Mälardalen University Licentiate Thesis
No.9

COMET: A Component-Based Real-Time Database for Vehicle Control-Systems

Dag Nyström

May 2003

Department of Computer Science and Engineering
Mälardalen University
Västerås, Sweden

ISBN 91-88834-46-8
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

Vehicle control-systems have evolved from small isolated controllers to complex distributed computer-systems. These systems include nodes spanning from simple 8-bit micro-controllers with a minimum of memory to complex 32-bit processors with vast resources. The main motivation for this evolution is the need for increased functionality in motor vehicles. Examples of such functionality include momentary fuel consumption measurements, anti-spin systems, and computerized diagnostics of vehicle-status. The control of the increased functionality requires the handling and maintenance of larger volumes of data, and has created a need for a uniform and efficient way to access and maintain this data. A real-time database management system could satisfy this requirement but an extensive survey of commercial and experimental database management systems has shown that there is currently no database system suitable for vehicle control systems available. In today's systems, data management is performed in an ad-hoc fashion at a low level of abstraction, using internal data-structures, e.g., shared variables and structures. This approach requires that the consistency of the data is maintained by the application, by, for example, resolving data access conflicts through the use of semaphores.

This thesis presents a flexible and configurable database management system designated COMET, suitable for embedded systems and in particular, vehicle control-systems. To be able to handle the varying requirements imposed by different systems, COMET emphasizes configurability and tailorability, by adopting a component-based architecture.

The result of this research is the implementation of COMET BaseLine, which is an instance of COMET suited to a particular vehicle control-system. The required behaviour of this database is based on requirements gathered from a case study performed at Volvo Construction Equipment Components AB in Eskilstuna. To fulfill these requirements, a concept called database pointers has been introduced and implemented. Database pointers provide controlled direct access to individual data elements in the database, efficiently and temporally deterministic, providing at the same time a high level of abstraction.

Acknowledgements

This thesis is presented as a partial fulfillement for the degree of Licentiate in Engineering, at the department of computer science and engineering, Mälardalen university, Sweden. The research presented in this thesis has been performed within the COMET (component-based real-time embedded database) project, jointly executed by Mälardalen university and Linköping university.

I sincerely thank my supervisors Christer Norström from Torshälla, and Jörgen Hansson for their invaluable support and the feedback they supplied on my work.

I wouldn't be here without the support and cooperation of my colleague and co-researcher, Aleksandra Tesanovic, at Linköping university.

This work has been funded by ARTES (A network for real-time research and graduate Education in Sweden). I would like especially to thank Hans Hansson, programme director of ARTES.

The cooperation of industry has been a vital part of this project and I would like to thank the staff at Volvo Construction Equipment Components AB, in Eskilstuna, Sweden, for the two successful weeks that we spent there. A special thanks to Nils-Erik Bånkestad for his support during the entire project. Furthermore, many thanks to Bengt Gunne at Upright Database Technology, Uppsala Sweden, for sharing his deep knowledge of embedded databases with us.

Also thanks to all my colleagues at the department for making these years so memorable. I would like to direct my special gratitude to the following: Thomas Nolte, my dear friend throughout our time at the university, Harriet Ekwall, for solving all those insolvable problems, Jukka Mäki-Turja, colleague and fellow skydiver, Joel, Daniel, Anders P., Mic, Anders "Fnurkelmannen" W., and Kristian.

Many thanks to Susanne Eriksson, Ingela Hedman, and Gisèle Mwepu for their great work implementing the COMET BaseLine. I wish you the best.

Finally a warm hug to my wife Jenny and my daughter Liv, for supporting me and loving me during these years.

So, buckle up folks, here we go!

Dag Nyström, Västerås in May 2003

Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Contribution . . . 3
- 1.3 The Research Work and Method . . . 3
- 1.4 Conclusions and Future Work . . . 5
- 1.5 Thesis Outline . . . 6

2 Paper A: Real-Time and Embedded Databases in Practice **9**
- 2.1 Introduction . . . 11
  - 2.1.1 An example application . . . 14
- 2.2 Database Systems . . . 18
  - 2.2.1 Traditional Database Systems . . . 18
  - 2.2.2 Embedded Database Systems . . . 18
  - 2.2.3 Commercial Embedded DBMS: a Survey . . . 21
  - 2.2.4 Current State-of-the-art From Research Point of View 44
  - 2.2.5 Real-Time Properties . . . 45
  - 2.2.6 Distribution . . . 49
  - 2.2.7 Transaction Workload Characteristics . . . 53
  - 2.2.8 Active Databases . . . 57
  - 2.2.9 Observations . . . 62
- 2.3 Summary . . . 63
  - 2.3.1 Conclusions . . . 63
  - 2.3.2 Future Work . . . 63

3 Paper B: Data Management Issues in Vehicle Control Systems: a Case Study **71**
- 3.1 Introduction . . . 73

3.2 The Case Study 74
3.2.1 Rubus 76
3.2.2 VECU 76
3.2.3 IECU 78
3.2.4 Data Management Requirements 80
3.3 Modeling the System to Support a RTDB 83
3.3.1 Data Management Implications 85
3.3.2 DBMS Design Implications 86
3.3.3 Mapping Data Requirements to Existing Database Platforms 88
3.4 Conclusions 89

4 Paper C: Database Pointers: a Predictable Way of Manipulating Hot Data in Hard Real-Time Systems 93
4.1 Introduction 95
4.2 Background and Related Work 96
4.2.1 Relational Query Processing 97
4.2.2 Tuple Identifiers 97
4.2.3 Related Work 98
4.3 Database Pointers 99
4.3.1 The `DBPointer` Data Type 100
4.3.2 The Database Pointer Table 100
4.3.3 The Database Pointer Interface 102
4.3.4 The Database Pointer Flag 103
4.3.5 Application Example 104
4.4 Concept Evaluation 106
4.5 Conclusions and Future Work 107

5 Paper D: The COMET BaseLine Database Management System 111
5.1 Introduction 113
5.2 Aspect-Oriented Software Development 114
5.3 ACCORD 116
5.4 RTCOM: Real-Time Component-Model 116
5.5 The Architecture of COMET 118
5.6 The Components in COMET 120
5.6.1 User Interface Component 120
5.6.2 Scheduling Management Component 121
5.6.3 Transaction Management Component 121
5.6.4 Index Management Component 123

5.6.5 Lock Management Component 123
5.6.6 Memory Management Component 124
5.6.7 Checkpointing and Recovery Component 124
5.7 The Aspects in COMET 125
5.7.1 Concurrency-Control Aspect 125
5.7.2 Logging and Recovery Aspect 125
5.8 COMET BaseLine 126
5.8.1 Relational Processing 126
5.8.2 Database Pointer Processing 128
5.8.3 COMET BaseLine Component-Model 128
5.9 COMET BaseLine Architecture 131
5.9.1 Relational User Interface Component 131
5.9.2 Relational Transaction Management Component . . . 132
5.9.3 Database Pointer User Interface Component 133
5.9.4 Database Pointer Transaction Management Component 134
5.9.5 Index Management Component 134
5.9.6 Memory Management Component 135
5.10 Conclusions and Future Work 135

List of Publications

Publications Included in Thesis

Paper A
Aleksandra Tešanović, Dag Nyström, Jörgen Hansson & Christer Norström
Embedded Databases for Embedded Real-Time Systems: A Component-Based Approach
MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-43/2002-1-SE
Mälardalen Real-Time Research Centre, Mälardalen University, January 2002.

This report is a survey that covers the following areas: (i) Real-time databases (ii) Embedded databases (iii) Component-based databases (iv) Component-based software engineering. It is a background study for the field of research for the entire COMET project, i.e., a state of the arts.
The surveys on real-time and embedded databases were written by Dag, and the surveys over component-based databases and component-based software engineering where written by Aleksandra

Paper B
Dag Nyström, Aleksandra Tešanović, Christer Norström, Jörgen Hansson & Nils-Erik Bånkestad
Data Management Issues in Vehicle Control Systems: a Case Study
In proceedings of the 14th Euromicro Conference on Real-Time Systems, pages: 249 -256, IEEE, Vienna, Austria, June 2002.

This paper presents a case-study on data management for a vehicle control-system developed at Volvo Construction Equipment Components AB, Eskilstuna, Sweden. Furthermore, it elaborates on how to redesign the system to incorporate a real-time database management system into it. Finally it discusses how to design a real-time database management system that suits the control-system.
Dag and Aleksandra jointly focussed on the data management issues in the current system. Dag also investigated the database requirements and its integration with the control-system. Nils-Erik provided valuable knowledge of the application domain.

Paper C
Dag Nyström, Aleksandra Tešanović, Christer Norström & Jörgen Hansson
Database Pointers: a Predictable Way of Manipulating Hot Data in Hard Real-Time Systems
In proceedings of the 9th International Conference on Real-Time and Embedded Computing systems and Applications, pages: 623 -634, Tainan, Taiwan, February 2003.

This paper proposes a new concept of interfacing a database. This concept, which resemble of pointer operations, writes data directly to the memory location inside the database where the data is stored, thus the index lookup routine is bypassed. This concept is beneficial for real-time control systems.
Responsible for the general ideas and main author and driving force of the paper during the writing process was Dag. Aleksandra aided in the writing process, as well as giving feedback and improvements on the general ideas.

Paper D
Dag Nyström, Aleksandra Tešanović, Christer Norström & Jörgen Hansson
The COMET BaseLine Database Management System
MRTC Report 98/2003, ISSN 1404-3041 ISRN MDH-MRTC-98/2003-1-SE
Mälardalen Real-Time Research Centre, Mälardalen University, April 2003.

This report presents the general ideas of the experimental component-based real-time database management system COMET. Furthermore, it describes the COMET BaseLine which is an instance of COMET suited for a particular vehicular control-system. Finally, it briefly explains the design method and component-model used during the development on COMET.
All authors are responsible for different parts of the main ideas presented in this report. The driving author of this report was Dag.

Publications by the Author, Not Included in Thesis

Aleksandra Tešanović, Dag Nyström, Jörgen Hansson & Christer Norström
Integrating Symbolic Worst-Case Execution Time Analysis with Aspect-Oriented System Development
In proceedings of OOPSLA 2002 Workshop on Tools for Aspect-Oriented Software Development, Seattle, USA, November 2002.
This workshop paper proposes a method to calculate worst case execution time for aspect oriented software.

Aleksandra Tešanović, Dag Nyström, Jörgen Hansson & Christer Norström
Towards Aspectual Component-Based Development of Real-Time Systems
In proceedings of the 9th International Conference on Real-Time and Embedded Computing systems and Applications, pages: 279 -298, Tainan, Taiwan, February 2003.
This conference paper presents the COMET component model which can be used in combination with aspect oriented programming. The paper also presents an idea how to implement the COMET DBMS using these components

Aleksandra Tešanović, Dag Nyström, Jörgen Hansson & Christer Norström
Aspect-Level Worst-Case Execution Time Analysis of Real-Time Systems Compositioned Using Aspects and Components
Proceedings of the 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming, Poland, Elsevier, May 2003. To appear.
This workshop paper proposes a method to calculate worst case execution time for aspect oriented real-time software.

Chapter 1

Introduction

Most functionality in a modern car is in one way or another controlled by a computer. For example, the speedometer on the dash board may, instead of being connected to the engine by a rotating wire, be connected to an onboard computer feeding it with the current speed of the vehicle.

The advantages of computer control of functionality are many. First of all, it reduces the amount of electrical wiring in the car. Instead of having an individual wire for each subsystem, different subsystems can share a single digital cable, called a data bus. Secondly, by supplying a central control-system with all vehicle data, such as speed, engine temperature, fluid levels, etc, this information is made available to different subsystems. The vehicle speed, for example, is supplied to the speedometer, but it can also be an important input to the anti-lock braking system (ABS). A third reason is that new functionality can be easily implemented in the car, requiring only computer software for its control. Examples of new functionality are, anti-skid systems to keep the vehicle steerable during emergency braking, air bags, and on-the-fly diagnostics and statistics about the conditions in different subsystems in the vehicle.

An investigation made by Volvo suggests that the use of computer systems in vehicle control is increasing, and is expected to continue to increase, by 7-10% annually [1]. Adding functionality to the car increases the volume of data to be handled by the control-system. This data requires more and more complex maintenance.

To further complicate matters, vehicular computer-systems are often distributed to several loosely coupled nodes, called electronic control units (ECUs), throughout the vehicle. These nodes vary, both in computational power and

the amount of resources available. Some nodes may be 8-bit micro-controllers with a small on-board memory while other nodes may be powerful processors with vast resources. These nodes need to be connected in order to share common information, i.e., the data distributed among the nodes.

In today's systems, data is handled in an ad hoc fashion, using internal data structures, i.e., shared variables. This means that the data must be kept consistent by the application. There are many types of consistency to handle, such as (i) concurrency consistency, i.e., data accesses must be serialized, and this is enforced using mechanisms such as semaphores. Further, (ii) data in real-time systems has a limited temporal validity, both absolute and relative [2]. Finally, (iii) data must be kept consistent among nodes in the distributed system. It is essential to consider all these issues while designing, maintaining and extending the application and to be aware of the increasing risk of software errors.

The current data management approach is becoming increasingly insufficient as systems become increasingly complex and a need for data management on a higher level of abstraction has emerged. One way of providing such a data management would be to incorporate a real-time database management system (DBMS) into the system. DBMSs provide a generic and uniform way of accessing data. Furthermore, they can automatically enforce consistency mechanisms, including temporal validity control and distribution control.

1.1 Motivation

A number of database management systems intended for embedded systems are currently available on the market. These include Polyhedra [3], Pervasive.SQL [4], Berkeley DB [5], and TimesTen [6]. Some of these systems are small enough to fit in an environment such as a vehicular control system. None, however, incorporates any real-time or deterministic behaviour. This means that they are not behaviourally analyzable, which is one requirement for deployment in a vehicular system.

On the other hand, there are a number of database management systems that do incorporate real-time algorithms, such as DeeDS [7], RODAIN [8], STRIP [9], and BeeHIVE [10]. Some of these systems, such as the DeeDS system, are intended for hard real-time systems, and might therefore be predictable and analyzable enough to be deployed in such a system. However, none is intended for embedded environments and none could, in its current form, fit into a vehicle control system.

Further, the varying requirements of data management in different vehicle control systems, require different behaviours of the database management system used. In other words, a DBMS suitable for use in one vehicle might not be suitable for use in a different vehicle. In fact, nodes within the same vehicle might be so heterogeneous that a DBMS for one type of node might not fit another.

One solution to this would be to develop an "in house" database manager to suit one particular vehicular system. However implementing, and even more importantly validating, a database management system for a vehicle control-system is not a trivial task, and thus this solution is not time-efficient.

Our approach is to develop a tailorable database management system using a component-based approach. This approach would enable the database to be configurable to suit a number of different types of vehicular control-systems.

1.2 Contribution

In this thesis we will identify the data management requirements of a specific industrial vehicle control-system. We show how the control-system can be redesigned to incorporate a real-time database management system. Further, we have designed and implemented a real-time database suited to this control-system. To permit this implementation, we have introduced a new data access concept for database management systems called database pointers.

The database, named COMET BaseLine, is an instance of the COMET DBMS, a component-based database management system, suited to embedded systems, in particular vehicle control-systems.

1.3 The Research Work and Method

One central aspect of this research has been close interaction with industry. This has enabled us to work on solving research questions relevant to industrial systems in practice. To achieve an understanding of both the academic problems surrounding real-time data management, and data management requirements from vehicle control-systems, extensive background work has been performed.

The research began by investigating the current state-of- the-art in real-time and embedded database management systems [11]. We performed an extensive survey of commercially available embedded DBMSs, investigating a handful of systems on the basis of a number of criteria. In addition to this survey, a second

survey was performed, this time, of experimental real-time DBMSs, developed in academia. The latter survey also provided us with documentation of basic database management systems theory, which was simultaneously investigated.

Equipped with this new knowledge, an industrial stay was arranged. During a period of two weeks, we visited Volvo Construction Equipment Components AB (Volvo CE), in Eskilstuna, Sweden. Volvo CE has developed a vehicular control-system used in construction vehicles such as wheel loaders, articulated haulers, and excavators. In this case study [12], we investigated the data management currently used in these systems, and discussed the possibilities of redesigning the system to incorporate a real-time DBMS. Furthermore, we considered how such a DBMS should be designed to fulfil the requirements of the control system.

Reviewing the earlier survey, we found no existing DBMS suitable for a system such as this. The commercial DBMS systems were, in some cases, small enough to fit in the nodes of the control-system but they had no hard real-time database properties and were therefore insufficiently reliable for integration with such a system. Some real-time DBMS systems on the other hand, have mechanisms which support a predictable behaviour but these are not intended for use in resource-constrained embedded systems.

This realization provided further motivation for the development of a tailorable DBMS that could be configured to suit a particular system. Initial work on a suitable architecture for COMET was begun. A component-based approach was used to enable parts of the database to be easily exchanged, to permit the necessary functionality to be plugged in and unnecessary functionality to be replaced. One problem, however, was that some functionality cross-cut several components, inhibiting the modularity we aimed at. To solve this problem, an aspect-oriented approach was used. Aspect-oriented software development (AOSD) enables certain functional properties (aspects) to be extracted and treated as separate artifacts [13]. These aspects could then be woven into the system at compile-time. An architectural framework for COMET was developed, this consisting of a set of components and a set of aspects.

The next step was to implement an instance of COMET suitable for the system investigated in the case study at Volvo CE. One problem still to be solved was how to allow the control-tasks to access the database in a predictable way without too much overhead. A database access method, called database pointers, was developed [14]. Database pointers are used to access individual data elements in the same way as shared variables are accessed but in this case the data elements reside inside a database controlled by a DBMS. To ensure database consistency, these accesses must be performed in a controlled way.

Finally the implementation of the first instance of COMET could begin. This instance, called COMET BaseLine, implements a main-memory DBMS. This DBMS can be accessed using the relational interface, which provides a subset of the database query language SQL. Furthermore, COMET can be accessed using database pointers. The database, which is suited to the control-system investigated at Volvo CE, is now fully functional and has a memory footprint of around 20kb.

1.4 Conclusions and Future Work

In this thesis, we have presented a real-time database management system, called COMET, suited to embedded real-time systems, in particular vehicular systems. COMET has a component-based architecture that supports aspects. This enables functionality to be tailored in many different ways, such as using parameterization, defining aspects, and replacing components. This is important because the requirements of different embedded applications vary. A first instance of COMET, called COMET BaseLine has been presented. This instance is specifically suited to a vehicular system developed at Volvo Construction Equipment Components AB (Volvo CE) in Eskilstuna, Sweden. The case study of the data management requirements of this system has been presented in the thesis. A concept called database pointers has been introduced to permit the hard real-time controlling system to efficiently access the database This concept has been successfully implemented in COMET BaseLine. This thesis leaves certain questions unanswered and therefore much work in this area remains to be performed. Examples of incomplete or pending research include:

- Integration, verification and testing of COMET BaseLine together with the Volvo CE control-system.
- Development of a concurrency control algorithm suited to database pointers. Ideally, hard database pointer transactions would be allowed to run without blocking or, at least, with bounded blocking times.
- Verification of the COMET architecture for other types of control-systems to determine the flexibility and configurability of COMET.
- Further investigation of the benefit of using aspects to tailor the behavior of a COMET instance.

- Development of configuration and analysis tools for COMET, to aid the developer of a control-system in choosing suitable components for COMET.

1.5 Thesis Outline

The thesis is outlined as follows: In paper A, we survey commercial embedded DBMSs as well as experimental research DBMSs, using a number of criteria. We continue in paper B with a case study of the data management in a vehicle control-system used in practice. In this paper we also elaborate on how to redesign the system to incorporate a real-time database. The demands on the real-time database are also presented. In paper C we introduce a new concept called database pointers that is used to access individual data elements in a real-time database in an efficient and temporally predictable way. Finally, in paper D we present the concepts of our experimental database management system, COMET. The architecture of COMET is described, as well as the individual components and aspects identified in COMET. Furthermore, the first implemented instance of COMET, the COMET BaseLine, is presented. COMET BaseLine implements database pointers and is tailored to fit the vehicle control-system described in paper B.

Bibliography

[1] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano - a Revolution in On-Board Communications. Technical report, Volvo Technology Report, 1998.

[2] K. Ramamritham. Real-Time Databases. *International Journal of distributed and Parallel Databases*, 1(2):199–226, 1993.

[3] Polyhedra Plc. http://www.polyhedra.com.

[4] Pervasive Software Inc. http://www.pervasive.com.

[5] Sleepycat Software Inc. http://www.sleepycat.com.

[6] TimesTen Performance Software. http://www.timesten.com.

[7] S. F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring. DeeDS Towards a Distributed and Active Real-Time Database System. *ACM SIGMOD Record*, 25(1):38–40, 1996.

[8] J. Lindstrom, T. Niklander, P. Porkka, and K. Raatikainen. A Distributed Real-Time Main-Memory Database for Telecommunication. In *Proceedings of the Workshop on Databases in Telecommunications*, pages 158–173. Springer, September 1999.

[9] B. Adelberg, B. Kao, and H. Garcia-Molina. Overview of the STanford Real-time Information Processor (STRIP). *SIGMOD Record*, 25(1):34–37, 1996.

[10] J. A. Stankovic, S. H. Son, and J. Liebeherr. *Real-Time Databases and Information Systems*, chapter BeeHive: Global Multimedia Database Support for Dependable, Real-Time Applications, pages 409–422. Kluwer Academic Publishers, 1997.

[11] Aleksandra Tešanović, Dag Nyström, Jörgen Hansson, and Christer Norström. Embedded Databases for Embedded Real-Time Systems: A Component-Based Approach. Technical Report MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-43/2002-1-SE, Dept. of Computer Engineering, Mälardalen University, January 2002.

[12] Dag Nyström, Aleksandra Tešanović, Christer Norström, Jörgen Hansson, and Nils-Erik Bånkestad. Data Management Issues in Vehicle Control Systems: a Case Study. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pages 249–256. IEEE Computer Society, June 2002.

[13] Aleksandra Tešanović, Dag Nyström, Jörgen Hansson, and Christer Norström. Towards Aspectual Component-Based Development of Real-Time Systems. In *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications*, pages 278–298, February 2003.

[14] Dag Nyström, Aleksandra Tešanović, Christer Norström, and Jörgen Hansson. Database Pointers: a Predictable Way of Manipulating Hot Data in Hard Real-Time Systems. In *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications*, pages 623–634, February 2003.

Chapter 2

Paper A: Real-Time and Embedded Databases in Practice

Aleksandra Tešanović, Dag Nyström, Jörgen Hansson & Christer Norström

This chapter contains the parts of the technical report “Embedded Databases for Embedded Real-Time Systems: A Component-Based Approach” written by me. The full technical report also investigates component-based software development and component-based databases.

MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-43/2002-1-SE, Mälardalen University.

Abstract

As complexity and the amount of data are growing in embedded real-time systems, the need for a uniform and efficient way to store data becomes increasingly important, i.e., database functionality is needed to provide support for storage and manipulation of data. However, a database that can be used in an embedded real-time system must fulfill requirements both from an embedded system and from a real-time system, i.e., at the same time the database needs to be an embedded and a real-time database. The real-time database must handle transactions with temporal constraints, as well as maintain consistency as in a conventional database. The main objectives for an embedded database are low memory usage, i.e., small memory footprint, portability to different operating system platforms, efficient resource management, e.g., minimization of the CPU usage, ability to run for long periods of time without administration, and ability to be tailored for different applications. In addition, development costs must be kept as low as possible, with short time-to-market and a reliable software. In this report we survey embedded and real-time database platforms developed in industrial and research environments. This survey represents the state-of-the-art in the area of embedded databases for embedded real-time systems. The survey enables us to identify a gap between embedded systems, real-time systems and database systems, i.e., embedded databases suitable for real-time systems are sparse. Furthermore, it is observed that there is a need for a more generic embedded database that can be tailored, such that the application designer can get an optimized database for a specific type of an application.

