Examensarbete

# Efficient Data Management in Engine Control Software for Vehicles

## Development of a Real-Time Data Repository

by

**Marcus Eriksson**

LiTH-IDA-Ex-03/13

2003-02-27

Supervisor: Thomas Gustafsson
Examiner: Dr. Jörgen Hansson

## Abstract

In all new cars, a computer is controlling the engine and this computer is part of the engine control unit. The software that runs on the computer is large and complex, which makes it hard to maintain. One way to deal with the complexity is to use a database where the shared data in the system is stored. For this purpose, a real-time data repository has been developed in this master's thesis project. The repository running on the real-time operating system Rubus will be used to evaluate future algorithms for data handling in real-time databases. In this context data repository is a small-scale real-time database. The implemented repository can handle concurrent transactions and it uses two different methods for concurrency control, namely two phase locking - high priority and an optimistic algorithm called broadcast commit. The transactions can be scheduled using the earliest deadline first algorithm. In addition, we have implemented the data dependency scheduling algorithm where a graph is kept in order to see how data items depend on each other. The graph is then used when updating a data item to keep its dependencies as fresh as possible.

# Contents

# Chapter 1

# Introduction

This chapter gives an introduction to the report, introducing the topic to the reader and explaining the background.

## 1.1 Purpose

The purpose of this master's thesis is to develop a real-time data repository for the engine control unit used in current models of Saabs. The data repository will be used to evaluate new algorithms for handling data validity.

## 1.2 Target reader

The target reading group is people with basic knowledge in real-time systems and databases. The background chapter provides enough information for a person with some computer science knowledge to enjoy the report.

## 1.3 Topic background

In all modern cars there is a computer that controls the behavior of the engine. The computer is called an engine control unit and it controls many parameters, for example, the amount of fuel to inject and at what time to ignite the air/fuel mixture. The software that runs on the computer is large and complex and it is hard to maintain the software. To address this, a database has been identified as one way to deal with complexity by centralizing the data. If the developer can focus on the actual algorithms instead of thinking about where to find a certain data item, much is gained.

Also, the reduced complexity makes the software more efficient and less memory consuming.

This master's thesis is a part of an ISIS project called *Real-time databases for engine control in automobiles*[1].

## 1.4 Disposition of the report

The report starts by giving a background to the topic in chapter 2 and continues with describing the problem in chapter 3. In chapter 4 a real-time operating system is introduced, namely Rubus; it is presented here since it gives a better understanding of the following chapters. Then in chapter 5 the data repository developed is described, and finally in chapter 6 a more thorough description of the solutions and implementation is presented. The two final chapters present related work (chapter 7) and the conclusions drawn from the project (chapter 8).

[1]Information about the project can be found at;
http://www.ida.liu.se/labs/rtslab/projects/ISIS_DB_EngineControl/

# Chapter 2

# Background

This chapter introduces fundamental knowledge needed to understand the problem. Four-stroke engines and the engine control unit that controls the engine are described. An introduction to the software that runs on the engine control unit and some real-time related information is also provided.

## 2.1 Four-stroke engines

The engine in most new cars is called a *four-stroke engine* [18]. The name comes from the four phases each cylinder goes through. Phases only change at bottom dead center (BDC) and top dead center (TDC). TDC is when the piston is at its highest point in the cylinder, that is, the pressure in the cylinder is at the highest level. BDC is when the piston is at its lowest point, here the pressure is at the lowest. The phases are described in figure 2.1 [18]:

1. Intake phase
   The intake valve is open and, while the piston moves downwards, the cylinder is filled with a fresh air/fuel charge from the intake manifold. In this phase the piston moves from TDC to BDC.

2. Compression phase
   Here the fuel/air-mixture is compressed to a higher temperature and pressure through the mechanical work produced by the piston. A moment before TDC (BTDC) a spark from the spark plug ignites the mixture and initiates the combustion. The combustion is started in this phase since it takes a little while for the flame to travel into the cylinder. In this phase the piston moves from BDC to TDC.

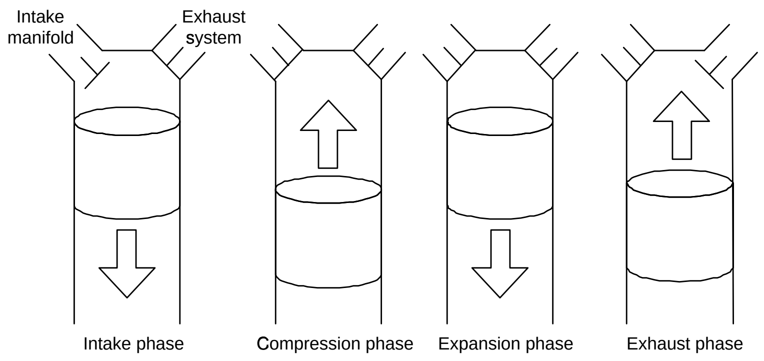


Figure 2.1: The phases in a four-stroke engine

3. Expansion phase
   Work is produced by the fuel/air-mixture during the expansion phase when the volume expands. Towards the end of this phase the exhaust port is opened and the blow out process starts, the gases are escaping the cylinder since the pressure in the cylinder is higher than in the exhaust system. In this phase the piston moves from TDC to BDC.

4. Exhaust phase
   The gases that are left in the cylinder are now pushed out into the exhaust system. When the piston reaches TDC a new cycle is started. In this phase the piston moves from BDC to TDC.

Since only one phase produces any actual work (the expansion phase), there are usually four or six cylinders in a four-stroke engine.

## 2.2 Engine control unit

The ECU controls a lot of things, among others, the amount of air to inject into each cylinder, when to ignite the air/fuel-mixture etc. [20]. It also has intelligence to, for example, add more air (the amount of fuel to inject is regulated by the amount of air that was sucked into the engine [20]) when the heat is turned on in the car. The increased speed of the engine then generates more electricity that is used to heat the car. The ECU is connected to a CAN-network[1] in order to get sensor values and to set actuators. The CAN network is also used to reprogram the flash-memory

[1]CAN stands for Controller Area Network. It is an ISO standard for serial data communication. The protocol was aimed at automotive applications.

and to connect from an external computer to the ECU in order to see and change different parameters.

### 2.2.1 Operation of the ECU

The ECU basically reads a lot of sensor values, for example temperatures and wheel speeds and then it uses these values in order to calculate on some action to do, this action is transmitted to the actuators. The actions must often be performed within a given time. The things sent out and read are electrical signals, there are drivers for the actuators and sensors in the ECU, these convert the values in the computer to the electrical signals. An actuator is for example a valve, i.e., something that changes the behavior of the engine. There are 70 input/output pins on the ECU and some are used for power supply etc. That gives us about 60 pins available for sensors and actuators [20].

### 2.2.2 Physical description

The data in this subsection is extracted from Jinnelöv [13]. The processor on the ECU is a Motorola MC68332, running at 16.778 MHz. The ECU has 64kb of RAM and 512kb of flash-memory. The flash memory contains the executable code and the RAM contains run-time data. There is also a slave processor on the board; it is primarily used to control the throttle.

## 2.3 Engine control software

The software in the engine control unit is layered. The bottom layer consists of the drivers for the sensors and actuators. The drivers get a current or a voltage level from a sensor and make these raw values readable in the software. The layer above then converts these raw values to usable ones, for example temperature or pressure. In the same way, if a value is sent to an actuator, the layers convert the value into a form that the actuators can understand. The values are stored in two global structures, one is called `In` and contains the sensor values and the other one is called `Out` and this is where the actuator values are stored.

### 2.3.1 Time bases

There are two time bases in the system, one is based on clock ticks, one tick every 5ms, the other one is based on the angle of a cogwheel connected to the engine. The cogwheel is supposed to have 60 cogs, but two cogs are missing in order to detect a new revolution of the cogwheel. Then there is a detector that detects these gaps and generates a signal. This signal is sent to the engine control unit where it generates an interrupt in the software. Since the cogwheel is mechanically connected to the engine, the number of interrupts generated is dependent on the speed of the engine. The interrupts can also be set to arrive at certain cogs before or after a gap. There are calculations to be done on each interrupt; therefore the load of the engine control unit is also increased when the engine speed is increased.

### 2.3.2 Diagnose

There is also a diagnose subsystem in the engine control software [13]. The diagnose checks if the sensors and actuators are working and if they do not, a flag is set in the subsystem. The flags can then be read with different tools by a mechanic at a car repair shop, in order to find the error and fix it. Some sensors and actuators have backups so that the diagnose system can chose to run a backup sensor or actuator if the primary one is broken.

## 2.4 Real-time systems

There are almost as many definitions of real-time systems as there are books and papers on the subject, this one was chosen because it fits the engine control system quite well [14]: “Any system where a timely response by the computer to external stimuli is vital is a real-time system”.

There are two distinct classes of real-time systems, namely *hard real-time* and *soft real-time* [21]. In hard real-time systems it is vital that all deadlines are met, i.e., that all tasks complete on time. This constraint implies that every action done in the system should have a bounded delay. There is usually no secondary storage in a hard real-time system since that can add unpredictable delays. In soft real-time systems, deadlines are not as important as in hard real-time systems, if a deadline is missed, it only reduces the value of the task. Tasks have priorities that the scheduler should follow; a high priority task should always run before a lower priority one. The soft real-time systems generally provide a lot more support for process synchronization and other advanced operating system services.

### 2.4.1 Scheduling

Scheduling is the task of devising a schedule given a set of tasks with corresponding precedence constraints, resource requirements and deadlines

[14]. A precedence constraint means that some task has to run before or after another task. Resource requirements are for example processing power and memory requirements. A deadline is a time when the task must have finished. There are three main groups of tasks when you consider periodicity; periodic, aperiodic and sporadic tasks. A task is periodic if it runs periodically, for example once every second, it is aperiodic if it can arrive at any time. Sporadic tasks are like aperiodic ones, only that there is a minimal time between the arrivals of the tasks.

