the strong sense. Also, the algorithms in Chapter 4 are greedy, meaning that to reduce running time and space requirements they traverse the data relationships, i.e., G, once trying to obtain a schedule fulfilling:

- data freshness,
- time constraints, and
- data relationships.

In addition to this, the algorithms described in Chapter 4 are, as discussed, either consistency- or throughput-centric. At a transient overload, the algorithms in Chapter 4 either miss the time constraints, i.e., deadlines, or cannot guarantee up-to-date data items. In this chapter, we study systems where data items can be divided into, with respect to calculations, *required* data items that constitute the most important data items required to be up-to-date in order to calculate a usable result, and *not required* data items that are less important (they are allowed to be stale). An algorithm is developed that can meet deadlines and schedule updates such that data freshness, time constraints, and data relationships are fulfilled. This developed algorithm addresses requirement R4, degrade performance in the case of a transient overload. Degraded performance means, in the context of data items needed to be up-to-date, that the ‘up-to-dateness’ of the data items is lowered.

The outline of this chapter is as follows. Section 7.1 gives an introduction to overload control and the problem of guaranteeing data items are up-to-date. Section 7.2 introduces an extension to the data model in Section 3.2. Section 7.3 describes the on-demand updating algorithm **A**dmission **C**ontrol **U**pdating **A**lgorithm (ACUA). Section 7.4 describes how admission control can be achieved using **A**dmission **C**ontrol **U**pdating **A**lgorithm. Section 7.5 introduces an algorithm that can estimate the total CPU utilization of a system. Section 7.6 contains our performance evaluations of ACUA. Finally, Section 7.7 concludes this chapter.

7.1 Introduction

The common way to check whether a system is overloaded or may become overloaded when admitting new jobs of tasks is to perform a feasibility test (see Section 2.1). Such an approach is used in the algorithms presented in Chapter 4, e.g., ODDFT that checks (line 3 in Figure B.1) whether an update can be accommodated within the deadline. The feasibility test compares CPU utilization of executing tasks, plus the ones that are in question of being admitted, to a bound, and if the test fails the system may be overloaded. The worst-case execution time (WCET) of the jobs is used to calculate their CPU utilization. The WCET may be pessimistic meaning that the real execution time of a job is less than its WCET. In order to calculate the CPU utilization the interarrival time of jobs is also used. The interarrival time can also be pessimistic—as is observed in Chapter 6—because the jobs may be conditioned, i.e., a job is invoked but not executed because a condition is false. Hence, in order to get an accurate CPU utilization of a system, interdependencies among tasks and values on data must be considered.

In addition, the workload of a system can also change on-line while the system is running. Two examples of such occurrences in the used engine control software are:

- When a set of tasks in engine control are triggered based on the speed of the engine. Hence, when the speed of the engine changes it affects the workload.
- When tasks have conditioned execution. Hence, the workload can change for systems where all tasks are periodic, because the tasks have if-statements checking values of data items. For instance, the actual frequency of when sensor values change depends on the external environment. The temperature of the cooling water in an engine increases fast when the engine is turned on and heated up, but then the engine reaches its working temperature and the temperature is fluctuating

around the working temperature. This means that the frequency of updates of temperature decreases with time until the working temperature is reached.

From the two examples above, we see that the workload of the engine control changes dynamically. When the driver presses the gas pedal, the workload of the system increases because the speed of the engine increases, but at the same time the workload might decrease because the engine temperature is reaching its working temperature. The total effect of these two events must be taken into consideration to determine whether the system is in a transient state and precautions must be taken.

In summary, we have two related problems with respect to transient overloads in a system. They are:

1. Accurately determining the system workload and decide whether the system is in a transient overload. Our initial work in this area has been reported in Chapter 6.

2. Reacting to an overload and produce (acceptable) results of the workload that can be used to control the system such that no damage to the system occurs.

The first problem is further investigated and addressed in this chapter by introducing the MTBIOfflineAnalysis algorithm that off-line estimates the workload of a system. This algorithm builds upon results presented in Chapter 6. Coarser estimates can be given by another algorithm denoted MTBIAlgorithm, where the running time of MTBIAlgorithm is considerably shorter compared to MTBIOfflineAnalysis. Hence, MTBIAlgorithm might be used on-line to estimate whether the system suffers a transient overload given workload changes. This algorithm is presented and evaluated in Chapter 8. The second problem is addressed in this chapter by the introduction of the **A**dmission **C**ontrol **U**pdating **A**lgorithm (ACUA), which adopts the notion of required data items and not required data items.

7.2 Extended Data and Transaction Model

As we concluded in Chapter 3 many embedded systems may become overloaded, and the software must be designed to cope with it, e.g., at high revolutions per minute of an engine the engine control software cannot perform all calculations. We have noticed that (in Chapter 2), for some calculations, only a subset of the data items used in a calculation are compulsory to derive a usable result (we denote such data items as *required* and other data items as *not required*). For instance, in the engine control software, the calculation of fuel amount to inject into a cylinder consists of several variables, e.g., temperature compensation factor, and a sufficiently good result can be achieved only by calculating a result based on a few of these compensation factors. Thus, at high revolutions per minute, only a few of the compensation factors are calculated.

Hence, the read set of a data item $R(d_i)$ can be divided into required data items, denoted $RR(d_i) \subseteq R(d_i)$, and not required data items, denoted $NRR(d_i) \subseteq R(d_i)$, $RR(d_i) \cap NRR(d_i) = \emptyset$. A data item that is required with respect to another data item can be distinguished by marking the relationship in the data structure describing G, e.g., with number 2 in the adjacency matrix of G. We assume the value of d_i is correct if at least all data items in $RR(d_i)$ are up-to-date when deriving d_i. We furthermore assume that values of data items in $RR(d_i)$ can be based on only up-to-date required data items. This means that the system still has a correct behavior if all transactions only use up-to-date values on required data items.

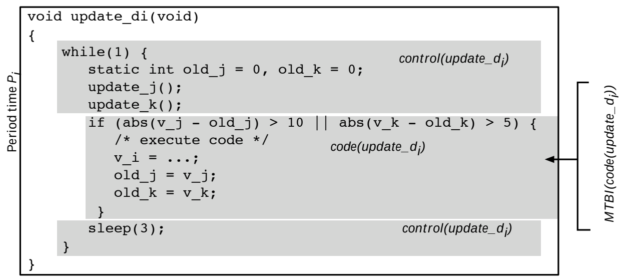
7.2.1 Update Functions

To calculate the CPU utilization of a system whose calculations and their relationships are described in the data dependency graph G and where an updating algorithm with a relevance check is used is difficult because we need to know at which times the updating algorithm must execute scheduled updates. This matter has been discussed in Chapter 6. In this chapter and the following, we use a data dependency graph with arbitrary number of levels instead of the special case discussed in Chapter 6 where the graphs only have two levels. A system is equivalent to a database system with on-demand updates if the following holds.

- Each node in G represents an update function which conceptually looks as in Figure 7.1, which is an extension of Figure 6.2 since the if-statement in $code(update\_d_i)$ may check the values of several derived data items. We see that the part denoted $code(update\_d_i)$ is executed whenever a scheduled triggered update needs to be executed.
- Periodic tasks call update functions of the leaf nodes.
- Periodic tasks call update functions of base item. This corresponds to step 1 of AUS.

In Figure 7.1, $code(update\_d_i)$ is the main functionality of the function and $control(update\_d_i)$ calls other update functions to ensure data items are up-to-date and also a control checking whether the main functionality should be performed. There is a probability associated with $update\_d_i$ changing the value of d_i that we denote $P_{update}(d_i)$. We are interested in finding a way to determine, exactly or as an estimate, the times $code(update\_d_i)$ are used. We see that, for a data item d_i, the invocation times of $code(update\_d_i)$ depend on how often leaf nodes, which are descendants of d_i, are called and how often values of data items in d_i's read set change.

Figure 7.2 shows a system with update functions that call each other. Figure 7.3 shows fractions of timelines containing occurrences of $code(update\_d)$,



```
void update_di(void)
{
   while(1) {
      static int old_j = 0, old_k = 0;
      update_j();
      update_k();
      if (abs(v_j - old_j) > 10 || abs(v_k - old_k) > 5) {
         /* execute code */
         v_i = ...;
         old_j = v_j;
         old_k = v_k;
      }
      sleep(3);
   }
}
```

Figure 7.1: The function $update\_d_i$ that derives data item d_i if data items d_j and d_k have changed more than 10 or 5, respectively.

$d \in \{b_1, b_2, d_k, d_i, d_m, d_n, d_o, d_p, d_q\}$. Timeline 1 (the timeline to the left of the encircled 1) shows the timepoints where sensors are updated. We assume every time any of the sensors is updated, data item d_k is affected by the change. Timeline 3 shows the time instances where $update\_d_k$ is called. Timeline 2 shows which calls of $update\_d_k$ that result in $code(update\_d_k)$ being executed.

In Figure 7.2, the mean interarrival time of calls of $update\_d_k$ and $update\_d_j$ is 70.3 ms (calculated using Equation (6.3) and setting $\mathbf{P}$ to $\{100, 500, 450\}$) because the same periodic calls make the requests of $update\_d_k$ and $update\_d_j$. The mean time between invocations of the main functionality of $update\_d_k$ is 93.8 ms and 150 ms for $update\_d_j$ when taking the frequency of value changes of b_1, b_2, b_3, and b_4 into consideration (using algorithm MTBIOfflineAnalysis, Figure 7.7), which is considerably larger than 70.3 ms. This example shows it is important to consider also the frequency of value changes in on-line CPU utilization calculations otherwise pessimistic results are obtained.

Over the course of the system and given fixed interarrival times on tasks[1] the time durations between calls of a function can be estimated by the mean time between the invocations of the function. The estimated total CPU utilization can then be described as

$$\sum_{\forall d_i} \left(\frac{wcet(control(update\_d_i))}{MTBI(control(update\_d_i))} + \frac{wcet(code(update\_d_i))}{MTBI(code(update\_d_i))} \right), \tag{7.1}$$

where $MTBI$ is the mean time between invocations of $control(update\_d_i)$ and $code(update\_d_i)$, respectively. $MTBI(control(update\_d_i))$ can be calculated by using Equation (6.3) and setting $\mathbf{P}$ to the period times of all leaf nodes being descendants of d_i. However, in this thesis, we assume $\frac{wcet(control(update\_d_i))}{MTBI(control(update\_d_i))}$ in Equation (7.1) is negligible compared to the other term, because $wcet(control(update\_d_i))$ is probably much smaller than $wcet(code(update\_d_i))$ since $control(update\_d_i)$ constitutes one if-statement

[1]Tasks executing leaf nodes and tasks executing periodic base item updates.

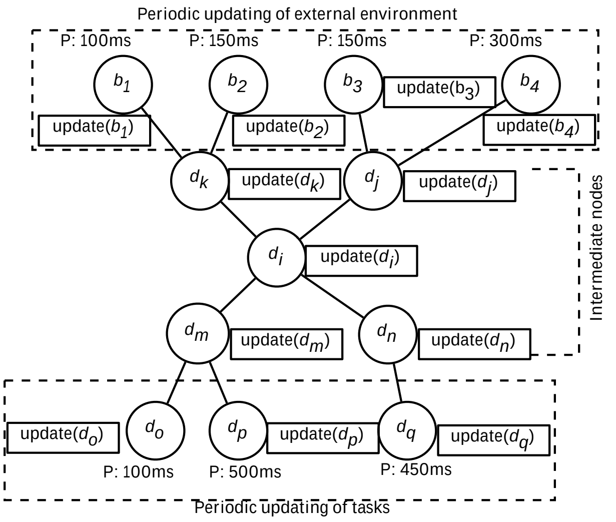


Figure 7.2: Data dependencies.

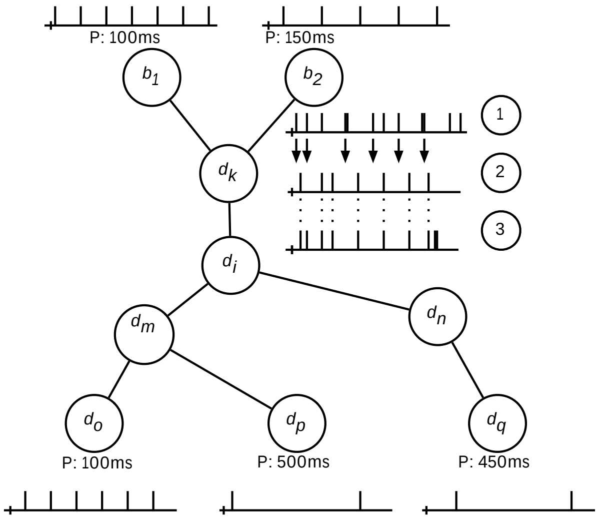


Figure 7.3: An example of invocation times of updates.

whereas $code(update\_d_i)$ constitute more code, and can thus be removed from Equation (7.1). Further, we assume $wcet(code(update\_d_i))$ is given. Hence, we must find a way to calculate $MTBI(code(update\_d_i))$. In this thesis, we describe two methods to calculate the mean time between invocations. One method is presented in Section 7.5 and the other in Section 8.1.3.

7.3 Admission Control Updating Algorithm

In this section we describe the **A**dmission **C**ontrol **U**pdating **A**lgorithm (ACUA) algorithm that decides which data items need to be updated when a transaction starts. The decision is based on markings by the AUS scheme and data relationships. ACUA is a top-bottom algorithm that knows about required and not required data items.

Before describing ACUA we contrast the differences between feasibility tests using admitted jobs and using total CPU utilization.

- Using admitted jobs and the job that is in question of being admitted, an overload is detected when it is about to happen. However, the possibility to react to it is limited. There are four cases.

  1. Do not admit the new job and leave the admitted jobs as they are.
  2. Admit the new job but its execution time is reduced in order to reduce the effects of the overload. The admitted jobs are left intact.
  3. Admit the new job and change it and other admitted jobs such that there is no overload.
  4. Admit the new job without changing it and change other admitted jobs such that there is no overload.

  The first bullet above is easy to implement. The second, third, and fourth bullets are more difficult to implement because data freshness and data relationships must be taken into account. In this chapter we concentrate on the scenario described in the second bullet since it is more straightforward to change jobs that have not yet been admitted compared to changing admitted jobs that may have already been started.

- Computing total CPU utilization also detects an overload. With total CPU utilization we mean the CPU utilization imposed by all tasks and all updates including those that are currently not active. An overload in a CPU utilization based feasibility test represents an arrival pattern of invocations of updates yielding a deadline miss. This arrival pattern happens sometime in the future. Thus, by calculating the total CPU utilization, given interarrival times of tasks and sensor updates, we have a prediction of the behavior of the system and can react to it immediately. Thus, using total CPU utilization there may be a longer time to react before the overload occurs compared to using feasibility testing at each task invocation. Chapter 8 presents an algorithm that can be used to estimate mean time between invocations of updates in a system with a data dependency graph with more than two levels. Using exactly two levels in G was studied in Chapter 6.

ACUA is implemented by traversing G top-bottom in a breadth-first approach. Data structures are used to keep information necessary to put updates in a schedule containing possibly stale data items. By using a top-bottom traversal with a data structure, ACUA can be extended with different functionality, e.g., in addition to schedule stale data items it is possible to calculate probabilities that updates get executed, i.e., in one traversal of G a schedule of updates and the probabilities that they get executed can be generated using ACUA.

ACUA is described in Figure 7.4. The parameter d is the data item a user transaction requests, $ancestors$ is the set of all ancestors sorted by increasing level (see definition 3.2.1), and $allMode$ is true if all data items should be considered for being updated and false if only required data items should be considered.

The set $ancestors$ is generated off-line by depth-first traversal of G from the node representing d, after the depth-first traversal the visited nodes are sorted according to increasing level. Line 7 of ACUA checks whether an immediate child of an ancestor should be considered for being updated. A data item that is required with respect to another data item can be distinguished by marking the edge in G, e.g., with number 2, in the adjacency matrix describing G. The function status(x) (see Figure 7.5) calculates the marking of x based on the inherited markings from ancestors of x (line 8). The inherited markings are traversed down G with the help of function inheritstatus(c,x) (see Figure 7.6). Further, ACUA can consider probabilities. We assume the probability that a scheduled update needs to execute is independent of the probability that other scheduled updates need to execute. The probability that a particular update needs to execute can then be written as follows:

$$(1 - \prod_{\forall x \in PAR(d_i)} (1 - P_{update}(x))) \times P_{update}(d_i), \tag{7.2}$$

where $PAR(d_i)$ is the set of potentially affected read set of d_i (see Section 3.3.1), and $P_{update}(x)$ is the probability that an update of x changes the value of x so much that an update of another data item must execute. We are interested in the probability that any of the members of $PAR(d_i)$ changes since this is the probability that the if-statement in $control(d_i)$ is true. Equation (7.2) uses the complementary event: “none of the scheduled updates changes” by taking

$$\prod_{\forall x \in PAR(d_i)} (1 - P_{update}(x)). \tag{7.3}$$

The complement of “none of the scheduled updates changes” is “any of the

```
ACUA($d$, $ancestors$, $allMode$)
 1: for all $x$ in $ancestors$ do
 2:     status($x$)
 3:     if $x.marked == true$ then
 4:         put an update for $x$ into schedule
 5:     end if
 6:     for all immediate children $c$ of $x$ do
 7:         if ($c$ is required and allMode is false) or (allMode is true) then
 8:             inheritstatus($c$, $x$)
 9:         end if
10:     end for
11: end for
```

Figure 7.4: The ACUA algorithm.

scheduled updates changes", which is calculated by

$$1 - \prod_{\forall x \in PAR(d_i)} (1 - P_{update}(x)). \tag{7.4}$$

In Figure 7.5, line 9 calculates Equation (7.3) and line 18 calculates the complement, i.e., Equation (7.4).

In summary, a marking by AUS is traversed down the graph and updates are scheduled as they are found to be needed (line 4 in Figure 7.4). When ACUA has constructed a schedule of updates as a response to an arrival of a user transaction, DIESIS starts to execute the updates before the UT commences (see Section 4.1.1). The updating scheme AUS is active whenever a data item is written to the database. This means that a data item might be in the schedule but it never becomes marked because an update in an immediate parent never resulted in a stale data item. Thus, only updates for the data items that are marked by AUS are started by DIESIS. In this way the workload is automatically adapted to how much data items change in the external environment. The experimental results presented in Section 7.6 confirm this.

Computational complexity

Computational complexity of ACUA is polynomial in the number of ancestors of a data item, i.e., $O(|N|^2)$, where N is the set of nodes of G, because ACUA loops through all ancestors of a data item and for each ancestor the algorithm loops through all its immediate children. In the worst-case, all nodes but one (the node itself) are ancestors. The number of immediate children is also in the worst-case in the order of number of nodes in the data dependency graph. Thus, the computational complexity is $O(|N|^2)$. However, in real-life situations, the number of ancestors and number of immediate children may be less than $|N|$ and the running time lower than $O(|N|^2)$.

```
status($x$)
1: if $x$ is marked then
2:     $x.marked = true$
3: else
4:     $x.marked = false$
5: end if
6: $prob = 1$
7: for all $p$ in $x.parents$ do
8:     $x.marked = x.marked \vee p.marked$
9:     $prob = prob * (1 - p.prob)$
10: end for
11: if $|x.parents| = 0$ then
12:     if $x.marked = true$ then
13:         $x.prob =$ probability that $x$ gets updates
14:     else
15:         $x.prob = 0$
16:     end if
17: else
18:     $x.prob = 1 - prob$
19: end if
```

Figure 7.5: The status help function.

```
inheritstatus($c$,$x$)
1: $c.parents[c.parentnum].marked = x.marked$
2: $c.parents[c.parentnum].prob = x.prob$
3: $c.parentnum + +$
```

Figure 7.6: The inheritstatus help function.

7.4 Admission Control using ACUA

The load represented by the admitted updates can be expressed as $U = \sum_{\forall \tau_i \in ActiveUT} \frac{wcet(\tau_i)}{period(\tau_i)}$, where $ActiveUT$ is the set of active user transactions, $wcet(\tau_i)$ is the sum of execution times of updates scheduled to need an update and the execution time includes the execution time of UT τ_i. $period(\tau_i)$ is the period time of UT τ_i. In order to successfully execute all UTs, U always needs to be below a specific bound. In the work on ACUA we choose to use RBound [87] since it gives a bound tighter than RMA. RBound says that if

$$\sum_{\tau_i \in ActiveUT} \frac{wcet(\tau_i)}{period(\tau_i)} \leq (m-1)(r^{1/(m-1)} - 1) + \frac{2}{r} - 1, \tag{7.5}$$

where m is the number of active UTs and r is the ratio

$$period(smallest)^{\log_2 \lfloor \frac{period(smallest)}{period(highest)} \rfloor} / period(highest),$$

where $period(smallest)$ is the smallest period time of active UTs and $period(highest)$ is the highest [87]. As with the well-known Liu and Layland bound [91], RBound is sufficient but not necessary.

Admission control of updates in DIESIS using RBound is done as follows. When a UT arrives to DIESIS, ACUA using $allMode$ set to true is used. This means that a schedule is generated where all data items are considered for being updated. Now, if Equation (7.5) is false, i.e., the system may be overloaded, then a new schedule using ACUA with $allMode$ set to false is generated. The execution time of a UT is estimated to the sum of execution times in the schedule.

In practice only one execution of ACUA is needed, because the not required data items can be marked, and removed from the schedule if Equation (7.5) is false. Using ACUA to schedule updates and the feasibility test RBound is denoted ACUA-RBound.

7.5 Analyzing CPU Utilization

This section describes an algorithm denoted MTBIOfflineAnalysis, described in Figure 7.7, which can be used to analyze the workload of a system using the data model described in Section 7.2. MTBIOfflineAnalysis is used in the section Performance Evaluations (Section 7.6) to analyze the workload of the transient and steady states of the system that is being used. In Chapter 8 we describe an extension of MTBIOfflineAnalysis that can be used on-line to estimate the CPU utilization of the system when the workload changes.

We now give a detailed example to illustrate the complexity of estimating the total CPU utilization of a system allowing arbitrary number of levels of the data dependency graph. Let us concentrate on the function calls presented in Figure 7.2 where every node, e.g., d_p represents both a data item and a function, $update\_d_p$, that is updating the value of

d_p. When $update\_d_p$ is called it results in calling the functions in the set $f_p = \{update\_d_k, update\_d_j, update\_d_i, update\_d_m\}$ in order to maintain freshness of data items (the calls are made in $control(update\_d_p)$, see Figure 7.1). The main functionality of one of these functions is executed if some conditions are true as illustrated in Figure 7.1. We say a function gets executed if its main functionality is executed. Assume a task calling $update\_d_p$ has an interarrival time of 500 ms and assume $update\_d_j \in f_p$ is executed every second time $update\_d_p$ is called due to value changes since the if-statement in $update\_d_j$ checks values of b_3 and b_4. The interarrival time of $update\_d_j$ is then 1000 ms. However, the functions in set f_p are also called when $update\_d_m$ is called. Hence, the interarrival time of $update\_d_j$ now depends on the interarrival times of $update\_d_p$ and $update\_d_m$ and with the frequencies sensors b_3 and b_4 are updated and how much they change every time they get updated.

Since DIESIS executes only the updates of data items that need to be updated, there is a need to off-line determine the mean time between invocations of updates of data items since this time can be used to calculate the CPU utilization by taking $\frac{wcet(d_i)}{MTBI(d_i)}$. Here $wcet(d_i)$ is the worst-case execution time of the update of d_i and $MTBI(d_i)$ is the mean time between invocations of the update of d_i. From Chapter 6 we know that an accurate estimate of the mean interarrival time of executions of on-demand updates is to draw a timeline of release times of tasks that may execute an update on-demand and then forming the MTBI of the tasks that is most probable to execute the update. MTBIOfflineAnalysis is an extension of the algorithm presented in Figure 6.4.