2.1 Introduction

Digital systems can be classified in two categories: general purpose systems and application-specific systems [1]. General purpose systems can be programmed to run a variety of different applications, i.e., they are not designed for any special application. Application-specific systems are designed for a specific application. Application-specific systems can also be part of a larger host system and perform specific functions within the host system [2], and such systems are usually referred to as *embedded systems*. An embedded system is implemented partly on software and partly on hardware. When standard microprocessors, microcontrollers or DSP processors are used, specialization of an embedded system for a particular application consists primarily on specialization of software. In this report we focus on such systems. An embedded system is required to be operational during the lifetime of the host system, which may range from a few years, e.g., a low end audio component, to decades, e.g., an avionic system. The nature of embedded systems also requires the computer to interact with the external world (environment). They need to monitor sensors and control actuators for a wide variety of real-world devices. These devices interface to the computer via input and output registers and their operational requirements are device and computer dependent.

Most embedded systems are also real-time systems, i.e., the correctness of the system depends both on the logical result of the computation, and the time when the results are produced [3]. We refer to these systems as *embedded real-time systems*[1] Real-time systems are typically constructed out of concurrent programs, called tasks. The most common type of temporal constraint that a real-time system must satisfy is the completion of task deadlines. Depending on the consequence due to a missed deadline, real-time systems can be classified as hard or soft. In a *hard real-time system* consequences of missing a deadline can be catastrophic, e.g., aircraft control, while in a *soft-real-time system*, missing a deadline does not cause catastrophic damage to the system, but affect performance negatively, e.g., multimedia. Below follows a list of examples where embedded real-time systems can be found.

- Vehicle systems for automobiles, subways, aircrafts, railways, and ships.
- Traffic control for highways, airspace, railway tracks, and shipping lines.
- Process control for power plants and chemical plants.

[1] We distinguish between embedded and real-time systems, since there are some embedded systems that do not enforce real-time behavior, and there are real-time systems that are not embedded.

- Medical systems for radiation therapy and patient monitoring.
- Military uses such as advanced firing weapons, tracking, and command and control.
- Manufacturing systems with robots.
- Telephone, radio, and satellite communications.
- Multimedia systems that provide text, graphic, audio and video interfaces.
- Household systems for monitoring and controlling appliances.
- Building managers that control such entities as heat, lights, doors, and elevators.

In the last years the deployment of embedded real-time systems has increased dramatically. As can be seen from the examples, these systems are now virtually embedded in every aspect of our lives. At the same time the amount of data that needs to be managed is growing, e.g., embedded real-time systems that are used to control a vehicle, such as a modern car, must keep track of several hundreds sensor values. As the amount of information managed by embedded real-time systems increases, it becomes increasingly important that data is managed and stored efficiently and in a uniform manner by the system. Current techniques adopted for storing and manipulating data objects in embedded and real-time systems are ad hoc, since they normally manipulate data objects as internal data structures. That is, in embedded real-time systems data management is traditionally built as a part of the overall system. This is a costly development process with respect to design, implementation, and verification of the system. In addition, such techniques do not provide mechanisms that support porting of data to other embedded systems or large central databases.

Database functionality is needed to provide support for storage and manipulation of data.

Embedding databases into embedded systems have significant gains: (i) reduction of development costs due to the reuse of database systems; (ii) improvement of quality in the design of embedded systems since the database provides support for consistent and safe manipulation of data, which makes the task of the programmer simpler; and (iv) increased maintainability as the software evolves. Consequently, this improves the overall reliability of the system. Furthermore, embedded databases provide mechanisms that support porting of data to other embedded systems or large central databases.

However, embedded real-time systems put demands on such embedded database that originate from requirements on embedded and real-time systems.

Most embedded systems need to be able to run without human presence, which means that a database in such a system must be able to recover from the failure without external intervention [4]. Also, the resource load the database imposes on the embedded system should be carefully balanced, in particular, memory footprint. For example, in embedded systems used to control a vehicle minimization of the hardware cost is of utmost importance. This usually implies that memory capacity must be kept as low as possible, i.e., databases used in such systems must have small memory footprint. Embedded systems can be implemented in different hardware environments supporting different operating system platforms, which requires the embedded database to be portable to different operating system platforms.

On the other hand, real-time systems put different set of demands on a database system. The data in the database used in real-time systems must be logically consistent, as well as temporally consistent [5]. Temporal consistency of data is needed in order to maintain consistency between the actual state of the environment that is being controlled by the real-time system, and the state reflected by the content of the database. Temporal consistency has two components:

- *Absolute consistency*, between the state of the environment and its reflection in the database.
- *Relative consistency*, among the data used to derive other data.

We use the notation introduced by Ramamritham [5] to give a formal definition of the temporal consistency.

A data element, denoted d, which is temporally constrained, is defined by three attributes:

- value d_{value}, i.e., the current state of data d in the database,
- time-stamp d_{ts}, i.e., the time when the observation relating to d was made, and
- absolute validity interval d_{avi}, i.e., the length of the time interval following d_{ts} during which d is considered to be absolute consistent.

A set of data items used to derive a new data item forms a relative consistency set, denoted R, and each such set is associated with a relative validity interval, R_{rvi}. Data in the database, such that $d \in R$, has a correct state if and only if [5]

1. d_{value} is logically consistent, and
2. d is temporally consistent, both
   - absolute $(t - d_{ts}) \leq d_{avi}$, and
   - relative $\forall d' \in R, |d_{ts} - d'_{ts}| \leq R_{rvi}$.

A transaction, i.e., a sequence of read and write operations on data items, in conventional databases must satisfy following properties: atomicity, consistency, isolation, and durability, called ACID properties. In addition, transactions that process real-time data must satisfy temporal constraints. Some of the temporal constraints on transactions in a real-time database come from the temporal consistency requirement, and some from requirements imposed on the system reaction time (typically, periodicity requirements) [5]. These constraints require time-cognizant transaction processing so that transactions can be processed to meet their deadlines, both with respect to completion of the transaction as well as satisfying the temporal correctness of the data.

2.1.1 An example application

We describe one example of a typical embedded real-time system. The example is typical for a large class industrial process control system that handles large volume of data, where data have temporal constraints. In order to keep the example as illustrative and as simple as possible we limit our example to a specific application scenario, namely the control of the water level in a water tank (see figure 2.1). Through this example we illustrate demands put on data management in such a system. Our example-system contains a real-time operating system, a database, an I/O management subsystem and a user-interface.

A controller task (PID-regulator) controls the level in the water tank according to the desired level set by the user, i.e., the highest allowed water level. The environment consists of the water level, the alarm state, the setting of the valve and the value of the user interface. The environment state is reflected by the content of the database (denoted a', x', y', r' in figure 2.1). Furthermore, PID variables that reflect internal status of the system are also stored in the database. The I/O manager (I/O MGNT) is responsible for data exchange between the environment and the database. Thus, the database stores the parameters from the environment and to the environment, and configuration data. Data in the database needs to be temporally consistent, both absolute and relative. In this case, absolute validity interval can depend on the water flow. That

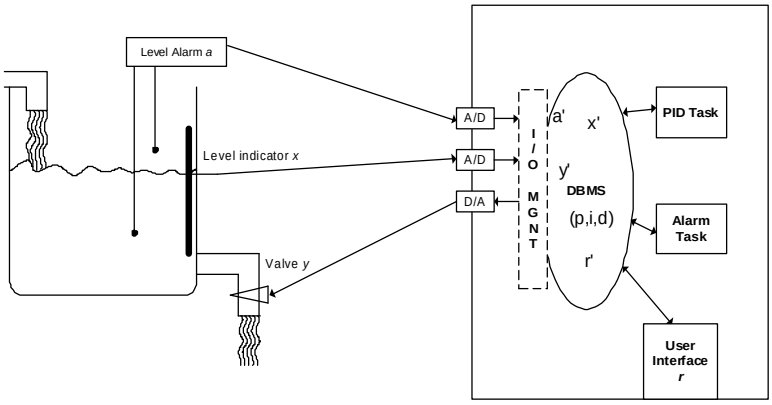

Figure 2.1: *An example system. On the left side a water tank is controlled by the system on the right side. The PID-task controls the flow out by the valve (y) so that the level in tank (x) is the same as the level set by the user (r). An alarm task shuts the system down if the level alarm (a) is activated.*

is, if the the tank in figure 2.1 is very big and the flow is small, absolute validity interval can be greater than in the case where the water-flow is big and the data needs to be sampled more frequently. The relative consistency depicts the difference between the oldest data sample and the youngest sample of data. Hence, if the alarm in our system is activated, due to high water level, and *x'* indicates a lover level, these two values are not valid even though they have absolute validity.

For this application scenario, an additional temporal constraint must be satisfied by the database, and that is an end-to-end deadline. This temporal constraint is important because the maximum processing time for the alarm event must be smaller than the end-to-end-deadline. Figure 2.2 shows the whole end-to-end process for an alarm event. When an alarm is detected, an alarm sensor sends the signal to the A/D converter. This signal is read by the I/O manager recording the alarm in the database. The alarm task then analyzes the alarm data and sends a signal back to indicate an emergency shutdown.

In our example, the database can be accessed by the alarm task, the PID task, the user interface, and the I/O manager. Thus, an adequate concurrency control must be ensured in order to serialize transactions coming from these four different database clients.

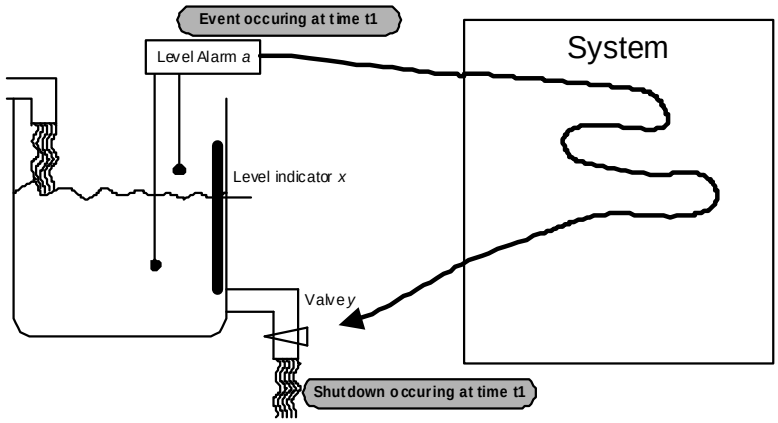

Figure 2.2: *The end-to-end deadline constraint for the alarm system. The emergency shutdown must be completed within a given time* dl_{alarm} *implying that* $t2 - t1 \leq dl_{alarm}$.

Let us now assume that the water level in the tank is just below the highest allowed level, i.e., the level when an alarm is triggered. The water flowing into the tank creates ripples on the surface. These ripples could cause the alarm to go on and off with every ripple touching the sensor, and consequently sending bursts of alarms to the system. In this case, one more temporal constraint must be satisfied, a delayed response. The delayed response is a period of time within which the water level must be higher than the highest allowed level in order to activate the alarm.

As we can see, this simple application scenario puts different requirements on the database. A complete industrial process control system, of which this example is part of, would put a variety of additional requirements on a database, e.g., logging.

Note that requirements placed on the database by the embedded real-time system are to some extent general for all embedded and real-time applications, but at the same time, there are requirements that are specific to an application in question (e.g., delayed response). Thus, an embedded database system must, in a sense, be tailored (customized) for each different application to give an optimized solution. That is, given the resource demands of the embedded

real-time system, a database must be tailored to have minimum functionality, i.e., only functionality that a specific application needs.

In recent years, a significant amount of research has focused on how to incorporate database functionality into real-time systems without jeopardizing timeliness, and how to incorporate real-time behavior into embedded systems. However, research for embedded databases used in embedded real-time systems, explicitly addressing the development and design process, and the limited amount of resources in embedded systems is sparse. Hence, the goal of our report is to identify the gap between the following three different systems: real-time systems, embedded systems, and database systems.

There are many embedded databases on the market, but, as we show in this report, they vary widely form vendor to vendor. Existing commercial embedded database systems, e.g., Polyhedra [6], RDM and Velocis [7], Pervasive.SQL [8], Berkeley DB [9], and TimesTen [10], have different characteristics and are designed with specific applications in mind. They support different data models, e.g., relational vs object-oriented model, and operating system platforms. Moreover, they have different memory requirements and provide different types of interfaces for users to access data in the database.

Application developers must carefully choose the embedded database their application requires, and find the balance between the functionality an application requires and the functionality that an embedded database offers. Thus, finding the right embedded database, in addition of being a quite time consuming, costly and difficult process, is a process with lot of compromises. Although a significant amount of research in real-time databases has been done in the past years, it has mainly focussed on various schemes for concurrency control, transaction scheduling, and logging and recovery. Research projects that are building real-time database platforms, such as ART-RTDB [11], BeeHive [12], DeeDS [13] and RODAIN [14], mainly address real-time performance, have monolithic structure, and are built for a particular real-time application. Hence, the issue of how to enable development of an embedded database system that can be tailored for different embedded real-time applications arises. The development costs of such database system must be kept low, and the development must ensure good software quality.

The outline of the report is as follows. In section 2.2.2, we investigate a number of commercially available database management systems. This survey is followed by a survey on experimental research database management systems in section 2.2.4. The report is concluded in 2.3.1.

2.2 Database Systems

2.2.1 Traditional Database Systems

Databases are used to store data items in a structured way. Data items stored in a database should be persistent, i.e., a data item stored in the database should remain there until either removed or updated. Transactions are most often used to read, write, or update data items. Transactions should guarantee serialization. The so-called database manager is accessed through interfaces, and one database manager can support multiple interfaces like C/C++, SQL, or ActiveX interfaces.

According to [15] most database management systems consist of three levels:

- The internal level, or physical level, deals with the physical storage of the data onto a media. Its interface to the conceptual level abstracts all such information away.
- The conceptual level handles transactions and structures of the data, maintaining serialization and persistence.
- The external level contains interfaces to users, uniforming transactions before they are sent to the conceptual level.

One of the main goals for many traditional database systems are transaction throughput and low average response time [5], while for real-time databases the main goal is to achieve predictability with respect to response times, memory usage and CPU utilization. We can say that when a worst case response time or maximum memory consumption for a database can be guaranteed, the system is predictable. However, there can be different levels of predictability. For a system that can guarantee a certain response time with some defined confidence, is said to have a certain degree of predictability.

2.2.2 Embedded Database Systems

Definitions

A device embedded database system is a database system that resides into an embedded system. In contrast, an application-embedded database is hidden inside an application and is not visible to the application user. Application-embedded databases are not addressed further in this survey. This survey focuses only on databases embedded in real-time and embedded systems. The

main objectives of a traditional enterprise database system often is throughput, flexibility, scalability, functionality etc., while size, resource usage, and processor usage are not as important, since hardware is relatively cheap. In embedded systems these issues are much more important. The main issues for an embedded database system can be summarized as [16, 4, 17]:

- Minimizing the memory footprint: The memory demand for an embedded system are most often, mainly for economical reasons, kept as low as possible. A typical footprint for an embedded database is within the range of some kilobytes to a couple of megabytes.
- Reduction of resource allocations: In an embedded system, the database management system and the application are most often run on the same processor, putting a great demand on the database process to allocate minimum CPU bandwidth to leave as much capacity as possible to the application.
- Support for multiple operating systems: In an enterprise database system, the DBMS is typically run on a dedicated server using a normal operating system. The clients, that could be desktop computers, other servers, or even embedded systems, connect to the server using a network connection. Because a database most often run on the same piece of hardware as the application in an embedded system, and that embedded systems often use specialized operating systems, the database system must support these operating systems.
- High availability: In contrast to a traditional database system, most embedded database systems do not have a system administrator present during run-time. Therefore, an embedded database must be able to run on its own.

Depending on the kind of system the database should reside in, some additional objectives might be more emphasized, while others are less important. For example, Pervasive, which manufactures Pervasive.SQL DBMS system, has identified three different types of embedded systems and has therefore developed different versions of their database to support these systems [18]:

- Pervasive.SQL for smart cards is intended for systems with very limited memory resources, typically a ten kilobytes. This version has traded off sophisticated concurrency control and flexible interfaces for size. Typical applications include banking systems like cash-cards, identification systems, health-care, and mobile phones.

- Pervasive.SQL for embedded systems is designed for small control systems like our example system in figure 2.1. It can also be used as a data pump, which is a system that reads data from a number of sensors, stores the data in a local database, and continuously "pumps" data to a large enterprise database in an unidirectional flow. Important issues here are predictability, with respect to both timing and system resources, as we are approaching a real-time environment. The interfaces are kept rather simple to increase speed and predictability. Since the number of users and the rate of transactions often can be predicted, the need for a complex concurrency control system might not be necessary.

- Pervasive.SQL for mobile systems is used in mobile devices like cellular phones or PDAs, where there is a need for higher degree of concurrency control. Consider that a user browses through e-mails on a PDA while walking into his/hers office, then the synchronizer updates the e-mail list using a wireless communication without interrupting the user. The interfaces support more complex ad-hoc queries than the embedded version. The interfaces are modular and can be included or excluded to minimize memory footprint.

An Industrial Case Study

In 1999, ABB Robotics who develops and manufactures robots for industrial use, wanted to exchange the existing in-house developed configuration storage management system into a more flexible and generic system, preferably some commercial-of-the- shelf (COTS) embedded database.

An industrial robot is a complex and computer-controlled system, which consists of many sub-systems. In each robot some configuration information about its equipment is stored. Today, this amounts to about 600 kilobytes of data. This data needs to be stored in a structured and organized way, allowing easy access.

The current system, called CFG, is a custom made application and resembles in many ways to a regular database application. However, it is nowadays considered to be too non-flexible and the user-interface is not user friendly enough. Furthermore, the internal structures and the choice of index system have lead to much longer response times as the data size has increased.

ABB Robotics decided to investigate the market for an existing suitable database system that would fulfill their demands. The following requirements were considered to be important[19]:

- Connectivity. It should be possible to access the database both directly from the robot controlling application and from an external application via a network connection. Furthermore the database should support *views* so data could be queried from a certain perspective. It would be preferable if some kind of standard interface, e.g., ODBC, could be used. If possible, the database should be backward compatible with the query language used in the CFG system. The database should also be able to handle simultaneous transactions and queries, and be able to handle concurrency.

- Scalability. The amount of data items in the system must be allowed to grow as the system evolves. Transaction times and database behavior must not change due to increase in data size.

- Security. It should be possible to assign different security levels for different parts of the data in the database. That is, some form of user identification and some security levels must exist.

- Data persistence. The database should be able to recover safely after a system failure. Therefore some form of persistent storage must be supported, and the database should remain internally consistent after recovery.

- Memory requirements. To keep the size of the DBMS system low is a key issue.

2.2.3 Commercial Embedded DBMS: a Survey

In this section we discuss and compare a number of commercial embedded database systems. We have selected a handful of systems with different characteristics. These databases are compared with each other with respect to a set of criteria, which represent the most fundamental issues related to embedded databases.

Criteria Investigated

- DBMS model, which describes the architecture of the database system. Two DBMS architectures for embedded databases are the client/server model and the embedded library model.

- Data model, which specifies how data in the database is organized. Common models are the relational, the object-oriented and the object relational.
- Data indexing, which describes how the data indexes are constructed.
- Memory requirements, which describes the memory requirements for the system, both data overhead and the memory footprint, which is the size of the database management system.
- Storage media, which specifies the different kinds of storage medias that the database supports.
- Connectivity, which describes the different interfaces the application can use to access the database. Network connectivity is also specified, i.e. the ability to access the database remotely via a network connection.
- Operating system platforms, which specifies the operating systems supported by the database system.
- Concurrency control, which describes how concurrent transactions are handled by the system.
- Recovery, which specifies how backup/restoration is managed by a system in case of failures.
- Real-time properties, which discusses different real-time aspects of the system.

Databases Investigated

In this survey we have selected a handful of systems that together represents a somewhat complete picture of the types of products currently on the market. This list of systems is not to be considered complete in any way, worth mentioning are, for example, the SQL.Anywhere system developed by Sybase [20] and the c-tree Plus system by FairCom [21].

- Pervasive.SQL by Pervasive Software Inc. This database has three different versions for embedded systems: Pervasive.SQL for smart-card, Pervasive.SQL for mobile systems, and Pervasive.SQL for embedded systems. All three versions integrate well with each other and also with their non embedded versions of Pervasive.SQL. Their system view and the

fact that they have, compared to most embedded databases, very small memory requirements was one reason for investigating this database [8].

- Polyhedra by Polyhedra Plc. This database was selected for three reasons, first of all it is claimed to be a real-time database, secondly it is a main memory database and third, it has active behavior [6].
- Velocis by Mbrane Ltd. This system is primarily intended for e-Commerce and Web applications, but has some support for embedded operating systems [7].
- RDM by Mbrane Ltd. Like Polyhedra, RDM also claims to be a real-time database. It is however fundamentally different from the Polyhedra system, by, for example, being an embedded library and, thus, does not adopt the client/server model [7].
- Berkeley DB by Sleepycat Software Inc. This database, which also is implemented as a library, was selected for the survey because it is distributed as open source and therefore interesting from a research point of view [9].
- TimesTen by TimesTen Performance Software. This relational database is, like the Polyhedra system a main memory real-time database system [10].

DBMS Model

There are basically two different DBMS models supported (see table 2.1). The first model is the client/server model, where the database server can be considered to be an application running separately from the real-time application, even though they run on the same processor. The DBMS is called using a request/response protocol. The server is accessed either through inter-process communication or some network. The second model is to compile the DBMS together with the system application into one executable system. When a task wants to access the database it only performs function calls to make the request. Transactions execute on behalf of their tasks' threads, even though internal threads in the DBMS might be used. There are advantages and drawbacks with both models. In a traditional enterprise database, the server most often run on a dedicated server machine, allowing the DBMS to use almost 100% of the CPU. However in an embedded client/server system, the application and the server process often share the same processor. This implies that for every transaction

DBMS system	Client/server	Library
Pervasive.SQL	x	
Polyhedra	x	
Velocis	x	
RDM		x
Berkeley DB		x
TimesTen	x	

Table 2.1: DBMS models supported by different embedded database systems.

at least two context switches must be performed, see figure 2.3. More complex transactions, like an update transaction might require even more context switches. Consider for example an real-time task that executes an update transaction that first reads the data element x, then derives a new value of x from the old value and finally writes it back to the database. This transaction would generate four context switches. Two while fetching the value of x and two while writing x back to the database.

These context-switches adds to the system overhead, and can furthermore make worst case execution time estimations of transactions more complex to predict. The network message passing or inter-process communication between the server and the client also add to the overhead cost. A drawback with an embedded library is that they lack standardized interfaces like ODBC. [4].

The problem with message passing overhead has been reduced in the Velocis system. They allow the process to be compiled together with the application, thus reducing the overhead since shared memory can be used instead.

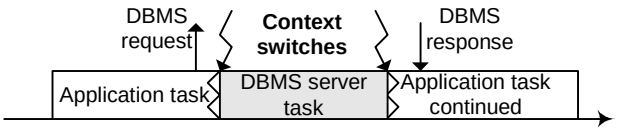

Figure 2.3: *In an embedded client/server DBMS at least two context switches are necessary for each transaction.*

Data Model

The data model concerns how data is logically structured. The most common model is the relational model where data is organized in tables with columns and rows. Databases implementing relational data model are referred to as relational databases (RDBMS). One advantage with a relational model is that columns in the table can relate to other tables so arbitrary complex logical structures can be created. From this logical structure queries can be used to extract a specific selection of data, i.e. a view. However, one disadvantage with the relational model is added data lookup overhead. This is because that data elements are organized with indexes in which pointers to the data is stored, and to perform the index lookup can take significant amount of time, especially for databases stored on hard drives. This can be a serious problem for databases that resides in time critical applications like our example application in figure 2.1.

The relational model, which was developed by Codd, only supports a few basic data types, e.g., integers, floating point numbers, currency, and fixed length arrays. This is nowadays considered a limitation which has lead to the introduction of databases which supports more complex data types, e.g., Binary Large Objects (BLOBs). BLOB is data, which is treated as a set of binary digits. The database does not know what a BLOB contains and therefore cannot index anything inside of the BLOB.