A schedule may be computed in advance (off-line scheduling) or obtained dynamically (online scheduling). Off-line scheduling is done before the actual execution of the task. Online scheduling is done when the task arrives in the system. There are several approaches to doing scheduling, the most classic ones are: *Rate monotonic* (RM) [16] where the tasks have static periods and they are assigned priorities that are inversely proportional to their period time, that is, if a task has a long period time, it will have a lower priority. Another approach is *earliest deadline first* (EDF) [16] where a task dynamically gets a higher priority if its deadline is closer. Both of these algorithms are optimal under certain assumptions, RM is optimal when the tasks have static period times and EDF is optimal when the tasks get dynamic priorities.

### 2.4.2 Semaphores

A semaphore is an integer variable that is only modified with the atomic functions `wait()` and `signal()` [21]. `wait()` decreases the value of the semaphore by one and `signal()` increases it. If the semaphore is less then zero, `wait()` stays in a loop until the semaphore is equal to, or greater than zero. A *mutex semaphore* is used for *mut*ual *ex*clusion between threads, i.e., to avoid that two threads enter a critical section at the same time.

A problem when using semaphores is called priority inversion [21]. It occurs when a higher priority thread needs a lock on a semaphore that a lower priority thread holds. The lower priority thread can then be interrupted by a third thread, this makes the waiting time for the high priority thread unbounded. A solution to the problem is called priority ceiling, where the threads run at a pre-set ceiling priority when they hold a semaphore. This makes the lower priority thread run at a higher priority as long as it holds the semaphore and therefore it cannot be interrupted by the third thread.

The priority ceiling protocol is an extension to the priority inheritance protocol [8]. The idea with priority inheritance is that a thread blocking a set of other threads inherits the priority of the blocked thread with the highest priority. For instance, if a thread holds a semaphore, it will execute at the highest priority among the threads which wait for the same semaphore. This reduces the blocking time of the threads since a low priority thread that holds a semaphore will be able to run at a higher priority as long as it holds the semaphore. The priority inheritance protocol has some problems that the priority ceiling protocol does not have [8], the most important one is that deadlocks are not prevented. A deadlock occurs when two threads are waiting for a semaphore locked by the other thread.

## 2.5 Real-time databases

A real-time database system is a system where time constraints are associated with transactions and data has specific time intervals for which the data is valid [22]. An ordinary database has no notion of time, i.e., all data in an ordinary database is valid as long as the values are in the database. In a temporal database the notion of time is added, i.e., it has validity intervals for its data. This solves problems with oversampling of sensor values since if a transaction tries to read an old value, the database can handle this event in some way. One difference between a temporal database and a real-time database is the fact that a real-time database has time constraints on the operations on the database, hence, you can make sure that a transaction is finished within a certain time [22]. Another difference is that a temporal database is often an ordinary database with added validity intervals and this gives unpredictable delays when accessing data items.

### 2.5.1 Concurrency control

Concurrency control is used to avoid that different transactions that run simultaneously interfere with each other [21]. There are two classes of methods to achieve this, one is called optimistic and the other one is called pessimistic [12]. The pessimistic approach means that conflicts are avoided during the execution of the transaction, this is achieved by using locks on data items. The optimistic approach on the other hand lets all transactions run and when the transaction is done, it checks that no conflicts occurred.

### 2.5.2 Transactions

Transactions should follow the ACID properties, that is, Atomicity, Consistency, Isolation and Durability [6]:

**Atomicity** means that an observer should see everything or nothing, that is, a user should not see any intermediate transaction results.

**Consistency** means that if a transaction violates an integrity constraint in the system, the system intervenes to cancel the transaction or to correct the violation of the constraint. In a temporal database consistency has two additional components, namely *absolute consistency* and *relative consistency* [19]. Absolute consistency is between the state of the environment and its reflection in the database. Relative consistency is among the data used to derive other data items. These notions are described more in section 2.5.3.

**Isolation** means that the same effect should be reached if we run the transactions in parallel as if we were to run them in serial order.

**Durability** means that if a transaction commits, the changes are never lost.

A transaction in a real-time database has a deadline when it should be done and often also a priority.

### 2.5.3 Validity intervals

A validity interval is a period of time after the data was created where the data is valid [19]. There are generally two different validity intervals in real-time databases; *absolute validity interval*, which is the interval when the data has the property of absolute consistency, and *relative validity interval* which is the interval when the data has the property of relative consistency.

A set of data items used to derive a new data item form a relative consistency set. Each such set $R$ is associated with a relative validity interval denoted by $R_{rvi}$. A data item $d$ from this set $R$ has a correct state if it is logically consistent, that is, satisfies all integrity constraints and if it is temporally consistent, which means that it should have both absolute consistency and relative consistency. A data item $d$ from the set $R$ is absolutely consistent if the difference between the timestamp of $d$ and the current time is less than the absolute validity interval, i.e., $d \in R$ is absolutely consistent if $(current\_time - d_{timestamp}) \leq |d_{avi}|$.

The data item $d$ is relatively consistent if the difference between the timestamp of $d$ and all the data items in the set $R$ is less than the relative validity interval, i.e., $d$ is relatively consistent if $\forall d' \in R, |d_{timestamp} - d'_{timestamp}| \leq |R_{rvi}|$. $d_{avi}$ represents the absolute validity interval and $d_{timestamp}$ is the timestamp for the data item $d$. $R_{rvi}$ is the allowed relative validity interval for the set $R$.

### 2.5.4 On demand updating

*On demand updating* is a way of updating a data item when the user requests it and the data item is not valid as it is [2]. A database with on-demand behavior must know how to update the values stored in the database.

## 2.6 Real-time operating system

A real-time operating system (RTOS) should provide efficient mechanisms and services to carry out good real-time scheduling and resource management [17]. It should also keep its own time and resource consumptions predictable and accountable. The RTOS should be extensible, modular and preferably small since it is often supposed to run in embedded systems where memory is limited. An RTOS is often built around a micro-kernel that provides certain services, for example scheduling, synchronization and interrupt handling.

# Chapter 3

# Problem description and statement

This chapter describes the current system and the problems that are the focus of this master's thesis work.

## 3.1 The system today

This section introduces the problems with the system today.

### 3.1.1 Multiple versions of similar data

The different tasks in the system use the values in the global structures mentioned in subsection 2.3 to derive new data and that particular derived data is often stored in other global structures. There are many of these global structures and that makes it hard to find out what the values actually represent, therefore a programmer might simply derive a new value instead of looking for the very same derived value in the global structures. If there was a way of storing derived values in a common repository, much could be gained, both in the processing power of actually doing the calculation only once and then the memory that we save by storing the derived value only once.

### 3.1.2 Sampling of sensor values

A function is called to start the process of updating a sensor value. There is no way of checking before such a call if refreshing of the sensor value is needed, instead the developer has to statically check the code and see where else the value is refreshed and if that is often enough. This is a tedious work and it is easy to miss an update of the values, therefore many sensor values are over-sampled, that is, updated more often than necessary.

### 3.1.3 Real-time operating system

Jinnelöv [13] describes that there is no real-time operating system available in the provided engine control software. Instead, everything is interrupt-driven, i.e., if a certain interrupt is triggered the corresponding code will run. Since then, a version of the software that runs on the real-time operating system Rubus has been provided.

## 3.2 Aims and objectives

To solve the identified problems, a set of aims and objectives for the thesis project was devised. The aims and objectives are presented in this section.

### 3.2.1 Evaluate Rubus

An evaluation of the Rubus-version of the engine control software was done, covering the current solution and the improvements that could be done.

### 3.2.2 Develop a centralized real-time data repository

The development of a centralized real-time data repository was the main task of the thesis work. The repository;

- is able to be extended and modified easily. This item is important since the main purpose of the repository is to evaluate the performance of new algorithms for handling validity intervals but also study other real-time database issues.
- is able to handle absolute validity intervals (AVI), i.e., check if a read data item is valid and collect statistics of the amount AVIs that are violated. If an AVI is violated, it is possible to handle it.
- is able to handle relative validity intervals (RVI). It is possible to tell a transaction how big the interval is and the repository then checks that the data items read do not violate this. The repository handles any violations.
- provides different ways of doing concurrency control. The repository can handle one protocol from each of the two main groups of concurrency control approaches, therefore it should be easy to implement other ones within these two groups.

- collects statistics of transaction. Currently information about the number of started transaction, execution time of the transactions etc. is collected.
- schedules transactions. Rubus (see chapter 4) schedules threads in a preemptive way, i.e., the highest priority thread that is ready to run is executed. It is interesting to study the performance if the scheduling algorithm is changed to a standard one, like earliest deadline first.
- supports the data dependency graph algorithm [10] where different data items depend on each other and updates are triggered to keep the data items as up to date as possible.

### 3.2.3 Analyze the existing system

The existing system was analyzed in order to find good transaction examples. These examples reflect the real system, i.e., with the load and common variables.

# Chapter 4

# Evaluation of Rubus

This chapter introduces the real-time operating system Rubus and evaluates how well it is suited for running the real-time data repository, described in chapter 5, on the engine control unit.

## 4.1 Introduction to Rubus

Rubus is a real-time operating system developed by Arcticus Systems. It contains services to provide an execution platform for the application software of safety critical systems [5]. The system is split into three kernels, the red kernel that runs time-triggered red threads, the blue kernel that runs event-triggered blue threads and the green kernel that runs interrupt-triggered green threads. There is also a set of basic services that any kernel can use. The kernels can communicate between each other using these basic services. Resources are statically allocated in Rubus, i.e., before run-time the developer allocates the needed memory and sets up every thread with their buffer sizes etc.