As mentioned in Section 7.2.1, there are two things that determine the mean time between invocations of an update of d_i: (i) the period times of UTs, and (ii) the probability that a recalculation of a member of the read set $R(d_i)$ results in a change in the value of d_i. See Figure 7.3 for an example. In this chapter, timelines have a length of 400000 time units which give accurate values on mean time between invocations. In order to get an accurate mean time between invocations, the length of the timelines needs to be equal to the hyperperiod[2] of period times of the read set and tasks. To shorten the execution time of MTBIOfflineAnalysis, the length of timelines can be fixed, but the length must be order of magnitudes longer than the period times of elements in the read set and of tasks in order to capture the arrival pattern of execution of updates. Line 10 determines whether an occurrence of an update of a read set member will make the value of d_i stale. The CPU utilization can easily be determined by calculating timeline $T3$ for each data item and then derive the mean time between invocations on that timeline followed by calculating $wcet(d_i)/MTBI(d_i)$. The total CPU utilization is the sum of $wcet(d_i)/MTBI(d_i)$ for each data item d_i. If the CPU utilization is below a threshold given by the schedulability test given with the deployed scheduling algorithm, then there should be no deadline misses.

[2]The hyperperiod of a set of period times is m such that $m = n_0 period(\tau_0) = n_1 period(\tau_1) \cdots$, all n_i are integers.

```
1: Use depth-first traversal of $G$ bottom-up and assign to each derived data
     item the period times of descendant leaf nodes.
2: Draw a timeline $T3$ for each base item with each occurrence of an update of
     it
3: for all levels of $G$ starting with level 2 do
4:     for all data items $d_i$ in the level do
5:         Merge all $T3$ timelines of $x \in R(d_i)$ and call the timeline $T1$
6:         Create $T2$ with possible updates of $d_i$, i.e., when derivatives of $d_i$ are
             called.
7:         $p = 0$
8:         for all Occurrences $ot2_i$ in $T2$ do
9:             for all Occurrences $ot1_i$ in $T1$ in the interval $]ot2_i, ot2_{i+1}]$ do
10:                if $r \in U(0,1) \leq p$ then
11:                    put $ot2_{i+1}$ into a timeline $T3$
12:                    $p = 0$
13:                    break
14:                else
15:                    increase $p$ with probability that an update of a read set member
                         affects the value of $d_i$.
16:                end if
17:            end for
18:        end for
19:    end for
20: end for
```

Figure 7.7: MTBIOfflineAnalysis algorithm.

Computational Complexity

The computational complexity of line 1 of MTBIOfflineAnalysis is $O(|N| + |E|)$. Line 2 has complexity $O(L)$ since there are $L/period$ number of task starts on timeline of length L. The number of times the for-loop on line 3 is called is in the worst case $O(|N|)$, and the for-loop on line 4 is called $O(|N|)$ number of times. Lines 5 and 6 have complexity $O(L)$ as described above. The complexity of the for-loop on line 8 is $O(L)$ and the complexity of line 9 is $O(period)$ because the number of occurrences in the given interval depends on the period times of tasks. Thus the computational complexity is $O(|N| \times |N| \times L \times period)$.

7.6 Performance Evaluations

This section contains a short description of the other algorithms that are used in the evaluations (Section 7.6.1), a description of the simulator setup (Section 7.6.2), and experiments with results (Section 7.6.3).

7.6.1 Evaluated Algorithms

In the evaluations the deadline miss ratio is used as a performance metric. We compare AUS, ACUA using all-mode (denoted ACUA-All), and ACUA-RBound to OD (see Section 4.4) in three different settings: OD-All, OD-(m, k), and OD-Skipover ((m, k)-firm scheduling and Skip-over scheduling were described in Section 2.1.1). When a UT arrives to the system, OD traverses G bottom-up from the data items written by the UT and visited data items are updated if they are stale according to AVIs (definition 2.2.1). Thus, using OD, data freshness is measured in the time domain.

The algorithm OD-(m, k) executes updates of data items according to OD and the priorities of UTs are set according to the (m, k) algorithm where $m = 1$ and $k = 3$, thus, four distances are possible (see Section 2.1.1 for details). The dynamic priorities of (m, k) are implemented in μC/OS-II by priority switches. Five priorities are set aside for each distance. When a UT starts its distance is calculated and its priority is switched to the first free priority within the set for that distance. OD-All uses OD from Section 4.4 and the UTs' priorities are fixed. OD-Skipover uses OD to update data items and the skip-over algorithm is red tasks only where, in the experiments in this section, every third instance of UTs are skipped. ACUA-RBound is the algorithm described in Section 7.4.

The evaluations presented in the remainder of this chapter show that using ACUA-RBound a transient overload is suppressed immediately. OD-(m, k) and OD-Skipover cannot reduce the overload to the same extent as ACUA-RBound. Thus, constructing the contents of transactions dynamically taking workload, data freshness, and data relationships into consideration is a good approach to overload handling.

7.6.2 Simulator Setup

The simulations are executed using DIESIS on μC/OS-II [84]. We choose to use DIESIS with ACUA on μC/OS-II since that enables us to execute simulations on a real-life system. The setup of the system should adhere to real-life systems and we choose to extend the simulator setup that was used in Chapter 5 with the notion of required and not required data items. The simulator setup is as follows in this chapter (the simulator setup used in Chapter 5 is described in Section C.1.1).

- 45 base items and 105 derived items were used.
- Tasks have specialized functionality so data items tend to seldom be shared between tasks, thus, the data dependency graph G is broad (in contrast to deep). The graph is constructed by setting the following parameters: cardinality of the read set, $|R(d_i)|$, ratio of $R(d_i)$ being base items, and ratio being derived items with immediate parents consisting of only base items. The cardinality of $R(d_i)$ is set randomly for each d_i in the interval 1–8, and 30% of these are base items, 60% are derived items with a read set consisting of only base items, and the remaining 10% are other derived items. These figures are rounded to nearest integer. The required data items are chosen by iteratively going through every member of $R(d_i)$ and set the member to be required with the probability $1/|R(d_i)|$. The iteration continues as long as $|RR(d_i)| = 0$. The number of derived items with only base item parents is set to 30% of the total number of derived items.
- To model changing data items, every write operation is taking a value from the distribution U(0,350) and divides it with a variable, $sensorspeed$, and then adds the value to the previous most recent version. To get a load of the system at an arrival rate of 20 UTs per second that shows the differences of the algorithms the following parameters are set as follows:
  - the data validity intervals are set to 900 for all data items, i.e., $\delta_{d_i} = 900$,
  - the absolute validity intervals are set to 500 because with a mean change of 175 and a period time of 100 ms on base items a base item's value is, on average, valid for at least 500 ms.

  The probability that an update must execute is 175/900=0.2 where 175 is the mean value change. Table 7.1 shows the CPU utilization, calculated using MTBIOfflineAnalysis in Section 7.5, where $sensorspeed$ is 1 and 10. We see that the system should be overloaded when ACUA-All is used in a transient state ($sensorspeed = 1$, i.e., sensors change much) and not overloaded when required-mode is used. In a steady state, i.e., $sensorspeed = 10$, the system is not overloaded.

The concurrency control algorithm that is used is High-Priority Two-Phase Locking (HP2PL).

Table 7.1: CPU utilizations.

Mode	$sensorspeed$	p base	p derived	U
all-mode	1	0.20	0.20	1.38
all-mode	10	0.02	0.20	0.72
required-mode	1	0.20	0.20	0.28

Table 7.2: Max mean deadline miss ratio (MMDMR) for transient and steady states.

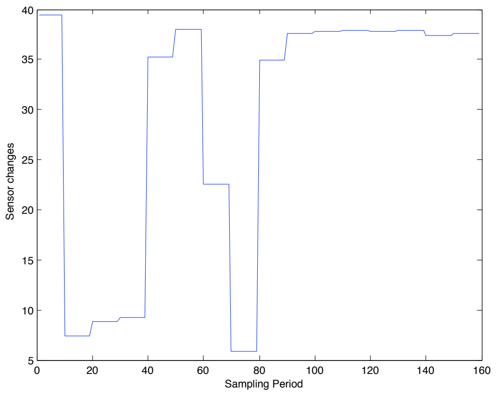
Algorithm	MMDMR transient; steady
OD-All	0.146;0.146
OD-(m, k)	0.178;0.146
OD-Skipover	0.090;0.080
ACUA-All	0.109;0.02
ACUA-RBound	0.029;0.002

7.6.3 Experiments

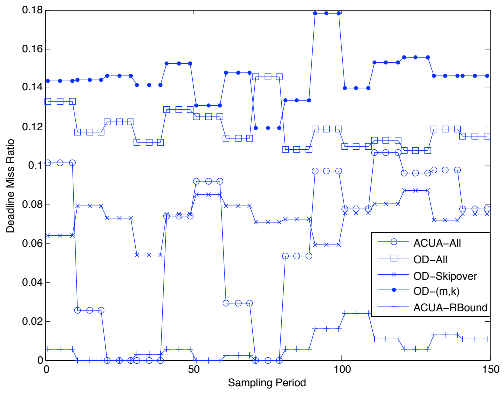
Figure 7.8(b) shows the performance of the algorithms where sampling periods are 500 ms. We show the mean deadline miss ratio for the intervals where $sensorspeed$ is set to the same value, which is periods of 5 s, i.e., 10 sampling periods. The max mean deadline miss ratio is shown in Table 7.2. The sensors change as showed in Figure 7.8(a). The deadline miss ratio of OD-All, OD-(m, k), and OD-Skipover is unaffected of the sensor changes which is expected because using AVIs for data freshness makes updating unaware of values of data items. The miss ratio drops using ACUA-All when the number of sensor changes per time unit is small as in the interval 15–40 sampling periods and 70–80 sampling periods. This is also expected since the entry $sensorspeed = 10$ in Table 7.1 says the system should be not overloaded.

The data consistency achieved by skip-over scheduling is worse than the consistency achieved by ACUA-All, ACUA-RBound, OD-All, and OD-(m, k), because using skip-over scheduling every third instance of a task never updates any data items. For ACUA-All and ACUA-RBound data items are always updated such that transactions use up-to-date values on required data items. OD-All and OD-(m, k) also use up-to-date values on data items.

Using skip-over scheduling improves the performance compared to OD-All. However, ACUA-All has similar performance as OD-Skipover. Thus, ACUA-All has similar deadline miss ratio compared to OD-Skipover and the data consistency is higher. OD-(m, k) does not perform, overall, better than OD-All and that is because the task having the highest priority according to RM gets a dynamic priority that might be lower than other running tasks with the same distance. Thus, the task with shortest period time misses more deadlines but other tasks meet more deadlines, and this is for instance showed in Figure 7.9 where the deadline miss ratio for tasks with second highest priority is lower



(a) Sensor changes per time unit



(b) On-demand updating algorithms

Figure 7.8: Performance of overload handling algorithms.

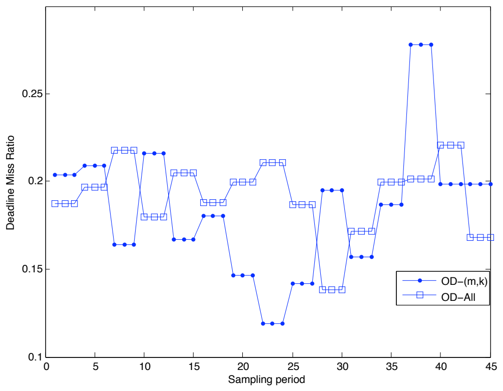


Figure 7.9: Deadline miss ratio for task with second highest priority.

for OD-(m, k) compared to OD-All. However, the performance of OD-(m, k) cannot be better than OD-Skipover because task instances are skipped using OD-Skipover, which they are not in OD-(m, k).

Skip-over gave the best effects on deadline miss ratio using the OD algorithm. Figure 7.10 shows the performance of ACUA-All using skip-over to skip every third task instance. The deadline miss ratio drops by introducing skip-over, but it is not affected much by the skips. Hence, to reduce workload in an overloaded system other means must be used than skipping invocations of tasks. The ACUA algorithm can generate schedules containing data items that might need to be updated, which can be seen in Figure 7.8(b). To improve the performance of ACUA-All, the schedules' lengths must be varied depending on the workload. However, data relationships must still be considered. One way to shorten the length of a schedule is to use the required-mode of ACUA. Switching to required-mode when the RBound feasibility test fails gives the performance denoted ACUA-RBound in Figure 7.8(b). As can be seen ACUA-RBound decreases the deadline miss ratio better than any of the other algorithms and suppresses the deadline miss ratio when the system goes from a steady to a transient state, e.g., sampling period number 80, where number of sensor changes from low to high. The mean deadline miss ratio is at maximum 0.029 in the interval 100 to 110 where sensors change much, i.e., the system is in a transient state, compared to OD-Skipover that has its maximum mean at 0.09. Using ACUA-RBound, the deadline miss ratio can be above zero because if the utilization bound in Equation (7.5) (Section 7.4) is false, required-mode of ACUA is used, but Equation (7.5) can still be false due to admitted UTs that have used all-mode. One way to resolve this is to reschedule updates of active UTs.

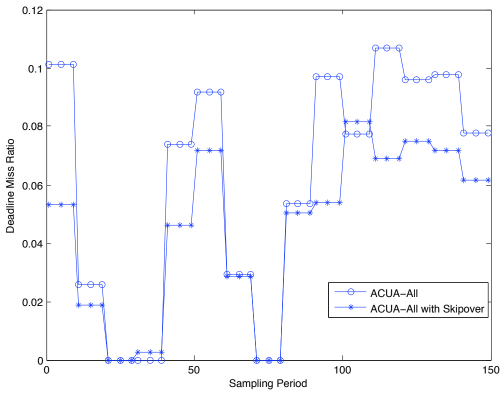


Figure 7.10: ACUA-All using skip-over.

7.7 Wrap-Up

This chapter has described an approach to handle overload situations when using on-demand updating. It has been recognized that for embedded systems it is possible to divide data items used in a calculation as either *required* to be up-to-date or *not required* to be up-to-date. The main idea is to focus CPU resources on important data items first. Performance results show that this approach yields better overload handling compared to existing approaches (Skipover and (m, k) scheduling).

Further, the MTBIOfflineAnalysis algorithm to estimate CPU utilization in a system where G can have arbitrary data relationships (in contrast to Chapter 6 where derived data items only are derived from base items) has been introduced. However, in the calculation of the CPU utilization by MTBIOfflineAnalysis, there are uncertainties that might affect the accuracy of the CPU estimation. They are:

- MTBIOfflineAnalysis considers mean time between invocations of updates. This delineates the typical behavior of the system. However, worst-case arrival patterns are not covered by the typical behavior, which means that the maximum CPU utilization of the system can be higher than indicated by the calculation performed by MTBIOfflineAnalysis.

- Also, the accuracy of MTBIOfflineAnalysis might be affected by the fact that it uses the release times of tasks and not the true start times. However, evaluations in Chapter 6 (Section 6.4.3) show that using release times yields accurate results.

- The probability used on line 10 in Figure 7.7 might not trigger the worst-case mean time between invocations.

- The running time of MTBIOfflineAnalysis is pseudo-polynomial in the period times or polynomial with a large constant if the length of timelines is fixed. This indicates that MTBIOfflineAnalysis is unsuitable for on-line execution. This is resolved in the next chapter by using multiple regression.

MTBIOfflineAnalysis can give accurate estimates, and if its estimate predicts a system to be schedulable but it is, at run-time, unschedulable then the ACUA algorithm presented in this chapter can reduce the workload during overload. Thus, MTBIOfflineAnalysis reduces pessimism in the calculated CPU utilization by accurately determining how often on-demand updates are used, and in situations where MTBIOfflineAnalysis gives a wrong prediction ACUA avoids the overload.

CHAPTER 8

On-line Estimation of CPU Utilization

As we saw in chapters 6 and 7, it is possible to estimate the total CPU utilization of a system using on-demand updating. However, as was also discussed in Chapter 7, the workload can change dynamically and by calculating the total CPU utilization given, e.g., a new interarrival time of a task, the question whether the system eventually will become overloaded is immediately answered. In addition, it probably takes a while before the phasings of the tasks causing the overload to occur, so there may be time to act early on the indicated overload. In this way, an overload could be avoided. However, for some systems it is impossible to exactly determine the total CPU utilization, and, thus, there are inherent inaccuracies in the schedulability test. We investigate the accuracy in this chapter.

In this chapter we construct a new algorithm, MTBIAlgorithm, that uses a linear model of tasks' interarrival times, change frequency of values, and the mean interarrival time of $code(update\_d_i)$. The mean time between invocations of functions is used to calculate the total CPU utilization. Calculating total CPU utilization addresses requirement R5, determining whether the system is in a transient overload.

The outline of this chapter is as follows. Section 8.1 presents the model being used in this chapter to estimate total CPU utilization and it gives an equation that calculates mean time between invocations of a set of periodic tasks. Section 8.2 presents the MTBIAlgorithm algorithm. Finally, Section 8.3 shows the performance results and Section 8.4 wraps-up the chapter.

8.1 MTBI in a System with Arbitrary Number of Levels in G

In this chapter, an efficient algorithm denoted MTBIAlgorithm is constructed that estimates the mean interarrival time of updates at any node in the data dependency graph G. The data model described in Section 7.2 is used also in this chapter since the model allows for arbitrary data dependency graphs (which is in contrast to the data model used in Chapter 6). This section describes the linear model that is used in MTBIAlgorithm to estimate mean interarrival time of updates. This section also contains a discussion on properties of systems reflected in G. Finally, generation of simulation data used in the multiple regression is discussed.

8.1.1 Model

In this section we describe how the mean time between invocations of the body of, e.g., $update\_d_k$, can be estimated given mean time between invocations on d_k's read set and interarrival times of tasks. We use regression analysis. The model is

$$Y = \beta_0 + \beta_1 \times x_1 + \beta_2 \times x_2 + \epsilon, \tag{8.1}$$

where Y is $MTBI(code(update\_d_i))$, x_1 is $MTBI(R(d_k))$, and x_2 is $MTBI(tasks(d_k))$. β_0, β_1, and β_2 are the parameters of the model. $MTBI(R(d_k))$ can be calculated by using Equation 6.3 and setting **P** to the estimated mean interarrival time for each immediate parent of d_k. $MTBI(tasks(d_k))$ can be calculated by using Equation 6.3 and setting **P** to the period times associated with leaf nodes being descendants of d_k. Figure 7.3 shows a data dependency graph. $MTBI(R(d_k))$ is $MTBI(\{period(b_1), period(b_2)\}) = 60$, and $MTBI(tasks(d_k))$ is $MTBI(\{period(d_o), period(d_p), period(d_q)\}) = 70.3$.

By using least square fit of collected data (see Section 8.1.3) we get estimates of β_0, β_1, and β_2 that give a prediction $\hat{Y}$ of Y.

The values of the estimates b_0, b_1, and b_2 depend on the number of elements in $R(d_k)$, the number of elements in $tasks(d_k)$, and whether $MTBI(R(d_k)) < MTBI(tasks(d_k))$ as is shown in Figure 8.1. The term ϵ is the error of the estimation of $MTBI(code(update\_d_k))$. If the error fulfills the Gauss-Markov conditions then least square fit is the best method to determine the values of b_0, b_1, and b_2 [113].

8.1.2 Analysis of Data Dependency Graphs

This section describes how data dependency graphs are constructed for the experiments in this chapter. This section also discusses the properties of data dependency graphs; that information is used to discuss, later in this chapter, why the results of MTBIAlgorithm look as they do. Data dependency graphs

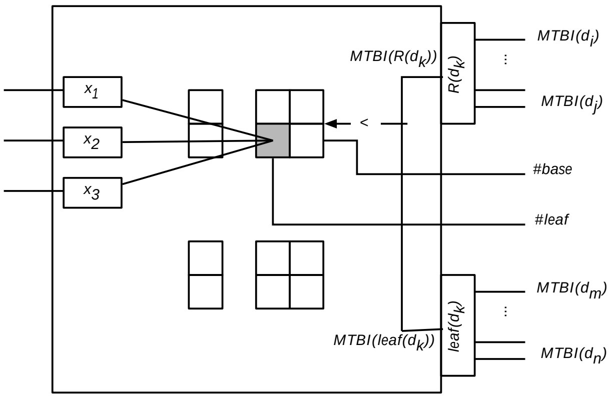


Figure 8.1: The model of estimating $MTBI(code(update\_d_k))$. The right of the figure constitutes the input, which is number of base items, number of leaf nodes, mean interarrival time updates of base items, and mean interarrival time of executed updates of leaf nodes. These inputs are used to find the estimates b_0, b_1, and b_2 of β_0, β_1, and β_2, respectively.

are constructed by setting the following parameters: cardinality of the read set, $|R(d_i)|$, ratio of $R(d_i)$ being base items, and ratio being derived items with only base item parents. The cardinality of $R(d_i)$ is set randomly for each d_i and a ratio of these are base items, a ratio are derived items with only base item parents, and the remaining parents are other derived items. These figures are rounded to nearest integer. The number of derived items with only base item parents is set to a ratio of the total number of derived items. We use a graph consisting of 150 data items. We believe such a graph represents the storage requirements of a hotspot of an embedded system, e.g., in an engine control software 128 data items are used to represent the external environment and actuator signals.

We expect the number of tasks making requests of a data item to be small, because functionality tends to be specialized and partitioned into tasks, e.g., in an engine control software the task deriving fuel amount to inject into a cylinder to a large extent uses other data items than the task diagnosing the lambda sensor.

The level of a node in G is defined in definition 3.2.1. The lower the level, the higher we expect the number of tasks calling a data item d_i to be, because a node with a low level tends to have more descendants than a node having a higher level, i.e., more descendants may result in more tasks. Figure 8.2 shows the number of derived items having a specified number of tasks requesting the data item. The data dependency graph is generated using the parameters in Table 8.1. We see that a majority of the derived data items in G is requested by one task. Hence, the algorithm for generating data dependency graphs gives graphs that reflect data dependencies in real-life embedded systems.

Table 8.1: Parameters of data dependency graph.

Parameter	Value
Cardinality of read set	U(1,8)
Ratio of $R(d)$ being base items	0.30
Ratio of $R(d)$ being derived item with only base item parents	0.60
Ratio of $R(d)$ being other derived items	0.10

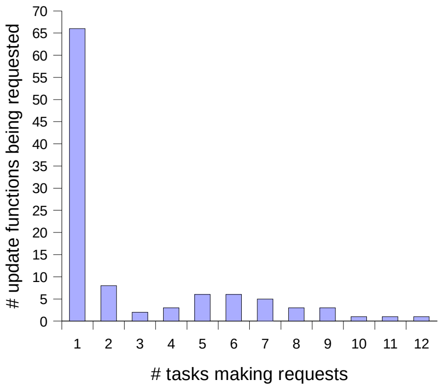


Figure 8.2: The number of tasks making requests of derived data items.