The object-oriented database (ODMBS) is a different kind of data model, which is highly integrated with object-oriented modeling and programming, The ODBMS is an extension to the semantics of an object-oriented language. An object-oriented database stores objects that can be shared by different applications. For example, a company which deals with e-Commerce has a database containing all customers. The application is written in an object-oriented language and a customer is modeled as an object. This object has methods, like `buy` and `changeAdress`, associated with it. When a new customer arrives, an instance of the class is created and then stored in the database. The instance can then be retrieved at any time. Controlled access is guaranteed due to concurrency control.

A third data model, which has evolved from both the relational and the object-oriented model, incorporates objects in relations, thus is called object-relational databases (ORDBMS).

As shown in table 2.2, all systems in the survey except Berkeley DB are relational. Furthermore the Polyhedra has some object-relational behavior through its Control language described below. However it is not fully object-

DBMS system	Relational	Obj.-oriented	Obj.-rel.	Other
Pervasive.SQL	x			
Polyhedra	x		(x)	
Velocis	x			
RDM	x			
Berkeley DB				x
TimesTen	x			

Table 2.2: Data models supported by different embedded database systems.

relational since objects itself cannot be stored in the database. To our knowledge, no pure object-oriented embedded databases exists on the market today. There are however some low requirements databases that are object-oriented, such as the Objectivity/DB [22] system. Furthermore some object-oriented client-libraries exists that can be used in an embedded system, such as the PowerTier [23] and the Poet Object system [24]. These client-libraries connects to an external database server. Since the actual DBMS and the database is located on a non-embedded system we do not consider them embedded databases in this survey.

Most systems have ways to "shortcut" access to data, and therefore bypassing the index lookup routine. Pervasive, for example, can access data using the Btrieve transactional engine that bypasses the relational engine. Mbrane uses a different approach in the RDM system. In many real-time systems data items are accessed in a predefined order (think of a controlling system where some sensor values are read from the database and the result is written back to the database). By inserting shortcuts between data elements such that they are directly linked in the order they are accessed, fast accesses can be achieved. As these shortcuts point directly to physical locations in memory, reorganization of the database is much more complex since a large number of pointers can become stale when a single data is dropped or moved.

The Polyhedra DBMS system is fundamentally different compared to the rest of the relational systems in this survey, because of its active behavior. This is achieved through two mechanisms, active queries and by the control language (CL). An active query looks quite like a normal query where some data is retrieved and/or written, but instead the query stays in the database until explicitly aborted. When a change in the data occurs that would alter the result of the query, the application is notified. The CL, which is a fully object-

oriented script language that supports encapsulation, information hiding and inheritance, can determine the behavior of data in the database. This means that methods, private or public, can be associated with data performing operations on them without involving the application. In our example application, the CL could be used to derive the valve setting y' from the level indicator x' and the PID parameters, thus removing the need for the PID task. The CL has another important usage, since it can be used in the display manager (DM) to make a graphical interface. The DM, which is implemented as a Polyhedra Client, together with the control language is able to present a graphical view on a local or remote user terminal. This is actually exactly the user interface in our example application, since DM also can take user input and forward that to the database. Active database management will be further discussed in section 2.2.8.

As mentioned above, Berkeley DB is the only non-relational system in this survey. Instead it uses a key-data relationship. One data is associated with a key. There are three ways to search for data, from key, part of key or sequential search. The data can be arbitrary large and of virtually any structure. Since the keys are plain ASCII strings the data can contain other keys so complex relations can be built up. In fact it would be possible to implement a relational engine on top of the Berkeley DB database. This approach claims for a very intimate relationship between the database and the programming language used in the application.

Data Indexing

To be able to efficiently search for specific data, an efficient index system should exist. To linearly search through every single key from an unsorted list would not be a good solution since transaction response times would grow as the number of data elements in the database increases. To solve this problem two major approaches are used, tree structures and hashed lists. Both approaches supply similar functionality, but differ somewhat in performance. Two major tree structures are used, B^+-tree indexing, which suits disk based databases, and T-tree indexing, which is used in main-memory databases.

The main issue for B^+-tree indexing is to minimize disk I/O, thus trading disk I/O for added algorithm complexity. B^+-tree indexing sorts the keys in a tree structure in which every node has a predefined number of children, denoted p. A large value of p results in a wide but shallow tree, thus the tree has a large fanout. A small value of p results in the opposite. For $p = 2$ tree is a binary tree. Since all data indexes reside in the leave nodes of the tree, while only the

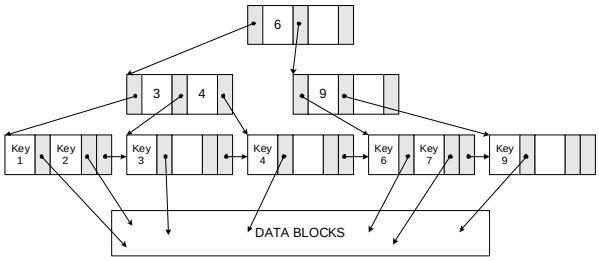

Figure 2.4: *The structure of a B^+-tree with fanout of 3. The inner nodes are only used to index down to the leave node containing the pointer to the correct data. Figure partially from [25].*

inner nodes are used to navigate to the right leave, the number of visited nodes will be fewer for a shallow tree. This results in fewer nodes that have to be retrieved from disk, thus reducing disk I/O. In figure 2.2.3 we see an example of a B^+-tree . The root directs requests to all keys that are less than six to the left child and keys from six to its middle child. As the value of p increases, the longer time it takes to pass the request to the correct child. Thus, when deciding the fanout degree, the time it takes to fetch a node from disk must be in proportion to the time it takes to locate which child to redirect to. Proposals on real-time concurrency control for B^+-tree indexing have been made [25].

DBMS system	**B^+-tree**	**T-tree**	**Hashing**	**Other**
Pervasive.SQL	x			
Polyhedra			x	
Velocis	n/a	n/a	n/a	n/a
RDM	x			
Berkeley DB	x		x	x
TimesTen	x	x	x	

Table 2.3: Data indexing strategies used by different embedded database systems.

For main-memory databases a different type of tree can be used, the T-tree structure [26]. The T-tree uses a deeper tree structure than the B^+-tree since it is a balanced binary tree, see figure 2.2.3, with a maximum of two children for

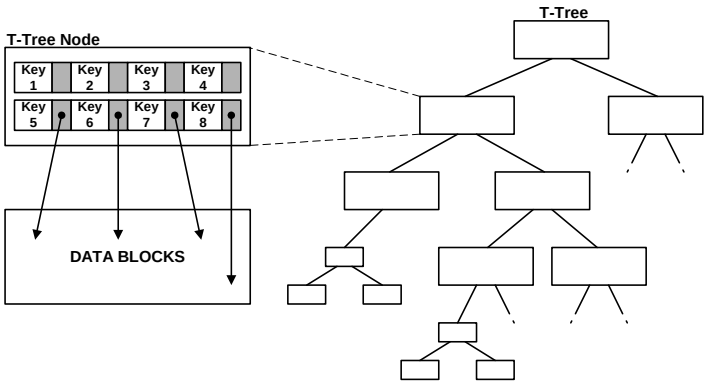

Figure 2.5: *The structure of a T-tree. Each node contains a number of entries that contains the key and a pointer to the corresponding data. All key values that are less then the boundary of this node is directed to the left child, and key values that are above the boundary are directed to the right.*

every node. To locate data, more nodes are normally visited, compared to the B^+-tree , before the correct node is found. This is not a problem for a main-memory database since memory I/O is significantly faster than disk I/O. The primary issue for T-tree indexing is that the algorithms used to traverse a T-tree have a lower complexity and faster execution time than corresponding algorithms for the B-tree. However, it has been pointed out that a T-tree traversal combined with concurrency control might perform worse than B-tree indexes due to the fact that more nodes in the T-tree had to be locked to ensure tree integrity during traversal [27] . They also proposed an improvement called T-tail, which reduces the number of costly re-balancing operations needed. Furthermore, they proposed two concurrency algorithms for T-tail and T-tree structures, one optimistic and one pessimistic (for a more detailed discussion about pessimistic and optimistic concurrency control, see section 2.2.3).

The second approach, hashing, uses an hash list in which keys are inserted according to a value derived from the key itself (different keys can be assigned the same value) [28]. Therefore, each entry in the list is a bucket that can contain a predefined number of data keys. If a bucket is full a rehash operation must be performed. The advantage with hashing is that the time to search for certain data is constant independent of the amount of data entries in the

database. However, hashing often cause more disk I/O than B-trees. This is because data is often used with locality of reference, i.e., if data element d_1 is used, it is more probable that data element d_2 will be used shortly. In a B-tree both elements would be sorted close to each other, and when d_1 is retrieved from disk, then it is probable that also d_2 will be retrieved. However in a hash list d_1 and d_2 are likely to be in different buckets, causing extra disk I/O if both data are used. Another disadvantage with hashing compared to tree-based indexing is that non-exact queries is time consuming to perform. For example, consider a query for all keys greater than x. After that x has been found, all keys are found to the right of x in a B^+-tree . For a hashed index, all buckets need to be searched to find all matching keys.

We can notice that main-memory databases, namely Polyhedra and TimesTen use hashing (see table 2.3). Additionally, TimesTen supports T-trees. Pervasive, RDM and Berkeley DB use B-tree indexing.

It is also noteworthy that Berkeley DB uses both B^+-tree and hashing. They claim that hashing is suitable for database schemes that are either so small that the index fits into main memory or when the database is so large that B^+-tree indexing will cause disk I/O upon most node fetching due to that the cache can only fit a small fraction of the nodes. Thus making the B^+-tree indexing suitable for database schemes with a size in between these extremes. Furthermore, Berkeley DB supports a third access method, queue, which is used for fast inserts in the tail, and fast retrieval of data from the head of the queue. This approach is suitable for the large class of systems that consume large volumes of data, e.g., state machines.

Memory Requirements

Memory requirement of the database is an important issue for embedded databases residing in environments with small memory requirements. For mass-produced embedded computer systems like computer nodes in a car, minimizing hardware is usually a significant factor for reducing development costs. There are two interesting properties to consider for embedded databases, first of all the memory footprint size, which is the size of the database without any data elements in it. Second, data overhead is of interest, i.e., the number of bytes required to store a data element apart from the size of the data element itself. An entry in the index list with a pointer to the physical storage address is typical data overhead. Typically client/server solutions seems to require

[1]The values given in the table are the “footprint”-values made available by database vendors.

DBMS system	Memory requirements[1]
Pervasive.SQL for smart cards	8kb
Pervasive.SQL for embedded systems	50kb
Pervasive.SQL for mobile systems	50 - 400kb
Polyhedra	1.5 - 2Mb
Velocis	4Mb
RDM	400 - 500kb
Berkeley DB	175kb
TimesTen	5Mb

Table 2.4: Different memory needs of investigated embedded database systems.

significantly more memory than embedded libraries, with Pervasive.SQL being the exception (see table 2.4). This exception could partially be explained by Pervasive's Btrieve native interface being a very low-level interface (see section 2.2.3), and that much of the functionality is placed in the application instead. In the case of Berkeley DB, there is a similar explanation to its small memory footprint. Additional functionality (not provided by the vendor) can be implemented on top of the database.

Regarding data overhead, Polyhedra has a data overhead of 28 bytes for every record. Pervasive's data overhead is not as easily calculated since it uses paging. Depending on record and page sizes different amount of fragmentation is introduced in the data files. There are however formulas provided by Pervasive for calculating exact record sizes. The other systems investigated in this survey supplies no information about actual data overhead costs.

For systems where low memory usage is considered more important than processing speed, there is an option of compressing the data. Data will then be transparently compressed and decompressed during runtime. This, however, requires free working memory of 16 times the size of the record being compressed/decompressed.

Storage Media

Embedded computer systems support different storage medias. Normally, data used on the computer is stored on a hard-drive. Hand-held computer use Flash or other non-volatile memory for storage of both programs and data. Since data in a database needs to be persistent even upon a power loss, some form of stable

DBMS system	Hard-drive	Flash	Smart card	FTP
Pervasive.SQL	x	x	x	
Polyhedra	x	x		x
Velocis	x			
RDM	x			
Berkeley DB	x	x		
TimesTen	x			

Table 2.5: Storage medias used by investigated embedded database systems.

storage technique must be used. As can be seen from table 2.5, most systems support Flash in addition to a hard-drive. The Polyhedra system also supports storage via a FTP-connection. This implies that the Polyhedra system can run without persistent storage, since data can be loaded via FTP-connection upon system startup.

Connectivity

Database interfaces are the users way to access the database. An interface normally contains functions for connecting to the server, making queries, performing transactions, and making changes to the database schema. However different interfaces can be specialized on specific types of operations, e.g., SQL is used mainly for schema modifications and queries, while a C/C++ API normally are specialized for transactional access. Figure 2.6 shows a typical interface configuration for an embedded database system.

The most basic interface is the native interface. This interface, which is for a specific programming language, e.g., C or C++, is often used by the application running on the embedded system. In our example in figure 2.1, tasks and I/O management would typically use the native interface to perform their transactions.

Microsoft has developed the Open Database Connectivity (ODBC) interface in the late 80's. This interface uses the query language SQL, and is today one of the largest standards of database interfaces. The advantage with ODBC is that any database that supports ODBC can connect with any application written for ODBC. ODBC is a low-level interface, that requires more implementation per database access then a high-level interface like ActiveX Data Object (ADO) interface, explained below in more detail. The ODBC interface cannot handle non-relational databases very well. Main benefits, beside the

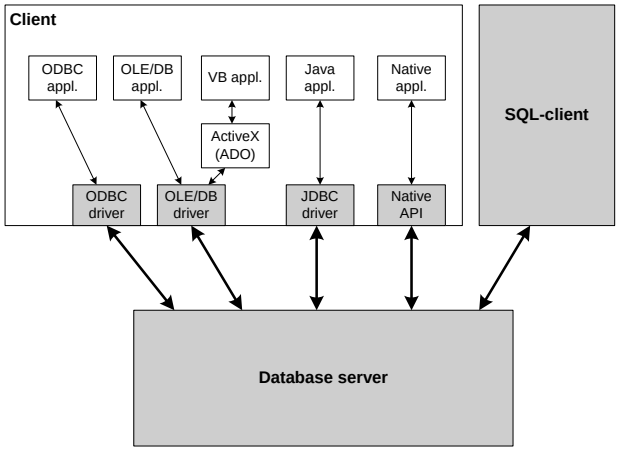

Figure 2.6: *Figure showing the architecture of applications that uses different types of database interfaces. The shaded boxes are components that normally are provided by the database vendor.*

interchangeability, is that the performance is generally higher than high-level interfaces, and the interface gives the possibility of detailed tuning.

A newer standard, also developed by Microsoft, is the OLE DB interface which is based on the Microsoft COM technology. OLE DB is, as ODBC, a low-level interface. The functionality of OLE DB is similar to ODBC, but object-oriented. However, one advantage is the possibility to use OLE DB on non-relational database systems. There are also drivers available to connect the OLE DB interface on top of an ODBC driver. In this case the ODBC driver would act as a server for the application but as a client to the DBMS.

On top of OLE DB, the high-level ADO interface could be used. Since both OLE DB and ADO are based on Microsoft COM, they can be reached from almost any programming language.

For Java applications, the JDBC interface can be used. It builds on ODBC but has been developed with the same philosophy as the OLE DB and ADO interfaces. For databases that do not directly support JDBC, an ODBC driver can be used as intermediate layer, just like OLE DB.

As can be seen in table 2.6, Pervasive.SQL uses the Btrieve interface for fast reading and updating transactions. Btrieve is very closely connected to the

DBMS system	C/C++	ODBC	OLE DB	ADO
Pervasive.SQL		x	x	x
Polyhedra	x	x	x	x
Velocis	x	x		
RDM	x			
Berkeley DB	x	x		
TimesTen				

DBMS system	JDBC	Java	Other
Pervasive.SQL	x		Btrieve RSI
Polyhedra	x		
Velocis		x	Perl
RDM			
Berkeley DB		x	TCL Perl
TimesTen	x		

Table 2.6: Interfaces supported by different embedded database systems.

physical storage of data. In Btrieve data is written as records. A record can be described as a predefined set of data elements, similar to a *struct* in the C programming language. Btrieve is not aware of the data elements in the record, but treats data elements as string of bytes that can be retrieved or stored. An index key is attached to every record. Records can then be accessed in two ways, physical or logical. When a record is located logically, an index containing all keys is used to lookup the location of the data. The indexes are sorted in either ascending or descending order in a B-tree. Some methods to access data, except by keyword, are `get first`, `get last`, `get greater than`, etc. However for very fast access, the physical method can be used. Then the data is retrieved using a physical location. This technique is useful when data is accessed in a predefined order, this can be compared to the pointer network used by RDM discussed previously. Physical access is performed using four basic methods, `step first`, `step last`, `step next` and `step previous`.

One limitation with the Btrieve interface is that it is not relational. Pervasive has therefore introduced their RowSet interface (RSI) that combines some functionality from Btrieve and some from SQL. Every record is a row and the

data elements in the records can be considered columns in the relation. This interface is available for the smart card version, embedded version, and mobile version only, and is not available for non-embedded versions of Pervasive.SQL.

The Btrieve and RSI interfaces are quite different from the native interface in Polyhedra, which does not require the same understanding of the physical data structure. The Polyhedra C/C++ interface uses SQL, thus operating on a higher level. Basically, three different ways to access the database are provided in the interface: queries, active queries, and transactions. A query performs a retrieval of data from the database immediately, using the SQL `select` command, and the query is thereafter deleted. An active query works the same way as an ordinary query but will not be deleted after the result has been delivered, but will be reactivated as soon as any of the involved data is changed. This will continue until the active query is explicitly deleted. One problem with active queries is that if data changes very fast, the number of activations of the query can be substantial, thus risking to overload the system. To prevent this a minimum inter-arrival time can be set, and thus the query cannot be reactivated until the time specified has elapsed. From a real-time perspective, the activation could then be treated as a periodic event, adding to the predictability of the system.

Using transactions is the only way to make updates to the database, queries are only for retrieval of data. A transaction can consist of a mixture of active update queries and direct SQL statements. Note that a query might update the database if it is placed inside of a transaction. Active update queries insert or update a single data element and also delete on single row. Transactions are, as always, treated as atomic operations and are aborted upon data conflict. To further speed up transaction execution time, SQL procedures can be used, both in queries and transactions. These procedures are compiled once and cannot be changed after that, only executed. This eliminates the compilation process and is very useful for queries that run often. However, procedures do not allow schema changes.

Noteworthy is that TimesTen only supports ODBC and JDBC, and it has no native interface like the rest of the systems. To increase the speed of ODBC connections, an optimized ODBC driver is developed that connects only to TimesTen. Like the Polyhedra DBMS, TimesTen uses precompiled queries, but in addition supports parameterized queries. A parameterized query is a precompiled query that supports arguments passed to it. For example the parameterized query

```
query( X )
```

```
SELECT * from X
```

would return the table specified by `X`.

The Berkeley DB, as mentioned earlier, is not a client/server solution, but is implemented as an embedded library. It is, furthermore, not relational but uses, similar to the Btrieve interface, a key that relates to a record. The similarities to Btrieve go further than that; data can be accessed in two ways, by key or by physical location, just as Btrieve. The methods `get()` and `put()` access data by the key, while cursor methods can be used to get data by physical location in the database. As in Btrieve, the first, last, next, current, and previous record number can be retrieved. A cursor is simply a pointer that points directly to a record. The functionality and method names are similar to each other independent of which of the supported programming language that is used. There are three different modes that the database can run in:

- Single user DB. This version only allows one user to access the database at a time, thus, removing the need for concurrency control and transaction support.
- Concurrent DB. This version allows multiple users to access the database simultaneously, but does not support transactions. This database is suitable for systems that has very few updates but multiple read-only transactions.
- Transactional database. This version allows both concurrent users and transactions.

The RDM database, which is implemented as a library, has a C/C++ interface as its only interface. This library is, in contrast to Berkeley DB, relational. The interface is a low-level interface that does not comply with any of the interface standards. As mentioned earlier, one problem with databases that does not use the client/server model is the lack of standard interfaces. However, since RDM is relational, a separate SQL-like interface *dbquery* is developed. It is a superset of the standard library, which makes use of a subset of SQL. Table 2.7 shows which databases that support network connectivity.

Operating System Platforms

Different embedded systems might run on various operating systems due to differences in the hardware environment. Also the nature of the application determines which operating systems that might be most useful. Thus, most

DBMS system	Network connectivity
Pervasive.SQL	x
Polyhedra	x
Velocis	x
RDM	x
Berkeley DB	
TimesTen	x

Table 2.7: Network connectivity in investigated embedded database systems.

DBMS system	Desktop OS				
	Windows	**UNIX**	**Linux**	**OS/2**	**DOS**
P.SQL for smart cards					
P.SQL for emb. syst.					
P.SQL for mob. syst.	x				
Polyhedra	x	x	x		
Velocis	x	x	x		
RDM	x	x	x	x	x
Berkeley DB	x	x	x		
TimesTen	x	x	x		

Table 2.8: Desktop operating systems supported by investigated embedded database systems.

embedded databases must be able to run on different operating system platforms.

Tables 2.8 and 2.9 give an overview of different operating systems supported by embedded databases we investigated. There are basically four categories of operating systems that investigated embedded databases support:

- Operating systems traditionally used by desktop computers. In this category you will find the most common operating systems like, Microsoft Windows, different versions of UNIX and Linux.

[0] Smart card operating system for Windows.
[1] Pervasive.SQL for smart cards.
[2] Pervasive.SQL for embedded systems.
[3] Pervasive.SQL for mobile systems.

DBMS system	Real-Time OS					
	VxWorks	OSE	pSOS	LynxOS	Phar-lap	QNX
P.SQL smart c.						
P.SQL emb.sys.	x				x	x
P.SQL mob.sys.						
Polyhedra	x	x	x		x	
Velocis						
RDM	x				x	
Berkeley DB	x					
TimesTen	x			x		

	Hand-held OS		Smart-card OS		
	PalmOS	WinCE	Java Card	Mult OS	WinSC
P.SQL smart c.			x	x	x
P.SQL emb.sys.					
P.SQL mob.sys.	x	x			
Polyhedra					
Velocis					
RDM					
Berkeley DB					
TimesTen					

Table 2.9: Real-time and embedded operating systems supported by investigated embedded database systems.

- Operating systems for hand-held computers. In this category you find Palm OS and Windows CE/PocketPC. These operating systems demand small memory requirements but still have most of the functionality of the ordinary desktop computer operating systems. A good interaction and interoperability between the operating systems on the hand-held computer and a standard desktop is also important. This is recognized by all databases, if you consider the complete Pervasive.SQL family as one database system.
- Real-time operating systems. In this category you find systems like VxWorks and QNX.
- Smart card operating systems. These systems, like Java Card and MultOS, are made for very small environments, typically no more than 32kb. The Java Card operating system is simply an extended Java virtual machine.

Most commercial embedded database systems in this survey support a real-time operating system (see table 2.9). Additionally, Pervasive.SQL supports

operating systems for hand-held computers and smart card operating systems, in the Pervasive.SQL for mobile systems version and the Pervasive.SQL for smart card version, respectively.

Concurrency Control

Concurrency control (CC) *serializes* transactions, retaining the *ACID properties*. All transactions in a database system must fulfill all four ACID properties. These are:

- Atomic: A transaction is indivisible, either it is run to completion or it is not run at all.
- Consistent: It must not violate logical constraints enforced by the system. For example, a bank transaction must follow the law of conservation of money. This means that after a money transfer, the sum of the receiver and the sender must be unchanged.
- Isolated: A transaction must not interfere with any other concurrent transaction. This is also referred to as serialization of transactions.
- Durable: A transaction is, once committed, written permanently into the database.

If two transactions that have some data elements in common are active at the same time, a data conflict might occur. Then it is up to the concurrency control to detect and resolve this conflict. This is most commonly done with some form of locking mechanism. It substitutes the semaphore guarding a global data in a real-time system. There are two fundamentally different approaches on achieving this serialization, an optimistic and a pessimistic approach.