### 4.1.1 The red kernel

As mentioned, the red kernel runs time-triggered red threads. The threads run by an off-line-generated static schedule where the threads have deadlines and maximal execution times which makes the red kernel good for modeling hard real-time systems where a deadline miss can lead to a catastrophe. There is also a possibility to switch between different red schedules if, e.g., there is a mode change in the system. The red threads always have priority over blue threads. The red kernel also supervises that the deadlines are respected.

Red threads support preemption, i.e., a red schedule can be composed of a set of red threads that interrupt each other. The threads use the same stack for storing execution time information, therefore a thread that preempts another thread must finish its execution before the preempted thread can execute again.

The red threads do not have any constructs for synchronization of the threads, no semaphores or signals to make sure that there are no conflicts when accessing common data. This makes the red kernel inappropriate to use when implementing a dynamic system where many threads are supposed to communicate with each other and access the same data.

### 4.1.2 The blue kernel

The blue kernel is an event-triggered kernel; it runs the threads by a priority based preemptive scheduling algorithm which guarantees that the thread with the highest priority among the ready threads is always executed first. The blue threads execute when the red threads do not utilize the processor. There are services for synchronization of blue threads included, namely message queues, signals and mutex semaphores [3]:

**Message queues** are used to communicate between blue threads, any blue thread can write a message to the queue and any blue thread can read the message. The messages and the queue itself are of fixed size, this enhances the performance of the message queue.

**Signals** are sent between blue threads. They do not carry any actual message other than a special bit-mask[1].

**Mutex semaphores** are used to synchronize blue threads.

Another service in the blue kernel is blue thread management where the developer for example can lock preemption, i.e., make sure that the blue threads do not interrupt each other. There are also possibilities to monitor the blue threads, check their stack usage and get information about their priorities etc.

### 4.1.3 The green kernel

Interrupt-triggered green threads run in the green kernel [4]. The green threads interrupt the red and blue ones if the processor's interrupt logic

[1]A bit-mask is a binary number-string that can be used to mask out information from another string.

allows it. The reason to use the green kernel instead of just letting the interrupts run as they want to is that the interrupts can be monitored, that is, one can set the maximal frequency and maximal execution time of them, and if they are violated, they can be handled. Some services in the green kernel are interrupt control, where the developer can lock interrupts, and thread management where for example information about executing threads can be collected.

Green threads could be good to use in the engine control system since they add some extra control over the interrupts such as maximal allowed frequency and maximal allowed execution time. It is recommended to signal a blue thread from the green interrupt threads since the blue threads have all the constructs for handling concurrent threads.

### 4.1.4 Basic services

There is a set of common services that all the kernels can use and the services can also be used to communicate between the different kernels. The services are [3]:

**Memory pools.** Since there is no possibility to dynamically allocate memory in Rubus (due to the fact that Rubus should be portable and all platforms do not support dynamic memory handling) there is an option to use memory pools. One can look at a memory pool as a part of the memory where applications can borrow memory and then return it.

**Basic message queue.** The difference between the basic message queue and the message queue in the blue kernel is that when using the basic queue, the threads cannot wait for a message to appear in the queue. The messages are consumed using first in - first out (FIFO).

**Mailbox.** The threads can send and receive messages from the mailbox. The mailbox contains references to the messages; this makes the mailbox fast.

**Event log.** It is used to log different events in the system. The log is fetched from the developed application with Rubus Visual Studio.

**Communication.** In order to communicate with the developed application from Rubus Visual Studio the developer needs to use the communication service.

### 4.1.5 Hardware Adaption Layer services

The hardware adaption layer contains services that adapt Rubus OS for a specific target processor. By using this approach the services that are target-specific are implemented in the most appropriate manner. The services include [4]:

**Basic timer control.** There are possibilities to disable the basic timer for a while. The basic timer is the timer that all other timers in Rubus depend on.

**Execution time measurement.** In order to measure time with high resolution this service is provided.

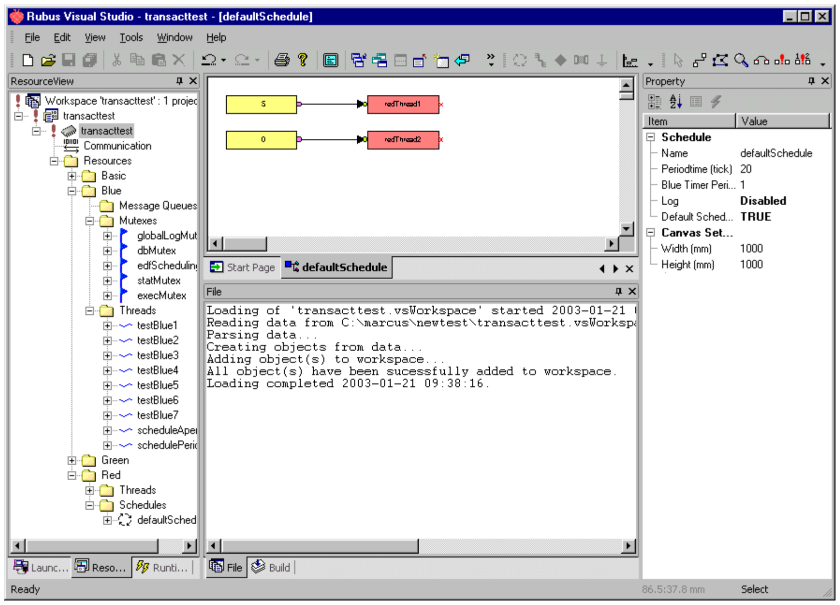
**Communication.** This is the hardware-dependent service of communication used by the basic communication service in order to communicate with the application from Rubus Visual Studio.

## 4.2 Windows simulator

Arcticus Systems provides a simulator in order to be able to simulate a run of the developed system in Windows. This greatly reduces the time needed to develop and debug a system. However when the system is deployed on the real target, the Rubus OS library file that is included is changed. Due to the fact that the simulator runs on top of another operating system there is a difference between the simulator environment and the real environment. The resolution of the basic timer (the timer that all other timers depend on) is always at least 10ms. In earlier versions of the simulator, the Windows NT clock was used to generate basic clock ticks in Rubus but sometimes ticks were missed since Windows NT is a multitasking system and the Rubus application might not be active when the NT clock tick is generated. To solve this problem, an interval of NT-ticks is used to generate a Rubus tick. This makes the system more robust but the resolution of the basic timer might be bigger than 10ms since ticks still can be missed in the interval.

## 4.3 Rubus Visual Studio

Rubus Visual Studio (RVS) is a tool, developed by Arcticus Systems, which is used to design, simulate and analyze the real-time system being developed. The developer allocates all the resources needed by the application



```
Loading of 'transacttest.vsWorkspace' started 2003-01-21
Reading data from C:\marcus\newtest\transacttest.vsWorkspa
Parsing data...
Creating objects from data...
Adding object(s) to workspace...
All object(s) have been sucessfully added to workspace.
Loading completed 2003-01-21 09:38:16.
```

| Item | Value |
|---|---|
| **Schedule** | |
| Name | defaultSchedule |
| Periodtime (tick) | 20 |
| Blue Timer Peri... | 1 |
| Log | **Disabled** |
| Default Sched... | **TRUE** |
| **Canvas Set...** | |
| Width (mm) | 1000 |
| Height (mm) | 1000 |

Figure 4.1: Rubus Visual Studio screen shot. To the left different resources that can be added or removed are shown. In the middle there is a small example of a red schedule, and to the right the developer can change different parameters.

and then Rubus Visual Studio generates a set of .c- and .h-files that are included in the project. It is also possible to connect to the application from RVS and collect a lot of information about the application, e.g. missed deadlines. All the parameters and services mentioned in the previous sections are configured using RVS.

## 4.4 Evaluation

This section evaluates the existing Rubus-version of the engine control software. The features of Rubus are also discussed.

### 4.4.1 The current system

In the Rubus-version of the engine control software the time functions are mapped to red threads. But since the engine control system was not designed to be hard real-time, a lot of errors occur. A better solution could be to signal blue threads from the red threads in order to capture the fact that the system is not hard real-time. The engine control software is not hard real-time since some calculations miss deadlines, i.e., some interrupt functions execute longer than the period between two interrupts when the load is high on the engine control unit. When a calculation misses a deadline an old value is used, for example in the function that calculates the amount of fuel to inject in the intake phase can use the values from the previous intake phase if the new calculations are not done in time.

### 4.4.2 Services

All services Rubus provides are easy to use and they are quite similar in the way the functions are called etc. A problem with the services is that they might be too limited depending on the application's needs. For example, in the original version of Rubus there is no way of turning off the priority ceiling protocol of the mutex semaphores, and there is no way of setting the priorities of the blue threads dynamically, even though that should not add much complexity or much code. The features needed are described in chapter 8.

### 4.4.3 Timer resolution

The fact that the basic timer only ticks every 10ms makes the Windows simulator bad for actually evaluating algorithms since the transaction load is too low. Instead, the actual evaluation would have to be done on the engine control unit or on some other platform where the timer resolution is higher. The simulator is good for developing and debugging since the developer can use the debug tools in Microsoft Visual Studio.

# Chapter 5

# A real-time data repository

This chapter describes a real-time data repository for the engine control unit running Rubus.

## 5.1 Preliminaries

The reason not to call the repository a database is the fact that not all features of a classic database are implemented. For example, there is no need to search the repository since everything is statically allocated; the only thing that will change in the repository is the values and properties of the data items. Also, there is no permanent log since there is no secondary storage. The repository was developed using the programming language C since the engine control software is developed in the same language.