8.1.3 Multiple Regression

In order to determine the values of the parameters β_0, β_1, and β_2 in the model represented by Equation (8.1) we use the well-established technique of multiple regression using least square fit [113]. In order to get values on the response variable $MTBI(code(update\_d_k))$ and on the explanatory variables $MTBI(R(d_k))$ and $MTBI(tasks(d_k))$, we use a program, denoted MTBIOfflineAnalysis and presented in Figure 7.7, which constructs a timeline with invocations of each base item update, as well as a timeline for each task invocation that may update the data item. A third timeline is constructed by taking the first distinct task invocation after a base item invocation. Taking the first task invocation after a base item invocation represents the outer if-statement in Figure 7.1 where we expect a base item update to make the if-statement true, i.e., on-demand updating is simulated. Thus, task invocations occurring before a new base invocation are skipped. This approach models that the code portion of updates is only executed when something has happened. In Figure 7.3, timelines 1, 2, and 3 are generated by MTBIOfflineAnalysis.

MTBIOfflineAnalysis runs with the parameters presented in Table 8.2. The arrival rates of base items and derived items are chosen such that they resemble an engine control software. When using MTBIOfflineAnalysis there is an equal chance that a task has any of the 5 possible period times. The 15 runs with same periods use random start times of each task, i.e., the start time of the first invocation of τ_i is derived from the following uniform distribution $U(0, P_i)$. The median of these 15 runs is derived and used as $MTBI(code(update\_d_k))$.

The multiple regression is done on two cases. One case is when $MTBI(R(d_k)) < MTBI(tasks(d_k))$ and the other when $MTBI(R(d_k) > MTBI(tasks(d_k))$. The reason we have chosen to divide into these two cases is because if the base items change more often than tasks making requests, then the occurrences of the tasks are the bottleneck of the $MTBI$. Hence, if base items change more seldom than tasks make requests, then the occurrences of the base items are the bottleneck.

We also do regression for each setting of number of base items and number of derived items. For a system needing at maximum 5 base items, and at maximum 9 derived items, $5 \times 9 \times 2 = 90$ regressions are needed. However, it is possible to automate the process of getting the correct data from MTBIOfflineAnalysis, and run the least square fit on the data. The 90 regressions are then stored in a table in the software. The values of b_0, b_1, and b_2 are derived by using the `regress` command in Matlab.

In tables 8.3 and 8.4, many of the regressions fail the Kolmogorov-Smirnov test, i.e., the distribution of the error might not be a normal distribution. This indicates that condition (2.9) of the Gauss-Markov conditions might not be fulfilled, i.e., we do not know if the variance of the error is constant, and therefore we do not know if least square fit is the best method to do the fitting of variables b_0, b_1, and b_2. However, in Section 8.3 we see that the CPU utilization estimation using the regression in Equation (8.1) is always overestimated and close to the simulated CPU utilization, therefore we chose to use the values we

Table 8.2: Parameters for MTBIOfflineAnalysis to feed into the least square fit to determine values on b_0, b_1, and b_2.

Name	Quantity
Base items	1–5
Tasks	1–9
Setup 1	
Base item periods	U(30,500)
Arrival rate	U(10,40)
Period time 1	$\frac{32}{arrivalRate} \times 60 \times U(1,5)$
Period time 2	$\frac{32}{arrivalRate} \times 120 \times U(1,5)$
Period time 3	$\frac{32}{arrivalRate} \times 250 \times U(1,5)$
Period time 4	$\frac{32}{arrivalRate} \times 500 \times U(1,5)$
Period time 5	$\frac{32}{arrivalRate} \times 1000 \times U(1,5)$
Runs	200
Runs with same periods	15
Setup 2	
Base item periods	U(30,2000)
Arrival rate	U(10,40)
Period time 1	$\frac{32}{arrivalRate} \times 60 \times U(1,5)$
Period time 2	$\frac{32}{arrivalRate} \times 120 \times U(1,5)$
Period time 3	$\frac{32}{arrivalRate} \times 250 \times U(1,5)$
Period time 4	$\frac{32}{arrivalRate} \times 500 \times U(1,5)$
Period time 5	$\frac{32}{arrivalRate} \times 1000 \times U(1,5)$
Runs	200
Runs with same periods	15

Table 8.3: $MTBI(base) < MTBI(leaf)$.

#base	#leaf	K-S	Skewness	#base	#leaf	K-S	Skewness
1	1	F	3.01	3	1	F	3.38
1	2		0.53	3	2	F	-0.16
1	3	T	-0.233	3	3	F	-0.35
1	4	T	-0.62	3	4	F	-0.12
1	5	T	-1.13	3	5	T	-0.53
1	6	T	1.16	3	6	T	-0.26
1	7	T	0.25	3	7	T	-0.54
1	8	T	0.30	3	8	T	-0.35
1	9	T	0.35	3	9	T	-0.017
1	10	T	1.14	4	1		3.01
1	11	T	0.56	4	2	F	0.53
1	12	T		4	3	F	0.55
1	13	T	0.57	4	4	F	0.34
2	1	F	2.88	4	5	F	-0.51
2	2	F	0.52	4	6	T	-0.12
2	3	F	-1.53	4	7	T	-1.21
2	4	F	0.36	4	8	T	-0.45
2	5	T	0.21	4	9	T	-0.53
2	6	F	-0.95	5	1	F	4.21
2	7	T	-0.18	5	2	F	0.17
2	8	T	0.60	5	3	F	-0.40
2	9		-0.67	5	4	F	0.17
2	10	T	-0.47	5	5	F	-0.47
2	11	T	0.21	5	6	F	-0.25
2	12	T	0.85	5	7	F	-0.40
2	13		0.45	5	8	T	-1.41
				5	9	T	

Table 8.4: $MTBI(base) > MTBI(leaf)$.

#base	#leaf	K-S	Skewness	#base	#leaf	K-S	Skewness
1	1	T	0.86	3	1	T	0.30
1	2	F	2.79	3	2	T	-0.36
1	3		1.66	3	3	T	-0.036
1	4		1.76	3	4	T	-0.075
1	5	F	2.13	3	5	F	-0.60
1	6	F	4.80	3	6	F	-0.92
1	7	F	2.46	3	7	F	-0.27
1	8	F	3.55	3	8	F	-0.21
1	9	F	4.15	3	9	F	-0.55
1	10	F	3.45	4	1	T	-0.44
1	11	T	1.86	4	2	T	0.34
1	12	T	3.41	4	3	T	0.44
1	13	F	2.60	4	4	T	-0.54
2	1	T	-0.50	4	5	T	-0.47
2	2	T	-0.26	4	6	T	-1.12
2	3	F	-0.10	4	7	T	-0.13
2	4	F	0.013	4	8	F	-0.75
2	5		-1.78	4	9	F	-0.31
2	6	F	-0.20	5	1	T	-0.91
2	7		0.44	5	2	T	-0.29
2	8	F	-0.30	5	3	T	-0.024
2	9	F	-0.54	5	4	T	-0.17
2	10	T	-0.070	5	5	T	-1.11
2	11	T	-0.095	5	6		-0.40
2	12	T	0.12	5	7	T	-0.11
2	13	T	0.50	5	8	T	-0.98
				5	9	T	

get from the regression, even though they might not be optimal.

8.2 CPU Estimation Algorithm

The algorithm MTBIAlgorithm is introduced in this chapter. The algorithm calculates $MTBI$ for each data item in increasing level in the data dependency graph. MTBIAlgorithm is presented in Figure 8.3.

The rational for the algorithm is described next. In order to get a CPU utilization estimation of an embedded system, the regression in Equation (8.1) is used on each function $update\_d_i$ in the system. However, for Equation (8.1) to work, $MTBI(R(d_k))$ and $MTBI(tasks(d_k))$ should be computable. $MTBI(tasks(d_k))$ is always possible to calculate by doing a bottom-up traversal from each node, n, with zero out-degree in the data dependency graph G. Every data item the traversal passes is annotated with the interarrival time of the corresponding task of node n. Duplicates are removed[1]. This step can be made off-line if the graph is not changing during run-time. $MTBI(R(d_k))$ can be calculated if $MTBI(code(update\_d_i)), \forall d_i \in R(d_k)$ are available. Thus, the order Equation (8.1) must be applied on data items is the same as a top-bottom order of G, because in a top-bottom order of G all data items in a read set have already been calculated by applying Equation (8.1), and $MTBI(R(d_k))$ can be calculated.

The pseudo-code presented in Figure 8.3 describes how, for each level of G, Equation (8.1) is used on each data item in the level. The values on b_0, b_1, and b_2 are fetched from one of two tables. Which table is used is determined by checking the condition $MTBI(R(d)) < MTBI(tasks(d))$.

The computational complexity of MTBIAlgorithm is linear in the number of levels in the graph that, in turn, is linear in the size of the graph; calculations of Equation (8.1) takes $O(1)$ time, i.e., the computational complexity of MTBI-Algorithm is $O(|N|)$.

8.3 Performance Results

This section shows the performance of the CPU estimation using the algorithm described above. The baseline to calculate CPU utilization of functions in the system is to assume functions get executed every time they are called.[2]

Figures 8.4, 8.5, 8.6, 8.7, and 8.8 show the ratio $(Uest - U)/U$ where $Uest$ is the estimated CPU utilization using MTBIAlgorithm (Figure 8.3) for determining mean time between invocations, and U is the estimated CPU utilization using program MTBIOfflineAnalysis. The arrival rate is fixed at 20 tasks per second, and the period time of base items and derived items are multiplied with an integer from the distribution U(1,5). The number of

[1]A duplicate can arise if there are several paths from n to a data item in G.

[2]The baseline is the same in Chapter 6.

```
**for all** levels $level$ from 1 to $\max_{\forall d \in G}(level(d))$ **do**
    **for all** data items $d$ with level $level$ **do**
        **if** $MTBI(R(d)) < MTBI(tasks(d))$ **then**
            $b_0 = coeffb0\_base\_less[\#R(d)][\#tasks(d)]$
            $b_1 = coeffb1\_base\_less[\#R(d)][\#tasks(d)]$
            $b_2 = coeffb2\_base\_less[\#R(d)][\#tasks(d)]$
            $MTBI(code(update(d))) =$
            $= b_1 \times MTBI(R(d)) + b_2 \times MTBI(tasks(d)) + b_0$
        **else**
            $b_0 = coeffb0\_base\_greater[\#R(d)][\#tasks(d)]$
            $b_1 = coeffb1\_base\_greater[\#R(d)][\#tasks(d)]$
            $b_2 = coeffb2\_base\_greater[\#R(d)][\#tasks(d)]$
            $MTBI(code(update(d))) =$
            $= b_1 \times MTBI(R(d)) + b_2 \times MTBI(tasks(d)) + b_0$
        **end if**
    **end for**
**end for**
```

Figure 8.3: Pseudo-code for MTBIAlgorithm which calculates $MTBI(code(update(d)))$ for each data item d in the graph G. $\#R(d)$ means the number of data items in the set $R(d)$. Similarly, $\#tasks(d)$ means the number of items in the set $tasks(d)$. The value $oldMTBI(R(d))$ is the value $MTBI(code(update(d)))$ had when $MTBI(code(update(d)))$ was last calculated.

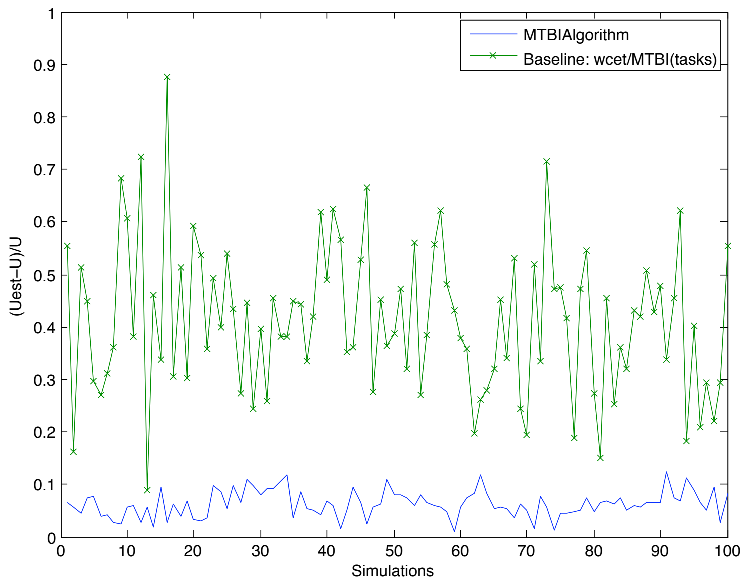


Figure 8.4: CPU utilization estimate of graph 1 using regression and using Equation (6.3).

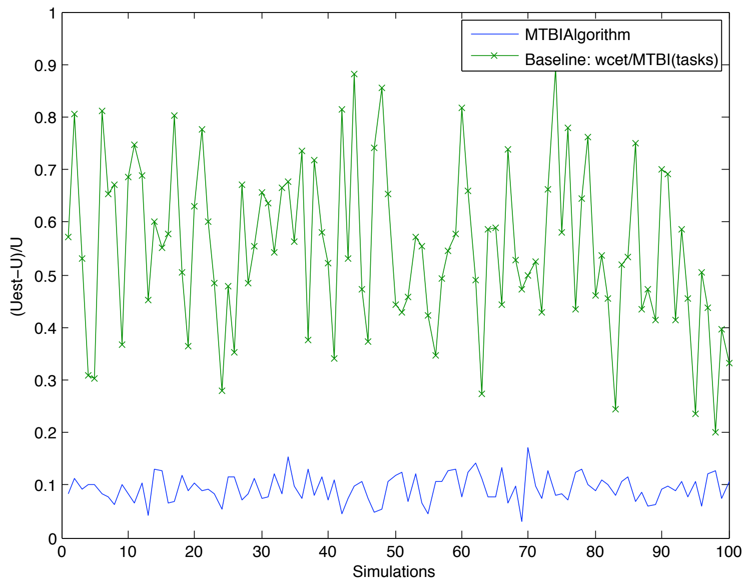


Figure 8.5: CPU utilization estimate of graph 2 using regression and using Equation (6.3).

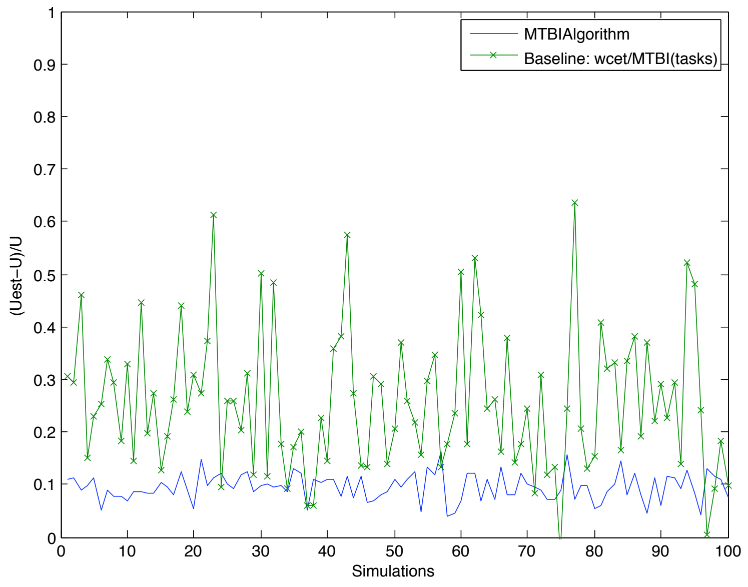


Figure 8.6: CPU utilization estimate of graph 3 using regression and using Equation (6.3).

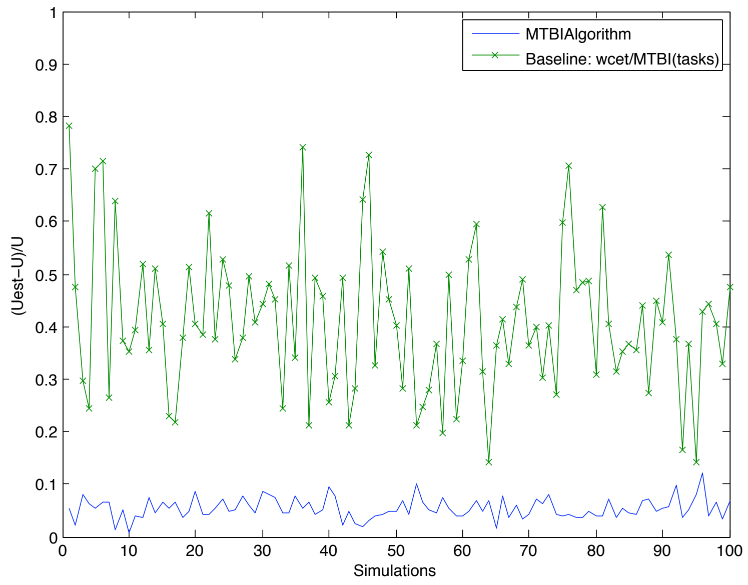


Figure 8.7: CPU utilization estimate of graph 4 using regression and using Equation (6.3).

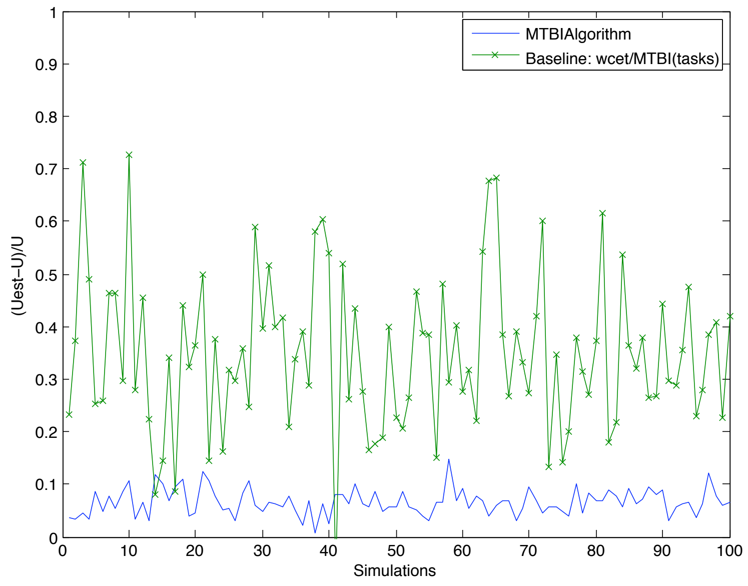


Figure 8.8: CPU utilization estimate of graph 5 using regression and using Equation (6.3).

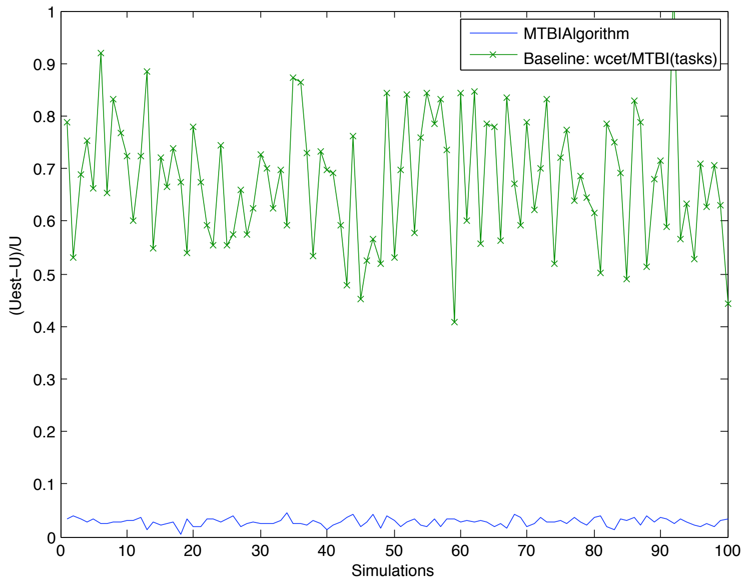


Figure 8.9: CPU utilization estimate of graph 1 using regression and using Equation (6.3).

simulations is 100, and the graphs are constructed using the parameters in Table 8.1. We see that the CPU estimate using MTBIAlgorithm is closer to the utilization compared to using Equation (6.3) to estimate mean time between invocations of $update(d_k)$. Also, the estimate is always larger than 0 which means that the estimate tends to be an overestimate.

Figures 8.9–8.13 show the ratio $(Uest - U)/U$ for the same five graphs as above, but the period times are multiplied with a number taken from the distribution $U(1, 5)$, i.e., integers are not used. We see that we get the same results as above. MTBIAlgorithm gives a tighter estimate than the baseline. In this setting, MTBIAlgorithm gives a negative ratio two times for graph 4 (Figure 8.12). The negative ratios are -0.005 and -0.01. Thus, of 500 simulations, only two give small negative ratios, all other simulations give tight overestimations.

The overestimate in Figures 8.4–8.8 and Figures 8.9–8.13 is due to that the skewness of the error is toward overestimating the CPU utilization, i.e., the mean time between invocations of requests of $update(d_i)$ is underestimated. Tables 8.3 and 8.4 list the skewness of error. The skewness is the largest, and positive, when there is one requesting leaf node. As we discussed in Section 8.1.2, a majority of the data items is requested by only one leaf node. Hence, for a majority of the data items, the CPU estimation based on the estimation on the mean time between requests of a data item tends to be overestimated. Figure 8.14 shows $(Uest - U)/U$ for each derived data item for a graph constructed from the parameters given in Table 8.1. Two things can be established from

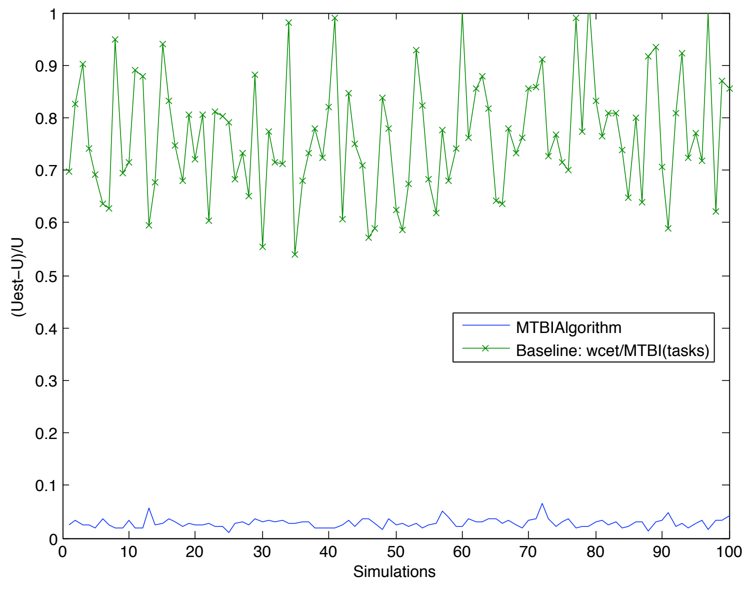


Figure 8.10: CPU utilization estimate of graph 2 using regression and using Equation (6.3).

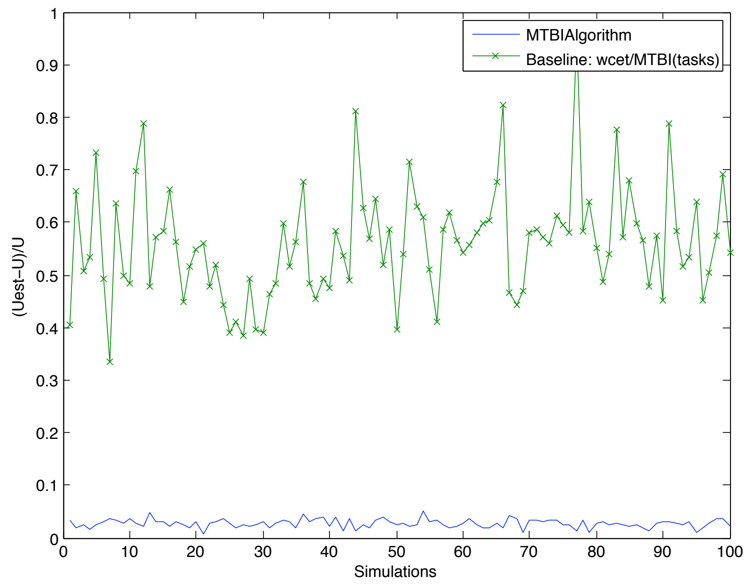


Figure 8.11: CPU utilization estimate of graph 3 using regression and using Equation (6.3).