The most common pessimistic algorithm is two-phase-locking (2PL) algorithm proposed in 1976 by Eswaran et al. [29]. This algorithm consists, as the name indicates, of two phases. In the first phase all locks are collected, no reading or writing to data can be performed before a lock has been obtained. When all locks are collected and the updates have been done, the locks are released in the second phase.

Optimistic concurrency control (OCC) was first proposed by Kung and Robinson [30] in 1981. This strategy takes advantage of the fact that conflicts in general are rather rare. The basic idea is to read and update data without regarding possible conflicts. All updates are, however, done on temporary data. At commit-time a conflict detection is performed and the data is written

DBMS system	Pessimistic CC	Optimistic CC	No CC
Pervasive.SQL	x		
Polyhedra		x	
Velocis	x		
RDM	x		
Berkeley DB	x		x
TimesTen	x		

Table 2.10: Concurrency control strategies used in different embedded database systems.

permanently to the database if no conflict was detected. However the conflict detection (verification phase) and the update phase need to be atomic, implying some form of locking mechanism. Since these two phases take much shorter time than a whole transaction, locks that involves multiple data, or the whole database, can be used. Since it is an optimistic approach, performance degrades when congestion in the system increases.

For real-time databases a number of variants of these algorithms have been proposed that suit these databases better [31, 32]. These algorithms try to find a good balance between missed deadlines and temporally inconsistent transactions. Song and Liu showed that OCC algorithms performed poorly with respect to temporal consistency in real-time systems that consist of periodic activities [33], while they performed very well in systems where transactions had random parameters, e.g., event-driven systems. However, it has been shown that the strategies and the implementation of the locking and abortion algorithms significantly determine the performance of OCC [34]. All databases except the Polyhedra DBMS use pessimistic CC (see table 2.10). Since Polyhedra is an event-driven system, OCC is a natural choice. Furthermore, Polyhedra is a main-memory database with very fast execution of queries, the risk of a conflict is thus reduced.

Pervasive.SQL has two kind of transactions: exclusive and concurrent. An exclusive transaction locks a complete database file for the entire duration of the transaction, allowing only concurrent non-transactional clients to perform read-only operations on the file. A concurrent transaction, however, uses read and write locks with much finer granularity, e.g., page or single data locks.

The Berkeley DB has three configurations: (i) The non-concurrent configuration allows only one thread at a time to access the database, removing the

need for concurrency control. (ii) The concurrent version allows concurrent readers and concurrent writers to access the database simultaneously. (iii) The concurrent transactional version allows for full transactional functionality with concurrency control, such as fine grain locking and database atomicity.

Similar to Berkeley DB, TimesTen also has a "no concurrency" option, in which only a single process can access the database. In addition, TimesTen supports two lock sizes: data-store level and record level locking.

Recovery

One important issue for databases is persistence, that is data written and committed to the database should remain until it is overwritten or explicitly removed, even if the system fails and have to be restarted. Furthermore, the state of all ongoing transactions must be saved to be able to restore the database to a consistent state upon recovery, since uncommitted data might have been written to the database. There is basically two different strategies for backup restoration: roll-back recovery, and roll-forward recovery, normally called "forward error recovery". During operation continuous backup points occurs where a consistent state of the database is stored on a non-volatile media, like a hard-drive. These backup points are called checkpoints.

In roll-back recovery you simply restart the database using the latest checkpoint, thus guaranteeing a consistent database state. The advantage with this approach is that it does not require a lot of overhead, but the disadvantage is that all changes made to the database after the checkpoint are lost.

When roll-forward recovery is used, checkpoints are stored regularly, but all intermediate events, like writes, commits and aborts are written to a log. In roll-forward recovery, the database is restored to the last checkpoint, and all log entries are performed in the same order as they have been entered into the log. When the database has been restored to the state it was at the time of the failure, uncommitted transactions are roll-backed. This approach requires more calculations at recovery-time, but will restore the database to the state it was in before the failure. Pervasive.SQL offers three different types of recovery mechanisms, as well as the option to ignore recovery (see table 2.11). This can also be selected parts of the database. Ignoring check-pointing for some data can be useful for systems where some data must be persistent while other data can be volatile. Going back to our example in section 2.1, the desired level given by the user interface, and the regulator parameters need to be persistent, while the current reading from the level indicator must not, since by the time the database is recovered the data will probably be stale. The second option is

DBMS system	Roll-forw.	Roll-back.	Journalling	None
Pervasive.SQL	x			x
Polyhedra			x	x
Velocis	x			
RDM	x			
Berkeley DB		x		
TimesTen		x		x

Table 2.11: Strategies for recovery used in different embedded database systems.

shadow-paging, which simply makes a copy of the database file and makes the changes there, and when the transaction is complete, the data is written back to the original page. A similar approach is the delta-paging, which creates an empty page for each transaction, and only writes the changes (delta-values) to it. At the end of a transaction the data is written back to the original, just as for shadow-paging. With both of these techniques, the database is consistent at all times. The last option is to use logging and roll-forward recovery.

The default configuration for Polyhedra is non-Jornalling, which means that no automatic backup is performed, but the application is responsible for saving the database to non-volatile memory. This is particularly important since this system is a main memory database, and if no backups is taken all data is, of course, lost. When journalling is used, the user can select which tables that should be persistent and the journaller will write an entry in the log when a transaction involving persistent data is committed. There are two approaches when data is persistently written, direct journalling and non-blocking journalling. The direct journalling approach writes data to persistent storage when the load on the DBMS is low. However, the database is blocked during this process and this can cause problems for systems with time-critical data. The second approach can then be used, and that is to use a separate journalling process responsible for writing entries persistent.

TimesTen supports three different levels of recovery control. The most stringent level guarantees transaction atomicity and durability upon recovery, in which case the transaction is not committed until the transaction is written onto disk. The second level of control guarantees transaction atomicity but not durability. Upon commit, the log entry is put into a queue and is later written to disk. Upon recovery the database will be consistent, but committed transactions that have not yet been written to disk might be lost. The lowest level

of recovery control is no recovery control. Neither atomicity nor durability is guaranteed. This option might not be as inappropriate as it might seem at a first glance, since TimesTen is a main-memory database, often used in systems with data that would be stale upon recovery, e.g., a process controller.

Real-Time Properties

Even though none of the commercial database systems in this survey can be classified as a real-time database from a hard real-time systems perspective, most of the systems are successfully used in time-critical systems (to different extent). Ericsson uses Polyhedra in their 3G platform for wireless communication. Delcan Corporation uses Velocis DB in a traffic control system that handles about 1200 traffic lights simultaneously even though this database primarily is used in web and e-Commerce applications. The database systems are simply so fast and efficient and have so many options for fine tuning their performance that the application systems works, even though the DB systems cannot itself guarantee predictability.

A question one could ask is: How would we use one of these systems in a hard real-time system? Are there any ways to minimize the unpredictability to such a degree that a pessimistic estimation would fulfill hard real-time requirements? In our opinion it is. For an event triggered real-time system, Polyhedra would fit well. Let us use our example application again. When new sensor values arrive the I/O management, the database is updated. If there is a change in the alarm data the CL code that is connected to that data is generating an event to trigger the alarm task. Built into the system is also the minimum inter-arrival interval mentioned in the interfaces section. So by using the Polyhedra database to activate tasks, that will then be scheduled by a priority based real-time operating system, a good enough degree of predictability is achieved. To further increase guarantees that critical data will be updated is to use a so called watchdog, that activates a task if it has not been activated for a predefined time. The memory and size predictability would be no problem since they have specified the exact memory overhead for every type of object. However, temporal validity is not addressed with this approach.

For statically scheduled real-time systems the RDM “Network access” could be able to run with predictable response times. This because the order of the data accesses is known a priori and can therefore be linked together in a chain, and the record lookup index is bypassed.

2.2.4 Current State-of-the-art From Research Point of View

There exists a number of databases that could be classified as pure real-time databases. However these databases are research project and are not yet on the commercial market. We have selected a number of real-time database systems and compared them with respect to the following criteria:

- Real-time properties. The criteria enables us to discuss real-time aspects of the systems and how they are implemented.
- Distribution. The criteria enables us to talk about different aspects with respect to distributing the database.
- Transactions workload characteristics. The criteria enables us to discuss how the transaction system is implemented.
- Active behavior. The criteria enables us to talk about the active aspects for some of the systems.

The systems selected in this survey represent some of the more recent real-time database platforms developed. These systems are representative of the ongoing research within the field.

STRIP The STanford Real-time Information Processor (STRIP) [35] is a soft real-time database system developed at Stanford University, US. STRIP is built for the UNIX operating system, is distributed and uses streams for data sharing between nodes. It is an active database that uses SQL3-type rules.

DeeDS The Distributed active, real-time Database System (DeeDS) [13] supports both hard and soft real-time transactions. It is developed at the University of Skövde, Sweden. It uses extended ECA rules to achieve an active behavior with real-time properties. It is built to take advantage of a multiprocessor environment.

BeeHive The BeeHive system [12] is a virtual database developed at the University of Virginia, Charlottesville, US, in which the data in the database can be located in multiple locations and forms. The database supports four different interfaces namely, a real-time interface, a quality of service interface, a fault-tolerant interface, and a security interface.

REACH The REACH system [36], developed at the Technical University of Darmstadt, Germany, is an active object-oriented database implemented on top of the object-oriented database openOODB. Their goal is to archive active behavior in an OO database using ECA rules. A benchmarking mode is supported to measure execution times of critical transactions. The REACH system is intended for soft real-time systems.

RODAIN The RODAIN system [37] developed at the University of Helsinki, Finland, is a firm real-time database system that primarily is intended for telecommunication. It is designed for high degree of availability and fault-tolerance. It is tailored to fit the characteristics of telecommunication transactions identified as short queries and updates, and long massive updates.

ARTS-RTDB The ARTS-RTDB system [38], developed at the University of Virginia, Charlottesville, US, supports both soft and hard real-time transactions. It uses imprecise computing to ensure timeliness of transactions. It is built on top of the ARTS real-time operating system [39].

2.2.5 Real-Time Properties

STRIP

The STRIP [35] system is intended for soft real-time systems in which the ultimate goal is to make as many transactions as possible commit before their deadlines. It is developed for the UNIX operating system which might seem odd, but since STRIP is intended to run in open systems UNIX was selected. Even though UNIX is not a real-time operating system it has, when used in the right way, turned out to be a good operating system for soft real-time systems since a real-time scheduling emulator can be placed on top of the UNIX scheduler which sets the priorities according to some real-time scheduling algorithm, a real-time behavior can be achieved. However, there have been some problems that needed to be overcome while implementing STRIP that is related to real-time scheduling in UNIX. For example to force a UNIX schedule not to adjust the priorities of processes as they are running. Non-real-time operating systems often have mechanisms to dynamically adjust the priorities of processes during run time to add to throughput and fairness in the system. By using Posix UNIX, a new class of processes whose priorities are not adjustable by the scheduler was provided. In [40] versions of the earliest deadline and least slack algorithms was emulated on top of a scheduler in a traditional operating

system, and the results showed that earliest deadline using absolute deadline and least slack for relative deadline performed equal or better than their real-time counterparts, while the emulated earliest deadline for relative deadline missed more deadlines than the original EDF algorithm for low systems load. However, during high load the emulated algorithm outperformed the original EDF. In STRIP EDF, highest value first, highest density value first or a custom scheduling algorithm can be used to schedule transactions. To minimize unpredictability, the STRIP system is a main-memory database, thus removing disk I/O.

DeeDS

DeeDS [13] is intended for both hard and soft real-time systems. It is built for the OSE delta real-time operating system developed by ENEA DATA [41]. This distributed hard real-time operating system is designed for both embedded uniprocessor and multiprocessor systems. If used in a multiprocessor environment the CPUs are loosely coupled. A more detailed description of OSE delta can be found in [42]. The DeeDS system consists of two parts, one part that handles non critical system services and one that handles the critical. All critical systems services are executed on a dedicated processor to simplify overhead cost and increase the concurrency in the system. The DeeDS system is, as the STRIP system, a main memory database.

BeeHive

BeeHive utilizes the concept of data-deadline, forced-wait and the data deadline based scheduling algorithms [43]. These concepts assure that transactions are kept temporally consistent by assigning a more stringent deadline to a transaction based on the maximum allowed age of the data elements that are used in the transaction, thus the name data deadline. The key concept of forced wait is to, if a transaction cannot complete before its data deadline, postpone it until data elements get updated, thus giving the transaction a later data deadline. These transactions can then be scheduled using, for example, the earliest data-deadline first (EDDF) or data deadline based least slack first (DDLSF) algorithms [43].

For real-time data storage a four level memory hierarchy is assumed: main memory, non-volatile RAM (e.g., Flash), persistent disk storage, and archival storage (e.g., tape storage) [12]. The main memory is used to store data that is currently in use by the system. The non-volatile RAM is used as a disk-

cache for data or logs that are not yet written to disk. Furthermore, system data structures like lock tables and rule bases can be stored in non-volatile RAM. The disk is used to store the database, just like any disk-based database. Finally, the tape storage is used to make backups of the system. By using these four levels of memory management a higher degree of predictability can be achieved than for entirely disk-based databases, since the behavior can be somewhat like a main-memory database even though it is a disk-based system.

REACH

REACH is built on top of Texas Instruments openOODB system, which is an open database system. By an open system we refer to a system that can be controlled, modified, and partly extended by developers and researcher [44]. The OpenOODB system is by itself not a real-time database system, but an object-oriented database. Some efforts has been done in the REACH project to aid the user with respect to system predictability [45] such as milestones, contingency plans, a benchmarking tool and a trace tool.

Milestones are used to monitor the progress of transactions. If a milestone is missed, the transaction can be aborted and a contingency plan is released instead. This contingency plan should be implemented to handle a missed deadline situation by either degrading the system in a safe way, or produce a good-enough result before the original transactions deadline. One could say that a contingency plan in cooperation with milestones works as a watch-dog that is activated when a deadline is about to be missed.

The REACH system has a benchmarking tool that can be used to measure execution times for method invocations and event triggering. The REACH trace tool can trace the execution order in the system. If the benchmarking tool is used together with the trace tool, system behavior and timeliness could be determined.

RODAIN

The RODAIN system is a real-time database system intended for telecommunication applications. The telecommunication environment is a dynamic and complex environment that deals with both temporal data and non-temporal data. A database in such a system must, support both soft and firm real-time transactions [37]. However, the believe is that hard real-time databases will not be used in telecommunication in the near future since they generally are to too expensive to develop or use to suit the market [37].

One of the key issues for RODAIN is availability. Therefore a fault-tolerant system is necessary. The RODAIN system is designed to have two nodes, thus giving full database replication. This is especially important since the database is a main-memory system. It uses a primary and a mirror node. The primary database sends logs to the mirror database which in turn acknowledges all calls. The communication between the nodes is assumed to be reliable and have a bounded message transfer delay. A watchdog monitor keeps track of any failing subsystem and can instantly swap the primary node and the mirror node in case of a failure, see figure 2.7. The failed node always recover as a mirror node and loads the database image from permanent storage. Furthermore, the User Request Interpreter system (URIS) can keep track of all ongoing transaction and take appropriate actions if a transaction fails. Since both the primary and the mirror node has a URIS no active transactions will be lost if a failure in the primary system leads to a node swap between the primary and the mirror. Figure 2.7 shows the architecture of RODAIN. The subsystem OODBMS handles all traditional database activities, like concurrency control, physical storage, schema management etc. There are three interfaces to the database, the traditional user interface, referred to as the Applications interface, the DBMS nodes interface, and the Mirror interface. Noteworthy is the distinction between mirror nodes and distributed DBMS nodes. The mirror nodes purpose is to ensure a fault-tolerant running, while the distribution nodes are used for normal database distribution.

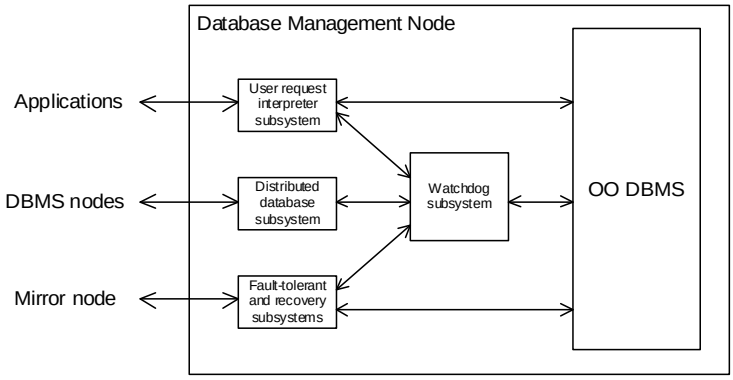

Figure 2.7: *RODAIN DBMS node. Figure from [37]*

ARTS-RTDB

The relational database ARTS-RTDB [38] incorporates an interesting feature called imprecise computing. If a query, when its deadline expires, is not finished, the result so far can be returned to the client if the result can be considered meaningful. Take a query that calculates the average value of hundreds of values in a relation column. If an adequate amount of values has been calculated at the time of deadline, the result could be considered meaningful but imprecise and it is therefore returned to the client anyway.

ARTS-RTDB is built on top of the distributed real-time operating system ARTS, developed by Carnegie Mellon University [39]. ARTS schedules tasks according to a time-varying value function, which specifies both criticality and semantics importance. It does support both soft and hard real-time tasks.

In ARTS-RTDB the most critical data operations has been identified to be the INSERT, DELETE, UPDATE and SELECT operations. Efforts has therefore been made to optimize the system to increase the efficiency for those four operations. According to [38] many real-time applications almost only use these operations at run-time.

2.2.6 Distribution

To increase concurrency in the system, distributed databases can be used. Distributed in a sense that the database copies reside on different computer nodes in a network. If the data is fully replicated over all nodes, applications can access any node to retrieve a data element. Unfortunately, maintaining different database nodes consistent with each other is not an easy task, especially if timeliness is crucial. One might have to trade consistency for timeliness. A mechanism that continuously tries to keep the database nodes as consistent with each other as possible is needed. Since the system is distributed all updates to a database node must be propagated via message passing, or similar, thus adding significantly to the database overhead because of the added amount of synchronization transactions in the system. One of the most critical moments for a transaction deployed in a distributed database is the point of commit. At this point all the involved nodes must agree upon committing the transaction or not.

One of the most commonly used commit protocols for distributed databases is the two-phase commit protocol [46]. As the name of the algorithm suggests the algorithm consists of two phases, the prepare-phase and the commit-phase. In the prepare phase, the coordinator for the transaction sends a `prepare`

message to all other nodes and then waits for all answers. The nodes then answers with a `ready` message if they can accept the commit. If the coordinator receives `ready` from all nodes a `commit` message is broadcasted, otherwise the transaction is aborted. The coordinator then finally waits for a `finished` message from all nodes to confirm the commit. Hereby is consistency guaranteed between nodes in the database.

DeeDS

The DeeDS [13] system has a less strict criteria for consistency, in order to enforce timeliness. They guarantee that each node has a local consistency, while the distributed database might be inconsistent due to different views of the system on the different nodes. This approach might be suitable for systems that mostly rely on data that is gathered locally, but sometimes uses data imported from other subsystems. Consider an engine that has all engine data, e.g., ignition control, engine speed and fuel injection, stored in a local node of a distributed database. The timeliness of these local data items are essential in order to run the engine. To further increase the efficiency of the engine, remotely gathered data, like data from the transmission box, with less critical timing requirements can be distributed to the local node.

STRIP

The concept of streams communicating between nodes in a distributed system has been recognized in STRIP [35]. Nodes can stream views or tables to each other on a regular basis. The user can select if whole tables/views or delta tables, which only reflects changes to the tables/views, should be streamed. Furthermore it is selectable if the data should be streamed, periodically when some data has reached a predefined age or only after an explicit instruction to stream. In figure 2.8 we can see the process architecture of STRIP. The middle part of the figure shows the execution process, the transaction queue, the result queue and the request process. This part makes the query layer of the database and can be compared to an ordinary database that processes transactions and queries from the application. Remote application addresses queries or transactions to the requesting process, which in turn places the transaction in the transaction queue. However, local applications can access the transaction queue directly via a client library, thus minimizing overhead. Each execution processes can execute a transaction from the transaction queue or an update transaction from one of the update queues. The update queues are fed from the

incoming update streams. When a table/view needs to be sent to an outgoing stream, one of the execution processes enqueues it to the update queue read by the export process. It has been shown that when the update frequency is low, the import and export processes can run at a high priority without significantly affecting transaction response times but when the frequency increases the need for a different update scheduling policy is necessary [35], see section 2.2.7.

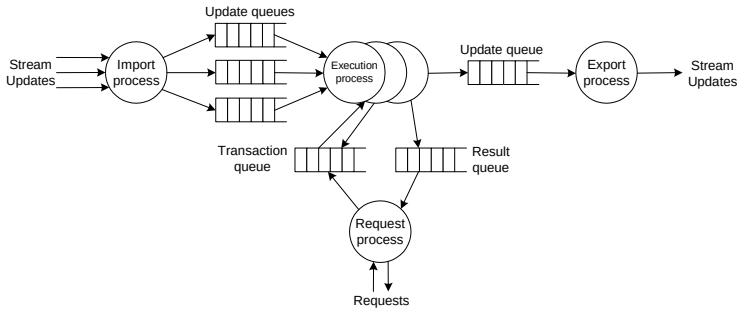

Figure 2.8: *The process architecture of the STRIP system. Figure from [35].*

BeeHive

Unlike most databases BeeHive is a virtual database, in a sense that in addition to its own data storage facility it can use external databases and incorporate them as one database. These databases can be located either locally or on a network, e.g., the Internet.

In [47], a virtual database is defined as a database that organizes data scattered through the Internet into a queriable database. Consider a web service that compares prices of products on the Internet. When you enter the database you can search for a specific product, and the service will return to you a list of Internet retailers and to what price they are selling the product. The database system consists of four parts [47]:

- Wrappers. A wrapper is placed between a web-page or database and the virtual database. In the web-page case, the wrapper converts the textual content of the web-page into a relational format understandable by the virtual database. When using a wrapper in the database case, the wrapper converts the data received from the database into the virtual

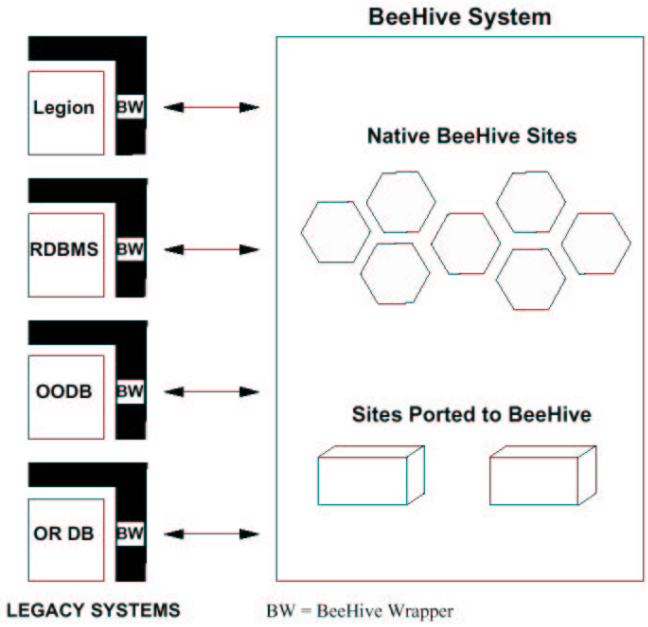

Figure 2.9: *The BeeHive virtual database. Figure from [12]*

database format. Such a wrapper can be implemented using a high level description language, e.g., Java applets.

- Extractors. The relations received from the wrapper most often contains unstructured textual data in which the interesting data needs to be extracted.
- Mappers. The mapper maps extracted relations into a common database schema.
- Publishers. The publisher is used to present the database to the user, it has similar functionality as a traditional database.

BeeHive can communicate with other virtual databases, web-browsers or independent databases using wrappers. These wrappers are Java applets and may use the JDBC standard, see figure 2.9.

RODAIN

RODAIN system supports data distribution. The nodes can be anything between fully replicated to completely disjoint. Their belief is that only a few requests in telecommunications need access to more than one database node and that the request distributions among the databases in the system can be arranged to be almost uniform [37].