## 5.2 Design

This section introduces the design and explains why the given design was chosen.

### 5.2.1 Interface to the data repository

In order to use the repository, a few functions need to be introduced:[1]

[1]Jinnelöv [13] proposed an interface to a real-time database, this is used with a few modifications. The deadline is given to BeginTransaction instead of CommitTransaction and the ReadDB and WriteDB functions are modified since there is no way to overload functions in C.

**BeginTransaction** starts a transaction, here the maximal execution time of the transaction and the maximal allowed relative validity interval of the data items read in the transaction are stated.

**ReadDB** reads a data item from the repository. The returned value is a union of the possible data types in the system.

**WriteDB** writes a data item to the repository. The absolute validity interval and the timestamp of the written data item are calculated from the previously read data items in the transaction.

**WriteDBavi** also writes a data item to the repository; the difference from WriteDB is that the allowed validity interval is expressed as a parameter. The timestamp is taken as the current time.

**UpdateDB** updates the parents to a given data item. This function is called when using the data dependency scheduling algorithm.

**CommitTransaction** verifies the transaction and makes the changes to the data repository permanent.

When accessing a data item in the repository, the user defines what data item is to be accessed, this is done using an enumerate data type. The data is then stored in an array and the enumerated constant represents an index in that array. As mentioned in section 2.3 the software in the original system is layered. The repository represents the layer below the actual applications, i.e., the repository will be the only place where an application in the engine control software retrieves any data.

### 5.2.2 Data storage

The data is, as mentioned in the previous subsection, stored in an array. This makes looking up indexes very fast and since every data is known in advance, no searching is required. Each position in the array consists of a structure where this information is stored. The structures are as follows:

**The data** is of course kept in the data structure. This could be any of the basic data types in the system[2]. A solution to this is to use a union-structure. A union represents an area in memory where different types of data could be stored.

[2]The data types in the system consist of u8, s8, u16, s16, u32 and s32. The number represents how many bits each type takes and the letter represents if the variable type is signed or unsigned.

**Timestamp** that represents when the data item was created.

**Absolute validity interval** is also kept here, this value represents how long time after the timestamp the data item is valid.

**Locking information** is stored in each data item. Depending on the concurrency control algorithm chosen, different locking information is needed. More about this in section 5.3.

Some more information is stored in each data item, depending on the current configuration of the repository. This is further discussed in chapter 6.

### 5.2.3 Transaction handling

To represent the transactions in the system, an array with the currently executing transactions is kept. The transaction number is given by the position in the array. Since there are only eight interrupt levels in the system, there could be at most eight concurrent transactions [13], this gives us that keeping an array with the executing transactions does not imply a large overhead in memory consumption[3]. If data dependency graph scheduling or on demand updating is used, the amount of data needed to store is doubled since each arriving transaction can trigger a series of triggered transactions. The information kept in the array is:

**Maximal execution time** given as a parameter to BeginTransaction.

**Starting time** of the transaction.

**Log pointer** is a pointer to the log of this transaction, this is described in more detail below.

**Minimal timestamp** of the current read-set.

**Maximal timestamp** of the current read-set. The maximal and minimal timestamps is kept in order to be able to check the relative validity interval of a transaction. The read-set $R$ is not relatively consistent if $max\_timestamp - min\_timestamp > R_{rvi}$.

**The relative validity interval** that is allowed for the transaction.

[3]There is 64kb of memory in the engine control unit and the information about executing transaction takes approximately 8*12=96 bytes since there can be at most eight concurrent transactions and the information needed to store about each active transaction is about 12 bytes

There is also a possibility to record the execution time of the transaction so far, this could be used in order to make good estimations on whether a restart of the transaction will be finished within the deadline.

### 5.2.4 Log handling

There is no permanent log in the system since there is no secondary storage and very little memory in the engine control unit. There is, however, a need to keep temporary logs for each transaction. Here the transactions store the actions they have taken. At commit time, the log is used in different ways depending on the concurrency control algorithm currently used. If the concurrency control algorithm is optimistic, no write operations actually change the repository before the commit point, therefore all writes are stored in the log. On the other hand, if pessimistic concurrency control is used, all old values of the data repository are stored in the log file in order to be able to roll back the changes.

## 5.3 Concurrency control

As mentioned before, concurrency control is needed to make sure that transactions running concurrently do not interfere with each other. There are two distinct approaches to this, the optimistic and the pessimistic.

### 5.3.1 Optimistic concurrency control

Optimistic concurrency control (OCC) [15] is provided in the data repository. OCC is what it sounds like, let a transaction run to the end and then detect any conflicts when the transaction is done. OCC is good when there are few conflicts in the system. There are three phases in a transaction that runs in an OCC-system, the first one is read, where the actual work of the transaction is done, all writes done in this phase are done to private storage [15]. The next phase is validation, where the transaction makes sure that no conflicts occurred during the execution of the transaction, and finally there is the write phase where all changes are written to the database. There are several approaches to achieve optimistic concurrency control, the one chosen here is a variant of the broadcast commit approach where a transaction detects an error and broadcasts this to all other active transactions [12]. The transactions involved in the conflict are then restarted. Experiments show that optimistic concurrency control outperforms two-phase locking (see subsection 5.3.2) when there are few conflicts [12]. To illustrate how optimistic concurrency control works in the

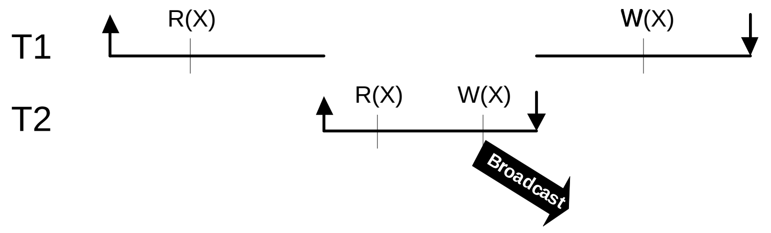


Figure 5.1: Optimistic concurrency control

data repository lets look at an example (Figure 5.1). Here we have two transactions, T1 and T2. We can see that T2 has higher priority than T1 since T2 interrupts T1. T1 starts by reading X, then T2 starts running and reads X, here we have no problem since many transactions can read the same data item without problem. When T2 then writes X, we get a problem since T1 has read it before. Then T2 broadcasts a message about this conflict to all active transactions. T2 then continues and writes all its changes to the database. Now T1 resumes its execution and writes to X, this is no problem since all write-operations are to local storage. Then T1 reaches the validation phase and sees the broadcasted message and realizes that it has to restart.

The locking information needed to store when using optimistic concurrency control is only a single 8-bit value for each data item. The value is used to mark what transaction first touched the data item.

### 5.3.2 Pessimistic concurrency control

Pessimistic concurrency control uses locks to ensure that no conflicts occur during the execution of the transaction. The most common pessimistic concurrency control protocol is called two-phase locking (2PL) [6]. The basic idea is that if a transaction needs to read a data item, it acquires a read-lock on the data item, and if it wants to write a data item, it acquires a write-lock. There can be many readers on the same data item at the same time, but there can only be one writer. Hence, read-locks are compatible with each other but conflicting with write-locks; write-locks are also conflicting with each other. There are two phases in a transaction in 2PL; the first one is a phase where the transaction gains locks, called the growing phase, and then a phase where it releases them, the shrinking phase. Once a transaction has released a lock it is not allowed to gain any

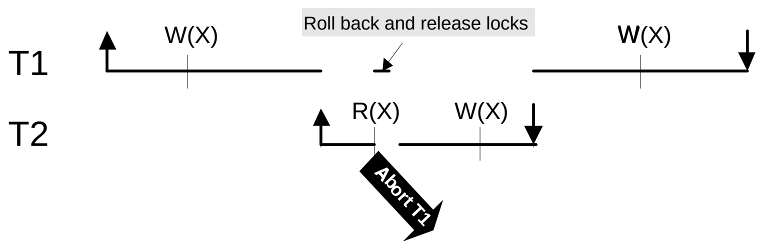


Figure 5.2: Pessimistic concurrency control

more locks. There is also a variant of 2PL that is called *strict two-phase locking*, here all locks are released at the commit point of the transaction [6]. The reason to use strict two-phase locking is that it does not allow dirty reads, that is, a transaction reads a data item that was written by a transaction that is roll-backed later.

The original two-phase locking algorithm is not well suited for real-time systems since it does not consider priorities. One variant of 2PL that considers priorities is called two-phase locking with high priority (2PL-HP) [1]. There is a set of rules that the transactions should follow in a 2PL-HP system:

1. If a higher priority transaction needs a lock on a data item, and a lower priority holds a conflicting lock on the same data item, the lower priority transaction should be aborted.
2. If a lower priority transaction needs a lock on a data item that a higher priority transaction holds a conflicting lock on, the lower priority transaction should wait until the higher priority transaction has released its lock.
3. A transaction can only join a set of readers if it has a higher priority than all transactions that wait for a write-lock on the data item.