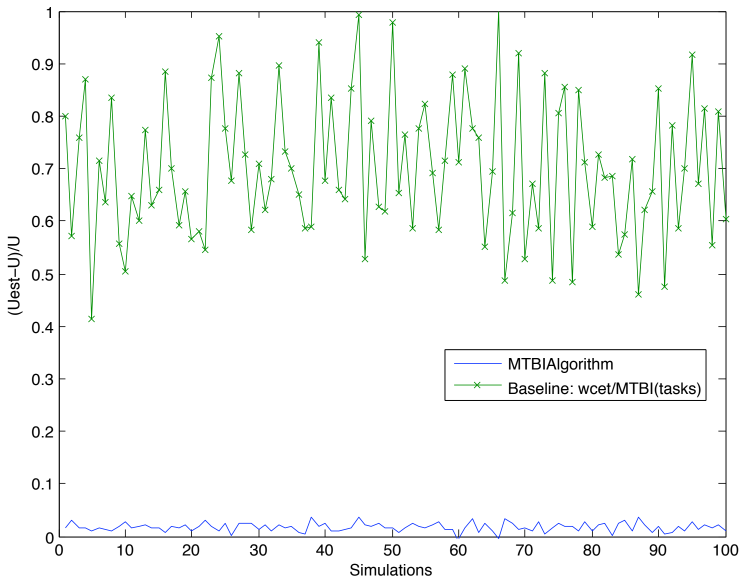


Figure 8.12: CPU utilization estimate of graph 4 using regression and using Equation (6.3).

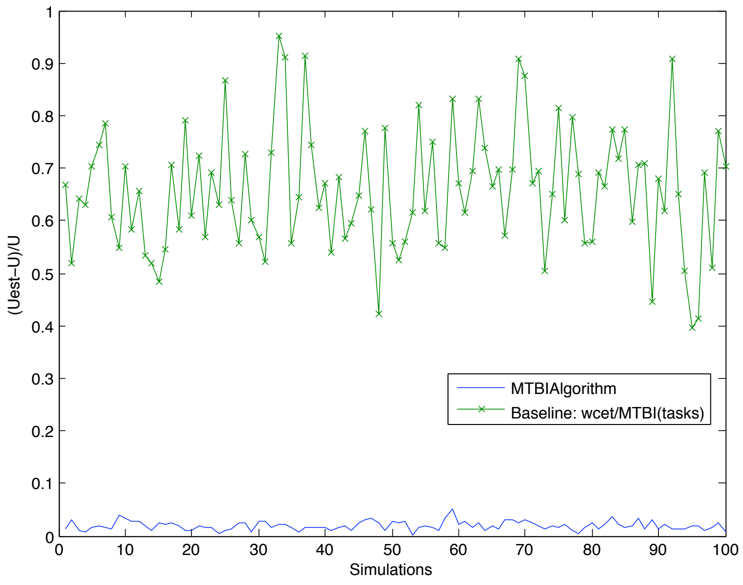


Figure 8.13: CPU utilization estimate of graph 5 using regression and using Equation (6.3).

Table 8.5:

Period times multiplied with integers	
Level	$(Uest - U)/U$
2	0.0810
3	0.08272
4	0.0426
5	-0.000844
6	0.00436
7	0.00918
Period times multiplied with floats	
Level	$(Uest - U)/U$
2	0.0287
3	0.0257
4	0.0235
5	0.00254
6	-0.0136
7	0.0244

Figure 8.14 (i) the relative error is larger the lower the level is, and (ii) the relative error tends to be greater than 0. Thus, the CPU utilization estimation on individual data items can be inexact, but, as Figure 8.14 shows, for derived data items with a high level, the CPU utilization estimation is close to the value found by MTBIOfflineAnalysis. Table 8.5 reports the CPU utilization estimation using MTBIAlgorithm for each level in the graph used in Figures 8.9 and 8.4.

Since all the estimations in Figures 8.4–8.13 are overestimated and the overestimates are considerably smaller than the baseline and tight (5% overestimate for some systems, e.g., Figures 8.9–8.13), we believe multiple linear regression can be a usable method to estimate CPU utilization for a system also with many levels in data dependency graph G. However, before the system is deployed using MTBIAlgorithm the system should first be simulated to check that MTBIAlgorithm gives good enough accuracy.

8.4 Wrap-Up

In this chapter we have shown that it is possible to linearize relationships existing in embedded systems by using monitored data of the relationship and fitting the data to a linear model using least square fit. We linearize the relationship between a set of periodic requests and periodic changes to data values and the MTBI of a conditioned calculation. Since the relationships are linearized the computational complexity of MTBIAlgorithm is polynomial in the size of G, which means that the total CPU utilization of a system may be

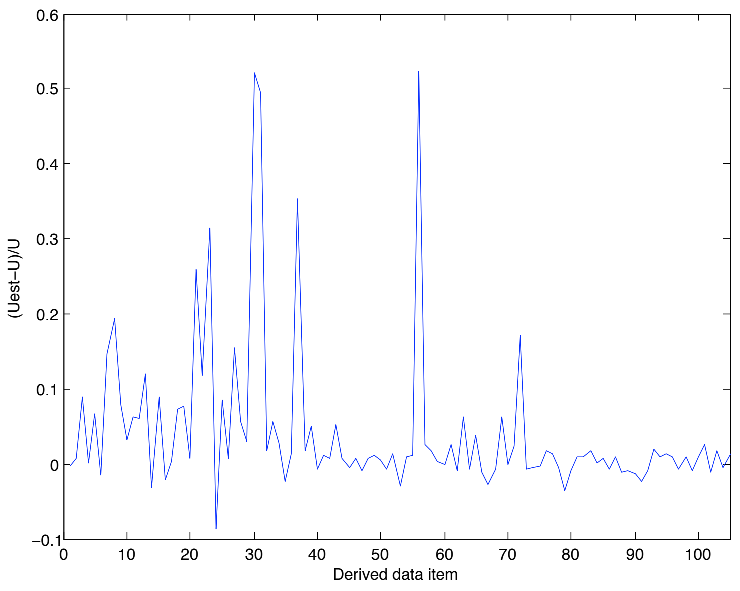


Figure 8.14: The relative error for each derived data item for a graph with 105 derived data items. The items 0–30 are in level 2, items 31–68 are in level 3, items 69–89 in level 4, items 90–98 in level 5, items 99–101 in level 6, and items 102–104 in level 7.

recalculated every time the interarrival time of a task changes. There are a number of factors that influence the accuracy of the linear model, namely:

- MTBIAlgorithm uses a model built upon linear multiple regression. Thus, the linear model gives errors in the estimates.
- The data fed to the linear regression is using MTBIOfflineAnalysis that gives estimates of mean interarrival times of updates. Thus, the model is built on estimated data.
- The inputs to the model in MTBIAlgorithm are outputs of the algorithm itself, i.e., the inputs are estimates.

The evaluations in this chapter show that the estimates can be accurate even though the model uses approximate data.

CHAPTER 9

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 [81], but has later on also been used when optimizing task period times [68], updating of data [131], and data dissemination on the web [41, 95, 109]. Similarity, as adopted in this thesis, can be divided into updating algorithms and concurrency control algorithms. The following sections relate the work achieved in this thesis to previous work done on data freshness (Section 9.1), concurrency control algorithms (Section 9.2), and admission control (Section 9.3).

9.1 Updating Algorithms and Data Freshness

We have previously discussed that in order to utilize the CPU resource efficiently unnecessary updates must be avoided. It is important for many applications that data items are up-to-date. The freshness of the value of a data item can be measured either in the time domain or in the value domain. Measuring freshness in the time domain has been used to set period times and/or deadlines of tasks [7, 9, 75, 76, 108, 135–137]. By predetermining the arrival pattern, e.g., fixing the period times of tasks [68, 76, 88, 136], avoidance of unnecessary updates cannot be achieved. Hamdaoui and Ramanathan introduced (m, k)-firm deadlines, where m deadlines out of k consecutive invocations of a task have to be met [63]. Hence, an invocation of a task can be skipped and it can be used to balancing 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, as we saw in Chapter 7, of data freshness in the value domain, and updates of data items are invoked even though values are unchanged. Thus, although skips of tasks are possible using (m, k)-firm deadlines, resources are not efficiently used at steady states as they are using similarity-aware updating algorithms, e.g., ACUA-All.

Kuo and Mok have introduced a *similarity bound* saying that two writes to a data item are similar if the time between them is less than the similarity bound [81, 82]. 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 [131]. The updating of data items works as follows [131]. 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 mangers 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) they 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. investigated the difference in performance between on-demand and updates-first and found that on-demand updating of data performs better than updates-first [7]. The evaluations of similarity-aware multiversion concurrency control in Chapter 5 show that ODTB with multiversion concurrency control supports more UTs commit within deadlines compared to using ODTB with single-version concurrency control with RCR as used by Wedde et al. in [131].

Tomic and Vrbsky introduced a new measure of data freshness that is used for derived data items [127]. They also discuss data freshness in the case of approximate queries, but we postpone that discussion until Section 9.3. Tomic and Vrbsky propose to define temporal consistency of data as follows. The read set being used deriving a data item d must contain versions of data items whose valid times overlap. This means that the values of the data items in the read set of a derived data item are up-to-date at the same time. In this thesis we implement this behavior by using definition 2.2.2. This definition is also used in the construction of the MVTO-S snapshot algorithm in Chapter 5 in order to guarantee that the values of data items used to derive a value are from the same system state.

Kao et al. introduce definitions of data freshness for discrete data objects in [74] 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. [8] 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 allowing 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.

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 [83]. This resembles the pa timestamp in ODDFT and ODKB_V where data items are assumed to be changed when at least one ancestor 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 [83] 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.

Blakely et al. have shown how it is possible to decide which updates to base and derived data objects affect the views, i.e., derived data objects [20]. 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 [20]. 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 [68, 76, 88, 134–136]. 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 and thesis, we have considered 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. One possible way to resolve this is to use multi mode operation of the system [111] where, for instance, specific updating frequencies for each mode could be assigned to the data items. In this context, for each mode the data items have specific updating frequencies 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 significantly when the engine is started, then the values gradually stabilize, and finally these compensation factors are not used any more. Our proposed updating schemas for freshness maintenance of data do not need mode changes since changes of updating frequencies are covered by the data validity intervals, because updating of data is done when needed.

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 [39]. 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.

Data-deadline and forced wait [137] are designed to achieve freshness at the deadline of a user transaction as in our work indicated by time t in the function $error$ 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 [137].

Ahmed and Vrbsky proposed three schemes for a database—based on the on-demand idea—that make data items up-to-date-before they are used [9]. One of the schemes corresponds to τ^{time}-tasks and the other two schemes extend the if-statement to check for available slack time and response times of tasks.

To summarize this section we conclude that there have been no comparisons, to the best of our knowledge, between results from the most recent research in periodic updating (More-Less [136] and DS [135]) and on-demand updating. Furthermore, there has been no work on estimating the interarrival times between on-demand updates. Such an estimate could be used to estimating the workload imposed by updates. In this thesis, we look into these issues.

9.2 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 [29, 65, 70]. 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 a system has been documented elsewhere.

Lam et al. have shown evaluations of concurrency control algorithms for mixed soft real-time transactions and non-real-time transactions [85]. They found that an integrated TO scheduler using OCC for soft real-time transactions and 2PL for non-real-time transactions performs 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 [116, 117, 121]. It has been found that 2PL performs better than MVTO and the single-version timestamp ordering concurrency control algorithm [121]. Song and Liu evaluate the 2PL and OCC multiversion algorithms in a hard real-time system [117]. 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. [86] and O2PL by Wedde et al. [131]. The proposed multiversion concurrency control algorithms, MVTO-S^{UV}, MVTO-S^{UP}, and MVTO-S^{CRC} 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 [123]. 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 [122, 123]. 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 [131]. 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-S^{UV}, MVTO-S^{UP}, and MVTO-S^{CRC} start restarting active transactions with lowest priority until there are enough memory for the current operation.

Epsilon-serializability also uses a form of similarity [107]. Epsilon-serializability is used in concurrency control to relax the serializability criterion (see Section 2.6) 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.

9.3 Admission Control

The research on admission control in real-time systems has been extensive [24, 38, 63, 66, 72, 77]. However, the work that has been done primarily focuses on admission control of independent tasks, whereas we in this thesis focus on admission control where data has relationships.

Work on maintaining data freshness can be classified into (i) off-line algorithms determining period times on tasks [88, 136, 137], and (ii) on-line algorithms [9, 41, 55, 56, 72, 95]. In Section 3.3.1, we showed that the general problem of choosing updates and considering data relationships is NP-hard in the strong sense and previous on-line algorithms are simplified (see Section 7.1) to reduce computational complexity in such a way that they reject updates when the updates cannot be fitted within the available time. We have investigated the case where data items can be divided into required data items and not required data items and devise an algorithm that updates data items and guarantees they get up-to-date at the same time as deadlines are met.

Tasks in the imprecise computation model can be described with one of the following approaches [92].

- *Milestone approach*: The result of a task is refined as its execution progresses. A task can be divided into a mandatory and an optional part, where the result after executing the mandatory part is acceptable, and the result after also executing the optional part is perfect, i.e., the error of the calculation is zero.

- *Sieve approach*: A task consists of operations where not all of them are compulsory [16]. A typical example is when a data item's value can be updated or used as is, i.e., the update is skipped.

- *Primary/alternative approach*: The task can be divided into a primary task containing functionality to produce a perfect result. The alternative task takes less time to execute and the result is acceptable. One of the primary and the alternative task is executed.

ACUA, described in Chapter 7, is consistent with the imprecise computation model because ACUA can construct a schedule that gives an imprecise but acceptable result. Other work that also fits in the imprecise computation model is [97, 103, 128–130]. The values of data items when using ACUA is acceptable because required data items are always updated. The approach of dividing data items into required and not required data items has industrial applications (see Section 7.1), e.g., a subset of fuel compensation factors must be updated in order to get an acceptable result. In this thesis, we have focused on on-line constructing the content of user transactions by considering workload, data freshness, and data relationships to get acceptable accuracy of produced results. To the best of our knowledge this is the first time such an approach is evaluated.

Kang et al. have described a flexible data freshness scheme that can reduce the workload by increasing the period times on updates of data items [72]. A feedback approach is used where a monitoring of changes in deadline miss ratio results in changing period times of updates within given bounds. The work in [72] does not consider data relationships nor data freshness measured in the value domain. Moreover, using a feedback approach introduces a settling time, i.e., it takes a time before the system stabilizes after a workload change. Some systems need fast reactions, and our evaluations show that using ACUA with a feasibility test lets the system react immediately on a workload change. Other work using the feedback control approach to perform admission control in the case of transient overloads, in the real-time area, are [12–14, 26, 94]. Lu et al. describe a model of a real-time database and give equations to design a stable feedback control admission controller using the root locus method [94]. Amirijoo et al. extend the milestone approach in a real-time database setting and use feedback control for decisions of admission of tasks [12–14]. Cervin et al. use feedback and feedforward control of control tasks [26].

Ramamritham et al. have investigated data dissemination on the Internet, where the problem of clients reading dynamic data from a server is discussed [41, 78, 95, 109, 115]. 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 [41, 95, 109]. 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. [131].

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. [41, 109]. A feedback control-theoretic approach is investigated in [95].

Feasibility tests are important for admission control and a wide range of feasibility tests have been proposed for different task models [10, 11, 18, 43, 87]. The task models consider different levels of granularity of the tasks. The models with high granularity model tasks as consisting of subtasks with precedence constraints, and branches [18]. Other models consider tasks as one unit of work [10, 11, 43, 87]. However, no model takes data values into consideration, which means that the CPU utilization, U, can be higher than the actual CPU utilization, because the feasibility test must assume, e.g., an if-statement is true which it might not be. Symbolic WCET analysis expresses the WCET of a task as a formula with parameters, e.g., maximum number of iterations of a loop [30]. The source code is divided into scopes that are associated with execution time and frequency. In order to get an accurate WCET of a source code, the frequencies of the scopes must be accurate. In this thesis, we have investigated a way to derive the frequency—that can change on-line—of a scope.

The results of chapters 6 and 8 can be applied in admission control of updates since the mathematical models of workload described in these chapters give a way to estimate workload imposed by updates. To the best of our knowledge, it is a novel approach to construct mathematical models estimating the workload of the system.

CHAPTER 10

Conclusions and Future Work

This chapter concludes the research results presented in this thesis and it ends with a discussion of possible future directions of our research.

10.1 Conclusions

The goal of this thesis has been to provide efficient data management for real-time embedded systems. Efficient data management can be provided in different ways. Our hypothesis has been, since databases have been used successfully for data management for many different problem domains during several decades, that a real-time database is a good means to manage data in real-time embedded systems. We argued in Chapter 3 that we could assume the hypothesis to be true if the real-time database has certain functionality. Thus, we focus our work on which functionality the real-time database should have in order to manage the data efficiently. With efficient data management we mean the following.

- Compared to the development methods being used in the industry, the development and maintenance of the software become more efficient when using a real-time database.
- Compared to how existing systems perform, they could perform more efficiently if they used a real-time database to manage the data.

Based on the experience of industrial partners developing a specific embedded system over many years, we have chosen to focus on **updating data items within the database** and **fast CPU workload analysis methods**. The motivations for these choices are elaborated below.

- Updating data items within the real-time database because:

  - It is important that data items have up-to-date values because the control of the external environment is made based on the values of these data items as they represent the current observed state.
  - It becomes easier to develop and maintain the software because the programmers do not have to implement functionality to maintain freshness of data items because it is managed by the real-time database.
  - The CPU utilization of updating data items constitutes a major part of the workload and if the database can reduce the CPU utilization of updating data it gives an immediate effect on the whole system. The fact is that calculations involving similar data values give similar results because calculations are deterministic and time-invariant. So, if this fact is exploited the CPU utilization of updating data can in reality be significantly reduced, which is shown in the thesis.

- Analysis of workload of the system when the real-time database is used is required because embedded systems can have dynamically changing workloads and can also be overloaded. Thus, during execution of the system the system can enter a transient overload. Analysis of the system helps the development and maintenance of the software in the following ways.
  - It is possible to both off-line and on-line predict whether the software may become overloaded.
  - It is possible to adjust the workload on-line such that a potential overload is avoided.

Our research has resulted in the real-time database, DIESIS, with the following novel functionality:

- Updating algorithms that use similarity to measure data freshness and that take data relationships into consideration, which are expressed in a directed acyclic graph of arbitrary depth. The devised algorithms ODDFT, ODBFT, ODDFT_C, ODBFT_C, ODKB_V, ODKB_C, and ODTB, are compared to the well-established updating algorithms OD, ODO, and ODKB [9].
- Concurrency control algorithms—OCC-S, MVTO-S^{UV}, MVTO-S^{UP}, and MVTO-S^{CRC}—that also use similarity. MVTO-S^{UV}, MVTO-S^{UP}, and MVTO-S^{CRC} use the updating algorithm ODTB to provide a transaction with data that are up-to-date with respect to the start time of the transaction.
- An updating algorithm, ACUA, that takes transient overloades into consideration. Data items are split into data items that must be up-to-date in order for calculations to produce acceptable results (these data items

are denoted *required*) and data items that may be stale (these data items are denoted *not required*). The updating algorithm detects transient overloads and switches between considering all or required data items.

Besides these functionalities of the database, the thesis presents mathematical models of mean time between updates in a system using periodic tasks that each issues updates on-demand. The mathematical models are shown to reduce pessimism in schedulability tests compared to other known models.

In Chapter 3 five requirements were stated that a data management of an embedded system should meet. These requirements were based on the experience of our industrial partners that develop embedded systems. We have seen that the results in chapters 4, 5, 6, 7, and 8 address requirements R3, R4, and R5. Requirement R2, monitoring data and reacting on events, was deemed too time-consuming to implement in DIESIS and also addressing R3–R5.[1]

Requirement R1, organize and maintain data efficiently, is indirectly addressed by using the concept of a real-time database, as we have stated already in Chapter 3. We have not conducted any measurements whether the use of, e.g., DIESIS, gives any benefits in terms of development and maintenance efficiency. However, since databases have successfully been used during several decades to handle large amounts of data we believe databases can address requirement R1 also for embedded systems. Thus, our conclusions are that a real-time database can be used to manage data such that the development and maintenance efforts of software are reduced. The database should contain specific functionality related to the application domain and for embedded systems this functionality is: updating of data, snapshots of data and overload handling.

Since the requirements R1–R5 have been addressed, also the goals G1–G3 have been met.

10.2 Discussions

The basis in reality of the performance evaluations in chapters 4–8 depend on the data and transaction model being used. We have had access to an engine control software that is the basis for the data and transaction model used. Thus, we believe that the results presented in this thesis are valid and representative with respect to a control unit in a vehicle. However, as we have discussed in Chapter 1, embedded systems are usually controlling systems that control the environment they are installed in and control units in vehicles belong to this family of systems. Thus, we believe our results are generalizable to a larger set of systems where similarity can be applied on a subset of the data items in the

[1]In collaboration with Aleksandra Tešanović and Master Thesis student Ying Du, requirement R2 has been addressed in papers [59, 124]. The COMET database was used, which was part of Tešanović's PhD work, instead of DIESIS. The reason for this decision was mainly that the papers [59, 124] address both active behavior in a database and implementing it using aspect-oriented programming. COMET was better suited for this. Due to that another database was used and that these papers were co-authored they are not included in the thesis.

system. In addition to this, mathematical models presented in Chapter 6 model both the use of absolute validity intervals and similarity so these models enlarge the set of systems covered in the thesis further.

Also, it is worth noting that the performance evaluations have been executed in the following systems: (i) RADEx++, (ii) engine control system, (iii) DIESIS, (iv) real-time system simulator implemented in Python, and (v) real-time system simulator implemented in Java. In addition, different instantiations of the data and transaction model have been used in these five systems. The found performance results have been consistent throughout the five systems, which indicate that they are correct.

In retrospect, is a database the best approach for efficient data management in an embedded system? The list of commercial alternatives given in Section 1.2 is short and does not really contain any viable alternatives for resource constrained embedded systems with (soft) real-time constraints. This fact works against the choice of using a database approach since commercial actors seem to have ruled it out. However, on the other hand embedded systems are becoming more and more software intensive, which increases the need for data management tools. Thus, the fact that there is a small number of commercial databases for embedded systems might depend on a small need, but the need would probably increase over time as the systems become more software intensive. This thesis suggests which functionality future databases for embedded systems should have and how they could be implemented.

10.3 Future Work

The CPU overhead of the algorithms have not been thoroughly measured. One problem is that the collection of statistical data is consuming CPU resources, i.e., the probe effect, and, thus, gives a negative effect on the performance of the algorithms. Hence, it is difficult to measure the true overhead of the algorithms in DIESIS. Furthermore, it is not clear to what the overhead costs should be compared. The original EECU software, for instance, stores data using registers and using no concurrency control and no transactions. The CPU overhead of DIESIS 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.

Related to overhead is also the question whether all data items in DIESIS should be subject to the updating scheme and the updating algorithm that are compiled into DIESIS. The data items could be divided into two sets: one containing the data items that the system can efficiently update using similarity and the other set containing those data items that are updated most efficiently by time-triggered tasks. DIESIS should be able to move data items between the two sets at run-time.