ARTS-RTDB

ARTS-RTDB has been extended to support distribution. The database nodes use a shared file which contains information that binds all relations to the server responsible for a particular relation. Since the file is shared between the nodes, it is to be treated as a shared resource and must therefore be accessed using a semaphore. It is believed that if relations are uniformly distributed between the nodes and if no hot-spot relations exist, an increase in performance will be seen. A hot-spot relation is a relation that many transactions use, and such a relation can lead to a performance bottleneck in the system.

2.2.7 Transaction Workload Characteristics

DeeDS

DeeDS supports sporadic (event-triggered) and periodic transactions. There are two classes of transactions: critical (hard transactions) and non-critical (soft transactions). To ensure that a deadline is kept for a critical transaction, milestone monitoring and contingency plans are used. A milestone can be seen as a deadline for a part of the transaction. If a milestone is passed, it is clear that the transaction will not make its deadline, a contingency plan can be activated and the transaction is aborted. A contingency plan should, if implemented correctly, deal with the consequences of a transaction aborting. One way to compute less accurate result and present it in time, similar to imprecise computing used by ARTS-RTDB (see section 2.2.5). Milestones and contingency plans are discussed further in [48]. The timing requirements for a transaction is passed to the system as a parameterized value function, this approach reduces the computational cost and the storage requirements of the system.

The scheduling of transactions is made online in two steps. First, a sufficient schedule is produced. This schedule meets the minimum requirements, for example all hard deadlines and 90 percent of the soft deadlines are met. Secondly, the event monitors worst-case execution time is subtracted from the

remaining allowed time for the scheduler during this time slot, and this time is used to refine and optimize the transaction schedule [49].

STRIP

STRIP supports firm deadlines on transactions [50], thus when a transaction misses its deadline it will be aborted. There are two classes of transactions in STRIP, low value and high value transactions. When transactions are scheduled for execution is determined by transaction class, value density, and choice of scheduling algorithm. The value density is the ratio between the transactions value and the remaining processing time. The scheduling of transactions is in competition with the scheduling of updates, therefore updates also have the same properties as transactions with respect to classes, values, and value density. STRIP has four scheduling algorithms, Updates First (UF), Transactions First (TF), Split Updates (SU), and Apply Updates on Demand (OD).

- Updates First schedules all updates before transactions, regardless of the value density of the transactions. It is selectable if a running transaction is preempted by an update of if it should be allowed to commit before the update is executed. For a system with a high load of updates this policy can cause long and unpredictable execution times for transactions. However, for systems where consistency between nodes and where temporal data consistency is more important than transaction throughput, this can be a suitable algorithm. Further, this algorithm is also suitable for systems where updates are prioritized over transactions. Consider a database running stock exchange transactions. A change in price for a stock must be performed before any pending transactions in order to get the correct value of the transaction.

- Transactions First is the opposite of update first. Transactions are always scheduled in favor of updates. However a transactions can not preempt a running update. This because updates most often have short execution time compared to transactions. This algorithm suits systems that values throughput of transactions higher than temporal consistency between the nodes. Transactions first might be useful in a industrial controlling system. If, in an overloaded situation, the most recent version of a data element is not available, the system might gain in performance from being able to run its control algorithms using an older value.

- Split Updates make use of the different classes of updates. It schedules updates and transactions according to the following order: high value

updates, transactions, and low value updates. This algorithm combines the two previous algorithms and would suit a system where one part of the data elements is crucial with respect to temporal consistency, at the same time as transaction throughput is important.

- Apply Updates on Demand execute transactions before updates, but with the difference that when a transaction encounters stale data, the update queue is scanned for an update that is waiting to update the stale data element. This approach would enable a high throughput of transactions with a high degree of temporal consistency. However, calculating the execution time for a transaction is more difficult since the worst case is that all data elements used in the transaction has pending updates. Applying updates on demand resembles about the ideas of triggered updates [51].

Since transaction deadlines are firm, thus transactions are valueless after their deadline has expired, a mechanism called feasible deadlines can be used for transactions. The mechanism aborts transactions that has no chance of committing before their deadline.

BeeHive

The fact that the database is virtual and may consist of many different kinds of underlying databases is transparent. The user accesses the database through at least one of the four interfaces provided by BeeHive: FT API, RT API, Security API, and QoS API (see figure 2.10). FT API is the fault-tolerant interface. Transaction created with this interface will have some protection against processor failures and transient faults due to power glitches, software bugs, race conditions, and timing faults. RT API is the real-time interface, which enables the user to specify time constraints and typical real-time parameters along with transactions. The security interface can be used by application that for instance demands authorization for users and applications. Furthermore encryption can be supported. The Quality of Service (QoS) interface is primarily used for multimedia transactions. The user can specify transactions demands on quality of service.

When a transaction arrives from any interface, it is sent to the service mapper, which transforms it to a common transaction format used for all transactions in the system, regardless of its type, e.g., real-time transaction. It is then passed on to the resource planner, which determines which resources that will be needed. Furthermore, it is passed to the admission controller, which in cooperation with the resource planner, decides if the system can provide a

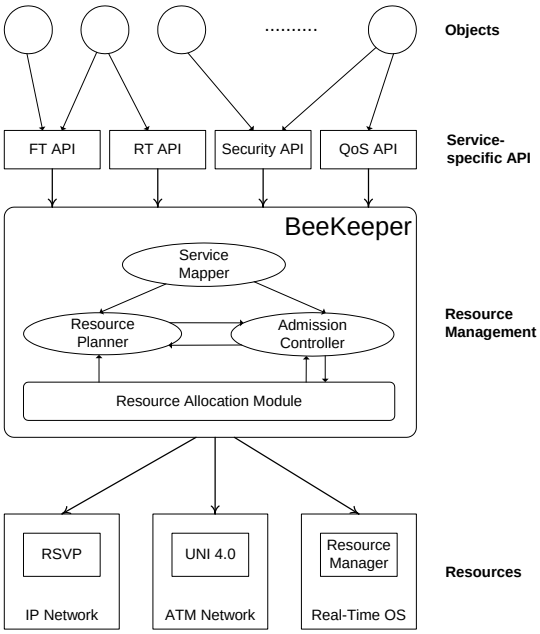

Figure 2.10: *The resource manager in BeeHive. Figure from [12]*

satisfactory service for the transaction. If that is the case, it is sent to the resource allocation module, which globally optimizes the use of resources in the system. Finally the transaction is sent to the correct resource, e.g., a network or the operating system etc.

RODAIN

Five different areas for databases in telecommunication is identified [52]:

1. Retrieval of persistent customer data.
2. Modifications of persistent customer data.
3. Authorization of customers, e.g., PIN codes.
4. Sequential log writing.

5. Mass calling and Tele voting. This can be performed in large blocks.

From these five areas, three different types of transactions can be derived [53]: short simple queries, simple updates, and long massive updates. Short simple queries are used when retrieving customer data and authorizing customers.Simple updates are used when modifying customer data and writing logs. Long massive updates are used when tele voting and mass calling is performed.

The concurrency control in RODAIN supports different kinds of serialization protocols. One protocol is the τ -serialization in which a transaction may be allowed to read old data as long as the update is not older than a predefined time [37].

Apart from a transactions deadline, an isolation level can also be assigned to a transaction. A transaction running on a low isolation level accepts that transactions running on a higher isolation level are accessing locked objects. However, it cannot access objects belonging to a transaction with a high degree of isolation.

ARTS-RTDB

The RTDB uses a pessimistic concurrency control, strict two phase locking with priority abort [11]. This means that as soon as a higher prioritized transactions wants a lock that is owned by a transaction with a low priority, the low level transaction is aborted. To avoid the costly process of rolling back the aborted transaction, all data writing is performed on copies. At the point of commit, the transaction asks the lock-manager if a commit can be allowed. If this is the case, the transaction invokes a subsystem that writes all data into the database.

2.2.8 Active Databases

Active databases can perform actions not explicitly requested from the application or environment. Consider a data element z, which is derived from two other data elements, x and y. The traditional way to keep z updated would be to periodically poll the database to determine if x or y have been altered. If that is the case, a new value of z would be calculated and the result written back to the database. Periodic polling is often performed at a high cost with respect to computational cost and will increase the complexity of the application task schedule. A different approach would be to introduce a trigger that would

cause an event as soon as either x or y has been changed. The event would in its turn start a transaction or similar that would update z. This functionality would be useful for other purposes than to update derived data elements. It could, for example, be used to enforce boundaries of data elements. If a data element, e.g., data that represents a desired water level in our example application in figure 2.1, has a maximum and minimum allowed value than this restriction can be applied in the database instead of in the application. An attempt to set an invalid value could result in some kind of action, e.g., an error exception or a correction of the value to the closest valid value. By applying this restriction in the database it is abstracted away from the application, thus reducing the risk of programming errors or inconsistency between transactions that use this data element.

One way to handle such active behavior is through the use of ECA rules [54], where ECA is an abbreviation of Event-Condition-Action-rules, see figure 2.11. The events in an active database need to be processed by an event

```
ON <event E>
   IF <condition C>
      THEN <action A>
```

Figure 2.11: *The structure of an ECA rule.*

manager, which sends them to the rule manager. The rule manager in its turn locates all ECA rules that are affected by the newly arrived event and checks their conditions. For those ECA rules whose conditions are true, the action is activated. There are, however, three distinct approaches on when and how to execute the actions [48]:

- Immediate: When an event arrives, the active transaction is suspended and condition evaluation and possibly action execution is done immediately. Upon rule completion, the transaction is resumed. This approach affects the response time of the transaction.
- Deferred: When an event arrives, the active transaction is run to completion and then condition evaluation and possibly action execution is done. This approach does not affect the response time of the triggering transaction.
- Decoupled: When using this approach, condition evaluation and possibly action execution are performed in one or several separate transac-

tions. This has the advantage that these updates will be logged even if the triggering transaction aborts.

One disadvantage with active databases is that the predictability, with respect to response times, is decreased in the system. This is because of the execution of the rules. For immediate or deferred rules the response time of the triggering transaction is prolonged because of the execution of rules prior to its point of commit. To be able to calculate a response time for such a transaction, the execution time of all possible rules that the transaction can invoke. These rules can in their turn activate rules, so called cascading, which can create arbitrary complex calculations. When decoupled rules are used, transactions can generate new transactions in the system. These transactions and their priorities, response times, and possible rule invocations must be taken in account when calculating transaction response time, resulting in very pessimistic worst-case-execution times.

Some extensions to ECA rules which incorporates time constraints have been studied in [5]. For a more detailed description about ECA rules, active databases and active real-time databases see [48, 55]

STRIP

The STRIP rule system checks for events at each transaction, T, commit-time and for all events whose condition is true a new transaction T' is created that executes the execution clause of the rule. Thus, the STRIP system uses a decoupled approach. T' is activated for execution after the triggering transaction has committed. By default T' is released immediately after commit of the triggering transaction, but it can be delayed using the optional `after` statement, see figure 2.12.

```
WHEN <eventlist>
   [IF <condition>]
   THEN
      EXECUTE <action>
      [AFTER <time-value>]
```

Figure 2.12: *A simplified structure of a STRIP-rule. The optional* `AFTER` *-clause enables for batching of several rule-executions in order to decrease computational costs.*

The `after` statement enables all rule execution clauses, activated during the time window between rule evaluation and the execution clause specified by the `after` statement, to be batched together. If multiple updates on a single data element exists in the batch, only the ones necessary to derive the resulting value of that data element is executed, thus saving computational cost [56]. To have the execution clause in a separate transaction simplifies rule processing. The triggering transaction can commit regardless of the execution of the executions clauses, since the condition clause of the rules cannot alter the database state in anyway. Possible side-effects of the rules are then dealt with in a separate transaction. One disadvantage with decoupled execution clauses is that the condition results and transition data are deleted when the triggering transaction commits, thus, data is not available for the triggered transaction. This has been solved in STRIP by the `bind as` statement. This statement binds a created table to a relation that can be accessed by the execution clause.

DeeDS

Since DeeDS supports hard real-time systems the rule processing must be designed to retain as much predictability with respect to timeliness as possible. This is achieved by restricting rule processing. Cascading, for example, is limited so that unbounded rule triggering is avoided when a rule is executed. The user can define a maximum level of the cascade depth. If the cascading goes deeper than that an exception is raised. Furthermore, condition evaluation is restricted to logical expressions on events and object parameters, and method invocations. DeeDS system an extended ECA rules [49] which also allows for specifying timing constraints to rules, e.g., deadlines.

Event monitoring has also been designed for predictability. Both synchronous and asynchronous monitoring is available. In synchronous mode the event monitor is invoked on time-triggered basis and results have shown that throughput is higher than if the asynchronous (event-triggered) event monitoring is used, but at the cost of longer minimum event delays. When synchronous mode is used, event bursts will not affect the system in an unpredictable way [49]. In DeeDS, the immediate and deferred coupling modes are not allowed. Instead when a rule is triggered and the condition states that the rule should be executed, a new transaction is created, which is immediately suspended. If a contingency plan exists for this rule, a contingency transaction is also created and suspended. All combinations of decoupled execution is supported in DeeDS [57], see table 2.12. Depending on which mode that is selected, the scheduler activates the rule transactions according to the mode. The exclusive

	Sequential	Parallel
Independent	T_r independent of Tt	
Dependent	T_r cannot *start* until Tt *commits*, otherwise it is *discarded*.	T_r cannot *complete* until Tt *commits*, otherwise it is *aborted*.
Exclusive	T_r cannot *start* until Tt *aborts*, otherwise it is *discarded*.	T_r cannot *complete* until Tt *aborts*, otherwise it is *aborted*.

Table 2.12: *Table showing all decoupled execution modes supported by the DeeDS system. Tt is the triggering transaction and T_r is the rule transaction. Table from [57].*

mode is used to schedule contingency plans.

REACH

REACH allows ECA rules to be triggered in all modes, immediate, deferred and decoupled. Decoupled rules might be dependent or independent, furthermore they can be executed in parallel, sequential, or exclusive mode. The exclusive mode is intended for contingency plans. The ECA rules used in REACH have no real-time extensions.

There are however some restrictions on when to use the different coupling modes. Rules triggered by a single method event can be executed in any coupling mode, while a rule triggered by a composite event only cannot be run in immediate mode. This is because if the events that make up composite event originate in multiple transactions, no identification of the spawning transactions is possible [36]. Rules that are invoked by temporal events (time-triggered) can only be run in an independent decoupled mode, since they are not triggered by a transaction [58].

To simplify the creation of the rule-base a tool, GRANT, has been developed. This graphical tool, prompts the user for necessary information, provides certain default choices, creates C++ language structures and maps rules to C functions stored in a shared library [45].

The event manager in REACH consumes events according to two policies: chronological or most recent. The chronological approach executes all events in the order they arrive while the most recent strategy only executes the most recent instance of every event. What policy to use is dependent of the semantics

of the application. Consider the example with the stock market database again. If, in an overloaded situation, the database is not able to immediately execute all stock price updates, only the most recent update of a stock is interesting, since this update contains the current price of this stock. For a system like this, the most recent approach would be sufficient, while in a telephone billing system it might not. In a system like that, all updates to the database consist of a value to add to the existing record, therefore most recent is not sufficient here.

2.2.9 Observations

The databases presented represent a set of systems that basically have the same purpose, i.e., to efficiently store information in embedded systems or real-time systems. They mainly consist of the same subsystems, e.g., interfaces, index system and concurrency control system. But yet the systems behave so differently. Some systems that are primarily event driven, and on the other hand systems that might suit well for static scheduling. Some systems are relational while other are object-oriented. Some systems are intended for a specific application, e.g., telecommunication, process control or multimedia application.

Furthermore, an interesting observation can be made about these systems. The research platforms focus more on functionality while the commercial systems are more interested in user friendliness, adaptability, and standards compliance. While the research platforms have detailed specification on new internal mechanisms that improve performance under certain circumstances, like new concurrency controls or scheduling policies, the commercial systems supports multiple interfaces to ease integration with the application and supports standards like ODBC, OLE/DB and JDBC. Looking back at the criteria mentioned by ABB in section 2.2.2, we can see that most of the criteria are fulfilled by the commercial products. To be able to combine these criteria with the technology delivered by the research platforms would be a contribution.

One important issue that arises is: Which embedded database system should one choose for a specific type of application? The task of choosing the best database system can be a long and costly process and probably not without compromises. A different approach would be to have a more generic database that can be customized, possibly with aid of a tool, to fit a specific application.

2.3 Summary

This final chapter starts with a summary of the main issues we have identified in the report with respect to current state-of-the-art in the area of embedded databases for embedded real-time systems (section 2.3.1).

2.3.1 Conclusions

Embedded systems, real-time systems, and database systems are research areas that have been actively studied. However, research on embedded databases for embedded real-time systems, explicitly addressing the development and design process, is sparse. A database that can be used in an embedded real-time system must handle transactions with temporal constraints, and must, at the same time, be suitable for embedded systems with limited amount of resources, i.e., the database should have small footprint, be portable to different operating system platforms, have efficient resource management, and be able to recover from a failure without external intervention.

As we have shown in this report, there are a variety of different embedded databases on the market. However, they vary significantly in their characteristics. Differences in products are typically the way data are organized in the database (data model), the architecture of the database (DBMS model), and memory usage. Commercial embedded databases also provide different interfaces for applications to access the database, and support different operating system platforms. Application developers must choose carefully the embedded database their application requires. This is a difficult, time consuming and costly process, with a lot of compromises. One solution could be to have a more generic embedded database platform that can be tailored and optimized such that it is suitable for different applications. Existing real-time database research platforms are unsuitable in this respect, since they are mainly monolithic systems, and, as such, they are not easily tailored for new applications having different or additional requirements.

2.3.2 Future Work

Our research is focused on providing an experimental research platform for building embedded databases for embedded real-time systems. At a high-level, the platform consists of two parts. First, we intend to develop a component library, which holds a set of methods, that can be used when building an embedded database. Initially, we will develop a set of components that deal with con-

currency control, scheduling, and main-memory techniques. At the next step, we develop tools that, based on the application requirements, will support the designer when building an embedded database using these components. More importantly, we want to develop application tools and techniques that: (i) support the designer in the composition and tailoring of an embedded database for a specific system using the developed components, where the application requirements are given as an input; (ii) support the designer when analyzing the total system resource demand of the composed embedded database system; and (iii) help the designer by recommending components and methods if multiple components can be used, based on the application requirements. Further, such a tool will help the designer to make trade-off analysis between conflicting requirements early in the design phase.

Bibliography

[1] R. K. Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1995.

[2] R. Camposano and J. Wilberg. Embedded System Design. *Design Automation for Embedded Systems*, 1(1):5–50, 1996.

[3] J. Stankovic. Misconceptions About Real-Time Computing: a Serious Problem for Next-Generation Systems. *IEEE Computer*, 21(10):10–19, October 1988.

[4] M. A. Olson. Selecting and Implementing an Embedded Database System. *IEEE Computers*, 33(9):27–34, Sept. 2000.

[5] K. Ramamritham. Real-Time Databases. *International Journal of distributed and Parallel Databases*, 1(2):199–226, 1993.

[6] Polyhedra Plc. http://www.polyhedra.com.

[7] MBrane Ltd. http://www.mbrane.com.

[8] Pervasive Software Inc. http://www.pervasive.com.

[9] Sleepycat Software Inc. http://www.sleepycat.com.

[10] TimesTen Performance Software. http://www.timesten.com.

[11] Y-K. Kim, M. R. Lehr, D. W. George, and S. H. Son. A Database Server for Distributed Real-Time Systems: Issues and Experiences. In *Proceedings of the Second IEEE Workshop on Parallel and Distributed Real Time Systems*, pages 66–75. IEEE Computer Society, April 1994.

[12] J. A. Stankovic, S. H. Son, and J. Liebeherr. *Real-Time Databases and Information Systems*, chapter BeeHive: Global Multimedia Database Support for Dependable, Real-Time Applications, pages 409–422. Kluwer Academic Publishers, 1997.

[13] S. F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring. DeeDS Towards a Distributed and Active Real-Time Database System. *ACM SIGMOD Record*, 25(1):38–40, 1996.

[14] J. Lindstrom, T. Niklander, P. Porkka, and K. Raatikainen. A Distributed Real-Time Main-Memory Database for Telecommunication. In *Proceedings of the Workshop on Databases in Telecommunications*, pages 158–173. Springer, September 1999.

[15] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley Longman Publishing Co., 2000.

[16] S. Ortiz. Embedded Databases Come Out of Hiding. *IEEE Computer*, 33(3):16–19, March 2000.

[17] Raima Corporation. Databases in Real-time and Embedded Systems. http://www.raimabenelux.com/, February 2001.

[18] Pervasive Software Inc. Pervasive.SQL 2000 Reviewers guide. Technical white paper.

[19] B. Liu. Embedded Real-Time Configuration Database for an Industrial Robot. Master's thesis, Linkping University, 2000.

[20] Sybase Inc. SQL.Anywhere. http://www.sybase.com.

[21] FairCom Corp. c-tree Plus. http://www.faircom.com.

[22] Objectivity inc. Objectivity/DB. http://www.objectivity.com.

[23] Persistence software inc. PowerTier. http://www.persistence.com.

[24] POET software corp. POET Object. http://www.poet.com.

[25] T-W. Kuo, C-H. Wei, and K-Y. Lam. Real-Time Data Access Control on B-Tree Index Structures. In *Proceedings of the 15th International Conference on Data Engineering*, pages 458–467. IEEE Computer Society, March 1999.

[26] T. J. Lehman and M. J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the 12th Conference on Very Large Databases*, pages 294–303. Morgan Kaufmann, August 1986.

[27] H. Lu, Y. Ng, and Z. Tian. T-Tree or B-Tree: Main Memory Database Index Structure Revisited. In *Proceedings of the 11th Australasian Database Conference*, pages 65–73. IEEE Computer Society, January 2000.

[28] W. Litwin. Linear Hashing: A New Tool for File and Table Addressing. In *Proceedings of the 6th International Conference on Very Large Databases*, pages 212–223. Springer, October 1980.

[29] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *The communications of the ACM*, 19(11):624–633, November 1976.

[30] H. T. Kung and J. T. Robinsson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[31] P. S. Yu, K. Wu, K. Lin, and S. H. Son. On Real-Time Databases: Concurrency Control and Scheduling. *Proceedings of the IEEE*, 82(1):140–157, January 1994.

[32] F. Baothman, A. K. Sarje, and R. C. Joshi. On Optimistic Concurrency Control for RTDBS. In *Proceedings IEEE Region 10 International Conference on Global Connectivity in Energy, Computer, Communication and Control*, volume 2, pages 615–618. IEEE Computer Society, December 1998.

[33] X. Song and J. Liu. Maintaining Temporal Consistency: Pessimistic vs. Optimistic Concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):786–796, October 1995.

[34] J. Huang, J.A. Stankovic, K. Ramamritham, and D.F. Towsley. Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 35–46. Morgan Kaufmann, September 1991.

[35] B. Adelberg, B. Kao, and H. Garcia-Molina. Overview of the STanford Real-time Information Processor (STRIP). *SIGMOD Record*, 25(1):34–37, 1996.

[36] J. Zimmermann and A. P. Buchmann. REACH. In *N. Paton (ed): Active Rules in Database Systems, Springer-Verlag*, 1998.

[37] J. Taina and K. Raatikainen. RODAIN: A Real-Time Object-Oriented Database System for Telecommunications. In *Proceedings of the workshop on Databases: active and real-time*, pages 10–14. ACM Press, November 1996.

[38] Y-K. Kim, M. R. Lehr, D. W. George, and S. H. Song. A Database Server for Distributed Real-Time Systems: Issues and Experiences. In *Proceedings of the Second IEEE Workshop on Parallel and Distributed Real-Time Systems*, pages 66–75. IEEE Computer Society, April 1994.

[39] H. Tokuda and C. Mercer. ARTS: A Distributed Real-Time Kernel. *ACM SIGOPS Operating Systems Review*, 23(3):29–53, July 1989.

[40] B. Adelberg, H. Garcia-Molina, and B. Kao. Emulating Soft Real-Time Scheduling Using Traditional Operating System Schedulers. In *Proceedings of IEEE Real-Time System Symposium*, pages 292–298. IEEE Computer Society, December 1994.