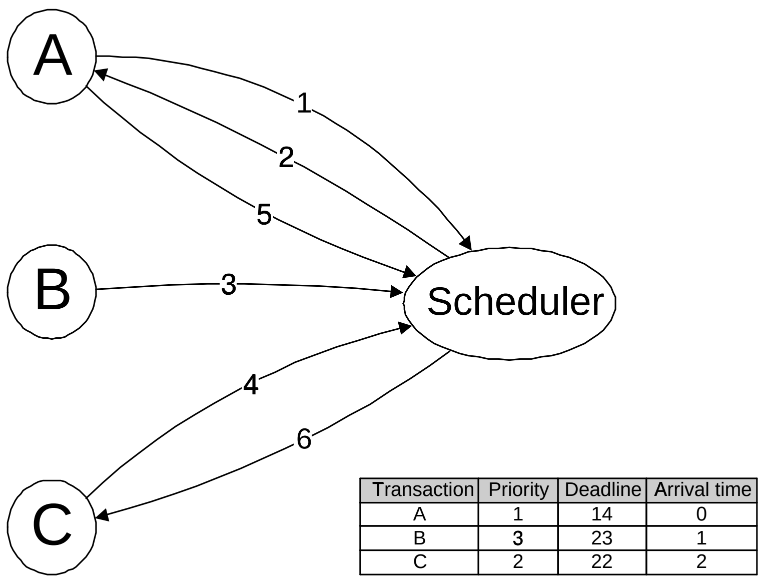
The example in figure 5.2 illustrates how 2PL-HP works. Here we have two transactions, T1 and T2. T1 starts by writing X, and thereby write-locking it. Then T2 with higher priority enters and wants to read X, according to the rules, T2 should abort T1, wait for it to rollback and release locks and then continue. T1 will restart and do its work when T2 is done. If, on the other hand T2 had lower priority, it would have to wait until T1 was ready with X and then continue.

The idea to allow several readers at the same time but only one writer is called the readers-writers problem [21]. The usual solution is to keep track of the amount of readers in the system and only let writers enter the system if the number of readers is equal to zero. When introducing priorities to the readers and writers, we get the readers-writers problem with priorities. To solve this a semaphore per priority level and data item is needed in order to make sure that lower priority transactions can wait for higher priority ones. In this specific application though, only one semaphore per data item is needed since a lower priority transaction never will have to wait for a higher one, since Rubus only lets higher priority threads interrupt lower priority ones. Hence, we know that if another transaction has a lock on a data item that we want to access, it must have lower priority, otherwise it would not have been interrupted. As we can see in figure 5.2, T2 will always have higher priority, otherwise T1 would finish before T2 gets to run.

## 5.4 Earliest deadline first scheduling

There is an option to schedule transactions by the earliest deadline first algorithm. The algorithm has been proven to be optimal [16]. Since Rubus schedules threads and the transactions run in threads, the EDF-approach is on top of the Rubus scheduling. This leads to unnecessary many context switches. Another drawback is that only higher priority transactions can enter the system and be scheduled. It would be good if there was a way of "turning off" the Rubus scheduling so that any thread that is ready gets to run, no matter what the priority of the thread is. This would give as many context switches but it would be true EDF. The scheduling in the repository works by signal sending and waiting for signals. Figure 5.3 shows a set of messages being sent between three transactions and the scheduler. The messages sent are described here:

1. Transaction A sends a message to the scheduler that it wants to run.
2. Since no other transaction is ready to run, the scheduler sends a message to A that it can run.
3. Transaction B starts and wants to be scheduled, so it sends a message to the scheduler and then waits for a reply.
4. Transaction C starts and also wants to be scheduled, it sends a message and waits for a reply. Since transaction A has an earlier deadline it continues to execute and transaction B and C waits.



| Transaction | Priority | Deadline | Arrival time |
|---|---|---|---|
| A | 1 | 14 | 0 |
| B | 3 | 23 | 1 |
| C | 2 | 22 | 2 |

Figure 5.3: Scheduling with EDF, three is the highest priority and one is the lowest.

5. Transaction A is now done and sends a message about this to the scheduler.

6. Since transaction C has an earlier deadline than transaction B it is allowed to execute.

We can see that a transaction can only send a message to the scheduler if it gets to execute for a short while, and it is only allowed to execute if it has a higher priority than the currently executing transaction, according to the Rubus scheduling algorithm. If we instead allowed any ready transaction to execute, they would all be allowed to send the message and be fairly scheduled according to EDF.

## 5.5 Data dependency graphs

This section describes data dependency graphs and how scheduling of transactions can be done using them. A more detailed description is given by Gustafsson [10].

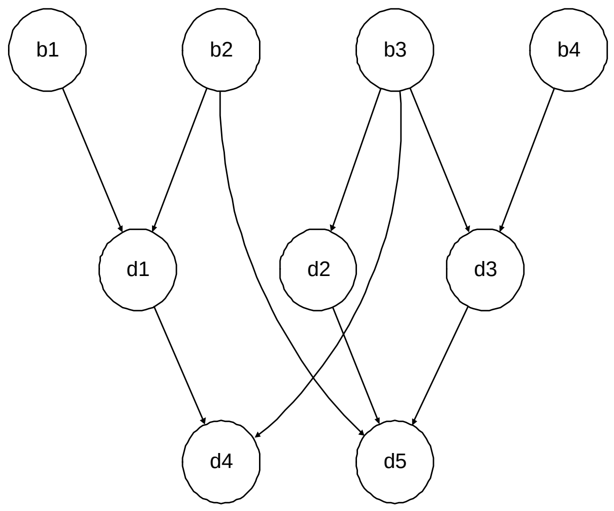


Figure 5.4: A data dependency graph

### 5.5.1 The graph

As mentioned in section 2.2.1, the engine control unit reads sensor values, makes calculations on these value and then emits them to the actuators. One can look at this as a data dependency graph, where sensor values are base items and then a lot of derived values are using these. Also, derived items can be used to derive new items. As we can see in the small example (figure 5.4), there can be several levels in the graph. The top level consists of the base items and all other items are derived from the base items.

### 5.5.2 Scheduling queues

To capture the properties of the current engine control system, there are two queues for arriving transactions, one has higher priority than the other. The higher priority queue is supposed to contain transactions started by angle-interrupts since they often are considered to be more important. The concurrency control algorithm used in the queues is optimistic. Earliest deadline first is used to schedule transactions in each queue. If there is a transaction ready in the higher priority queue, it gets to run before any transactions in the low priority one.

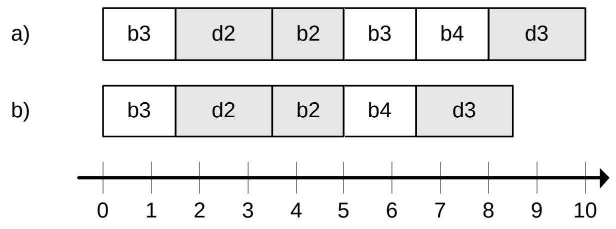


Figure 5.5: Data dependency schedule generated for d5 with the depth-first approach.

### 5.5.3 Triggered transactions

There are two different types of transactions in the system: arrived and triggered. The arrived transactions are ordinary transactions that start by standard execution of the system. Triggered transactions are transactions started in order to make the parents of a data item fresh. Triggered transactions are not put in the high or low priority queues described in subsection 5.5.2 since they are considered to be part of the currently executing arrived transaction. Though, they get a transaction number in order to be able to keep a transaction-local log.

### 5.5.4 Changed flag

In every data item there is a changed flag that states if the data item needs an update. If a data item gets updated and the new value is far from the old value on the data item, there is a possibility that the result of a recalculation of the children of this data item changes much from the stored value. Thus all children of the newly updated data item need to get their changed-flag set.

### 5.5.5 Parent updates

As seen in figure 5.4, derived item d5 depends on items b2, d2, and d3; d2 then depends on b3, and d3 depends on b3 and b4. If d5 is to be updated and the parents of it are marked as changed, they have to be updated. There are generally two approaches to doing these updates, either depth-first or breadth-first. Depth-first updates one parent-branch bottom-up entirely before the next one. Figure 5.5a shows one example of a schedule generated by the depth-first approach, figure 5.5b shows how the schedule would look if the duplicates were removed. Breadth-first updates all items

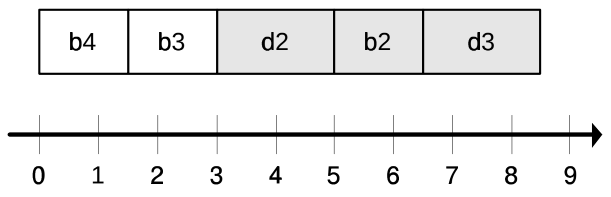


Figure 5.6: Data dependency schedule generated for d5 with the breadth-first approach.

in the level above before doing the next level, figure 5.6 shows an example of a schedule when traversing the graph breadth first.

### 5.5.6 Generating data dependency schedules

As we can see in figures 5.5 and 5.6 the generated schedules can look quite different depending on the way to traverse the data dependency graph. The order to pick parents when generating schedules also makes a big difference on the final result. To capture this, the parent that would add the most error if it is not updated is scheduled first so that it has a higher probability of being updated. To approximate the error that would be introduced if the data item is not updated, a function is called. The function considers how old the data item is. Also, a static weight is kept for each data item, if it is important to have one data item fresh, it is given a high weight.

To update the parents of a data item, a function is called, namely `UpdateDB(...)`. This function takes a maximal update time as parameter, a schedule is then generated that fits within this time. If all updates do not fit within the given time, some have to be left out. For example, in figure 5.5, the total amount of time needed for the schedule with duplicates is 10 time units. If the time is not enough, the schedule is generated so that it will have time to finish the schedule. If there are five time units for executing the updates, the schedule only includes b2, b4 and d3. The d2 branch is left as it is.

If, like in figure 5.5a, there are duplicate updates of data items, the one that is run first is kept and the other ones are removed. It is considered that a data item that is updated early in a generated schedule is fresh during the whole duration of the schedule.

# Chapter 6

# Implementation details

This chapter describes the implementation of the real-time data repository. The chapter is meant to simplify any future development of the repository and to give more detailed descriptions of the solutions chosen.