The admission control algorithm ACUA-RBound has been tested with the concurrency control algorithm HP2PL. For applications where both admission control and snapshots of data are important, MVTO-S must be able to function with ACUA-RBound. We must investigate how snapshots are affected by dividing data items into mandatory and optional data items.

The similarity-aware multiversion concurrency control algorithms proposed in this thesis store a new version when the write timestamp is larger than the timestamp of the oldest active transaction. Further optimizations can be made if many versions are not needed by the transactions. Other policies 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.

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, DIESIS 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.

BIBLIOGRAPHY

[1] Arcticus AB homepage. http://www.arcticus.se.

[2] Autosar light version v1.5f. http://www.autosar.org/download/AUTOSAR_Light%20Version_V1_5_f.pdf.

[3] CRC16. http://www.ptb.de/en/org/1/11/112/infos/crc16.htm. Url-date: 2004-10-22.

[4] The mathworks homepage. http://www.mathworks.com.

[5] Nist/sematech e-handbook of statistical methods: Kolmogorov-smirnov goodness-of-fit test. http://www.itl.nist.gov/div898/handbook/eda/section3/eda35g.htm.

[6] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions: a performance evaluation. *ACM Transactions on Database Systems (TODS)*, 17(3):513–560, 1992.

[7] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying update streams in a soft real-time database system. In *Proceedings of the 1995 ACM SIGMOD*, pages 245–256, 1995.

[8] B. Adelberg, B. Kao, and H. Garcia-Molina. Database support for efficiently maintaining derived data. In *Extending Database Technology*, pages 223–240, 1996.

[9] Q. N. Ahmed and S. V. Vbrsky. Triggered updates for temporal consistency in real-time databases. *Real-Time Systems*, 19:209–243, 2000.

[10] K. Albers and F. Slomka. An event stream driven approximation for the analysis of real-time systems. In *Proceedings of the 16th Euromicro*

Conference on Real-Time Systems (ECRTS04), pages 187–195. IEEE Computer Society Press, 2004.

[11] K. Albers and F. Slomka. Efficient feasibility analysis for real-time systems with edf scheduling. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 492–497, Washington, DC, USA, 2005. IEEE Computer Society.

[12] M. Amirijoo, J. Hansson, and S. H. Son. Algorithms for managing QoS for real-time data services using imprecise computation. In *Proceedings of the Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA)*, 2003.

[13] M. Amirijoo, J. Hansson, and S. H. Son. Error-driven QoS management in imprecise real-time databases. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2003.

[14] M. Amirijoo, J. Hansson, and S. H. Son. Specification and management of QoS in imprecise real-time databases. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)*, 2003.

[15] P. Atzeni, S. Ceri, S. Paraboschi, and R. Torlone. *Database Systems Concepts, Languages and Architectures*. The McGraw-Hill Companies, 1999.

[16] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Incorporating unbounded algorithms into predictable real-time systems. Technical report, Real-Time Systems Research Group, Department of Computer Science, University of York, 1993.

[17] E. Barry, S. Slaughter, and C. F. Kemerer. An empirical analysis of software evolution profiles and outcomes. In *ICIS '99: Proceeding of the 20th international conference on Information Systems*, pages 453–458, Atlanta, GA, USA, 1999. Association for Information Systems.

[18] S. K. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, (24):93–128, 2003.

[19] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Publishing Company, 1987.

[20] J. A. Blakeley, N. Coburn, and P.-Å. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems*, 14(3):369–400, 1989.

[21] G. Blom. *Sannolikhetsteori och statistikteori med tillämpningar. Bok C.* Studentlitteratur, 1989.

[22] M. Broy. Challenges in automotive software engineering: From demands to solutions. `http://www.softwareresearch.net/site/other/EmSys03/docs/12.Broy.pdf`.

[23] M. Broy. Automotive software engineering. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 719–720, Washington, DC, USA, 2003. IEEE Computer Society.

[24] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1997.

[25] G. C. Buttazzo. Rate monotonic vs. edf: Judgment day. *Real-Time Systems*, 29(1):5–26, 2005.

[26] A. Cervin and J. Eker. Control-scheduling codesign of real-time systems: The control server approach. *Journal of Embedded Computing*, 1(2):209–224, 2005.

[27] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, Oct. 1989.

[28] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Journal of Real-Time Systems*, (2), 1990.

[29] A. Chiu, B. Kao, and K.-Y. Lam. Comparing two-phase locking and optimistic concurrency control protocols in multiprocessor real-time databases. In *Proceedings of the Joint Workshop on Parallel and Distributed Real-Time Systems*, 1997.

[30] A. Colin and G. Bernat. Scope-tree: A program representation for symbolic worst-case execution time analysis. In *ECRTS '02: Proceedings of the 14th Euromicro Conference on Real-Time Systems*, page 50, Washington, DC, USA, 2002. IEEE Computer Society.

[31] V. Consultants. Quality crisis hits the embedded software industry. `http://www.electronicstalk.com/news/vru/vru102.html`.

[32] J. E. Cooling. *Software Engineering for Real-Time Systems*. Addison-Wesley Publishing Company, 2003.

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. The MIT Press, 2 edition, 2001.

[34] E. Coskun and M. Grabowski. Complexity in embedded intelligent real time systems. In *ICIS '99: Proceeding of the 20th international conference on Information Systems*, pages 434–439, Atlanta, GA, USA, 1999. Association for Information Systems.

[35] D. R. Cox and W. L. Smith. The superposition of several strictly periodic sequences of events. *Biometrika*, 40(1/2):1–11, June 1953.

[36] I. Crnkovic and M. Larsson, editors. *Building reliable component-based software systems*. Artech House, 2002.

[37] D. P. Darcy and C. F. Kemerer. Software complexity: Toward a unified theory of coupling and cohesion. In *Proceedings of ICSC Workshop 2002*, 2002.

[38] A. Datta, S. Mukherjee, P. Konana, I. R. Viguier, and A. Bajaj. Multiclass transaction scheduling and overload management in firm real-time database systems. *Information Systems*, 21(1):29–54, 1996.

[39] A. Datta and I. R. Viguier. Providing real-time response, state recency and temporal consistency in databases for rapidly changin environments. Technical report, TimeCenter, 1997.

[40] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The hipac project: combining active databases and timing constraints. *SIGMOD Rec.*, 17(1):51–70, 1988.

[41] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: disseminating dynamic web data. In *Proceedings of the tenth international conference on World Wide Web*, pages 265–274. ACM Press, 2001.

[42] P. Deutsch and J.-L. Gailly. Rfc 1950 - zlib compressed data format specification version 3.3. `http://www.faqs.org/rfcs/rfc1950.html`, 1996.

[43] U. Devi. An improved schedulability test for uniprocessor periodic task systems. In *Proceedings of the 15th Euromicro conference on Real-Time Systems (ECRTS03)*, pages 23–30. IEEE Computer Society Press, 2003.

[44] N. R. Draper and H. Smith. *Applied Regression Analysis*. John Wiley & Sons, 1998.

[45] U. Eklund, Ö. Askerdal, J. Granholm, A. Alminger, and J. Axelsson. Experience of introducing reference architectures in the development of automotive electronic systems. In *SEAS '05: Proceedings of the second international workshop on Software engineering for automotive systems*, pages 1–6, New York, NY, USA, 2005. ACM Press.

[46] Encirq. DeviceSQL. `http://www.encirq.connectthe.com`.

[47] M. Eriksson. Efficient data management in engine control software for vehicles - development of a real-time data repository. Master's thesis, Linköping University, Feb 2003.

[48] M. R. Garey and D. S. Johnson. *Computers and Intractability A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[49] O. Goldsmith, D. Nehme, and G. Yu. Note: On the set-union knapsack problem. *Naval Research Logistics*, 41:833–842, 1994.

[50] A. Göras, S.-A. Melin, and J. Hansson. Inbyggda realtidsdatabaser för motorstyrning. Technical report, Linköping University, 2001.

[51] G. R. Goud, N. Sharma, K. Ramamritham, and S. Malewar. Efficient real-time support for automotive applications: a case study. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA06)*, 2006.

[52] M. H. Graham. How to get serializability for real-time transactions without having to pay for it. In *Proceedings of the Real-Time Systems Symposium 1993*, pages 56–65, 1993.

[53] T. Gustafsson, H. Hallqvist, and J. Hansson. A similarity-aware multiversion concurrency control and updating algorithm for up-to-date snapshots of data. In *ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 229–238, Washington, DC, USA, 2005. IEEE Computer Society.

[54] T. Gustafsson and J. Hansson. Scheduling of updates of base and derived data items in real-time databases. Technical report, Department of computer and information science, Linköping University, Sweden, 2003.

[55] T. Gustafsson and J. Hansson. Data management in real-time systems: a case of on-demand updates in vehicle control systems. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, pages 182–191. IEEE Computer Society Press, 2004.

[56] T. Gustafsson and J. Hansson. Dynamic on-demand updating of data in real-time database systems. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 846–853. ACM Press, 2004.

[57] T. Gustafsson and J. Hansson. Data freshness and overload handling in embedded systems. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA06)*, pages 173–182. IEEE Computer Society Press, 2006.

[58] T. Gustafsson and J. Hansson. On the estimation of cpu utilization of real-time systems. Technical report, Department of Computer and Information Science, Linköping University, Sweden, 2006.

[59] T. Gustafsson and J. Hansson. Performance evaluations and estimations of workload of on-demand updates in soft real-time systems. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA07). To appear*, 2007.

[60] T. Gustafsson, J. Hansson, A. Göras, J. Gäddevik, and D. Holmberg. 2006-01-0305: Database functionality in engine management system. *SAE 2006 Transactions Journal of Passenger Cars: Electronic and Electrical Systems*, 2006.

[61] T. Gustafsson, A. Tešanović, Y. Du, and J. Hansson. Engineering active behavior of embedded software to improve evolution and performance: an aspect-oriented approach. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 673–679. ACM Press, 2007.

[62] H. Hallqvist. Data versioning in a real-time data repository. Master's thesis, Linköping University, 2004.

[63] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m, k)-firm deadlines. *IEEE Transactions on Computers*, 44(12):1443–1451, December 1995.

[64] J. Hansson. *Value-Driven Multi-Class Overload Management in Real-Time Database Systems*. PhD thesis, Institute of technology, Linköping University, 1999.

[65] J. R. Haritsa, M. J. Carey, and M. Livny. On being optimistic about real-time constraints. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 331–343. ACM Press, 1990.

[66] J. R. Haritsa, M. Livny, and M. J. Carey. Earliest deadline scheduling for real-time database systems. In *IEEE Real-Time Systems Symposium*, pages 232–243. IEEE Computer Society Press, 1991.

[67] H. Heinecke, K.-P. Schnelle, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Maté, K. Nishikawa, and T. Scharnhorst. AUTomotive Open System ARchitecture - an industry-wide initiative to manage the complexity of emerging automotive e/e-architectures. In *Proceedings of Convergence*, number SAE-2004-21-0042, 2004.

[68] S.-J. Ho, T.-W. Kuo, and A. K. Mok. Similarity-based load adjustment for real-time data-intensive applications. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, pages 144–154. IEEE Computer Society Press, 1997.

[69] W. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quartely*, (21), 1974.

[70] J. Huang, J. A. Stankovic, and K. Ramamritham. Experimental evaluation of real-time optimistic concurrency control schemes. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 35–46, September 1991.

[71] S. S. Inc. `http://www.sleepycat.com`.

[72] K.-D. Kang, S. H. Son, and J. A. Stankovic. Managing deadline miss ratio and sensor data freshness in real-time databases. *IEEE Transactions on Knowledge and Data Engineering*, 2003.

[73] K.-D. Kang, S. H. Son, J. A. Stankovic, and T. F. Abdelzaher. A QoS-sensitive approach for timeliness and freshness guarantees in real-time databases. In *Proceedings of the 14th Euromicro International Conference on Real-Time Systems*, pages 203–212. IEEE Computer Society Press, 2002.

[74] B. Kao, K.-Y. Lam, B. Adelberg, R. Cheng, and T. Lee. Maintaining temporal consistency of discrete objects in soft real-time database systems. *IEEE Transactions on Computers*, 2002.

[75] B. Kao, K.-Y. Lam, B. Adelberg, R. Cheng, and T. Lee. Maintaining temporal consistency of discrete objects in soft real-time database systems. *IEEE Transactions on Computers*, 52(3):373–389, March 2003.

[76] Y.-K. Kim and S. H. Son. Supporting predictability in real-time database systems. In *2nd IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, pages 38–48. IEEE Computer Society Press, 1996.

[77] G. Koren and D. Shasha. Skip-over: algorithms and complexity for overloaded systems that allow skips. In *RTSS '95: Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*, page 110, Washington, DC, USA, 1995. IEEE Computer Society.

[78] R. kr. Majumdar, K. M. Moudgalya, and K. Ramamritham. Adaptive coherency maintenance techniques for time-varying data. In *Proceedings of the 24th Real-Time Systems Symposium (RTSS'03)*, pages 98–107. IEEE Computer Society Press, 2003.

[79] C. M. Krishna and K. G. Shin. *Real-Time Systems*. The McGraw-Hill Companies, 1997.

[80] H. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[81] T.-W. Kuo and A. K. Mok. Application semantics and concurrency control of real-time data-intensive applications. In *Proceedings of IEEE 13th Real-Time Systems Symposium*, pages 35–45. IEEE Computer Society Press, 1992.

[82] T.-W. Kuo and A. K. Mok. Real-time data semantics and similarity-based concurrency control. *IEEE Transactions on Computers*, 49(11):1241–1254, November 2000.

[83] A. Labrinidis and N. Roussopoulos. Update propagation strategies for improving the quality of data on the web. In *The VLDB Journal*, pages 391–400, 2001.

[84] J. J. Labrosse. *MicroC/OS-II The Real-Time Kernel Second Edition*. CMPBooks, 2002.

[85] K.-Y. Lam, T.-W. Kuo, B. Kao, T. S. Lee, and R. Cheng. Evaluation of concurrency control strategies for mixed soft real-time database systems. *Information Systems*, 27:123–149, 2002.

[86] K.-Y. Lam and W.-C. Yau. On using similarity for concurrency control in real-time database systems. *The Journal of Systems and Software*, (43):223–232, 2000.

[87] S. Lauzac, R. Melhem, and D. Mossé. An improved rate-monotonic admission control and its applications. *IEEE Transactions on Computers*, 52(3):337–350, 2003.

[88] C.-G. Lee, Y.-K. Kim, S. Son, S. L. Min, and C. S. Kim. Efficiently supporting hard/soft deadline transactions in real-time database systems. In *Third International Workshop on Real-Time Computing Systems and Applications, 1996.*, pages 74–80, 1996.

[89] E. A. Lee. What's ahead for embedded software? *Computer*, pages 18–26, 2000.

[90] K. Lee and S. Park. Classification of weak correctness criteria for real-time database applications. In *Proceedings of the 20th International Computer Software and Applications Conference 1996 (COMPSAC'96)*, pages 199–204, 1996.

[91] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[92] J. W. S. Liu, K.-J. Lin, W.-K. Shih, A. C. shi Yu, J.-Y. Chung, and W. Zhao. Algorithms for scheduling imprecise computations. *Computer*, 24(5):58–68, 1991.

[93] D. Locke, L. Sha, R. Rajikumar, J. Lehoczky, and G. Burns. Priority inversion and its control: An experimental investigation. In *IRTAW '88: Proceedings of the second international workshop on Real-time Ada issues*, pages 39–42, New York, NY, USA, 1988. ACM Press.

[94] C. Lu, J. A. Stankovic, and S. H. Son. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems*, 23(1–2):86–126, 2002.

[95] R. Majumdar, K. Ramamritham, R. Banavar, and K. Moudgalya. Disseminating dynamic data with qos guarantee in a wide area network: A practical control theoretic approach. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Synmposium (RTAS'04)*, pages 510–517. IEEE Computer Society Press, May 2004.

[96] McObject LLC. eXtremeDB. `http://www.mcobject.com/`.

[97] P. Mejia-Alvarez, R. Melhem, and D. Mossé. An incremental approach to scheduling during overloads in real-time systems. In *Proceedings of the 21th Real-Time Systems Symposium (RTSS'00)*, pages 283–293. IEEE Computer Society Press, 2000.

[98] Microsoft. Microsoft SQL Server 2005 Compact Edition. `http://www.microsoft.com/downloads/details.aspx?FamilyId=%2085E0C3CE-3FA1-453A-8CE9-AF6CA20946C3&displaylang=en`.

[99] L. Nielsen and L. Eriksson. *Course Material Vehicular Systems*. Linköping Institute of Technology, Vehicular Systems, ISY, 2001.

[100] L. Nielsen and U. Kiencke. *Automotive Control Systems For Engine, Driveline, and Vehicle*. Springer-Verlag, 1999.

[101] D. Nyström, A. Tešanović, C. Norström, J. Hansson, and N.-E. B. nkestad. Data management issues in vehicle control systems: a case study. In *Proceedings of the 14th Euromicro International Conference on Real-Time Systems*, pages 249–256, Vienna, Austria, June 2002. IEEE Computer Society Press.

[102] M. A. Olson. Selecting and implementing an embedded database system. *IEEE Computer*, 2000.

[103] C. Olston, B. T. Loo, and J. Widom. Adaptive precision setting for cached approximate values. *SIGMOD Rec.*, 30(2):355–366, 2001.

[104] C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1987.

[105] R. Pellizzoni. Efficient feasibility analysis of real-time asynchronous task sets. Master's thesis, Scuola Superiore S. Anna, Pisa, Italy, 2004.

[106] J. Philip J. Koopman. Embedded system design issues (the rest of the story). In *Proceedings of the International Conference on Computer Design (ICCD 96)*, 1996.

[107] C. Pu and A. Leff. Epsilon-serializability. Technical report, Department of Computer Science, Columbia University, 1991.

[108] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1(2):199–226, 1993.

[109] K. Ramamritham, P. Deolasee, A. Katkar, A. Panchbudhe, and P. Shenoy. Dissemination of dynamic data on the internet. In *Proceedings of Databases in Networked Information Systems, International Workshop (DNIS) 2000*, pages 173–187, 2000.

[110] K. Ramamritham, S. H. Son, and L. C. DiPippo. Real-time databases and data services. *Real-Time Systems*, (28):179–215, 2004.

[111] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, (26):161–197, 2004.

[112] J. T. Robinson. *Design of concurrency controls for transaction processing systems*. PhD thesis, 1982.

[113] A. Sen and M. Srivastava. *Regression Analysis Theory, Methods, and Applications*. Springer-Verlag, 1990.

[114] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28:101–155, 2004.

[115] S. Shah, K. Ramamritham, and P. Shenoy. Resilient and coherence preserving dissemination of dynamic data using cooperating peers. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):799–812, 2004.

[116] L. Shu and M. Young. Versioning concurrency control for hard real-time systems. *The Journal of Systems and Software*, (63):201–218, 2002.

[117] X. Song and J. W. Liu. Maintaining temporal consistency: Pessimistic vs. optimistic concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):786–796, 1995.

[118] SPSS. SPSS regression models. `http://www.spss.com/regression/`.

[119] J. A. Stankovic, M. Spuri, M. D. Natale, and G. C. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6):16–25, 1995.

[120] J. Stone, M. Greenwald, C. Partridge, and J. Hughes. Performance of checksums and CRC's over real data. *IEEE/ACM Transactions on Networking*, 6(5):529–543, 1998.

[121] R. Sun and G. Thomas. Performance results on multiversion timestamp concurrency control with predeclared writesets. In *Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 177–184. ACM Press, 1987.

[122] H. Sundell. *Ecient and Practical Non-Blocking Data Structures*. PhD thesis, Chalmers University of Technology, 2004.

[123] H. Sundell and P. Tsigas. Simple wait-free snapshots for real-time systems with sporadic tasks. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applicatins (RTCSA04)*, 2004.

[124] A. Tešanović, T. Gustafsson, and J. Hansson. Separating active and on-demand behavior of embedded systems into aspects. In *Proceedings of the International Workshop on Non-functional Properties of Embedded Systems (NFPES'06)*, 2006.

[125] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Embedded databases for embedded real-time systems: a component-based approach. Technical report, Department of Computer Science, Linköping University, 2002.

[126] M. E. Thomadakis and J.-C. Liu. Linear time on-line feasibility testing algorithms for fixed-priority, hard real-time systems. Technical Report TR00-006, 26, 2000.

[127] S. Tomic, S. V. Vrbsky, and T. Camp. A new measure of temporal consistency for derived objects in real-time database systems. *Inf. Sci. Inf. Comput. Sci.*, 124(1-4):139–152, 2000.

[128] S. V. Vrbsky. A data model for approximate query processing of real-time databases. *Data Knowl. Eng.*, 21(1):79–102, 1996.

[129] S. V. Vrbsky and J. W. S. Liu. Approximate: a query processor that produces monotonically improving approximate answers. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):1056–1068, 1993.

[130] S. V. Vrbsky and S. Tomic. Satisfying temporal consistency constraints of real-time databases. *J. Syst. Softw.*, 45(1):45–60, 1999.

[131] H. F. Wedde, S. Böhm, and W. Freund. Adaptive concurrency control in distributed real-time systems. Technical report, University of Dortmund, Lehrstuhl Informatik 3, 2000.

[132] E. W. Weisstein. Correlation coefficient. From MathWorld – A Wolfram Web Resource. `http://mathworld.wolfram.com/CorrelationCoefficient.html`.

[133] R. N. Williams. A painless guide to CRC error detection algorithms. ftp://ftp.rocksoft.com/papers/crc_v3.txt.

[134] M. Xiong, S. Han, and D. Chen. Deferrable scheduling for temporal consistency: Schedulability analysis and overhead reduction. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA06)*, pages 117–124, 2006.

[135] M. Xiong, S. Han, and K.-Y. Lam. A deferrable scheduling algorithm for real-time transactions maintaining data freshness. In *Proceedings of the 26th IEEE Real-Time Systems Symposium, 2005. RTSS 2005*, 2005.

[136] M. Xiong and K. Ramamritham. Deriving deadlines and periods for real-time update transactions. In *Proceedings of the 20th Real-Time Systems Symposium*, pages 32–43. IEEE Computer Society Press, 1999.

[137] M. Xiong, R. Sivasankaran, J. Stankovic, K. Ramamritham, and D. Towsley. *Scheduling access to temporal data in real-time databases*, chapter 1, pages 167–192. Kluwer Academic Publishers, 1997.

[138] J. Zweig and C. Partridge. Rfc 1146 - TCP alternate checksum options. http://www.faqs.org/rfcs/rfc1146.html, 1990.

APPENDIX A

Abbreviations and Notation

Symbol	Denotes
D	Set of (derived) data items
B	Set of base items
d_i^j	Version j of data item i
d_i	The current version of data item i
b_j	Base item corresponding to a sensor value
$V(d_i)$	Set of all versions of d_i
$V(\boldsymbol{x})$	Variance of a set of values
$R(d_i)$	Set of data items read when deriving d_i
$\#R(d_i) = \|R(d_i)\|$	Number of members in the read set
$RR(d_i)$	Subset of read set containing required data items
$NRR(d_i)$	Subset of read set containing not required data items. Note that $NRR(d_i) \cap RR(d_i) = \emptyset$
$PAR(d_i)$	Set of currently potentially affected members of the read set of d_i
$PAA(d_i)$	Set of currently potentially affected ancestors of d_i
$A(d_i)$	Set of currently affected ancestors of d_i
$G = (N, E)$	Data dependency graph
τ	Task or transaction in a real-time task
$\tau_{d_i}^{value}$	Task using on-demand updating measuring data freshness in value domain of data item d_i
	Continues on next page.