[41] ENEA Data. OSE Real-time system. http://www.enea.se.

[42] J. Hansson. Dynamic Real-Time Scheduling in OSE Delta. Technical Report HS-IDA-TR-94-007, Dept. of Computer Science, Univ. of Skövde, 1994.

[43] M. Xiong, R. Sivasankran, J. Stankovic, K. Ramamritham, and D. Towsley. Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics. In *Proceedings of the 17th IEEE Real-time Systems Symposium*, pages 240–251. IEEE Computer Society, December 1996.

[44] D. L. Wells, J. A. Blakeley, and C. W. Thompson. Architecture of an Open Object-Oriented Database Management System. *IEEE Computer*, 25(10):74–82, October 1992.

[45] A. P. Buchmann, A. Deutsch, J. Zimmermann, and M. Higa. The REACH active OODBMS. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 476–476. ACM Press, May 1995.

[46] J. Gray. Notes on Database Operating Systems. In *Operating Systems: An Advanced Course*, number 60 in Lecture Notes in Computer Science, pages 393–481. Springer-Verlag, 1978.

[47] A. Gupta, V. Harinarayan, and A. Rajaraman. Virtual Database Technology. In *Proceedings of 14th International Conference on Data Engineering*, pages 297–301. IEEE Computer Society, February 1998.

[48] J. Eriksson. Real-Time and Active Databases: A Survey. In *Proceedings of the Second International Workshop on Active, Real-Time and Temporal Databases*, nr. 1553 in Lecture note series., pages 1–23. Springer-Verlag, December 1998.

[49] J. Mellin, J. Hansson, and S. Andler, editors. *Real-Time Database Systems: Issues and Applications*, volume 396 of *The Kluwer International Series in Engineering And Computer Science*, chapter Refining Timing Constraints of Application in DeeDS. Kluwer Academic Publishers, 1997.

[50] B. S. Adelberg. *STRIP: A Soft Real-Time Main Memory Database for Open Systems*. PhD thesis, Stanford University, 1997.

[51] Q. N. Ahmed and S. V. Vrbsky. Triggered Updates for Temporal Consistency in Real-Time Databases. *Real-Time Systems*, 19(3):209–243, November 2000.

[52] K. Raatikainen and J. Taina. Design Issues in Database Systems for Telecommunication Services. In *Proceedings of IFIP-TC6 Working conference on Intelligent Networks*, pages 71–81. Kluwer Academic Publishers, August 1995.

[53] T. Niklander and K. Raatikainen. RODAIN: A Highly Available Real-Time Main-Memory Database System. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, pages 271–278, September 1998.

[54] S. Chakravarthy et al. HiPAC: A Research Project in Active, Time-Constrained Database Management. Technical Report XAIT-8902, Xerox Advanced Information Technology, 1989.

[55] R. Sivasankaran, J. Stankovic, D. Towsley, B. Purimetla, and K. Ramamritham. Priority Assignment in Real-Time Active Databases. *The VLDB Journal*, 5(1):19–34, 1996.

[56] B. Adelberg, H. Garcia-Molina, and J. Widom. The STRIP rule system for efficiently maintaining derived data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 147–158. ACM Press, May 1997.

[57] J. Eriksson. *Specifying and Managing Rules in an Active Real-Time Database System*. Licenciate Thesis, Number 738. Linköping University, Sweden, December 1998.

[58] A. P. Buchmann, J. Zimmermann, J. A. Blakeley, and D. L. Wells. Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions. In *Proceedings of the 11th International Conference on Data Engineering*, pages 117–128. IEEE Computer Society Press, 1995.

Chapter 3

Paper B: Data Management Issues in Vehicle Control Systems: a Case Study

Dag Nyström, Aleksandra Tešanović, Christer Norström, Jörgen Hansson, and Nils-Erik Bånkestad

In Proceedings of 14th EUROMICRO Conference on Real-Time Systems Vienna, Austria, June 2002

Abstract

In this paper we present a case study of a class of embedded hard real-time control applications in the vehicular industry that, in addition to meeting transaction and task deadlines, emphasize data validity requirements. We elaborate on how a database could be integrated into the studied application and how the database management system (DBMS) could be designed to suit this particular class of systems.

3.1 Introduction

In the last ten years, control systems in vehicles have evolved from simple single processor systems to complex distributed systems. At the same time, the amount of information in these systems has increased dramatically and is predicted to increase further with 7-10% per year [1]. In a modern car there can be several hundreds of sensor values to keep track of. Ad hoc techniques that are normally used for storing and manipulating data objects as internal data structures in the application result in costly development with respect to design, implementation and verification of the system. Further, the system becomes hard to maintain and extend. Since the data is handled ad hoc, it is also difficult to maintain its temporal properties. Thus, the need for a uniform and efficient way to store and manipulate data is obvious. An embedded real-time database providing support for storage and manipulation of data would satisfy this need.

In this paper we study two different hard real-time systems developed at Volvo Construction Equipment Components AB, Sweden, with respect to data management. These systems are embedded into two different vehicles, an articulated hauler and a wheel loader. These are typical representative systems for this class of vehicular systems. Both systems consist of a number of nodes distributed over a control area network (CAN).

The system in the articulated hauler is responsible for I/O management and controlling of the vehicle. The system in the wheel loader is, in addition to controlling the vehicle, responsible for updating the driver display. We study structures of the systems and their data management requirements to find that today data management is implemented as multiple data storages scattered throughout the system. The systems are constructed out of a finite number of tasks. Each task in the system is equipped with a finite amount of input and output ports, through which inter-task communication is performed. Due to intense communication in both systems, several hundred ports are used. These ports are implemented as shared memory locations in main memory, scattering the data even more.

We study temporal properties of the data in the systems and conclude that the they could benefit from a real-time database (RTDB). Furthermore, we discuss how the current architecture could be redesigned to include a RTDB. The important feature of a RTDB in these systems is to guarantee temporal consistency and validity [2] rather than advanced transaction handling. In a typical vehicular system, nodes vary both in memory size and computation and, hence, there is a need for a scalable RTDB that can be tailored to suit different kinds

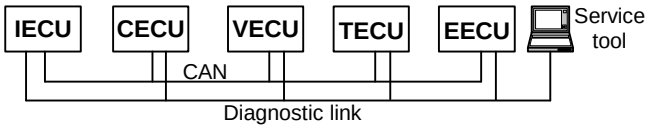

Figure 3.1: The overall architecture of the vehicle controlling system.

of systems. In this paper transactions refer to a number of reads and/or updates of data in a database. Thus, tasks can contain transactions.

The contribution of this paper is a detailed case-study of the two Volvo applications. Furthermore, we elaborate on how the existing hard real-time system could be transformed to incorporate a RTDB. This architectural transition would allow data in the system to be handled in a structured way. In this architecture, the database is placed between the application and the I/O management. We elaborate on why concurrency control, for this transformed system, is not necessarily needed for retaining the integrity of transactions. Moreover, we argue that a hard real-time database that would suit this system could be implemented using passive components only, i.e., a transaction is executed on the calling task's thread of execution. This implies that the worst-case transaction execution time is added to the worst-case execution time of the task, retaining a bounded execution time for all tasks.

In section 2 we study the existing vehicle systems and their data management requirements in detail. In section 3 we discuss: how the systems could be redesigned to use a RTDB, the implications for the application and the RTDB, and how existing real-time database platforms would suit the studied application. We conclude our work and present future challenges in section 4.

3.2 The Case Study

The vehicle control system consists of several subsystems called electronic control units (ECU), connected through two serial communication links: the fast CAN link and the slow diagnostic link, as shown in the figure 3.1. Both the CAN link and the diagnostic link are used for data exchange between different ECUs. Additionally, the diagnostic link is used by diagnostic (service) tools. The number of ECUs can vary depending on the way functionality is divided between ECUs for a particular type of vehicle. For example, the articulated

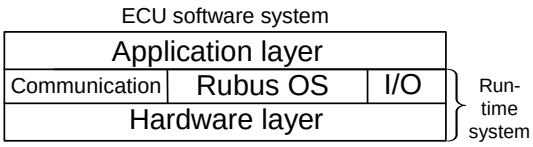

Figure 3.2: The structure of an ECU.

hauler consists of five ECUs: instrumental, cabin, vehicle, transmission and engine ECU, denoted IECU, CECU, VECU, TECU, and EECU, respectively. In contrast, the wheel loader control system consists of three ECUs, namely IECU, VECU, and EECU.

We have studied the architecture and data management of the VECU in the articulated hauler, and the IECU in the wheel loader. The VECU and the IECU are implemented on hardware platforms supporting three different storage types: EEPROM, Flash, and RAM. The memory in an ECU is limited, normally 64Kb RAM, 512Kb Flash, and 32Kb EEPROM. Processors are chosen such that power consumption and cost of the ECU are minimized. Thus, processors run at 20MHz (VECU) and 16MHz (IECU) depending on the workload.

Both VECU and IECU software systems consist of two layers: a run-time system layer and an application layer (see figure 3.2). The run-time system layer on the lower level contains all hardware-related functionality. The higher level of the run-time system layer contains an operating system, a communication system, and an I/O manager. Every ECU uses the real-time operating system Rubus. The communication system handles transfer and reception of messages on different networks, e.g., CAN. The application is implemented on top of the run-time system layer. The focus of our case study is data management in the application layer. In the following section we briefly discuss the Rubus operating system. This is followed by sections where functionality and a structure of the application layer of both VECU and IECU, are discussed in more detail (in following sections we refer to the application layer of the VECU and IECU as the VECU (software) system and the IECU (software) system).

3.2.1 Rubus

Rubus is a real-time operating system designed to be used in systems with limited resources [3]. Rubus supports both off-line and on-line scheduling, and consists of two parts: (i) red part, which deals with hard real-time; and (ii) blue part, which deals with soft real-time.

The red part of Rubus executes tasks scheduled off-line. The tasks in the red part, also referred to as red tasks, are periodic and have higher priority than the tasks in the blue part (referred to as blue tasks). The blue part supports tasks that can be invoked in an event-driven manner. The blue part of Rubus supports functionality that can be found in many standard commercial real-time operating system, e.g., priority-based scheduling, message handling, and synchronization via semaphores. Each task has a set of input and output ports that are used for communication with other red tasks. Rubus is used in all ECUs.

3.2.2 VECU

The vehicle system is used to control and observe the state of the vehicle. The system can identify anomalies, e.g., an unnormal temperature. Depending on the criticality of the anomaly, different actions, such as warning the driver, system shutdown etc., can be taken. Furthermore, some of the vehicle's functionality is controlled by this system via sensors and actuators. Finally, logging and maintenance via the diagnostics link can also be performed using a service tool that can be connected to the vehicle.

All tasks in the system, except the communication task, are non-preemptive tasks scheduled off-line. The communication task uses its own data structures, e.g., message queues, thus no resources are shared with other tasks. Since non-preemptive tasks run until completion and cannot be preempted, mutual exclusion is not necessary. The reason for using non-preemptive off-line scheduled tasks is to minimize the runtime overhead and to simplify the verification of the system.

The data in the system can be divided into five different categories: (1) sensor/actuator raw data, (2) sensor/actuator parameter data, (3) sensor/actuator engineering data, (4) logging data, and (5) parameter data.

The *sensor/actuator raw data* is a set of data elements that are either read from sensors or written to actuators. The data is stored in the same format as they are read/written. This data, together with the *sensor/actuator parameter data*, is used to derive the *sensor/actuator engineering data*, which can be

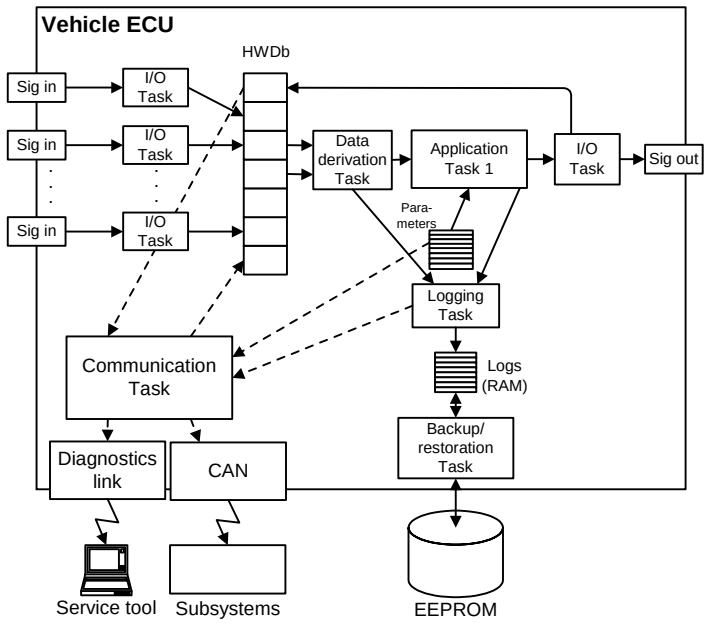

Figure 3.3: The original architecture of the VECU.

used by the application. The sensor/actuator parameter data contains reference information about how to convert raw data received from the sensors into engineering data. For example, consider a temperature sensor, which outputs the measured temperature as a voltage T_{volt}. This voltage needs to be converted to a temperature T using a reference value T_{ref}, e.g., $T = T_{volt} \cdot T_{ref}$.

In the current system, the sensor/actuator (raw and parameter) data are stored in a vector of data called a hardware database (HW Db), see figure 3.3. The HW Db is, despite its name, not a database but merely a memory structure. The engineering data is not stored at all in the system but is derived "on the fly" by the data derivation tasks. Apart from data collected from local sensors and the application, sensor and actuator data derived in other ECUs is stored in the HW Db. The distributed data is sent periodically over the CAN bus. From the application's point of view the locality of the data is transparent in the sense that it does not matter if the data is gathered locally or remotely.

Some of the data derived in the applications is of interest for statistical and

maintenance purposes and therefore the data is logged (referred to as *logging data*) on permanent storage media, e.g., EEPROM. Most of the logging data is cumulative, e.g., the vehicle's total running time. These logs are copied from EEPROM to RAM in the startup phase of the vehicle and are then kept in RAM during runtime, to finally be written back to EEPROM memory before shutdown. However, logs that are considered critical are copied to EEPROM memory immediately at an update, e.g., warnings. The *parameter data* is stored in a parameter area. There are two different types of parameters, permanent and changeable. The permanent parameters can never be changed and are set to fulfill certain regulations, e.g., pollution and environment regulations. The changeable parameters can be changed using a service tool.

Most controlling applications in the VECU follow a common structure residing in one precedence-graph. The sensors (Sig In) are periodically polled by I/O tasks (typically every 10 ms) and the values are stored in their respective slot in the HW Db. The data derivation task then reads the raw data from the HW Db, converts it, and sends it to the application task. The application task then derives a result that is passed to the I/O task that both writes it back to the HW Db and to the actuator I/O port.

3.2.3 IECU

The IECU is a display electronic control unit that controls and monitors all instrumental functions, such as displaying warnings, errors, and driver information on the driver display. The IECU also controls displaying service information on the service display (a unit for servicing the vehicle). It furthermore controls the I/O in the driver cabin, e.g., accelerator pedal, and communicates with other ECUs via CAN and the diagnostic link.

The IECU differs from the VECU in several ways. Firstly, the data volume in the system is significantly higher since the IECU controls displays and, thus, works with a large amount of images and text information. Moreover, the data is scattered in the system and depending on its nature, stored in a number of different data structures as shown in figure 3.4. Similarly to the HW Db, data structures in the IECU are referred to as databases, e.g., image databases, menu databases and language databases. Since every text and image information in the system can be displayed in thirteen different languages, the interrelationships of data in different data storages are significant.

A dominating task in the system is the task updating the driver display. This is a red task, but it differs from other red tasks in the system since it can be preempted by other red tasks in the IECU. However, scheduling of all tasks

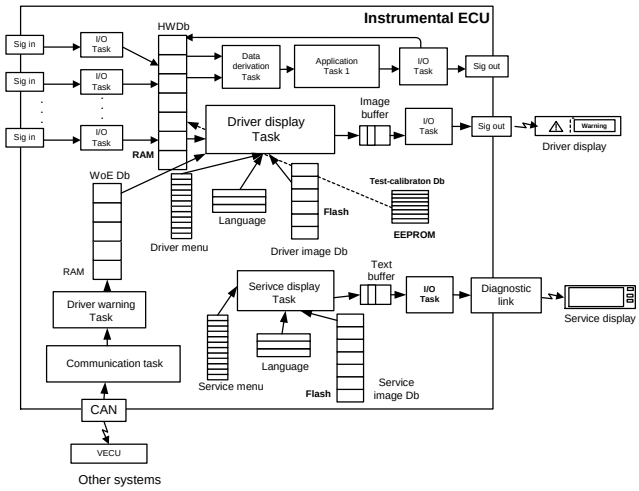

Figure 3.4: The original architecture of the IECU.

is performed such that all possible data conflicts are avoided.

Data from the HW Db in the IECU is periodically pushed on to the CAN link and copied to the VECU's HW Db. Warnings or errors (WoE) are periodically sent through the CAN link from/to the VECU and are stored in the dedicated part of RAM, referred to as the WoE database (WoE Db). Hence, the WoE Db contains information of active warnings and errors in the overall wheel loader control system. While WoE Db and HW Db allow both read and write operations, the image and menu databases are read-only databases.

The driver display is updated as follows (see figure 3.4). The driver display task periodically scans the databases (HW Db, WoE Db, menu Db) to determine the information that needs to be displayed on the driver display. If any active WoE exists in the system, the driver display task reads the corresponding image, in the specified language, from the image database located in a persistent storage and then writes the retrieved image to the image buffer. The image is then read by the blue I/O task, which then updates the driver display with an image as many times as defined in the WoE Db. Similarly, the driver display task scans the HW Db and menu database. If the hardware database has been updated and this needs to be visualized on the driver display, or if

data in the menu organization has been changed, the driver display task reads the corresponding image and writes it to the driver display as described previously. In case the service tool is plugged into the system, the service display task updates the service display in the same way as described for the driver display, but using its own menu organization and image database, buffer, and the corresponding blue I/O task.

3.2.4 Data Management Requirements

The table 3.1 gives an overview of data management characteristics in the VECU and IECU systems. The following symbols are used in the table: As can be seen from the table 3.1, all the data in both systems are scattered in groups of different flat data structures referred to as databases, e.g., HW Db, image Db, WoE Db and language Db. These databases are flat because data is structured mostly in vectors, and the databases only contain data with no support for DBMS functionality.

The nature of the systems put special requirements on data management (see table 3.1): (i) static memory allocation only, since dynamic memory allocation is not allowed due to the safety-critical aspect of the systems; (ii) small memory consumption, since production costs should be kept as low as possible; and (iii) diverse data accesses, since data can be stored in different storages, e.g., EEPROM, Flash, and RAM.

Most data, from different databases and even within the same database, is logically related. These relations are not intuitive, which makes the data hard to maintain for the designer and programmer as the software of the current system evolves. Raw values of sensor readings and actuator writings in the HW Db are transformed into engineering values by the data derivation task, as explained in section 3.2.2. The engineering values are not stored in any of the databases, rather they are placed in ports (shared memory) and given to application tasks when needed.

The period times of updating tasks ensure that data in both systems (VECU and IECU) are correct at all times with respect to absolute consistency. Furthermore, task scheduling, which is done off-line, enforces relative consistency of data by using an off-line scheduling tool. Thus, data in the system is temporally consistent (we denote this data property in the table as temporal validity). Exceptions are permanent data, e.g., images and text, which is not temporally constrained (see table 3.1).

[1]The feature is true only for some engineering data in the VECU.

v — feature is true for the data type in the VECU,
i — feature is true for the data type in the IECU, and
x — feature is true for the data type in both VECU and IECU.

Data types / Management characteristics		*Sensor*	*Actuator*	*Engineering*	*Parameters*	*WoE*	*Image&Text*	*Logs*
Data source	HW Db	x	x			i		
	Parameter Db				x			
	WoE Db					i		
	Image Db						i	
	Language Db						i	
	Menu Db						i	
	Log Db							v
Memory type	RAM	x	x	x	x	x		v
	Flash						i	
	EEPROM				x			v
Memory allocation	Static	x	x	x	x	x	i	v
	Dynamic							
Interrelated with other data		x	x	x	x	x	i	v
Temporal validity		x	x	x		x		v
Logging	Startup							v
	Shutdown							v
	Immediately			v[1]				
Persistence		x	x	v[1]	x	x		
Logically consistent		x	x	x	x			
Indexing							i	
Transaction type	Update	x	x	x	x	x		v
	Write-only	x		x				
	Read-only		x	x	x		i	
	Complex update	x	x	x				v
	Complex queries	x	x	x	x	x	i	v

Table 3.1: Data management characteristics for the systems.

One implication of the systems' demand on reliability, i.e., the requirement that a vehicle must be movable at all times, is that data must always be temporally consistent. Violation of temporal consistency is viewed as a system error, in which case three possible actions can be taken by the system: use a predefined default data value (most often), use an old data value, or shutdown of the functions involved (system exposes degraded functionality).

Some data is associated with a range of valid values, and is kept logically consistent by tasks in the application (see table 3.1). The negative effect of enforcing logical consistency by the tasks is that programmers must ensure consistency of the task set with respect to logical constraints.

Persistence in the systems is maintained by storing data on stable storage, but there are some exceptions to the rule, e.g., RPM data is never copied to stable storage. Also, some of the data is only stored in stable storage, e.g., internal system parameters. In contrast, data imperative to systems' functioning is immediately copied to stable storage, e.g., WoE logs are copied to/from stable storage at startup/shutdown.

Several transactions exist in the VECU and IECU systems: (i) update transactions, which are application tasks reading data from the HW Db; (ii) write-only transactions, which are sensor value update tasks; (iii) read-only transactions, which are actuator reading tasks; and (iv) complex update transactions, which originate from other ECUs. In addition, complex queries are performed periodically to distribute data from the HW Db to other ECUs.

Data in the VECU is organized in two major data storages, RAM and Flash. Logs are stored in EEPROM and RAM (one vector of records), while 251 items structured in vectors are stored in the HW Db. Data in the IECU is scattered and interrelated throughout the system even more in comparison to the VECU (see table 3.1). For example, the menu database is related to the image database, which in turn is related to the language Db and the HW Db. Additionally, data structures in the IECU are fairly large. HW Db and WoE Db resides in RAM. HW Db contains 64 data items in one vector, while WoE Db consists of 425 data items structured as 106 records with four items each. The image Db and the language Db reside in Flash. All images can be found in 13 different languages, each occupying 10Kb of memory. The large volume of data in the image and language databases requires indexing. Indexing is today implemented separately in every database, and even every language in the language Db has separate indexing on data.

The main problems we have identified in existing data management can be summarized as follows:

- all data is scattered in the system in a variety of databases, each representing a specialized data store for a specific type of data;
- engineering values are not stored in any of the data stores, but are placed in ports, which enlarges maintenance complexity and makes adding of functionality in the system a difficult task;
- application tasks must communicate with different data stores to get the data they require, i.e., the application does not have a uniform access or view of the data;
- temporal and logical consistency of data is maintained by the tasks, increasing the level of complexity for programmers when maintaining a task set; and
- data from different databases exposes different properties and constraints, which complicates maintenance and modification of the systems.

3.3 Modeling the System to Support a RTDB

To be able to implement a database in the real-time system, the system needs to be redesigned to support a database. For the studied application, this could be done by separating I/O management from the application.