## 6.1 Storage of data items

The amount of data needed to store with each data item depends on the scheduling and concurrency control approaches chosen. With the compiler option `-Xenum-is-best` the compiler will choose the smallest possible integer data type for the enumeration data type. If there, for example, are 200 data items, they can fit within an unsigned char data type and therefore only occupy one byte of memory instead of four [23]. The data items are stored in an array that is called `DataBaseArray`. This is further discussed in sections 6.2, 6.4 and 6.5.

## 6.2 Concurrency control implementation

### 6.2.1 Optimistic concurrency control

The basic idea is that a transaction that wants to write a data item already "locked" by some other transaction, broadcasts a message to all other active transactions that there is a conflict in the system. The broadcast works by having a global list where a transaction writes the message. The global list is protected by a mutex semaphore called `globalLogMutex`. When a transaction tries to verify that no conflicts occurred, it sees the message, deletes the message and then restarts. Thus, the transaction that first enters the verification phase is able to commit.

The classic OCC-BC [11] algorithm works in a slightly different way; instead of detecting the conflicts when the actual data item is accessed, it does it during the verification. Also, in the classical algorithm a transaction that gets a broadcast message can restart immediately without having to reach the verification phase. The performance difference could be quite large since many transactions can be affected by one broadcast message, i.e., a transaction that detects a conflict can alert other transactions that are then restarted, and the total time gained by restarting immediately is then bigger than the time gained by being able to send the broadcast message immediately when the conflict occurs.

The extra information needed to store on each data item when using this concurrency control method is:

**Lock** that represents the transaction that first touched the item. This is used when validating transactions.

### 6.2.2 Pessimistic concurrency control

Two phase locking - high priority is implemented in the following way, each data item has an array where the current readers of the data item are kept. The position in the array is the reader's transaction number and the size of the array is equal to the maximum number of transactions. Then there is a semaphore on each data item that represents a write-lock on a data item. A transaction entering the system always has a higher priority than all currently running transactions. Thus, all that is needed to do when a conflicting lock is requested is to abort the one that currently holds the lock and then wait for it to roll back and release its locks. This is also true when running earliest deadline first scheduling since the deadlines are fixed during the execution of the transaction and therefore an interrupted transaction will not start again before the one that interrupted it is finished. Abortions are managed by setting a flag on the aborted transaction in the `executingTransaction` array. The flag is checked in every operation on the database to detect if the currently executing transaction is being aborted, and then roll back changes and release the locks. The transaction has to finish all calculations before it can restart.

The semaphores used in the pessimistic concurrency control implementation do not have the priority ceiling protocol. If the semaphores had the priority ceiling protocol, a transaction that got a write-lock on a data item would execute at the set ceiling priority. Then all threads that would have started executing, if the first thread was running on its original priority, would be blocked.

The extra information needed when using this concurrency control algorithm consists of:

**Current readers.** Since there can be many readers an array is needed to keep track of the transactions reading the item. This information is used when aborting transactions.

**Writer** that keeps track of the transaction number of the current writer. This is used when aborting transactions.

**Writer-semaphore** is used when waiting for an aborted transaction to release its locks.

## 6.3 Earliest deadline first scheduling

To keep track of the transactions that want to be scheduled there is a global array called `readyArray` where each transaction that wants to start executing stores its deadline and a pointer to the thread instance. The scheduler thread uses the thread instance pointer in order to be able to send a message that starts the transaction. It is not possible, when using signals in Rubus, to see who is the the sender of a message, therefore the pointer to the thread has to be stored in the global array. Rubus has a function to get a pointer to the currently executing thread, the function is called `blueSelf()`. The array is protected by a mutex semaphore called `edfSchedulingMutex`.

When a transaction is done, it removes its deadline and thread pointer from the global array, signals the scheduler that it is done so another transaction can start. During the execution of the transaction, other transactions can arrive. They will signal the scheduler that they want to run, but if the currently executing transaction still has the earliest deadline, the scheduler sends a signal to it even if it is already started. This does not matter since transactions never wait for any signals after they are started. Instead, all pending signals are cleared when the transaction is done.

## 6.4 On demand updating of data items

One assumption taken when using on demand updating is that transactions that update a given data item only update that item. Since the data repository is supposed to know how to update a data item, the transactions need to be kept in functions so that a pointer to the function can be stored with the data item that the transaction updates. This makes it possible to, when trying to read a data item that is invalid, call this function and then get an update of the data item.

Some extra information is needed on each data item when using on demand updating of data items:

**Transaction pointer** is a pointer to the transaction that updates the data item.

## 6.5 Data dependency graph scheduling

The data dependency graph is kept as a constant adjacency matrix to reduce the memory usage. If the matrix is constant it will be kept in the flash memory while if it was dynamic it would be in RAM memory. Generally there is more flash memory available than RAM memory. An adjacency matrix is a way to represent a graph [9]. The adjacency matrix for the graph in figure 5.4 is shown in figure 6.1. The items in the example graph are numbered 0–8 and then that number is mapped to a row and column number. Each column marked as “1” represents that the item with that row number has a parent that has that column number. The transpose[1] of the matrix represents the children of a node in the same way.

There is an assumption in this algorithm, namely that every transaction reads a set of data items and then writes another item. In this way, all the read items in a transaction are the parents of the data item that is going to be written. To capture this an extra function call is needed when doing a transaction that uses the dependency graph algorithm. The function was mentioned in subsection 5.5.5 and is called `UpdateDB`. It takes three parameters. The first one is the transaction number, the second is which data item that is updated in this transaction. The third is a maximal execution time for the updates, i.e., how long time can be spent on updating parents. Each data item in the repository keeps a maximal execution time that represents how long its update transaction takes. This value is used when generating the schedule. `UpdateDB` uses recursion to create a schedule

[1]The transpose of a matrix is constructed by letting the rows of the original matrix become the columns of the transposed one [9].

$$\begin{matrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0
\end{matrix}$$

Figure 6.1: Data dependency graph adjacency matrix

and it stops when the accumulated time is about to exceed the maximal update time given as a parameter to the function.

The prioritization between different parents mentioned in section 5.5.6 is also considered when generating the schedules. The solution is to use a priority queue and call the recursive function on each item in the queue. The priority queue consists only of the immediate parents of a given data item and it is used to select which branch to update first. A new priority queue is generated for each parent of a data item.

The extra information needed on each data item when implementing the data dependency graph algorithm is:

**Transaction pointer** that points to the transaction that updates the data item.

**Update time** for the data item, i.e., the time needed to execute the transaction that updates the data item. This is used when generating schedules.

**Changed flag** that is set by the parents to a data item if they are changed enough.

**Change delta,** the value of the data item can change within this delta without getting the change flag set.

## 6.6 Function description

This section describes the functions implemented in the data repository.

### 6.6.1 User functions

The functions that a user needs to know are:

- `void initDatabase();`
  This function initializes the database and sets all values to what they should be. This function should always be called before any transactions start.

- `char BeginTransaction(s8 *transNr,u32 met,u16 rvi);`
  This function starts a transaction. It finds a transaction number and sets all needed variables. Also, if earliest deadline first scheduling is used, it signals the scheduler and waits for an answer. If the value of `*transNr` is equal to the global constant BEGIN_TRANSACTION it starts a transaction, otherwise it is considered that a transaction is restarted. `met` is the maximal execution time of the transaction and `rvi` is the relative validity interval allowed for the transaction. Two extra parameters are needed when using data dependency scheduling; the priority of the transaction, i.e., if the transaction should be scheduled in the high or the low priority queue; the second extra parameter is the type of transaction it is, i.e. if it is an arrived or triggered transaction.

- `void CommitTransaction(s8 *transNr);`
  This function validates the commiting transaction and if the validation was successful, it makes the changes done during the transaction permanent. It sets `transNr` to the global constant TRANSACTION_DONE in order for BeginTransaction to know that everything was successful. If the validation fails on the other hand, it restarts the transaction by leaving the transaction number as it is.

- `void WriteDB(s8 *transNr, DataEnum di, DB_Data data);`
  This function writes a data item to the repository. The parameters given are; transaction number, what data item to write and the new data on that data item. Depending on the concurrency control algorithm used, it either stores the new value directly in the data repository (2PL) and the old values in a transaction-local list, or it stores the new value in the list and leaves the data repository unchanged

```
DB_Data dbdata;
s8 transNr=TRANSACTION_START;
while(BeginTransaction(&transNr,1000,5))
{
  ReadDB(&transNr,IN_N_ENGINE,&dbdata);
  CommitTransaction(&transNr);
}
```

Figure 6.2: An example of a transaction. The transaction is allowed to execute for 1000$\mu$s and the allowed relative validity interval is 5ms.

(OCC). The function calculates the AVI using `calculateAvi`. If on-demand updating or data dependency scheduling is used, a function pointer to the transaction that is calling `WriteDB` needs to be provided.

- `void WriteDBavi(..., u16 avi);`
  This function works in exactly the same way as `WriteDB` only that with this function we can explicitly state the absolute validity interval that should be stored for the data item. The function takes the same parameters as `WriteDB` with the addition of the `avi` parameter.

- `DB_Data ReadDB(s8 *transNr, DataEnum di, DB_Data *data);`
  This function reads a data item from the repository, returns the value and sets the reference parameter to the value. The function checks if AVI and RVI are violated using the functions `checkRvi` and `checkAvi`.

- `void UpdateDB(s8 *transNr, u32 dataItem, u32 deadline);`
  This function takes a data item and a deadline as parameter, then it generates a dependency graph schedule that fits within the deadline and that updates the parents of the given data item.

Two examples of transactions are given in figures 6.2 and 6.3. The reason to use different units for time is that there is never any need to give a validity interval that is less than 1ms, but a transaction might be allowed to execute for example 750$\mu$s.