Continued from previous page

Symbol	Denotes
$\tau_{d_i}^{time}$	Task using on-demand updating measuring data freshness in time domain of data item d_i
$dt(\tau)$	Relative deadline of transaction τ
$ts(\tau)$	Unique timestamp of transaction τ
$gvts$	Global virtual timestamp
$wt(d_i^j)$	Write timestamp of version j of d_i
$v_{d_i}^t$	Value of d_i at time t
$pa(d_i)$	Timestamp when latest version of d_i was affected by any $d_k \in R(d_i)$
$change(d_i)$	Binary marking which is true if d_i is potentially affected and false if d_i is not potentially affected by a change in an ancestor
$fixedint_{d_i}(v_{d_i}^t)$	Interval number of d_i at time t
$level(d_i)$	The level of data item d_i
$avi(d_i)$	Allowed age of d_i's value.
δ_{b_j,d_i}	Allowed changes of b_j with respect to d_i. If δ_{b_j,d_i} equal for all d_i being immediate children of b_j the denote it δ_{b_j}
$wcet(\tau)$	WCET of task excluding on-demand update
$wcet(d_i) = wcet(\tau_{d_i})$	WCET of update of d_i
$wcet_i(b_j)$	1 if task i has an on-demand update of b_j, otherwise $wcet_i(b_j) = 0$
$update\_d_i$	Update of d_i
$code(update\_d_i)$	The code of the update of d_i
$wcet(update\_d_i)$	WCET of update of d_i
$wcet(code(update\_d_i))$	WCET of update of d_i
$period(\tau)$	Period time of τ
$MTBI$	Mean time between invocations
MIT	Mean interarrival time
$MTBI(\mathbf{P})$	MTBI of executions of periodic tasks in set $\mathbf{P}$
$MTBI(y, \mathbf{P})$	MTBI of executions of periodic tasks in set $\mathbf{P}$ if they are not started if the latest execution was at maximum y time units ago

APPENDIX B

On-Demand Updating Algorithms in Value-Domain

This appendix discusses algorithms and shows performance results that are related to results presented in Chapter 4. Algorithm descriptions and performance evaluations are presented in this appendix in order not to clutter the text of Chapter 4 too much. The outline of this appendix is as follows. Section B.1 describes the updating schemes AUS and PAUS that are being used by top-bottom and bottom-up on-demand updating algorithms respectively (see Section 4.5.3 for ODTB and Section B.3 for ODDFT). Section B.2 discusses why a binary marking of stale data items is not sufficient for correctly marking stale data items. Sections B.3–B.6 give a detailed description of the updating algorithms ODDFT, ODBFT, ODKB_C, and ODTB. Section B.7 describes the functionality of the auxiliary functions BeginTrans, ExecTrans, and AssignPrio that are used by the RADEx++ simulator in Chapter 4. Remember that DIESIS is also used to evaluate performance of algorithms, and the performance evaluations using DIESIS are described in Section 5.5 and in Appendix C. Finally, Section B.8 describes the results of experiments 1b, 1c, 1e, 2a, and 2b. Experiments 1a and 1d are described in Section 4.5.5.

B.1 Updating Schemes

This section describes the AUS and the PAUS updating schemes that are used by the on-demand updating algorithms.

The reason there are two updating schemes is that the data dependency graph *G* can be traversed either top-bottom or bottom-up. AUS should be used together with top-bottom algorithms, e.g., ODTB, and PAUS should be used together with bottom-up algorithms, e.g., ODDFT. The steps of AUS are.

- AUS_S1: Update base items periodically in order to keep an up-to-date view of the external environment.
- AUS_S2: Mark immediate children of a data item that is found to be stale when it is updated.
- AUS_S3: Determine which data items should be updated before a user transaction starts its execution. This must be done by traversing the data dependency graph, G, top-bottom since step AUS_S2 only marks the immediate children. An example of an updating algorithm that could be used is ODTB.

The steps of PAUS are.

- PAUS_S1: Update base items periodically in order to keep an up-to-date view of the external environment.
- PAUS_S2: Mark all descendants of the data item that is found to be stale when it is updated.
- PAUS_S3: Determine which data items should be updated before a user transaction starts its execution. This can be done by a bottom-up traversal of G since all descendants are marked as potentially marked by a change and are therefore found in a bottom-up traversal. An example of an algorithm is ODDFT.

The reason top-bottom algorithms may be better than bottom-up algorithms is that the scheduling algorithm and the updating scheme can be implemented in a more efficient way, because the marking of (potentially) stale data items is, generally, performed on a less number of data items using AUS compared to using PAUS, because immediate children instead of all descendants are marked.

Performance results using these marking techniques are evaluated in Section 4.5.5 and Section B.8.

B.2 Binary marking of stale data items

This section discusses why the pa timestamp is needed and why using the simpler approach of binary flags may fail.

The solution that first comes to mind to mark whether a data item is potentially affected by a change in an ancestor is to use a binary flag taking the values false, i.e., the data item is not potentially affected, and true, i.e., the data item is potentially affected. Using the PAUS updating scheme and binary flags, there are cases where PAUS would set the marking to false where it instead should remain set at true. This is illustrated with the example below.

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

Figure B.1: The ODDFT algorithm.

Example B.1. *Consider data item d_7 in Figure 2.6 and $change(d_7)$ is the binary marking of d_7 and it equals true. A triggered update τ_{d_7} has started to update d_7 because $change(d_7)$ equals true (and therefore $pa(d_7) > 0$). Assume a UT starts for another data item in the system and the UT has a higher priority than τ_{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 $change$ flag, when τ_{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 ancestors, which is not correct.*

The example above shows that the pa timestamp should be used when using the PAUS updating scheme.

B.3 Bottom-Up Traversal: Depth-First Approach

This section describes the on-demand depth-first traversal (ODDFT) algorithm that implements the on-demand scheduling of updates in step PAUS_S3.

The ODDFT algorithm is given in Figure B.1. The input parameters are d, the data item that might be included in the schedule, t, the release time of an update, $UTrt$, which is the release time of the UT, 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 arrived UT. The variable $blockingf$ on line 7 is used to regard interruptions from higher prioritized transactions and their updates.

Section 4.5.1 gives four algorithmic steps of ODDFT. They 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, (iii) put needed updates in a schedule, and (iv) execute the updates in the schedule. Algorithmic step (i) is implemented by the for-loop on line 1 and the recursive call on line 9. The prioritized order is determined by algorithm AssignPrio that is described in Section B.7. Algorithmic step (ii) is implemented with the if-statement on line 2, where $error(x, freshness\_deadline)$ 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. Algorithmic step (iv) only involves taking the top entry of the schedule and start a TU of the data item represented in that entry.

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.

The attentive reader may have seen that the depth-first traversal in ODDFT is not implemented with a check if a node in G has already been visited. The reason is that data items needing an update must be put in the schedule in an order that obeys data relationships in G. So, if ODDFT has scheduled ancestors of data item d_i, where $d_i \in R(d_{UT})$, then when scheduling ancestors of $d_j \in R(d_{UT}), d_i \neq d_j$, the same ancestors may be found. These ancestors must be put before d_j in the schedule. One way to achieve this is to put them in the schedule and remove the duplicates (line 6). However, checking for duplicates is a computational expensive task since the whole schedule $S_{d_{UT}}$ must be traversed.

ODDFT is now described by an example using G in Figure 2.6.

Example B.2. *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 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 τ_{d_7} is put in the schedule followed by updates for d_2, d_3, and d_4 according to a priorization. Finally, the algorithm continues with the d_8 branch and τ_{d_8} is put in the schedule followed by τ_{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 B.2. 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 τ_{d_9} the algorithm is terminated since no more updates can be executed (assuming WCETs on updates).*

Computational Complexity

Since ODDFT is a recursive algorithm possibly traversing all branches from a node in G, the computational complexity can be derived in the same way as for

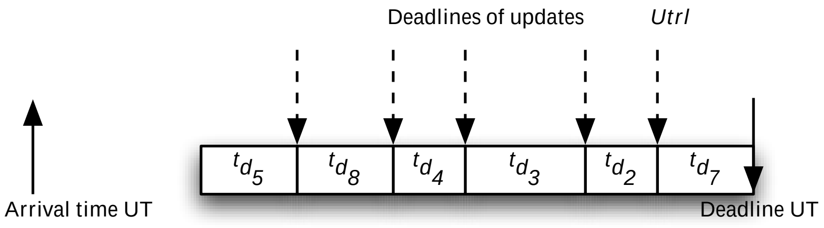


Figure B.2: A schedule of updates generated by ODDFT.

OD. Consider the graph in Figure 4.5 and the following recurrence relation:

$$\begin{cases} T(n) & = mT(n+1) + O(m \log m) \\ T(k) & = 1 \end{cases}$$

where $O(m \log m)$ is the running time of an algorithm that prioritizes nodes (AssignPrio described in Section B.7) 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 4.5, 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.

B.4 Bottom-Up Traversal: Breadth-First Approach

This section describes the ODBFT algorithm that implements the on-demand scheduling of updates in the PAUS_S3 step. ODBFT has four algorithmic steps. 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, and step (iv) is to execute updates of the scheduled data items in the order they appear in the schedule. Algorithmic step (i) is described now. The breadth-first algorithm (see [33]) 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 $\sqsupset$ is introduced, and $x \sqsupset y$, where x and y are data items in the database, is defined as:

$$\begin{aligned} x \sqsupset y \text{ iff } & level(x) > level(y) \vee \\ & (level(x) = level(y) \wedge Sprio(x) > Sprio(y)) \vee \\ & (level(x) = level(y) \wedge Sprio(x) = Sprio(y) \wedge id(x) > id(y)), \end{aligned}$$

where $Sprio$ is the product of the priority of the data item and the weight, $level$ is the level the data item resides at (see definition 3.2.1), 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 $d_5 \sqsupset d_4$ if they reside in the same level and are assigned the same priority by algorithm AssignPrio.

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 $\sqsupset$. The head of Q is used by ODBFT to start a new search for undiscovered nodes in the graph. Since $\sqsupset$ 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.

The ODBFT algorithm is described in Figure B.3. The input parameters are the arrived UT τ, t initially set to the time $dt(\tau) - wcet(\tau)$, 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 arrived 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 B.7). The for-loop on line 14 cycles through all immediate 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 2.6.

Example B.3. *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. Ancestors 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 ancestor 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 $\sqsupset$ sorts*

```
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:      $dt(\tau_u) = t$
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:     $priority\_queue = AssignPrio(u, t)$
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** $pa(v) > 0 \wedge error(v, freshness\_deadline) > \delta_{d,v}$ **then**
18:                 Put $v$ in $Q$ sorted by relation $\sqsupset$
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 B.3: The ODBFT algorithm.

these items to be placed after d_8, that still resides in Q, because $\sqsupset$ 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 ancestor 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 determines their order. None of d_2–d_5 has derived data items as immediate 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 $[\tau_{d_2}\tau_{d_3}\tau_{d_4}\tau_{d_5}\tau_{d_8}\tau_{d_7}\tau_{d_6}]$.

Computational Complexity

The total running time of algorithm ODBFT is $O(|N| + |E|)$ if the operations for enqueuing and dequeuing Q take $O(1)$ time [33]. In algorithm ODBFT, the enqueuing takes in the worst-case $O(\log |N|)$ since the queue can be kept sorted and elements are inserted in the sorted queue. The total running time of algorithm AssignPrio 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 $O(|N| \log |N| + |E| \log p)$.

B.5 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 (see Section 4.5). Before a scheduled update is execute is a check performed if the current value of the data item being updated is affected by any changes in its immediate parents in G. If the data item is unaffected by any such changes, the update is not triggered. Hence, a relevance check (see definition 4.3.2) 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 update generated by ODKB_V, and ODKB_V has polynomial time complexity.

B.6 Top-Bottom Traversal: ODTB With Relevance Check

This section gives a detailed description of ODTB. An overview of ODTB can be found in Section 4.5.3. The steps of ODTB are:

(i) Traverse G top-bottom to find affected data items.

(ii) Top-bottom traversal from found affected data items to d_{UT}.

(iii) Execute updates of scheduled data items.

Data dependency graph $G = (N, E)$ describes the relation $<_G$. To obtain a pregenerated schedule that can be used by ODTB, a bottom node is added, denoted *bottom*, to N and all leaf nodes are connected to it by adding edges to E. A schedule is generated using BuildPreGenSchedule, which is described in Figure B.4, for the added *bottom* node[1] and denote it S.

```
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 B.4: The BuildPreGenSchedule algorithm.

Theorem B.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 Build-PreGenSchedule, which origin from the same node, always pick branches in the same order. BuildPreGenSchedule has no memory of which nodes that have already been visited. Hence, the outgoing edge that is picked by BuildPreGen-Schedule 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 B.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 B.6.1. □

[1]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.

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

Figure B.5: Top-Bottom relevance check algorithm (pseudo-code).

By corollary B.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 (immediate parents and immediate 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 canceled.

The ODTB algorithm is shown in Figure B.5. Algorithmic step (i) of ODTB is implemented on lines 2, 3, and 4. The for-loop on line 2 cycles through all immediate parents of d_{UT}, and for every immediate 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 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.

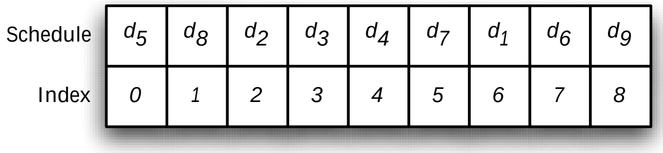


Figure B.6: Pregenerated schedule by BuildPreGenSchedule for G in Figure 2.6.

Example B.4. *A UT τ_{d_7} arrives to a system that has a data dependency graph as given in Figure 2.6. The fixed schedule S is given in Figure B.6, 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 ancestor 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 B.5). If such a data item is found, WCET for the data item u and the remaining data items in S_x, denoted $wcet\_from\_u\_to\_x$, has to fit in the available time $availt$ of τ_{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 canceled 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.*

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 immediate parent of d_{UT}. There are a polynomial number of ancestors 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 $|N|$, 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 ancestors are changed. These two timestamps are a combination of the second step of the updating scheme for ODDFT and ODBFT and the second step for ODTB. Hence, more time is spent marking data items when they change, but when data items do not change a fresh data item can immediately be detected.

B.7 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.

B.7.1 BeginTrans

A database system is notified by a UT when it starts to execute the BeginTrans algorithm. In this section, we only discuss the generation of timestamps and execution of updating algorithms. The algorithm is presented in Figure B.7 and $gvts$ is the global virtual timestamp. As can be seen on line 2, the $gvts$ variable is monotonically increasing implying that UTs get unique timestamps.

```
BeginTrans
1: Begin critical section
2: $gvts = gvts + 1$
3: End critical section
4: $ts(\tau) = gvts$
5: Execute updating algorithm
6: ExecTrans($\tau_{UT}$)
```

Figure B.7: Pseudo-code of the BeginTrans algorithm.

B.7.2 ExecTrans

This section 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 immediate parents. If the update τ_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.

B.7.3 AssignPrio

When the updating algorithms ODDFT, ODBFT, and ODDFT_C can choose from several nodes in G, the AssignPrio algorithm (see Figure B.9), prioritizes the nodes, and the updating algorithm chooses branches in priority order. A function $error(d, t)$ is used in AssignPrio to approximate the error in the stored value of d at time t. This function is application-specific and can look as in Figure B.10 where the error is approximated by how much the value can possibly change during the duration until t. Time t in AssignPrio is the future time at which data items should be fresh for the transaction to derive a fresh value [54]. 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. [75].

B.8 Performance Results

The discrete-event simulator RADEx++ is used to conduct experiments to test the performance of the updating algorithms [64]. A benefit of using a discrete event simulator is that code taking long time to execute but are not part of the algorithms does not affect the performance of the algorithms. The experiments performed in the RADEx++ simulator conform to an absolute consistent system [75] which is defined in Definition 4.5.1. 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, by performing recalculations, the new value of the data item.

```
ExecTrans($\tau$)
 1: for all $x \in S_{d_{UT}}$ 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 (definitions 4.2.1
            or 4.2.2) 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 descendants 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 B.8: Pseudo-code of the ExecTrans algorithm.