As mentioned in section 3.2.2 and shown in figure 3.3, the data flow goes from the I/O tasks, via the HW Db and application tasks to the I/O tasks to the right, sending the values to the actuators. The transition of such a system could, at a high level, be performed in three steps. The first step is to separate all I/O tasks from the application. This can be viewed as “folding the architecture”. By doing this an I/O management is formed that is separated from the control application. The second step is to place the real-time database between the I/O management and the control application as shown in figure 3.5. In the Volvo case, the HW Db is replaced by a RTDB which is designed using a passive library. The desired properties of this RTDB are described more in detail in section 3.3.1. The I/O tasks are modified to communicate with the database instead of the data derivation tasks. The application is, analogue to the I/O tasks, also modified to communicate with the database only. At this stage the database splits two domains, the I/O domain and the application domain. The last step is to collect additional data that might be scattered in the system into the database, e.g., parameter and logging data. The tasks that communicate with these data stores are, similar to the I/O and application tasks, modified

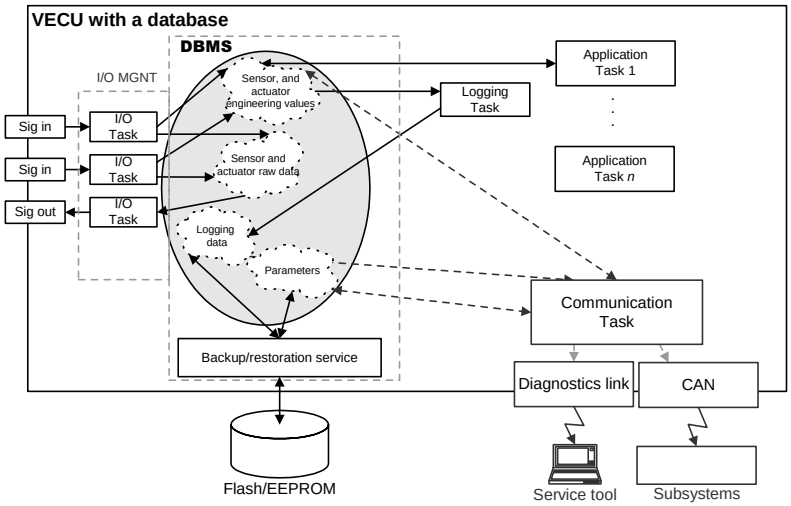

Figure 3.5: The new architecture of the VECU.

to communicate with the database only. With this architecture we have separated the application from the I/O management and the I/O ports. The database could be viewed as a layer between the application and the operating system, extending the real-time operating system functionality to embrace data management, see figure 3.6. All data in the system is furthermore collected in one database, satisfying the need for a uniform and efficient way to store data. Another important issue, shown in figure 3.5, is that both the raw sensor data and the engineering data, previously derived by the data derivation task, are now included in the database. The actual process of deriving the engineering values could be performed in multiple ways. The I/O tasks could be modified to embrace this functionality, so that they write both the raw value and the engineering value to the database. Another, perhaps more elegant, way of solving this is to use database rules, where a rule is triggered inside the database as soon as a data item is updated. This rule would execute the code that derive the engineering value.

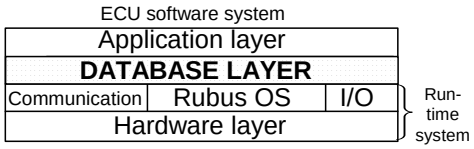

Figure 3.6: The structure of an ECU with an embedded database.

3.3.1 Data Management Implications

When designing a system running on the described hardware, one of the main goals is to make it run with as small processor and memory footprint as possible. Traditionally, for data, this is achieved by using as small data structures as possible. A common misconception is that a database is a very large and complex application that will not fit into a system such as this [4]. However, there are, even commercial DBMSs that are as small as 8Kb, e.g., Pervasive.SQL. It should be added, though, that even if the size of the DBMS is very small, the total memory used for data storage can increase because of the added overhead for each data element stored in the database. This is because memory is used to store the database indexing system, data element locks, etc. Clearly, there is a trade-off between functionality and memory requirements. The most important issue in this application is timeliness. The system cannot be allowed to miss deadlines and behave unpredictable in any way. It is off-line scheduled with non-preemptable tasks. This fact provides some interesting implications. No task, except the driver display task (see section 3.2.3), can preempt another task. Thus, database conflicts are automatically avoided since the tasks themselves are mutually exclusive. This makes database concurrency control and locking mechanisms unnecessary because only one transaction can be active in such a system at any given time, thus serialization of transactions are handled “manually”. This is similar to why semaphores are not needed for non-preemptive real-time systems [5].

Implementing a database into the existing system will have benefits. All data, regardless of on which media it is stored, can be viewed as one consistent database. The relations between the data elements can be made clearer than today. For example, currently an image retrieval in the IECU is performed by first looking in the image Db, then in the language Db, and finally in the HW Db. A database query asking for an image, using the current language and the

correct value from the HW Db, can be done in one operation. Furthermore, constraints on data can be enforced centrally by the database. If a data element has a maximum and a minimum value, the database can be aware of this and raise an exception if an erroneous value is inserted. Today, this is performed in the application, implying a responsibility that constraints are made consistent between all tasks that use the data.

In this system the transaction dispatching delay is removed since a database scheduler is not needed. Also, conflict resolution is removed since no conflicts will occur because only one transaction is running at any given time. Regarding the data access time, it will increase as the database grows larger. However, this can be tolerated since the increase can be controlled in two ways. First of all, as the database is a main-memory database, any access to data will be significantly shorter than the execution times of the transactions. To decrease the transaction response times various indexing strategies especially suited for main-memory databases can be used, e.g., t-tree [6] and hashing algorithms [7].

The application investigated in this paper consists of, as previously mentioned, primarily non-preemptable tasks, hence no concurrency control is needed. One interesting question is how this approach would fit into a preemptable off-line scheduled system. This would call for some kind of concurrency control in the database, thus possibly resulting in unpredictable response times for transactions due to serialization conflicts. However, this could be avoided by solving all conflicts off-line. Since all transactions in the system are known a priori, we know all data elements that each transaction touches. This allows us to feed the off-line scheduler with information about which transactions might cause conflicts if preempted by each other. The scheduler can then generate a schedule where tasks containing possibly conflicting transactions do not preempt each other.

3.3.2 DBMS Design Implications

If we can bound the worst case response time for a specific transaction, we can add this time to the calling tasks worst-case execution time (WCET) without violating the hard real-time properties of the system.[1] Execution of the transaction on its task's thread instead of having separate database tasks, decreases the number of tasks in the schedule, making it easier for the off-line scheduling tool to find a feasible schedule. However a question one should ask is: How do

[1] The response time is defined as the time from transaction initiation to the completion of the transaction.

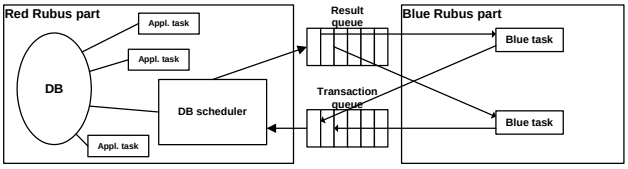

Figure 3.7: A database that supports non-periodic transactions via an external scheduler.

we find the worst case response time for a transaction? There are basically four different circumstances that define the response time of a transaction, namely: (i) the time it takes from the instant a transaction is released until the instant it is dispatched; (ii) the actual execution time of the code that needs to be executed; (iii) the time it takes to access the data elements through the indexing system; and (iv) the time it takes to resolve any serialization conflicts between transactions. For an optimistic concurrency control this would imply the time it takes to run the transaction again, and for a pessimistic concurrency control it would be the time waiting for locks.

In this system the transaction dispatching delay is removed since a database scheduler is not needed in this system. Also, conflict resolution is removed since no conflicts will occur because only one transaction is running at any given time. Regarding the data access time, it will increase as the database grows larger. However this can be tolerated since the increase is bounded if a suitable indexing structure is used, such as the T-tree [6] or the hashing [7] algorithms.

In future versions of this application, it is expected that some of the functionality is moved to the blue part, thus requiring concurrency control and transaction scheduling since we cannot predict the arrival times of blue tasks. Moving parts of the application to the blue part could imply restructuring the data model if a database is not used. If new functionality from the database will be needed in the future, the database schema can be reused. Still, this would not allow non-periodic transactions. Furthermore, it would not allow tasks scheduled online, e.g., blue tasks. However, an extension that would allow this is shown in figure 3.7. A non-preemptable scheduler task is placed in the red part of Rubus. Since this task is non-preemptable it is mutually exclusive towards all other tasks and can therefore have access to the entire database. If this task is scheduled as a periodic task, it acts like a server for transaction

scheduling. Thus, the server reads all transactions submitted to the transaction queue, process them and return the results in the result queue (blue tasks are preemptable and, hence, their execution can be interleaved).

From the blue tasks' perspective, they can submit queries, and since we know the periodicity of the scheduler task we can determine the worst-case execution time for these transactions. From the red tasks' perspective, nothing has changed, they are still, either as in the current system, non-preemptive resulting in no conflicts, or they are scheduled so that no conflicts can occur. It is important to emphasize that this method is feasible only if any transaction processed by the scheduler task can be finished during one instance of the scheduling task. If this requirement cannot be met an online concurrency control is needed.

3.3.3 Mapping Data Requirements to Existing Database Platforms

Today there are database platforms, both research and commercial platforms, which fulfill a subset of the system requirements. The DeeDS [8] platform, for example, is a hard real-time research database system that support hard periodic transactions. It also has a soft and a hard part. Furthermore, the DeeDS system uses milestones and contingency plans. These hard periodic transactions would suit the red Rubus tasks and would, if used with milestones and contingency plans, suit the Volvo application. The milestones would check that no deadlines are about to be missed, and the contingency plans would execute alternate actions if that is the case. DeeDS is, as the STRIP system [9] a main memory database that would suit this application. The Beehive [10] system implements the concept of temporal validity, that would ensure that temporal consistency always exists in the database. These platforms are designed as monolithic databases with the primary intent to meet multiple application requirements with respect to real-time properties, and on a lesser extent the embedded requirements. As such, they are considered to provide more functionality than needed, and as a consequence, they are not optimal for this application given the need to minimize resource usage as well as overall system complexity.

On the commercial side, embedded databases exist that are small enough to fit into the current system, e.g., the Berkeley DB by Sleepycat Software Inc. and the Pervasive.SQL database for embedded systems. There are also pure main-memory databases on the market, e.g., Polyhedra and TimesTen. Polyhedra, DeeDS, STRIP, and REACH [11] are active database systems, which

can enforce consistency between the raw values and the engineering values, and thereby removing the need for the data derivation task. However, integrating active behavior in a database makes timing analysis of the system more difficult. The Berkeley DB system allows the user to select between no concurrency control and an pessimistic concurrency control [12]. If Volvo should decide upon moving part of the functionality to the blue part, concurrency in the database would be necessary. The option of choosing whether or not to use concurrency control would enable the use of the same DBMS, database scheme, and database interface regardless of the strategy being used. Unfortunately, none of the commercial systems mentioned have any real-time guarantees and are therefore not suitable for this type of application.

3.4 Conclusions

We have studied two different hard real-time systems from the vehicular industry with respect to data management, and we have found that data is scattered throughout the system. This implies that getting a full picture of all existing data and its interrelations in the system is difficult.

Further, we have redesigned the architecture of the system to support a real-time database. In this new architecture all tasks communicate through the database instead of using ports, and the database provides a uniform access to data. This application does not need all the functionality provided by existing real-time database research platforms, and issues like concurrency and scheduling have been solved in an easy way. Currently the application is designed so that all tasks are off-line scheduled. All tasks, except the driver display task, are non-preemptive. However, future versions of the application are expected to embrace preemption as well as online scheduled tasks.

Finally, we have discussed mapping the data management requirements to existing databases. Some of the database platforms, both research and commercial, offer functionality that is needed by the system, but at the same time they introduce a number of unnecessary features.

Our future work will focus on the design and implementation of a tailorable real-time embedded database [13]. This includes: (i) developing a set of real-time components and aspects, (ii) defining rules for composing these components into a real-time database system, and (iii) developing a set of tools to support the designer when composing and analyzing the database system. A continuation of this case study where we will implement our database in the Volvo system is planned.

Bibliography

[1] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano - a Revolution in On-Board Communications. Technical report, Volvo Technology Report, 1998.

[2] K. Ramamritham. Real-Time Databases. *International Journal of distributed and Parallel Databases*, 1(2):199–226, 1993.

[3] Rubus OS - reference manual. Articus Systems, 1996.

[4] J. Stankovic, S. Son, and J. Hansson. Misconceptions About Real-Time Databases. *IEEE Computer*, 32(6):29–36, June 1999.

[5] J. Xu and D. L. Parnas. Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations. *IEEE Transactions on Software Engineering*, 16(3):360–369, 1990.

[6] H. Lu, Y. Ng, and Z. Tian. T-Tree or B-Tree: Main Memory Database Index Structure Revisited. In *Proceedings of the 11th Australasian Database Conference*, pages 65–73. IEEE Computer Society, January 2000.

[7] W. Litwin. Linear Hashing: A New Tool for File and Table Addressing. In *Proceedings of the 6th International Conference on Very Large Databases*, pages 212–223. Springer, October 1980.

[8] S. F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring. DeeDS Towards a Distributed and Active Real-Time Database System. *ACM SIGMOD Record*, 25(1):38–40, 1996.

[9] B. Adelberg, B. Kao, and H. Garcia-Molina. Overview of the STanford Real-time Information Processor (STRIP). *SIGMOD Record*, 25(1):34–37, 1996.

[10] J. A. Stankovic, S. H. Son, and J. Liebeherr. *Real-Time Databases and Information Systems*, chapter BeeHive: Global Multimedia Database Support for Dependable, Real-Time Applications, pages 409–422. Kluwer Academic Publishers, 1997.

[11] A. P. Buchmann, A. Deutsch, J. Zimmermann, and M. Higa. The REACH active OODBMS. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 476–476. ACM Press, May 1995.

[12] X. Song and J. Liu. Maintaining Temporal Consistency: Pessimistic vs. Optimistic Concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):786–796, October 1995.

[13] Aleksandra Tešanović, Dag Nyström, Jörgen Hansson, and Christer Norström. Embedded Databases for Embedded Real-Time Systems: A Component-Based Approach. Technical Report MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-43/2002-1-SE, Dept. of Computer Engineering, Mälardalen University, January 2002.

Chapter 4

Paper C: Database Pointers: a Predictable Way of Manipulating Hot Data in Hard Real-Time Systems

Dag Nyström, Aleksandra Tešanović, Christer Norström, and Jörgen Hansson

In Proceedings of The 9th International Conference on Real-Time and Embedded Computing Systems and Applications, Tainan, Taiwan, February 2003

Abstract

Traditionally, control systems use ad hoc techniques such as shared internal data structures, to store control data. However, due to the increasing data volume in control systems, these internal data structures become increasingly difficult to maintain. A real-time database management system can provide an efficient and uniform way to structure and access data. However the drawback with database management systems is the overhead added when accessing data. In this paper we introduce a new concept called database pointers, which provides fast and deterministic accesses to data in hard real-time database management systems compared to traditional database management systems. The concept is especially beneficial for hard real-time control systems where many control tasks each use few data elements at high frequencies. Database pointers can co-reside with a relational data model, and any updates made from the database pointer interface are immediately visible from the relational view. We show the efficiency with our approach by comparing it to tuple identifiers and relational processing.

4.1 Introduction

In recent years, the complexity of embedded real-time controlling systems has increased. This is especially true for the automotive industry [1]. Along with this increased complexity, the amount of data that needs to be handled has grown in a similar fashion. Since data in real-time systems traditionally is handled using ad hoc techniques and internal data structures, this increase of data is imposing problems when it comes to maintenance and development.

One possible solution to these problems is to integrate an embedded real-time database management system (RTDBMS) within the real-time system. A RTDBMS can provide the real-time system with a uniform view and access of data. This is especially useful for distributed real-time systems where data is shared between nodes. Because of the uniform access of data, the same database request is issued regardless if the data is read at the local node or from a distributed node. Furthermore, RTDBMSs can ensure consistency, both logical and temporal [2]. Finally, RTDBMSs allow so called ad hoc queries, i.e., requests for a view of data performed during run-time. This is especially useful for management and system monitoring. For example, consider a large control system being monitored from a control room. Suddenly, a temperature warning is issued. An ad hoc query showing the temperatures and pressures of multiple sub-systems might help the engineers to determine the cause of the overheating.

Integrating a RTDBMS into a real-time system also has drawbacks. There will most certainly be an added overhead for retrieving data elements. This is partly because of the indexing system used by most database management systems (DBMS). The indexing system is used to locate where in the memory a certain data element is stored. Usually, indexing systems use some tree structure, such as the B-tree [3] and T-tree [4] structures, or a hashing table [5].

An increase of the retrieval times for data has, apart from longer task execution, one additional drawback. Since shared data in a concurrent system needs to be protected using semaphores or database locking systems, the blocking factor for hot data can be significant. Hot data are data elements used frequently by multiple tasks. Hot data is sensitive to congestion and therefore it is of utmost importance to lock hot data for as short time as possible. Furthermore, it is important to bound blocking times to allow response time analysis of the system. Examples of hot data are sensor readings for motor control of a vehicle, e.g., rpm and piston position. These readings are continuously stored by I/O tasks and continuously read by controlling tasks. A congestion involving these heavily accessed data elements might result in a malfunction. On the

other hand, information regarding the level in the fuel tank is not as crucial and might be accessed less frequent, and can therefore be considered non-hot data.

In this paper we propose the concept of database pointers, which is an extension to the widely used tuple identifiers [6]. Tuple identifiers contain information about the location of a tuple, typically a block number and an offset. Database pointers have the efficiency of a shared variable combined with the advantages of using a RTDBMS. They allow a fast and predictable way of accessing data in a database without the need of consulting the DBMS indexing system. Furthermore database pointers provide an interface that uses a "pointer-like" syntax. This interface is suitable for control system applications using numerous small tasks running at high frequencies. Database pointers allow fast and predictable accesses of data without violating neither temporal or logical consistency nor transaction serialization. It can be used together with the relational data model without risking a violation of the database integrity.

The paper is outlined as follows. In section 4.2 we describe the type of systems we are focusing on. In addition, we give a short overview of tuple identifiers and other related work. Database pointers are explained in section 4.3, followed by an evaluation of the concept, which is presented in section 4.4. In section 4.5 we conclude the paper.

4.2 Background and Related Work

This paper focuses on real-time applications that are used to control a process, e.g., critical control functions in a vehicle such as motor control and brake control. The flow of execution in such a system is: (i) periodic scanning of sensors, (ii) execution of control algorithms such as a PID-regulators, and (iii) propagation of the result to the actuators.

The execution is divided into a number of tasks, e.g., I/O-tasks and control tasks. The functions of these tasks are fixed and often limited to a specific activity. For example, an I/O-task's only responsibility could be to read the sensor-value on an input-port and write it to a specific location in memory, e.g., a shared variable [7].

In addition to these, relatively fixed control tasks, a number of management tasks exists, which are generally more flexible than the control tasks, e.g., management tasks responsible for the user interface.

4.2.1 Relational Query Processing

Relational query processing is performed using a data manipulation language (DML), such as SQL. A relational DML provides a flexible way of viewing and manipulating data. The backside of this flexibility is performance loss.

Figure 4.1 shows a typical architecture of a DBMS. The DBMS provides access to data through the SQL interface. A query, requesting value x, passed to this interface will go through the following steps:

1. The query is passed from the application to the SQL interface.
2. The SQL interface requests that the query should be scheduled by the transaction scheduler.
3. The relational query processor parses the query and creates an execution plan.
4. The locks needed to process the query are obtained by the concurrency controller.
5. The tuple containing x is located by the index manager.
6. The tuple is then fetched from the database.
7. All locks are released by the concurrency controller.
8. The result is returned to the application.

Finally, since the result from a query issued to a relational DBMS is a relation in itself, a retrieval of the data element x from the resulting relation is necessary. This is done by the application.

In this example we assume a pessimistic concurrency control policy. However, the flow of execution will be roughly the same if a different policy is used.

4.2.2 Tuple Identifiers

The concept of tuple identifiers was first proposed back in the 70's as internal mechanisms for achieving fast accesses to data while performing relational operations, such as joins and unions. It was implemented by IBM in an experimental prototype database called System R [6]. A tuple identifier is a data type containing a pointer to one tuple stored either on a hard drive or in main

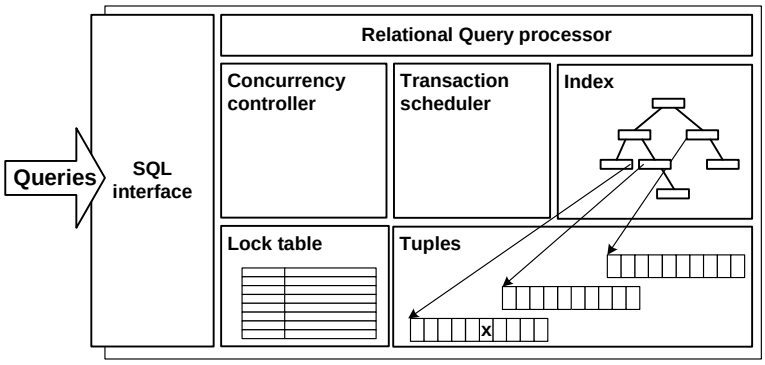

Figure 4.1: Architecture of a typical Database Management System.

memory. Usually, a tuple is a rather short array of bytes containing some data. For a relational model, one tuple contains the data for one row of a relation.

A decade later, it was proposed in [8] that tuple identifiers could be used directly from the application via the DBMS interface. This would enable applications to create shortcuts to hot data, in order to retrieve them faster. The concept is also implemented in the Adabas relational DBMS [9] under the name *Adabas Direct Access Method*. In Adabas, tuple identifiers are stored in a hash table and can be retrieved by the user for direct data access. A disadvantage of this concept is the inability to move or delete tuples at run-time. To be able to perform deletions or movements of tuples in Adabas, a reorganization utility must be run, during which the entire database is blocked.

Applications using tuple identifiers must be aware of the structure of the data stored in the tuples, e.g., offsets to specific attributes in the tuple. This makes it difficult to add or remove attributes from relations, since this changes the structure of the tuples.

4.2.3 Related Work

Apart from tuple identifiers, the concept of bypassing the index system to achieve faster data access has been recognized in other database systems. The RDM database [10] uses a concept called network access, which consist of a network of pointers. Network pointers shortcut data used in a predefined order. The implementation is, however, static and cannot be dynamically changed

during run-time.

In the Berkeley database [11], a concept called queue access is implemented, which allows enqueueing and dequeueing of data elements without accessing the index manager. The approach is primarily suited for data production and consumption, e.g., state machines.

The Pervasive.SQL database [12], uses the interface Btrieve to efficiently access data. Btrieve supports both physical and logical accesses of tuples. Logical accesses uses a tuple key to search for a tuple using an index, while physical access retrieves tuples based on their fixed physical locations. One database file contains tuples of the same length in an array. Btrieve provides a number of operations that allows stepping between the tuples, e.g., `stepNext` or `stepLast`. The Btrieve access method is efficient for applications in which the order of accesses is predefined and the tuples are never moved during runtime. Furthermore, restructuring the data within the tuples is not possible.

Some database management systems use the concept of database cursors as a part of their embedded SQL interface [13]. Despite the syntactical similarities between database pointers and database cursors they represent fundamentally different concepts. While database cursors are used to access data elements from within query results, i.e., result-sets, database pointers are used to bypass the index system in order to make data accesses more efficient and deterministic.

4.3 Database Pointers

The concept of database pointers consists of four different components:

- The `DBPointer` data type, which is the actual pointer defined in the application.
- The database pointer table, which contains all information needed by the pointers.
- The database pointer interface, which provides a number of operations on the database pointer.
- The database pointer flag, which is used to ensure consistency in the database.

Using the concept of database pointers, the architecture of the DBMS given in figure 4.1, is modified to include database pointer components, as shown in

figure 4.2. To illustrate the way database pointers work, and its benefits, we use the example presented in section 4.2.1, i.e., the request for retrieving the data x from the database.

Using the database pointer interface, the request could be made significantly faster and more predictable. First, a read operation together with the database pointer would be submitted to the database pointer interface. The database pointer, acting as an index to the database pointer table array would then be used to get the corresponding database pointer table entry. Each database pointer table entry consists of three fields: the physical address of data element x, information about the data type of x, and eventual locking information that shows which lock x belongs to. Next the lock would be obtained and x would be read. Finally, the lock would be released and the value of x would be returned to the calling application. The four components of the database pointer and its operations are described in detail in sections 4.3.1 to 4.3.4.