### 6.6.2 Timing functions

This subsection describes the time measurement functions implemented.

```
DB_Data locald2,locald3,locald5;
s8 transNr=TRANSACTION_START;
while(BeginTransaction(&transNr,100,5,LOW_QUEUE,ARRIVED))
{
  UpdateDB(&transNr,D5,80);
  ReadDB(&transNr,D2,&locald2);
  ReadDB(&transNr,D3,&locald3);
  locald5=calculateStuff(locald2,locald3);
  WriteDB(&transNr,D5,locald5,&funcPtr);
  CommitTransaction(&transNr);
}
```

Figure 6.3: An example of a transaction when using data dependency graphs. UpdateDB updates as many parents to D5 as possible within 80$\mu$s.

```
u16 time;
time=startValidityTimer();
doLotsOfStuff();
time=stopValidityTimer(time);
```

Figure 6.4: A time measurement example.

- `u16 startValidityTimer();`
  This function returns the value of the basic clock.

- `u16 stopValidityTimer(u16 startTime);`
  This function does not really stop the timer, it only returns the time lapsed between `startTime` and the current time. The time given as parameter is given by `startValidityTimer()`.

An example of how to use these functions is given in figure 6.4. These functions are used when checking RVI and AVI in the repository. The functions are similar to the Rubus functions for high resolution time measurement, i.e.,halBsExecTimeStart() and halBsExecTimeStop(...).

### 6.6.3 Data repository help functions

This subsection describes the helper functions in the data repository.

- `void logCommit(LogItem *head);`

This function is used when using OCC, it writes all data items that are stored in list pointed to by `head`.

- `int calculateTimestamp(LogItem *head);`
  This function takes a pointer to the transaction-local list and reads it to determine the read set of the transaction. Then it calculates a timestamp using the timestamps of the data items in the read set.

- `int calculateAvi(LogItem *head);`
  This function works in a similar way as the function in the previous bullet, but it calculates an absolute validity interval using the read set.

- `u8 verify(s8 transNr);`
  This function is called when using OCC, it checks the global log for conflicts.

- `u8 checkRvi(DataEnum dataItem, s8 transNr);`
  This function takes a data item as parameter and then checks whether reading this data item violates the relative validity interval. The solution is to keep the maximal and minimal timestamp of the current read set of the transaction and check whether the new timestamp is bigger or smaller than these values. If so, the function checks whether the new interval is bigger than the allowed relative validity interval.

- `u8 checkAvi(DataEnum dataItem);`
  This function checks whether the data item given as parameter is absolutely valid.

- `void rollback(LogItem *li, s8 transNr);`
  This function is used in pessimistic concurrency control to reset all changes done by a transaction.

### 6.6.4 Locking functions

- `u8 r_lock(DataEnum dataItem, s8 transNr);`
  This function read-locks a data item when using OCC.

- `u8 w_lock(DataEnum dataItem, s8 transNr);`
  This function write-locks a data item when using OCC.

- `u8 r_lock_2pl(DataEnum dataItem, s8 transNr);`
  This function read-locks a data item when using 2PL.

- `u8 w_lock_2pl(DataEnum dataItem, s8 transNr);`
  This function write-locks a data item when using 2PL.

- `void releaseLocks(LogItem *li,s8 transNr);`
  This function releases all locks held. Depending on the concurrency control algorithm used, it works in different ways, if optimistic concurrency control is used, the data item is simply marked as unused, and if pessimistic concurrency control is used, the semaphore is unlocked.

### 6.6.5 Other functions

- `SimpleItem* getDependencySchedule(DataEnum d,s8 transNr);`
  This function returns a pointer to a dependency graph schedule. This function is only called from `UpdateDB`. The parameters are: the data item to generate a schedule for, and the transaction number of the caller.

- `void printStatistics();`
  This function prints the collected statistics using `printf(...)` and therefore the function is only available when using the Rubus Windows simulator.

## 6.7 Implementation limitations

Currently there is no way of restarting a transaction immediately since there is no way of "jumping" back to `BeginTransaction` whenever it is needed. This could likely be solved using some sort of a state for each thread. The state, including registers and the stack, is saved when a transaction enters `BeginTransaction` and a transaction that detects that another transaction needs a restart should reset the saved state on the thread that is executing the conflicting transaction.

## 6.8 Statistics

There is a possibility to collect statistics of the transactions. A data structure called `Statistics` is available, which contains a lot of variables that are changed during the execution of the system. For example, the number of started transactions and number of restarts are stored. The data structure is protected by a mutex semaphore called `statMutex`. It is very easy to collect more statistics in the system, a variable is added to the structure and then it is used to collect the wanted statistics. The following statistical data is currently collected:

- Number of started transactions
- Number of committed transaction
- Number of restarts
- Number of missed AVIs
- Number of missed RVIs
- Number of missed deadlines
- Number of transactions who read data items with the change flag set
- Maximal number of concurrent transactions at a given time
- Total run time of the system
- Total execution time of transactions, can, for example, be used to calculate the mean execution time of the transactions
- Total deadline miss time, i.e., if a transaction misses its deadline, the excess time is added to this variable
- Number of triggered calls

A lot more is derived using this information. For example, throughput in transactions per second and mean execution time for transactions. The whole statistics structure could be read with some external tool when the system is deployed on the engine control unit.

## 6.9 Compiling with Diab 5.0a

The Rubus-ported version of the engine control was compiled by Saab and Mecel using an older version of the Diab-compiler. The version available during the project was Diab 5.0a and some parameters to the compiler were needed for it to work.

1. `-Xmismatch-warning`. This option turns off the fact that the compiler treats warnings of type mismatches as errors.
2. `-Xpreprocessor-old`. To use the same preprocessor as the old compiler.

## 6.10 Benchmark suite

A small benchmark suite was developed during the thesis project, two subsystems in the engine control software was studied, FuelMaster and the task that is run every 20ms. FuelMaster calculates the amount of fuel to inject into the engine. The common variables of the two subsystems were found and some transactions that use the variables were implemented. The common variables are:

- In.n_Engine [w/r]
- In.p_AirBefThrottle [w/r]
- In.p_AirInlet [w/r]
- In.Q_AirInlet [w/r]
- In.p_AirAmbient [w]
- In.T_Engine [w]
- In.ST_IgnitionKey [w]
- In.v_Vehicle [w]
- In.U_Batt [w]

A [w] in the list means that the variable is written in both subsystems and [r] means that it is read. Further needed development of the benchmark suite is left as future work.

# Chapter 7

# Related work

This chapter presents some related work in the areas real-time databases, concurrency control, scheduling and engine control.

## 7.1 Real-time databases

Ramamritham [19] gives a good overview of real-time databases, characterizing data and transactions in real-time database systems. Also, active databases are discussed with their pros and cons and some other issues with real-time databases are also covered, like I/O and overload management.

### 7.1.1 Characteristics of transactions

The characteristics of transactions are introduced in the paper, the transactions are split into three groups, namely *write-only*, *update* and *read-only*. Write only transactions are, if the engine control system is considered, reading the sensors and then writing the values into the database. Update transactions are the ones that read a set of data items, perform some calculations and then write the results into the database. A read-only transaction get a derived value from the database and sends it to an actuator.

Another distinction between transactions is done by the way they handle deadline misses. Three types are introduced, *hard*, *soft* and *firm*. Hard and soft deadlines are described in section 2.4. Firm deadlines mean that if a deadline is missed, the value of the transaction is equal to zero but it does not imply a catastrophe, instead the transaction can be aborted without loosing any value in the system.

### 7.1.2 Active databases

Berndtsson, Hansson [7] and Ramamritham [19] discuss the issues in merging active databases and real-time databases. It is described that the classical active databases follow the event-condition-action rule (ECA). ECA means that when a specified event E occurs and a condition C holds, perform the action A. The papers also describe how this ECA-rule could be made more appropriate for real-time applications; "on event E and if condition C holds, perform action A within t seconds". The active behavior described earlier in the report and that is implemented in the data repository is restricted to updating invalid data items when trying to read them. If the implementation is mapped to the real-time ECA-rule, we can see the event as reading a data item, the condition is that the data item is invalid and the action is to update it. The timing comes in when using the data dependency graph scheduling where a deadline for updating data items is given.

The active behavior in the data repository is related to the work by Adelberg, Garcia-Molina and Kao [2] where they talk about on-demand updating of stale[1] data items. They have separated the user-transactions that both read and write data in the database from updates that only update data items; each update only updates one data item. The updates are enqueued upon arrival in the system and when a user-transaction tries to read a stale object, the update queue is searched for an update of the data item. If the update is found, then it is applied. On demand updating minimizes the number of stale data read. There are two other algorithms introduced in the article, namely "do updates first" where updates are applied as soon as they arrive in the system and "do transactions first" where updates only are applied when no transactions exist in the system. Also, "split updates" is described, here updates are applied to high priority data on arrival and only when no transactions exist on low priority data. The results in the article show that the on demand approach gives the best overall performance. Although, it can be difficult to see what updates are applicable to a certain item, for example, if an item that represents a mean value of a set of stocks is read it can be hard to know what updates to apply in order to get a fresh mean value. This work can be compared to the implemented data repository where the "mean-value-problem" is solved by having the data dependency graph. There is a difference between updates and transactions in the repository too, even though updates are

[1]A stale data item is one that is too old to use in a calculation.

transactions, the update transactions do not read any data items from the repository before writing. Also, the update transactions are not enqueued, instead they are considered as ordinary transactions and scheduled like them. The update-functionality is only used when one of these updates have not been applied often enough.

## 7.2 Concurrency control

This section will cover related work in the concurrency control area, mostly for real-time applications.