```
AssignPrio($d$,$t$)
 1: for all $x \in R(d)$ do
 2:     if $error(x,t) \geq \delta_{d,x}$ then
 3:         $total\_error = total\_error + error(x,t)$
 4:         Put $x$ in queue $Q_1$
 5:     end if
 6: end for
 7: for all $x \in Q_1$ do
 8:     $Sprio(x) = error(x,t)$
 9:     Multiply $Sprio(x)$ with $weight(x)$
10:     Put $x$ in queue $Q_2$ sorted by priority
11: end for
12: Return $Q_2$
```

Figure B.9: The AssignPrio algorithm.

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

Figure B.10: Example of *error* function.

Two experiments can be found in Chapter 4, Experiments 1a and Experiment 1d. In this section are the experiments Experiment 1b, Experiment 1c, Experiment 1e, Experiment 2a, and Experiment 2b presented.

B.8.1 Experiment 1b: Deriving Only Actuator User Transactions

The objective of this experiment is to investigate the performance of a system that only derives data items in leaf nodes. 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 B.11. Figure B.11(a) shows the number of valid committed UTs deriving data items that have at least one derived data item as immediate 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 Experiment 1a, 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 ancestors 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 B.12 showing the number of triggered updates. Comparing these numbers to those in Figure 4.8(b) shows that more updates are needed.

Figure B.11(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.

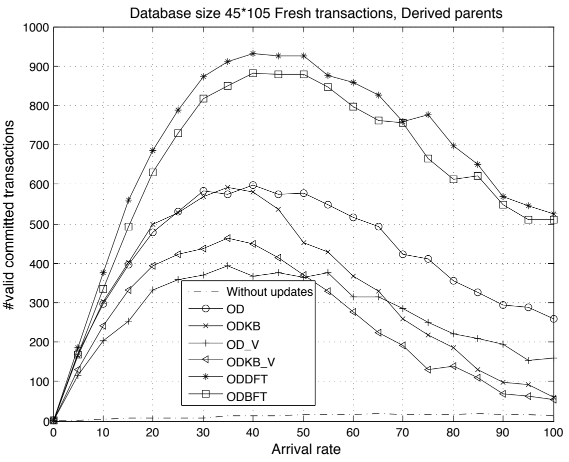
B.8.2 Experiment 1c: Comparison of Using Binary Change Flag or pa Timestamp

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 [54–56], 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 B.1 shows this.

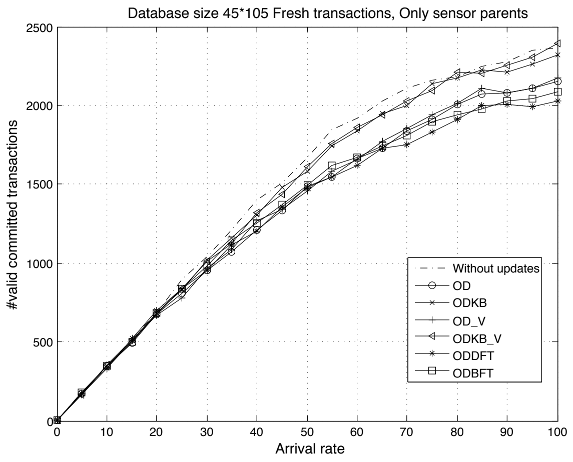
The reason the pa timestamp is used is to correctly determine if an update is needed. This is important when relative consistency of data is considered. The updating algorithms that are implemented in the RADEx++ simulator are implemented with the choice of skipping late updates. This destroys the possibilities 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 B.13 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.

B.8.3 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 $blockingf > 1$. In essence, the workload



(a) Number of valid committed UTs where the UTs are deriving leaf nodes having at least one derived data item as an immediate parent.



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

Figure B.11: Experiment 1b: Consistency and throughput of UTs that only derive leaf nodes.

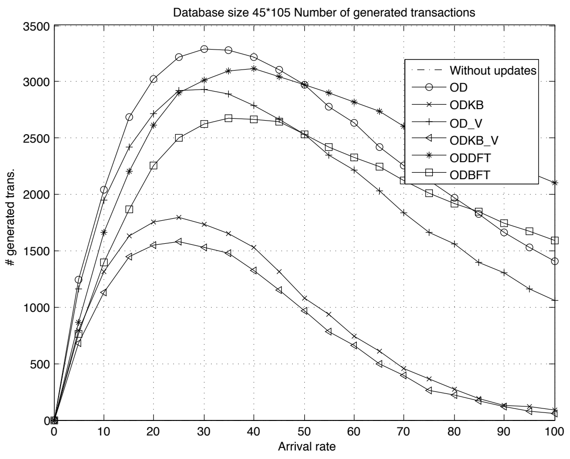


Figure B.12: Experiment 1b: Number of generated triggered updates where UTs derive leaf nodes (confidence intervals are presented in Figure D.3).

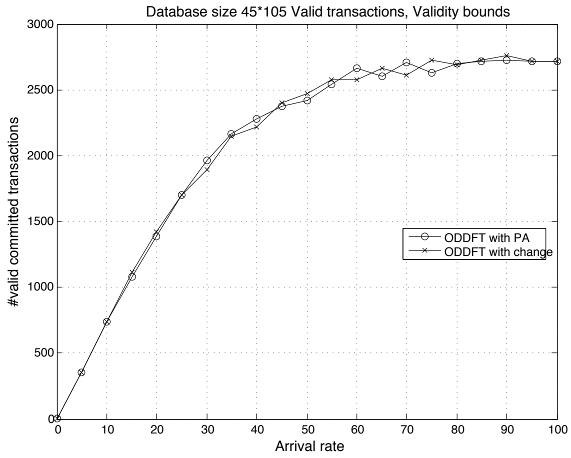


Figure B.13: Experiment 1c: Number of valid committed UTs using either the pa timestamp or a change flag to indicate potentially affected data items (confidence intervals are presented in Figure D.4).

from updates can be reduced by increasing $blockingf$, but the risk is that more user transactions become invalid.

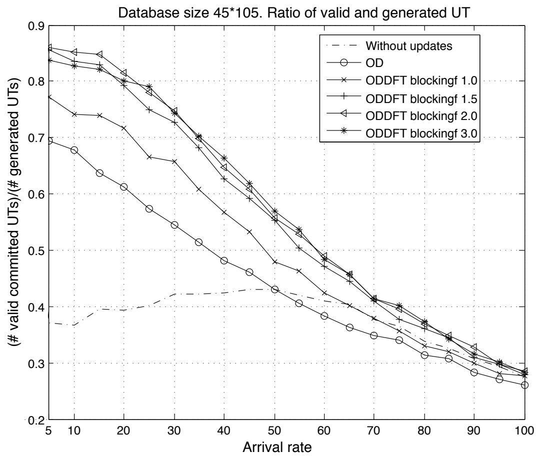
The blocking factors are part of the algorithms ODDFT, ODBFT, and ODTB, and Figure B.14(a) shows the performance of ODDFT using different blocking factors where UTs derive random data items. The figure shows the ratio of valid committed UTs and the generated UTs. The blocking factor can give a positive effect on the number of valid committed UTs. The reason is that fewer triggered updates are started which can be seen in Figure B.14(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 B.14(a) where ODDFT with $blockingf = 3.0$ lets fewer valid UTs to commit compared to when a blocking factor of 2.0 is used.

Note that, as indicated by Figure B.15, 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 $blockingf$ to.

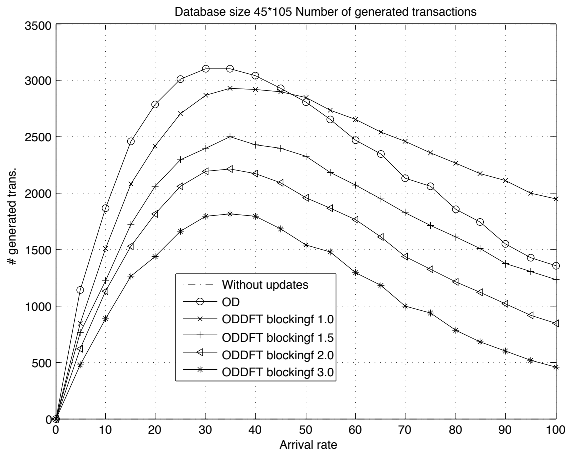
Figure B.16 shows the effect of blocking factors where UTs derive only leaf nodes, i.e., the system only executes actuator user transactions. In this setting, only triggered updates can make intermediate nodes valid while in the case of UTs deriving random data items an intermediate node can be made 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 immediate 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 B.16 shows the ratio of valid committed UTs and generated UTs in an experiment deriving only leaf nodes and changing $blockingf$. Comparing the results in Figure B.16 to the results in Figure B.14(a), we see that a larger value of $blockingf$ is needed to drop the performance for the only leaves experiment. This confirms the fact that more triggered updates are generated for data items in the only leaves experiment since the larger the $blockingf$ the more updates are not scheduled.

B.8.4 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

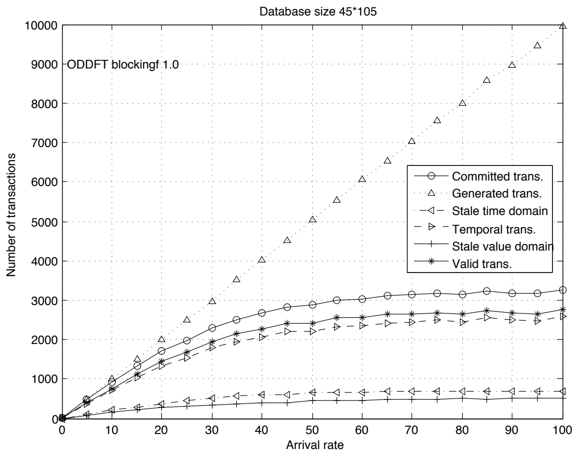


(a) Ratio of valid committed UTs and generated UTs.

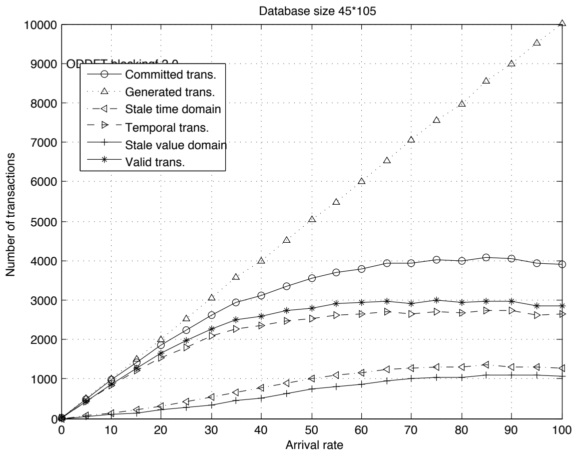


(b) Number of generated triggered updates.

Figure B.14: Experiment 1e: The effect of $blockingf$ on the number of valid committed UTs where a UT randomly derives a data item.



(a) ODDFT with $blockingf = 1.0$.



(b) ODDFT with $blockingf = 2.0$.

Figure B.15: Experiment 1e: Statistical data for ODDFT using two different blocking factors.

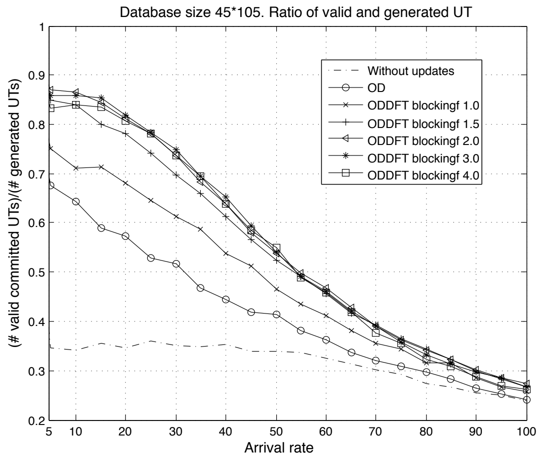
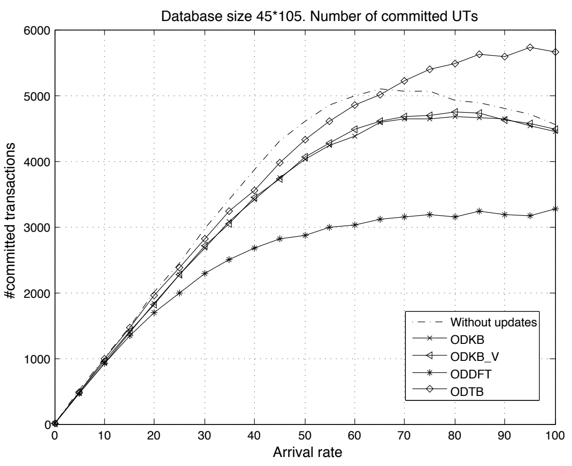


Figure B.16: Experiment 1e: The effect of *blockingf* on the number of valid committed UTs where a UT derives leaf nodes.

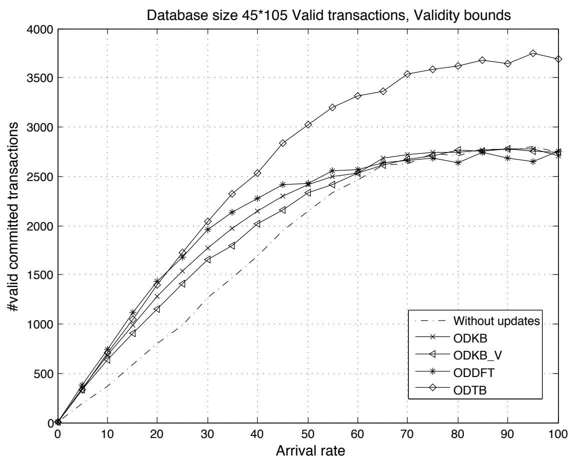
on the system whenever possible.

Figure B.17(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 B.17(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 B.18. 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.

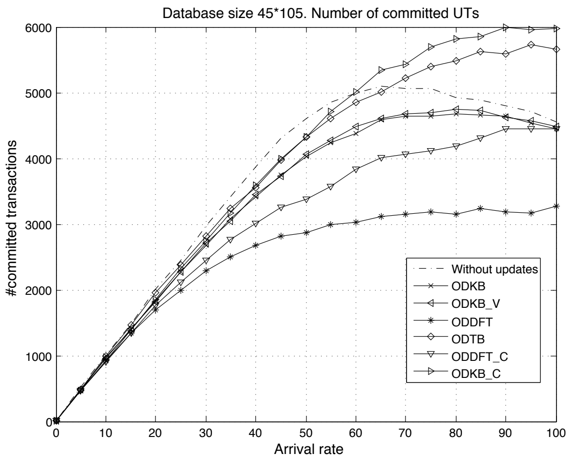


(a) Number of committed UTs.

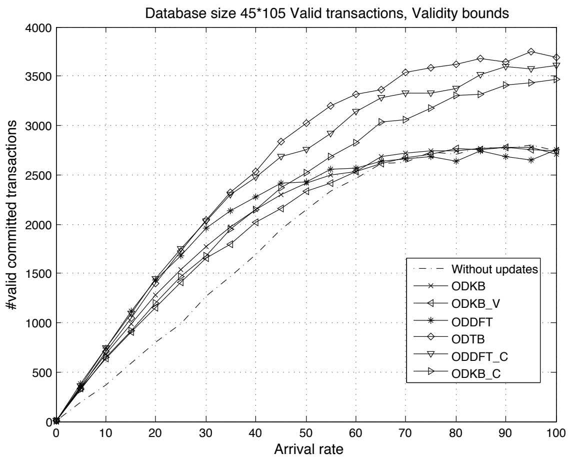


(b) Number of valid committed UTs.

Figure B.17: Experiment 2a: Consistency and throughput of UTs with no relevance check on ODDFT and ODKB_V.



(a) Number of committed UTs.



(b) Number of valid committed UTs.

Figure B.18: Experiment 2a: Consistency and throughput of UTs with a relevance check on triggered updates on ODDFT (confidence intervals are presented in Figure D.5.)

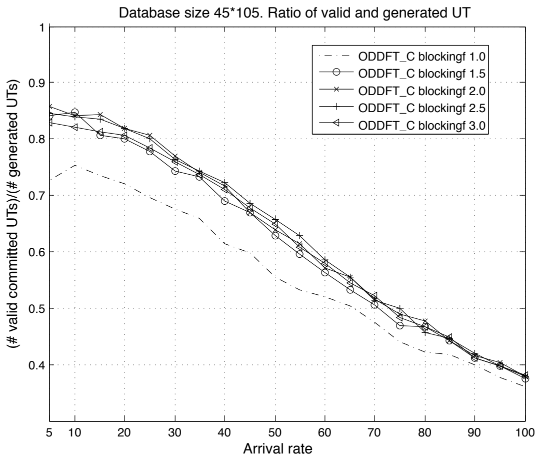
From Figure B.18 we see that ODKB_C lets slightly more UTs commit than ODTB, but more UTs are valid for ODTB. ODTB also has more valid committed UTs than ODDFT_C. A comparison between ODTB and ODDFT_C using blocking factors greater than 1 is presented in Experiment 2b.

B.8.5 Experiment 2b: Effects of Blocking Factors and Only Deriving Actuator Transactions

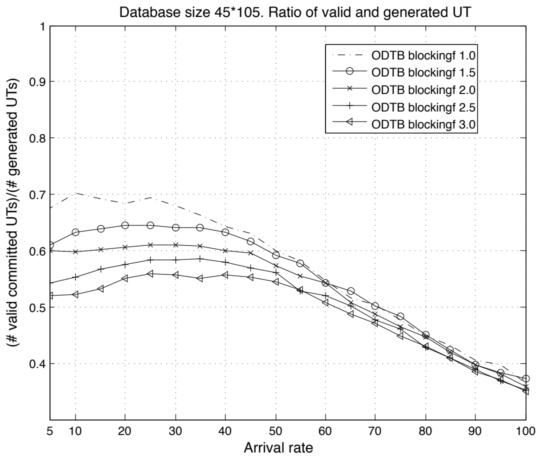
Figure B.19 shows the behavior of updating algorithms that can skip updates and their behavior with different values on $blockingf$. ODDFT_C increases the performance with $blockingf > 1$ while ODTB decreases the performance for all $blockingf > 1$. The reason ODTB drops in performance is that a check is done for the execution time of the remainder of a path to the data item being updated (the if-statement on line 7 of ODTB in Figure B.4). If the check fails, none of the updates are put in the schedule of updates. Since $blockingf > 1$ 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 updates in the schedule, which makes it a more consistency-centric algorithm than ODTB.

Figure B.20(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 shown in Figure B.19(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 immediate parents is needed before a triggered update is started. The scheduling step and the validity check are cheaper in the ODTB algorithm.

Figure B.21 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.

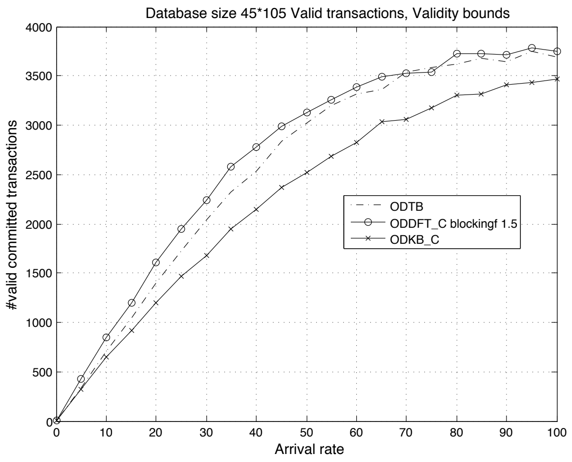


(a) Ratio of valid committed UTs and generated UTs for ODDFT_C.

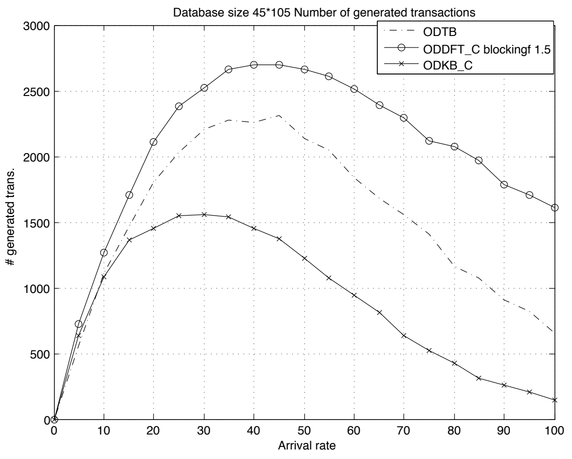


(b) Ratio of valid committed UTs and generated UTs for ODTB.

Figure B.19: Experiment 2b: Performance for updating algorithms that have the possibility to skip updates.

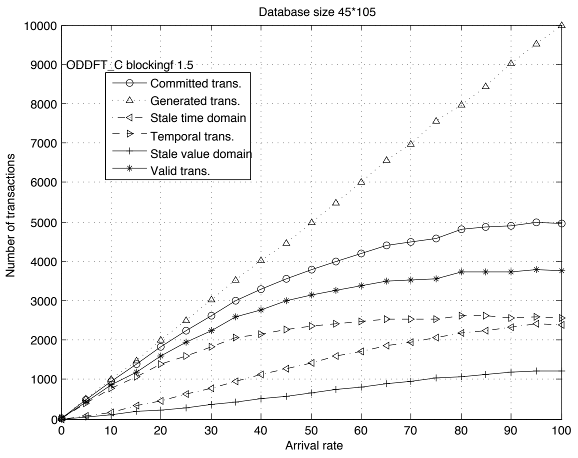


(a) Number of valid committed UTs for the ODDFT_C, ODTB, and ODKB_C with parameters such that they perform the best.

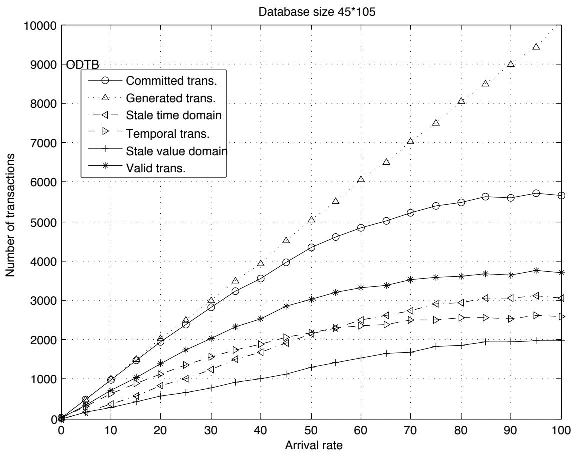


(b) Number of generated triggered updates.

Figure B.20: Experiment 2b: Performance for updating algorithms that has the possibility to skip updates (confidence intervals are presented in Figure D.6).



(a) ODDFT_C with $blockingf = 1.5$.



(b) ODTB

Figure B.21: Experiment 2b: Performance metrics for ODDFT_C with $blockingf = 1.5$ and ODTB.

APPENDIX C

Multiversion Concurrency Control with Similarity

This appendix describes the simulator setup of DIESIS and performance results of MVTO-S^{UV}, MVTO-S^{UP}, and MVTO-S^{CRC} where ODTB is used as the updating algorithm. The performance results are contrasted to well-established techniques to implement concurrency control. The sections in this chapter complement sections 5.1–5.5.

C.1 Performance Results

C.1.1 Simulator Setup

This section describes the settings of RADEx++ being used in the experiments in sections 5.5 and C.1.2–C.1.6.

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 $32/arrival\_rate$, where 32 is the number of invoked tasks using the base period times, and $arrival\_rate$ 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 45×105 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 μC/OS-II [84] 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. $blockingf$ is set to 1 and execution times on updates are not used since all scheduled updates are executed.

C.1.2 Experiment 4b: Memory Pool Size

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

Results in Figure C.1 support the hypothesis. MVTO-S$^{\text{UP}}$ 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 5.11(b) shows the number of committed user transactions before their deadlines using multiversion concurrency control algorithms using fixed validity intervals. MVTO-S$^{\text{CRC}}$ and MVTO-S$^{\text{UV}}$ have the same performance, whereas MVTO-S$^{\text{UP}}$ has worse performance for small pool sizes. If data items in a system are best modeled by fixed validity intervals, then MVTO-S$^{\text{CRC}}$ is a good choice of a concurrency control algorithm.

C.1.3 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

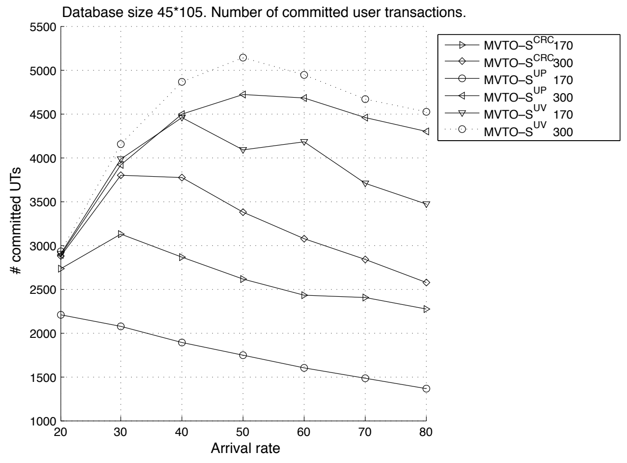


Figure C.1: Experiment 4b: The performance when using different sizes of the memory pool.

restart mechanisms of the MVTO concurrency control algorithms are designed for restarting low priority transactions first. Figure C.2 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.

C.1.4 Experiment 4d: Overhead

The purpose of this experiment is to investigate how the overhead affects 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 C.3(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.

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 C.3(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 C.3(b) where the OCC/OCC-C2 plot is above the NOCC/NOCC-C2 plot.

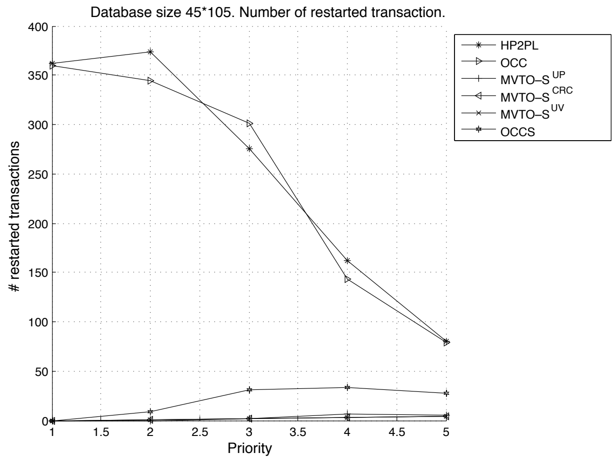


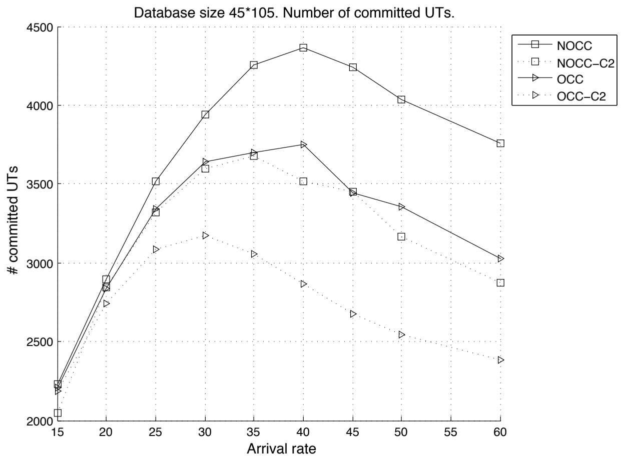
Figure C.2: Experiment 4c: Number of restarts at different priority levels at an arrival rate of 40 UTs per second. Level one represents the highest priority.

C.1.5 Experiment 4e: Low Priority

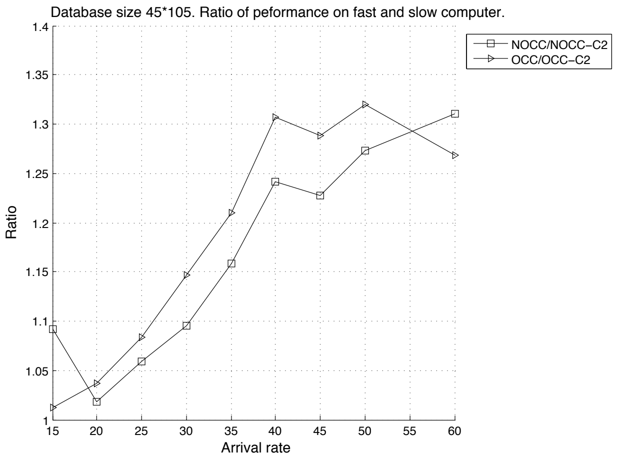
In this experiment an additional task with the lowest priority has been added, issuing one UT reading 25 data items. The period time of the new task is 1000 ms. The throughput of this transaction shows how the system can cope with, e.g., a diagnosis task executing with the lowest priority as in the EECU. The results in Figure C.4(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 C.4(b) plotting the restarts of the lowest prioritized transaction. This indicates that the values read by the transaction are not relative consistent. 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.

C.1.6 Experiment 4f: Transient State

In this experiment, the simulator is set up 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 C.5 to those in Figure 5.7 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

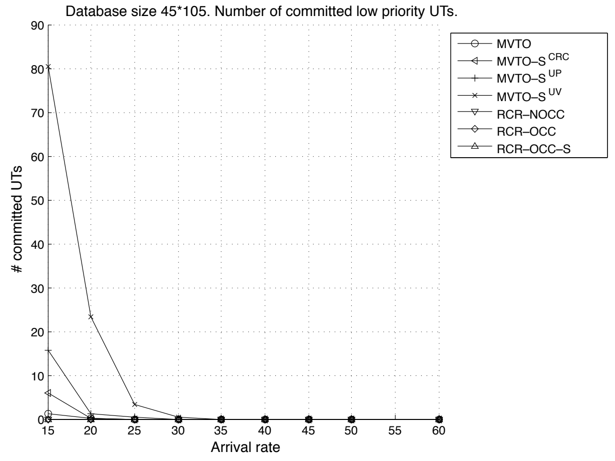


(a) Number of committed UTs on fast and slow computers.

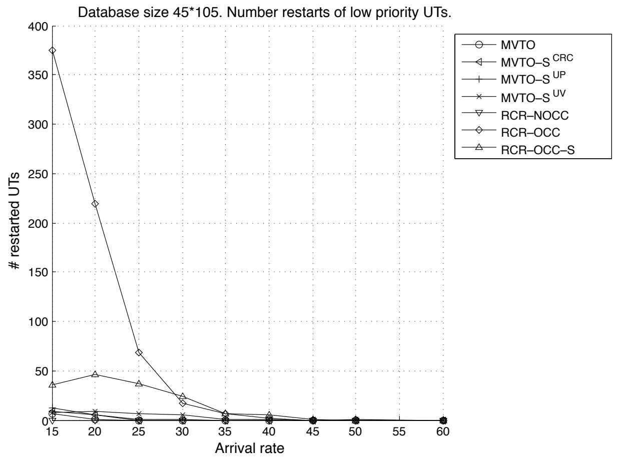


(b) Ratio of committed UTs of fast and slow computer.

Figure C.3: Experiment 4d: An experiment executed on fast and slow computers.