### 7.2.1 Pessimistic concurrency control

Atzeni, Ceri, Paraboschi and Torlone [6] give a nice introduction to pessimistic concurrency control, describing two-phase locking and strict two-phase locking. Abbott and Garcia-Molina [1] introduce the two-phase locking - high priority (2PL-HP) algorithm which is more suited for real-time applications. They also cover some priority assignment policies like earliest deadline first and least slack first. In the results from their experiments they find that two-phase locking-high priority performs worst of the algorithms tested, especially when combined with earliest deadline first scheduling. Their test bed is a single processor and a database without any secondary storage so it is similar to the set up in the data repository in this report. Two-phase locking - high priority was implemented in the data repository since it provides the constructs for pessimistic concurrency control and should make it easier to implement other pessimistic algorithms.

### 7.2.2 Optimistic concurrency control

Kung and. Robinson [15] introduced optimistic concurrency control. They let transactions run unhindered until the commit point. There the transactions try to validate and if the validation fails the transaction restarts. Haritsa, Carey and Livny [11] describe and evaluate the optimistic concurrency control - broadcast commit (OCC-BC). In the description, a transaction that gets a broadcast message can restart immediately, this is not possible in the data repository since Rubus does not provide the necessary constructs for it. The results from the article show that OCC-BC performs better than 2PL-HP when there are few conflicts in the system.

## 7.3 Scheduling

Liu and Wayland [16] give a view of the classic scheduling algorithms for real-time systems, namely earliest deadline first and rate monotonic. They conclude that the algorithms are optimal under certain conditions. Rate monotonic is optimal among the fixed priority algorithms and earliest deadline first if it is possible to schedule dynamically.

# Chapter 8

# Conclusions

## 8.1 Summary

The real-time data repository implemented can handle transactions in a real-time fashion. Transactions have deadlines and can be scheduled using earliest deadline first [16] or by using Rubus static priority scheduling where transactions are scheduled by the priority of the threads they execute in. The following is implemented in the repository:

- A variant of the broadcast commit [11] protocol.
- The two-phase locking - high priority [1] concurrency control algorithm.
- Earliest deadline first scheduling algorithm [16].
- Data dependency graph scheduling algorithm [10].
- Mechanisms for gathering statistics.

The broadcast commit protocol uses a global list as a broadcast channel. The transactions cannot restart immediately when they get a broadcast message, instead they have to execute until their verification-points, a solution to this problem is outlined in chapter 8. A transaction that is successfully validated gets to commit and a transaction with higher priority has a higher probability to get successfully validated. The implementation is not true broadcast commit [11] since the transactions do not restart until the validation point and they send broadcast messages whenever they detect a conflict.

The idea with two-phase locking - high priority is that a higher priority transaction should never have to wait for a lower priority one, instead the higher priority transaction simply aborts the lower priority one. The abortion functionality is implemented in the data repository by having the high priority transaction set a flag on the lock holding lower priority transaction and then waiting for it to roll back its changes and release the locks. The waiting done here is very short and required since otherwise the higher priority transaction could read intermediate results that the lower priority transaction has written to the repository. This is the only waiting needed since a transaction that aborts another one always has higher priority.

When scheduling with the earliest deadline first (EDF) algorithm, the transaction with the earliest deadline executes first. It is implemented by sending and receiving signals. A transaction that wants to be scheduled sends a message to a scheduler thread that decides what transaction to start. There is a restriction in the current implementation, namely that the EDF algorithm works on top of Rubus scheduling. As described in section 5.4, a transaction with a very early deadline can be delayed because of the fact that it is executing in a low priority Rubus-thread. A solution to this is outlined in section 5.4. The data dependency graph algorithm keeps track of how a data item is dependent on other data items in the repository. This is done by keeping a graph that shows the data dependencies. The graph in the repository is stored as an adjacency matrix [9]. To get the parents of a data item updated, the function `UpdateDB` is called. It takes a deadline and data item as parameters and generates a schedule that updates as many as possible of the data item's parents.

It is possible to collect statistics on transactions. Currently quite a lot of information is provided, for example throughput in transactions per second, AVI misses, RVI misses and mean execution time. It is easy to collect statistics, a structure with all information is kept and the whole structure could be read from the application with some external tool.

Some features missing in Rubus have been identified as vital to the future development of the data repository, they are:

- Mutex semaphores without priority ceiling and ownership. In order to implement two-phase locking with high priority a mutex semaphore without priority ceiling was needed. Ownership makes standard solutions to synchronization problems more complex too, for example, monitors can not be implemented when the mutex semaphores have owners. Arcticus Systems have given us a version of Rubus with this feature implemented but the feature is not available in the original Rubus version.

- Possibility to set dynamic priorities on blue threads. This could be useful when doing future improvements on the repository.
- Possibility to "turn off" Rubus thread scheduling. In section 5.4 the problem with using earliest deadline first scheduling of transactions is described. A transaction with an early deadline that is in a thread with a low priority is delayed until all transactions in higher priority threads are finished. The solution would be to turn off Rubus scheduling of blue threads and instead let every thread run when it is ready to do so, then they can tell the EDF-scheduler their deadlines and it can decide what transaction to start.
- State functionality on blue threads. In order to to restart a transaction immediately; it would be nice to be able to store the current state of a thread when the transaction starts and then be able to set this state on the thread so that it continues from where the state was stored. This functionality exists in Unix and the information needed to store should be about the same as when doing a context switch. In Unix, the function call needed to save the state is called `setjmp(...)` and to reset the state, `longjmp(...)`.

## 8.2 Future work

A few suggestions to future work is provided here:

- Develop a data dependency graph tool. It is very difficult to create the graphs with their adjacency matrixes even if they are quite small as the example in figure 6.1. It would be good to have a tool that handles the graphs, perhaps in a graphical way. The tool should be able to add new data items in the graph, add dependencies between data items etc. The tool could handle the actual data repository too, adding and removing the data items there as well. Perhaps could "stubs" for the transactions be generated so that the correct data items are read and written, the developer could then just add the calculations needed in order to derive the data item. When everything is configured, the tool should generate a set of .c and .h-files that should be included in the project.

- Possibility to restart transactions. As mentioned earlier, there is no way of restarting a transaction when it is needed. This gives quite a big overhead since a transaction could be quite long.

- The scheduling in Rubus should somehow be overridden in order to get true earliest deadline first scheduling.

- Set up a better simulation environment. In order to get good results from the runs of a system using the repository, a better simulation platform is needed. As mentioned in section 4.2 the resolution of the basic timer in the Rubus Windows simulator is only 10ms. This is way to big to see any real differences between different algorithms since the execution times of the transactions in the engine control software is much smaller than 10ms. Instead, Rubus could be run as the only operating system on an i386-processor to both get the file handling so that logs could be kept, and get a higher resolution of the timer. Another option is to clean up the engine control software so that the engine control unit can be used as an evaluation platform. This adds some problems though, for example, a tool to read data from the engine control unit is needed, this could be the software that Saab is using today or some sort of a CAN-analyzer that simply reads data from the CAN network.

- Currently an improved benchmark suite is being developed in order to evaluate the new algorithms. It will cover many issues with the engine control system and provide a good evaluation platform.

- Evaluate the data dependency graph algorithm. The dependency graph algorithm is new and untested, therefore it is interesting to evaluate the two update schedule generating approaches available, i.e., depth first and breadth first.

# Bibliography

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

[2] 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.

[3] Arcticus Systems AB. *Rubus OS - API Services.*

[4] Arcticus Systems AB. *Rubus OS - Reference manual.*

[5] Arcticus Systems AB. *Rubus OS - Tutorial*, December 2001.

[6] P. Atzeni, S. Ceri, S. Paraboschi, and R. Torlone. *Database Systems - Concepts, Languages and Architectures.* McGraw-Hill Publishing Company, 1999.

[7] M. Berndtsson and J. Hansson. Issues in active real-time databases. In *Proceedings of the International Workshop on Active and Real-Time Database Systems*, pages 142–157, 1995.

[8] G.C. Buttazzo. *Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and applications.* Kluwer Academic Publishers, 1997.

[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to algorithms.* McGraw-Hill Book Company, second edition, 2001.

[10] T. Gustafsson. Scheduling of updates of base and derived data items in real-time databases. Ongoing work, 2003.

[11] J.R. Haritsa, M.J. Carey, and M. Livny. On being optimistic about real-time constraints. In *In Proceedings of the 9 ACM Symposium on*

*Principles of Database Systems*, pages 313–343, Nashville, 1990. ACM Press.

[12] J. Huang, J.A. Stankovic, K. Ramamritham, and D. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *Proceedings of the 17th Conference on Very Large Databases*, Sept 1991.

[13] M. Jinnelöv. Analysis of an engine control unit in preparation of a real time database. Master's thesis, Linköping university, October 2002.

[14] C.M. Krishna and G. Shin. *Real-time systems.* McGraw-Hill, 1997.

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

[16] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of ACM*, 20(1), January 1973.

[17] J.W.S. Liu. *Real-time Systems.* Prentice Hall, Inc., 2000.

[18] L. Nielsen and L. Eriksson. Course material vehicular systems. Technical report, Department of Information Systems, Linköping University, 2001.

[19] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1(2):199–226, 1993.

[20] Saab Automobile, Södertälje. *Verkstadsmanual 9-5*, 1996.

[21] A. Silberschatz and P. Galvin. *Operating system concepts.* Addison-Wesley Publishing Company, 5th edition, 1998.

[22] J. Stankovic, S. Son, and J. Hansson. Misconceptions about real-time databases. *Computer*, 32:29–36, 1999.

[23] WindRiver Systems. *Diab C/C++ Compiler for 68K/CPU32*, May 2002.

# Index