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



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

Figure C.4: Experiment 4e: Performance of an additional task with the lowest priority issuing a transaction reading 25 data items.

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 5.9. 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.

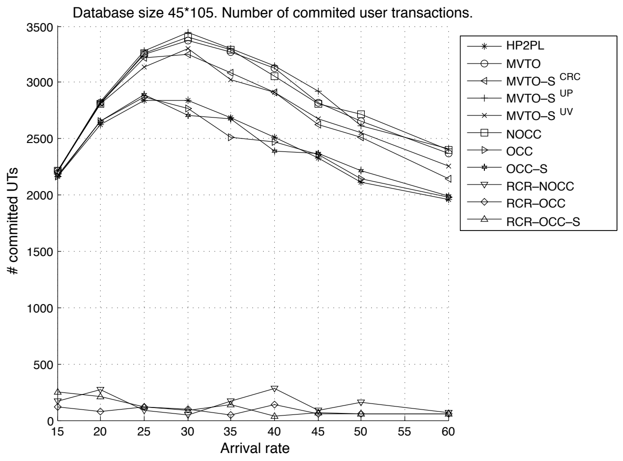
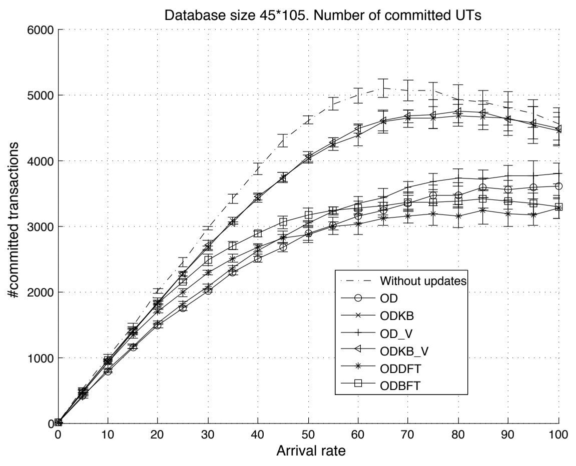


Figure C.5: Experiment 4f: Every write of a data item changes its value with 450 (confidence intervals are presented in Figure D.9).

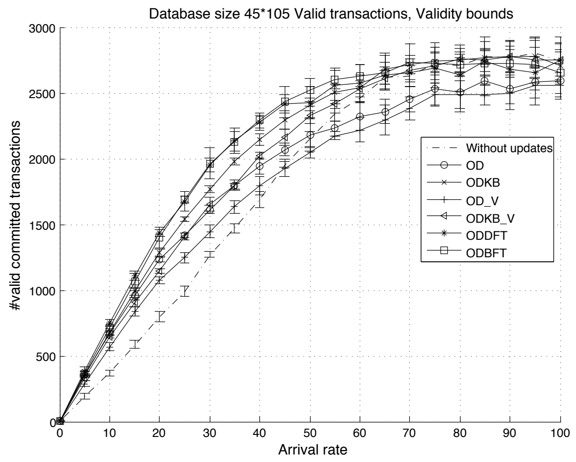
APPENDIX D

Confidence Intervals

This chapter presents 95% confidence intervals (see Section 2.4), using the t-distribution, of some of the performance evaluations.

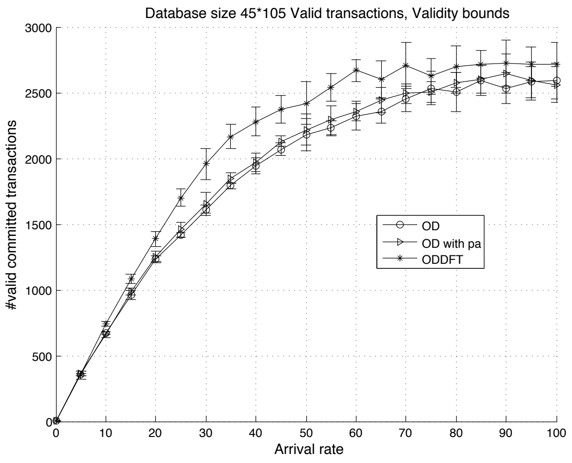


(a) Number of committed UTs.

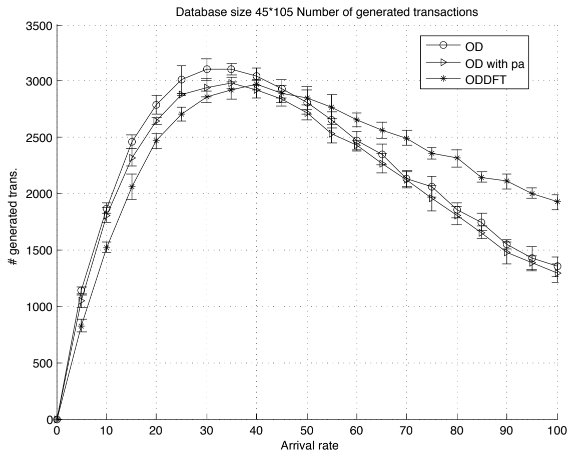


(b) Number of valid committed UTs.

Figure D.1: Experiment 1a: Consistency and throughput of UTs.



(a) Number of valid committed UTs.



(b) Number of generated triggered updates.

Figure D.2: Experiment 1a: Effects of measuring staleness of data items at deadline of UT.

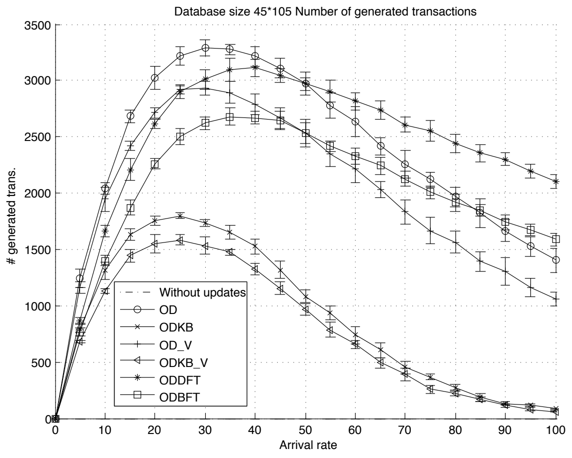


Figure D.3: Experiment 1b: Number of generated triggered updates where UTs derive leaf nodes.

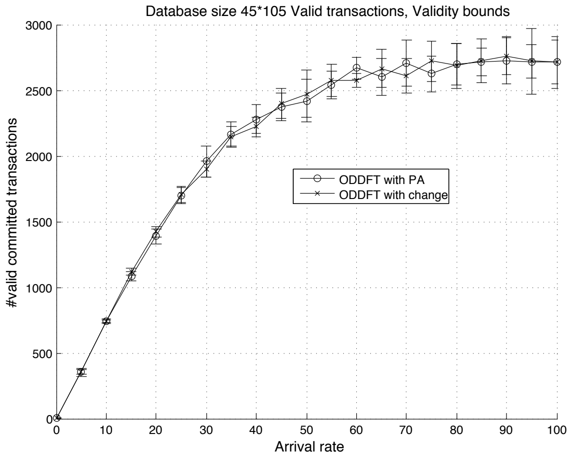
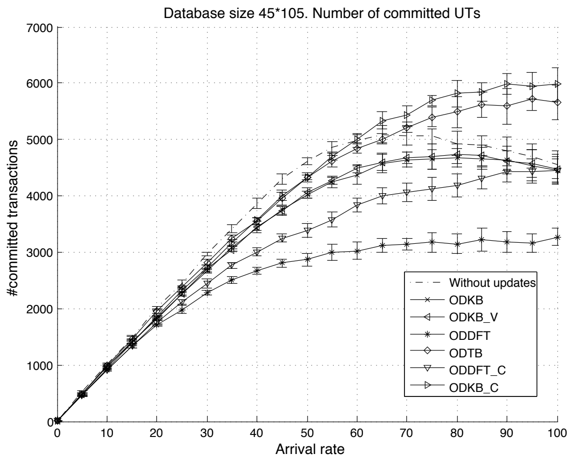
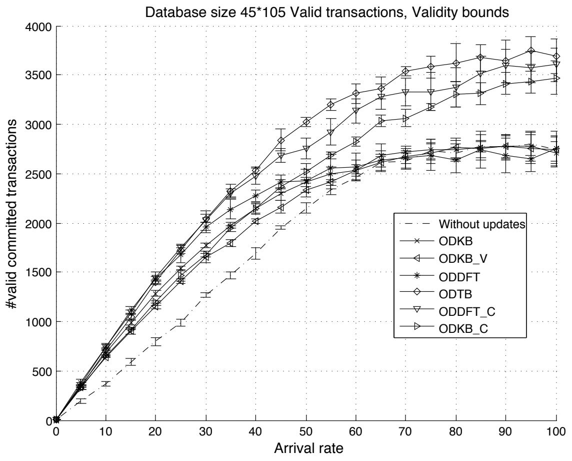


Figure D.4: Experiment 1c: Number of valid committed UTs using either the pa timestamp or a change flag to indicate potentially affected data items.

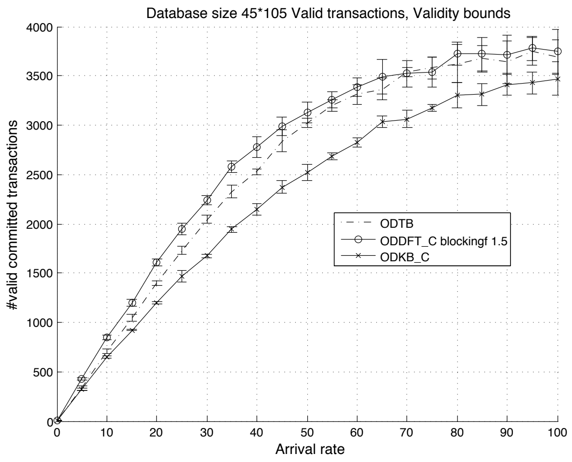


(a) Number of committed UTs.

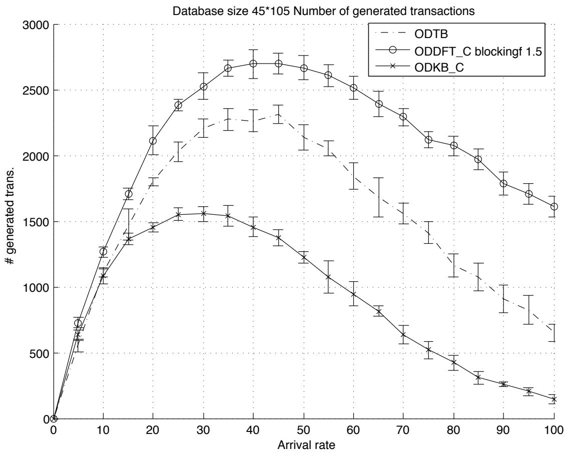


(b) Number of valid committed UTs.

Figure D.5: Experiment 2a: Consistency and throughput of UTs with a relevance check on triggered updates using ODDFT.

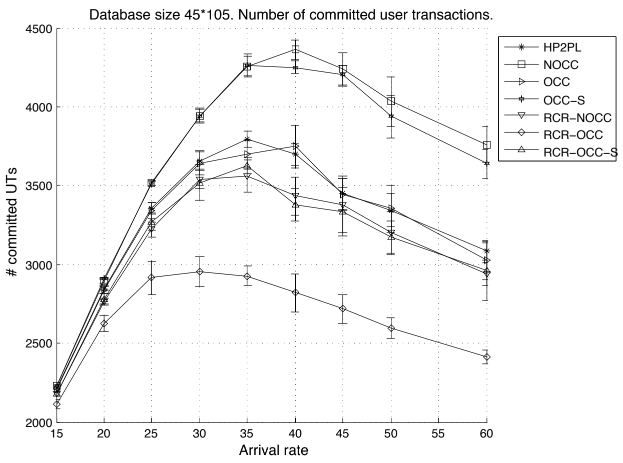


(a) Number of valid committed UTs for the ODDFT_C, ODTB, and ODKB_C with parameters such that they perform the best.

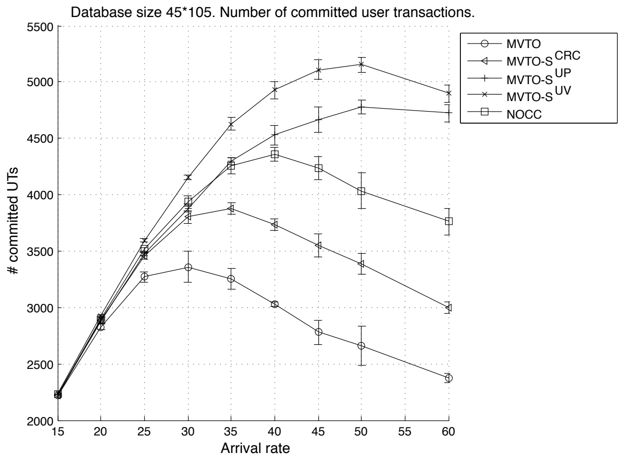


(b) Number of generated triggered updates.

Figure D.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 D.7: Experiment 4a: Number of UTs committing before their deadlines using single- and multiversion concurrency control algorithms.

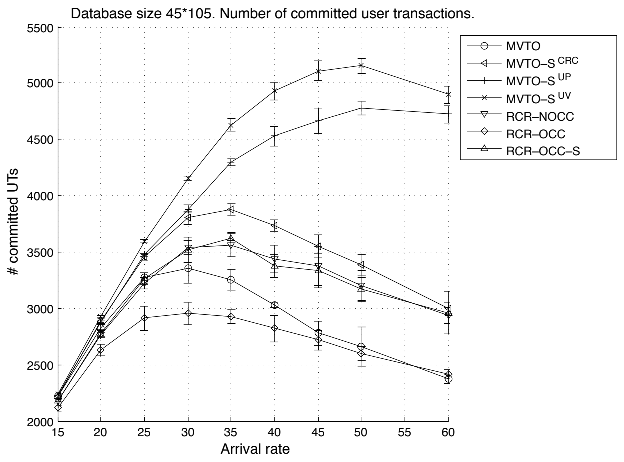


Figure D.8: Experiment 4a: A comparison of single-version concurrency control algorithms enforcing relative consistency and multiversion concurrency control algorithms.

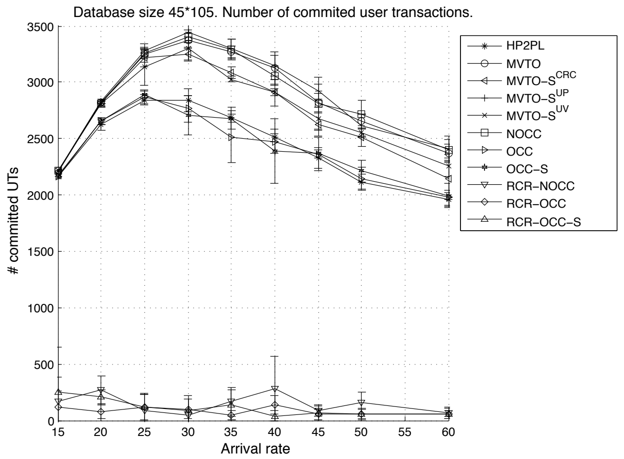


Figure D.9: Experiment 4f: Every write of a data item changes its value with 450.

INDEX

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology

No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall:** Semantisk analys av processbeskrivningar i naturligt språk, 1977, ISBN 91-7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91-7372-232-4.

No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91-7372-527-7.

No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation,1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372-805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies,1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity

Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi**: Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN-91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.

No 725 **Tim Heyer**: Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.

No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.

No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl:** Development of IT-suppor-ted Inter-organisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.

No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.

No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.

No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.

No 774 **Choong-ho Yi:** Modelling Object-Oriented Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.

No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.

No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.

No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.

No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems´ Development, 2003, ISBN 91-7373-604-X

No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informationsystem, 2003, ISBN 91-7373-618-X.

No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.

No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.

No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.

No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.

No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing - An Emperical Study in Software Engineering, 2003, ISBN 91-7373-779-8.

No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.

No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.

No 869 **Jo Skåmedal:** Telecommuting's Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.

No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.

No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.

No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.

No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.

No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5

No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004. ISBN 91-7373-966-9.

No 887 **Anders Lindström:** English and other Foreign Linquistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.

No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modellling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.

No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.

No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.

No 918 **Jonas Lundberg:** Shaping Electronic News: Genre Perspectives on Interaction Design, 2004, ISBN 91-85297-14-3.

No 900 **Mattias Arvola:** Shades of use: The dynamics of interaction design for sociable use, 2004, ISBN 91-85295-42-6.

No 920 **Luis Alejandro Cortés:** Verification and Scheduling Techniques for Real-Time Embedded Systems, 2004, ISBN 91-85297-21-6.

No 929 **Diana Szentivanyi:** Performance Studies of Fault-Tolerant Middleware, 2005, ISBN 91-85297-58-5.

No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.

No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.

No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.

No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN: 91-85297-97-6.

No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.

No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.

No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005. ISBN 91-85457-07-8.

No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.

No 974 **Dan Lawesson:** An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.

No 979 **Claudiu Duma:** Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.

No 983 **Sorin Manolache:** Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.

No 986 **Yuxiao Zhao:** Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.

No 1004 **Patrik Haslum:** Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.

No 1005 **Aleksandra Tešanovic:** Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components, 2006, ISBN 91-85497-29-0.

No 1008 **David Dinka:** Role, Identity and Work: Extending the design and development agenda, 2006, ISBN 91-85497-42-8.

No 1009 **Iakov Nakhimovski:** Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis, 2006, ISBN 91-85497-43-X.

No 1013 **Wilhelm Dahllöf:** Exact Algorithms for Exact Satisfiability Problems, 2006, ISBN 91-85523-97-6.

No 1016 **Levon Saldamli:** PDEModelica - A High-Level Language for Modeling with Partial Differential Equations, 2006, ISBN 91-85523-84-4.

No 1017 **Daniel Karlsson:** Verification of Component-based Embedded System Designs, 2006, ISBN 91-85523-79-8.

No 1018 **Ioan Chisalita:** Communication and Networking Techniques for Traffic Safety Systems, 2006, ISBN 91-85523-77-1.

No 1019 **Tarja Susi:** The Puzzle of Social Activity - The Significance of Tools in Cognition and Cooperation, 2006, ISBN 91-85523-71-2.

No 1021 **Andrzej Bednarski:** Integrated Optimal Code Generation for Digital Signal Processors, 2006, ISBN 91-85523-69-0.

No 1022 **Peter Aronsson:** Automatic Parallelization of Equation-Based Simulation Programs, 2006, ISBN 91-85523-68-2.

No 1023 **Sonia Sangari:** Some Visual Correlates to Focal Accent in Swedish, 2006, ISBN 91-85523-67-4.

No 1030 **Robert Nilsson:** A Mutation-based Framework for Automated Testing of Timeliness, 2006, ISBN 91-85523-35-6.

No 1034 **Jon Edvardsson:** Techniques for Automatic Generation of Tests from Programs and Specifications, 2006, ISBN 91-85523-31-3.

No 1035 **Vaida Jakoniene:** Integration of Biological Data, 2006, ISBN 91-85523-28-3.

No 1045 **Genevieve Gorrell:** Generalized Hebbian Algorithms for Dimensionality Reduction in Natural Language Processing, 2006, ISBN 91-85643-88-2.

No 1051 **Yu-Hsing Huang:** Having a New Pair of Glasses - Applying Systemic Accident Models on Road Safety, 2006, ISBN 91-85643-64-5.

No 1054 **Åsa Hedenskog:** Perceive those things which cannot be seen - A Cognitive Systems Engineering perspective on requirements management, 2006, ISBN 91-85643-57-2.

No 1061 **Cécile Åberg:** An Evaluation Platform for Semantic Web Technology, 2007, ISBN 91-85643-31-9.

No 1073 **Mats Grindal:** Handling Combinatorial Explosion in Software Testing, 2007, ISBN 978-91-85715-74-9.

No 1075 **Almut Herzog:** Usable Security Policies for Runtime Environments, 2007, ISBN 978-91-85715-65-7.

No 1079 **Magnus Wahlström:** Algorithms, measures, and upper bounds for satisfiability and related problems, 2007, ISBN 978-91-85715-55-8.

No 1083 **Jesper Andersson:** Dynamic Software Architectures, 2007, ISBN 978-91-85715-46-6.

No 1086 **Ulf Johansson:** Obtaining Accurate and Comprehensible Data Mining Models - An Evolutionary Approach, 2007, ISBN 978-91-85715-34-3.

No 1089 **Traian Pop:** Analysis and Optimisation of Distributed Embedded Systems with Heterogeneous Scheduling Policies, 2007, ISBN 978-91-85715-27-5.

No 1091 **Gustav Nordh:** Complexity Dichotomies for CSP-related Problems, 2007, ISBN 978-91-85715-20-6.

No 1106 **Per Ola Kristensson:** Discrete and Continuous Shape Writing for Text Entry and Control, 2007, ISBN 978-91-85831-77-7.

No 1110 **He Tan:** Aligning Biomedical Ontologies, 2007, ISBN 978-91-85831-56-2.

No 1112 **Jessica Lindblom:** Minding the body - Interacting socially through embodied action, 2007, ISBN 978-91-85831-48-7.

No 1113 **Pontus Wärnestål:** Dialogue Behavior Management in Conversational Recommender Systems, ISBN 978-91-85831-47-0.

No 1120 **Thomas Gustafsson:** Management of Real-Time Data Consistency and Transient Overloads in Embedded Systems, ISBN 978-91-85831-33-3.

Linköping Studies in Information Science

No 1 **Karin Axelsson:** Metodisk systemstrukturering- att skapa samstämmighet mellan informa-tionssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.

No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998. ISBN-9172-19-299-2.

No 3 **Anders Avdic:** Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN-91-7219-606-8.

No 4 **Owen Eriksson:** Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000. ISBN 91-7219-811-7.

No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X

No 6 **Ulf Melin:** Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.

No 7 **Pär J. Ågerfalk:** Information Systems Actability - Understanding Information Technology as a Tool for Business Action and Communication, 2003, ISBN 91-7373-628-7.

No 8 **Ulf Seigerroth:** Att förstå och förändra systemutvecklingsverksamheter - en taxonomi för metautveckling, 2003, ISBN91-7373-736-4.

No 9 **Karin Hedström:** Spår av datoriseringens värden - Effekter av IT i äldreomsorg, 2004, ISBN 91-7373-963-4.

No 10 **Ewa Braf:** Knowledge Demanded for Action - Studies on Knowledge Mediation in Organisations, 2004, ISBN 91-85295-47-7.

No 11 **Fredrik Karlsson:** Method Configuration - method and computerized tool support, 2005, ISBN 91-85297-48-8.

No 12 **Malin Nordström:** Styrbar systemförvaltning - Att organisera systemförvaltningsverksamhet med hjälp av effektiva förvaltningsobjekt, 2005, ISBN 91-85297-60-7.

No 13 **Stefan Holgersson:** Yrke: POLIS - Yrkeskunskap, motivation, IT-system och andra förutsättningar för polisarbete, 2005, ISBN 91-85299-43-X.

No 14 **Benneth Christiansson, Marie-Therese Christiansson:** Mötet mellan process och komponent - mot ett ramverk för en verksamhetsnära kravspecifikation vid anskaffning av komponentbaserade informationssystem, 2006, ISBN 91-85643-22-X.