4.3.1 The `DBPointer` Data Type

The `DBPointer` data type is a pointer declared in the application task. When the pointer is initialized, it points to a database pointer table entry, which in its turn points to the actual data element. Hence the `DBPointer` could be viewed as a handle to a database pointer. However, due to the database pointer's syntactical similarities with a pointer variable, we have chosen to refer to it as a pointer.

4.3.2 The Database Pointer Table

The database pointer table contains all information needed for the database pointer, namely:

1. A pointer to the physical memory location of the data element inside the tuple. Typically, the information stored is the data block the tuple resides in, an offset to the tuple, and an offset to the data element within the tuple.

2. The data type of the data element pointed by the database pointer. This is necessary in order to ensure that any write to the data element matches its type, e.g., it is not feasible to write a floating point value to an integer.

3. Lock information describing the lock that corresponds to the tuple, i.e., if locking is done on relation granules, the name of the relation should be

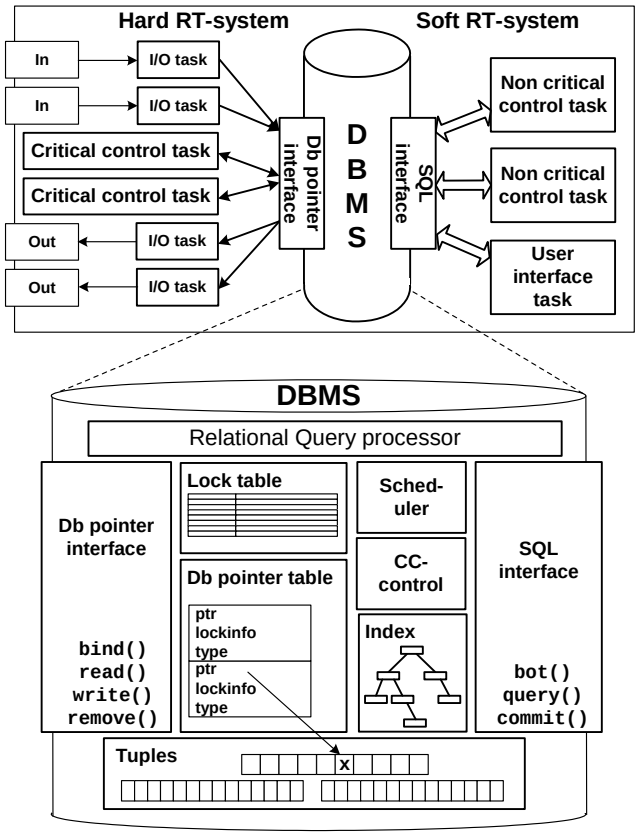

Figure 4.2: Architecture of a controlling system that uses a DBMS with database pointers.

stored in as lock information. Note, if locks are not used in the DBMS, i.e., if optimistic concurrency control is used, some other serialization information can be stored in the database pointer table entry instead of the lock information.

4.3.3 The Database Pointer Interface

The database pointer interface consists of four operations:

1. **`bind(ptr,q)`** This operation initializes the database pointer *ptr* by binding it to a database pointer table entry, which in turn points to the physical address of the data. The physical binding is done via the execution of the query *q*, which is written using a logical data manipulation language, e.g., SQL. The query should be formulated in such a way that it always returns the address of a single data element. By using the bind operation, the binding of the data element to the database pointer is done using a logical query, even though the result of the binding is physical, i.e., the physical address is bound to the database pointer entry. This implies that no knowledge of the internal physical structures of the database is required by the application programmer.

2. **`remove(ptr)`** This operation deletes a database pointer table entry.

3. **`read(ptr)`** This operation returns the value of the data element pointed by *ptr*. It uses locking if necessary.

4. **`write(ptr,v)`** This operation writes the value *v* to the data element pointed by *ptr*. It also uses locking if necessary. Furthermore, the type information in the database pointer entry is compared with the type of *v* so that a correct type is written.

The pseudo codes for the `write` and `read` operations are shown in figure 4.3. The `write` operation first checks that the types of the new value matches the type of the data element (line 2), and then obtains a write lock for the corresponding lock (line 4), i.e., locks the relation that the data element resides in. The data element is then updated (line 5), and finally the lock is released (line 6). The `read` operation obtains the corresponding read lock (line 10), reads the data element (line 11), releases the lock (line 12), and then returns the value to the application (line 13).

```
1  write(DBPointer dbp, Data value){
2     if(DataTypeOf(value) != dbp->type)
3        return DATA_TYPE_MISMATCH;
4     DbGetWriteLock(dbp->lockInfo);
5     *(dbp->ptr) = value;
6     DbReleaseLock(dbp->lockInfo);
7     return TRUE;
   }

8  read(DBPointer dbp){
9     Data value;
10    DbGetReadLock(dbp->lockInfo);
11    value = *(dbp->ptr);
12    DbReleaseLock(dbp->lockInfo);
13    return value;
   }
```

Figure 4.3: The pseudo codes for the `write` and `read` operations

4.3.4 The Database Pointer Flag

The database pointer flag solves the problem of inconsistencies between the index structure and the database pointer table, thus enabling tuples to be restructured and moved during run time.

For example, if an additional attribute is inserted into a relation, e.g., a column is added to a table, it would imply that all tuples belonging to the relation need to be restructured to contain the new data element (the new column). Hence, the size of the tuples changes, relocation of the tuples to new memory locations is most probable. Since a schema change is performed via the SQL interface, it will use and update the index in the index manager. If one of the affected tuples is also referenced from a database pointer entry, inconsistencies will occur, i.e., the database pointer entry will point to the old physical location of the tuple.

Each database pointer flag that is set in the index structure indicates that the tuple flagged is also referenced by a database pointer. This informs the index manager that if this tuple is altered, e.g., moved, deleted, or changed, the corresponding database table entry must be updated accordingly.

4.3.5 Application Example

To demonstrate how a real-time control system could use a RTDBMS with a database pointer interface, we provide an application example. Consider the system shown in figure 4.2 which is divided into two parts:

1. A hard real-time part that is performing time-critical controlling of the process. The tasks in this part use the database pointer interface.

2. A soft real-time part that handles user interaction and non-critical controlling. It uses the flexible SQL interface.

A hard real-time controlling task that reads a sensor connected to an I/O port is shown in figure 4.4. The task reads the current sensor value and updates the corresponding data element in the database. The task consists of two parts, an initialization part (line 2-4), which is run one time, and an infinite loop that is periodically polling the sensor and writing the value to the database (line 5-8).

The initialization of the database pointer is done by first declaring the database pointer (line 3) and then binding it to the data element containing the oil temperature in the engine (line 4). The actual binding is performed in the following four steps:

1. A new database pointer table entry is created.

2. The SQL query is executed and the address of the data element in the tuple is stored in the database pointer table entry.

3. The data type information is set to the appropriate type, e.g., `unsigned int`.

4. The locking information is set, e.g., if locking is done at relation granules, the locking information would be set to `engine`.

After performing these four steps, the database pointer is initialized and ready to be used. The control loop is entered after the initialization (line 5). In the control loop a new sensor value is collected (line 6), the value is then written to the RTDBMS using the database pointer operation `write` (line 7). Finally, the task sleeps until the next period arrives (line 8).

```
1 TASK OilTempReader(void){
2   int s;
3   DBPointer *ptr;
4   bind(&ptr, "SELECT temperature
               FROM engine WHERE
               subsystem=oil;");
5   while(1){
6     s=read_sensor();
7     write(ptr,s);
8     waitForNextPeriod();
    }
  }
```

engine

subsystem	temperature	pressure
hydraulics	42	27
oil	**103**	10
cooling water	82	3

Figure 4.4: An I/O task that uses a database pointer and its corresponding relation.

Criteria		TiD's	DbP's	Rel
Interface	Pointer based	x	x	
	Relational			x
Data access	Physical	x	x	
	Logical		x	x
Characteristics	Can handle tuple movements		x	x
	Can handle attribute changes		x	x

Table 4.1: A comparison between tuple identifiers, database pointers, and relational processing.

4.4 Concept Evaluation

In table 4.1 we compare the different access methods: tuple identifiers (TiD's), database pointers (DbP's), and relational processing (Rel). Both tuple identifiers and database pointers use a pointer based interface, which provides fast and predictable accesses to data inside a DBMS. However, it is not as flexible as most relational interfaces, e.g., SQL.

Furthermore, database pointer and tuple identifiers both access data based on direct physical references, in contrast to relational accesses that use logical indexing to locate data. However, database pointers bind the pointer to the data element using logical indexing, but access the data element using physical access.

Tuple identifiers have two drawbacks, firstly they are sensitive to schema changes, and secondly the physical structure of the database is propagated to the users. The former results in a system that can only add tuples instead of moving or deleting them, while the latter requires that the application programmer knows of the physical implementation of the database. Database pointers remove both of these drawbacks. Due to the flag in the index system, the database pointer table can be updated whenever the schema and/or index structure is changed, allowing attribute changes, tuple movements and deletions. Moreover, since the database pointer is bound directly to a data element inside the tuple instead of to the tuple itself, no internal structures are exposed.

The major advantage with accessing the data via pointers instead of going through the index system is the reduction of complexity. The complexity for the T-tree algorithm is $O(log_2 n + \frac{1}{2} log_2 \frac{n}{k})$, where n is the number of tuples in the system and k is the number of tuples per index node [14]. The complexity for database pointers and tuple identifier is $O(1)$. As can be seen, there is a constant execution time for accessing a data element using a database pointer or a tuple identifier, while a logarithmic relationship exists for the tree-based approach. There is however one additional cost for using the relational approach which we will illustrate with the following example.

We already showed how the oil temperature of an engine can be accessed using database pointers. Figure 4.5 shows the pseudo code for the same task, which now uses an SQL interface instead of the database pointer interface. In line 5, the `Begin of transaction` is issued and the actual update is performed in line 6, using a C-like syntax that resembles of the function `printf`. The actual `commit` is performed in line 7. In figure 4.5 all tuples in the relation `engine` have to be accessed to find all that fulfill the condition `subsystem = oil`. This requires accessing all three tuples.

```
1 TASK OilTempReader(void){
2   int s;
3   while(1){
4      s=read_sensor();
5      DB_BOT();
6      DB_Op("UPDATE engineSET temperature=%d
              WHERE subsystem = oil;",s);
7      DB_COMMIT();
8      waitForNextPeriod();
    }
  }
```

Figure 4.5: An example of a I/O task that uses a Relational approach.

It can, of course, be argued that precompiled transactions would be used in a case like this. Precompiled transactions are transactions that have been evaluated and optimized pre-run time. Such transactions can be directly called upon during run-time, and is normally executed much more efficient than an ad-hoc query. However, this does not influence the number of tuples accessed, since no information of the values inside the tuples are stored there. Therefore, all three tuples have to be fetched anyway.

4.5 Conclusions and Future Work

In this paper we have introduced the concept of database pointers to bypass the indexing system in a real-time database. The functionality of a database pointer can be compared to the functionality of an ordinary pointer. Database pointers can dynamically be set to point at a specific data element in a tuple, which can then be read and written without violating the database consistency. For concurrent, pre-emptive applications, the database pointer mechanisms ensure proper locking on the data element.

We have also showed an example of a real-time control application using a database that supports both database pointers and a SQL interface. In this example the hard real-time control system uses database pointers, while the soft real-time management system utilizes the more flexible SQL interface.

The complexity of a database operation using a database pointer compared to a SQL query is significantly reduced. Furthermore, the response time of a database pointer operation is more predictable.

Currently we are implementing database pointers as a part of the COMET

DBMS, our experimental database management system [15]. This implementation will be used to measure the performance improvement of database pointers for hard real-time controlling systems. Furthermore, different approaches for handling the interference between the hard real-time database pointer transactions and the soft real-time management transactions are investigated.

Bibliography

[1] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano - a Revolution in On-Board Communications. Technical report, Volvo Technology Report, 1998.

[2] K. Ramamritham. Real-Time Databases. *International Journal of distributed and Parallel Databases*, 1(2):199–226, 1993.

[3] T-W. Kuo, C-H. Wei, and K-Y. Lam. Real-Time Data Access Control on B-Tree Index Structures. In *Proceedings of the 15th International Conference on Data Engineering*, pages 458–467. IEEE Computer Society, March 1999.

[4] H. Lu, Y. Ng, and Z. Tian. T-Tree or B-Tree: Main Memory Database Index Structure Revisited. In *Proceedings of the 11th Australasian Database Conference*, pages 65–73. IEEE Computer Society, January 2000.

[5] W. Litwin. Linear Hashing: A New Tool for File and Table Addressing. In *Proceedings of the 6th International Conference on Very Large Databases*, pages 212–223. Springer, October 1980.

[6] M. M. Astrahan et al. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1(2):97–137, 1976.

[7] Dag Nyström, Aleksandra Tešanović, Christer Norström, Jörgen Hansson, and Nils-Erik Bånkestad. Data Management Issues in Vehicle Control Systems: a Case Study. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pages 249–256. IEEE Computer Society, June 2002.

[8] R. P. Van de Riet et al. High-Level Programming Features for Improving the Efficiency of a Relational Database System. *ACM Transactions on Database Systems*, 6(3):464–485, 1981.

[9] Software AG / SAG Systemhaus GmbH. Adabas Database . http://www.softwareag.com.

[10] Birdstep Technology ASA. http://www.birdstep.com.

[11] Sleepycat Software Inc. http://www.sleepycat.com.

[12] Pervasive Software Inc. http://www.pervasive.com.

[13] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley Longman Publishing Co., 2000.

[14] T. J. Lehman and M. J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the 12th Conference on Very Large Databases*, pages 294–303. Morgan Kaufmann, August 1986.

[15] Aleksandra Tešanović, Dag Nyström, Jörgen Hansson, and Christer Norström. Towards Aspectual Component-Based Development of Real-Time Systems. In *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications*, pages 278–298, February 2003.

Chapter 5

Paper D: The COMET BaseLine Database Management System

Dag Nyström, Aleksandra Tešanović, Christer Norström, and Jörgen Hansson

MRTC Report 98/2003, ISSN 1404-3041 ISRN MDH-MRTC-98/2003-1-SE, Mälardalen University.

Abstract

As complexity and the amount of data are growing in embedded real-time systems, the need for a uniform and efficient way to store data becomes increasingly important, i.e., database functionality is needed to provide support for storage and manipulation of data. However, a database that can be used in an embedded real-time system must be resource efficient with respect to memory and CPU resources and predictable in the temporal domain.

This report presents COMET, a component-based embedded real-time database management system, which is developed for embedded real-time systems, in particular vehicle control-systems. The key issues when designing COMET was to allow the possibility to tailor it for different types of application requirements and make it predictable in the temporal domain. The mean for achieving tailorability is by a flexible component model called RTCOM which supports the use of aspects.

We introduce COMET BaseLine, which is the first instance of COMET. This instance is tailored to suit a particular vehicle control-system developed at Volvo Construction Equipment Components AB, Eskilstuna, Sweden.

5.1 Introduction

The COMET DBMS (component-based embedded real-time database management system) is an experimental database platform. COMET is intended for resource-constrained embedded vehicle control-systems.

The complexity of modern vehicle control systems is rapidly increasing [1], as is the volume of data to be handled and maintained. A need for a structured way of handling this data has emerged. A real-time database management system would fulfil this need by providing a higher level of abstraction of the data management.

Different vehicular control-systems have different requirements, i.e., some might be static and non-preemptive while others might be much more flexible, allowing preemption. Many vehicular systems are distributed, but some are not. This places different requirements on the database. The natural approach would then be to develop an "in-house" DBMS that fulfil the requirements of the application. However, implementing a real-time database management system, and getting it certified as a component in a vehicular system is not a small task. A different approach would be to develop a tailorable database management system that can be configured to meet different kinds of requirements.

COMET is, too a high degree, tailorable to meet different kinds of requirements, such as different task-models, architectures, temporal constraints, and resource constraints. To achieve this, COMET is designed using both a component-based and an aspect-oriented approach.

Component-based software development enables systems to be built by assembling ready-made components. Each component is responsible for a well-defined task. A specialized database can be created by selecting a set of components that meets the requirements of the application, In COMET we have identified the following components, user-interface component, transaction management component, index management component, memory management component, lock management component, check pointing and recovery management component. The tasks of all of these components are described individually in section 5.5.

Aspect-oriented software development allows some concerns of the system to be separated into an aspect. We have identified a number of aspects in COMET. Some aspects, such as the concurrency control-aspect determine the behaviour of the system, while others, such as the temporal aspect, are used to describe non-functional properties of the system. Aspect-oriented software development is further described in section 5.2.

The current COMET concept consists of a set of components, a set of as-

pects, a DBMS architecture and a tool to calculate worst-case execution times. Future versions of COMET will contain, configuration tools, more components and aspects, as well as further analysis tools.

In this report we describe the underlying concepts of COMET. The architecture of COMET, the individual components and aspects identified in COMET, as well as our component-model, RTCOM, and our design model ACCORD are also described. Finally we describe the COMET BaseLine implementation, which is the first instance of COMET intended for a particular vehicle control-system developed at Volvo Construction Equipment Components AB [2].

5.2 Aspect-Oriented Software Development

Aspect-Oriented Software Development (AOSD) has emerged as a new principle for software development, and is based on the notion of separation of concerns [3]. Typically, AOSD implementation of a software system has the following constituents [3]:

- components, written in a language, which has support for aspect-oriented programming, e.g., C, C++, Java,
- aspects, written in the corresponding aspect language[1], e.g., AspectC [4], AspectC++ [5], AspectJ [6], and
- an aspect weaver, which is a special compiler that combines the components and aspects.

Components used for system composition in AOSD are not black box components (as they are in CBSE), rather they are *white box* components as we can modify their internal behavior by weaving different aspects in the code of a component.

Aspects are commonly considered to be a property of a system that affects its performance or semantics, and that crosscuts the system's functionality [3]. Aspects of software such as persistence and debugging can be described separately and exchanged independently of each other without disturbing the modular structure of the system. Each aspect declaration consists of advices and pointcuts.

[1] All existing aspect languages are conceptually very similar to AspectJ, developed for Java.

A *pointcut* in an aspect language consists of one or more join points, and is described by a pointcut expression. Pointcuts define points in the static structure or a dynamic control flow of the program. A *join point* refers to a point in the component code at which aspects should be weaved, e.g., a method, a type (struct or union).

Figure 5.1 shows the definition of a named pointcut `getLockCall`, which refers to all calls to the function `getLock()` and exposes a single integer argument of that call[2].

```
pointcut getLockCall(int lockId) =
   call(''void getLock(int)'') && args(lockId);
```

Figure 5.1: An example of the pointcut definition

```
advice getLockCall(lockId):
   void after (int lockId)
{
   cout << ''Lock requested is'' << lockId << endl;
}
```

Figure 5.2: An example of the advice definition

An *advice* is a declaration used to specify the code that should run when the join points, specified by a pointcut expression, are reached. Different kinds of advices can be declared, such as: (i) *before advice*, which is executed before the join point, (ii) *after advice*, which is executed immediately after the join point, and (iii) *around advice*, which is executed in place of the join point. Figure 5.2 shows an example of an after advice. With this advice each call to `getLock()` is followed by the execution of the advice code, i.e., printing of the lock id.

[2]The examples presented are written in AspectC++.

5.3 ACCORD: Aspectual Component-Based Real-Time System Development

The COMET database is developed using ACCORD (aspectual component-based real-time system development) [7], which is a method of designing aspectual component-based real-time systems.

In ACCORD the system is first decomposed into a set of components, and then further decomposed into a set of aspects. A component performs a specific task and is accessed using one or more well-defined interfaces. Aspects on the other hand are semantic functions or properties that crosscut the system, influencing several components.

ACCORD consider three different categories of aspects, namely, (i) Application aspects which change the functional behaviour of the application, (ii) run-time aspects which describe extra-functional properties of the compiled application, and (iii) composition aspects which describe which components and aspects can be combined. For further reading about these categories of aspects, we refer to [7].

Application aspects are aspects that change the functional behaviour of one or more components. This is done by writing aspect-code that is weaved into components using an aspect weaver. An example of a functional aspect is an earliest deadline first policy that is weaved into a scheduling component, and debug information that is weaved into a component during systems testing.

In sections 5.5, 5.7.1, and 5.7.2 we will further elaborate on aspects derived during the development of the COMET DBMS.

5.4 RTCOM: Real-Time Component-Model

The component-model RTCOM [8] (real-time componenent model) is used in COMET to support the different kinds of aspects used in ACCORD,.

The component consists of three parts, each corresponding to one type of aspect in ACCORD, the different parts can be seen in figure 5.3. The functional part, which is the actual code, corresponds to application aspects, the run-time part corresponds to run-time aspects and the compositional part corresponds to compositional aspects.

The functional part consists of two sub parts, the policy part and the mechanism part. The mechanism part consists of mechanisms that are fixed, while the policy part consists of operations that can be modified by application aspects. A component is accessed through a number of required and provided

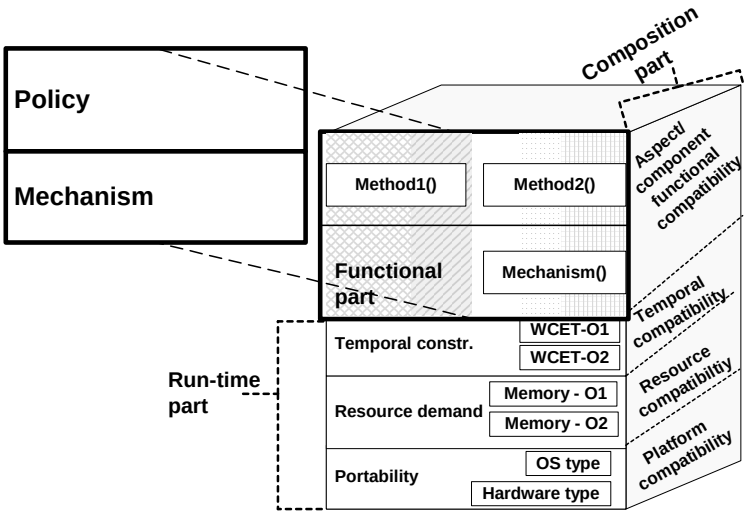

Figure 5.3: The real-time component-model

interfaces. Each interface consists of a subset of the component's operations.

For example, consider a component that implements a FIFO-list. Internally the list is implemented using a linked list. In the mechanism part, a number of mechanisms exist that are used to modify the linked list, i.e., `linkIn`, `linkOut`, `getFirst` and `getNext`. The policy part consists, however, of two operations, namely `enqueue` and `dequeue`.

The operations can call all the mechanisms in the component, as well as operations of other components. It is important to write the operations as structured and as modular as possible.

Further, consider that the aspect `priority` is weaved into the component. The aspect contains aspects code that is weaved into the operations in such a way that the list is sorted, based on a priority, i.e., the `enqueue` operation is modified to link the new element in its appropriate place in the list.

The run-time part contains information about different run-time aspects of the component, such as memory requirements, platform compatibility and worst-case execution time etc. Several of these aspects are dependent on the application aspects, such as the memory requirements and the worst-case execution time aspects. If an application aspect is weaved into the operations

of the components, it will affect the worst-case execution time and memory requirements of the component. To handle this, a component description language that supports adding application aspects to components is used [9]. For a formal definition of the RTCOM component-model, we refer to [10].

5.5 The Architecture of COMET

When we designed COMET, we used the ACCORD approach, and thus we started by identifying the different activities in a database management system. We identified the following activities:

1. User interface management
2. Transaction management
3. Scheduling management
4. Index management
5. Memory management
6. Lock management
7. Concurrency controlling
8. Checkpoint and recovery management
9. Logging management

One could argue that a database management system might consist of more activities, e.g., authorization management, query-optimizing management, and distribution management. We drew the conclusion, however, that the above-mentioned activities were the most central for vehicle control-systems, based on the case study by Nyström et al. [2]. However COMET could, without too much effort, be extended to include these additional activities also.

The second step was to extract components from these activities, and we found that activity 1 to 5 were candidate components, since these activities are well defined and isolated. Further, the use of these components when assembled would result in a functional database management system. The limitations to such a database would be that it could not handle concurrent transactions in addition to recovery.

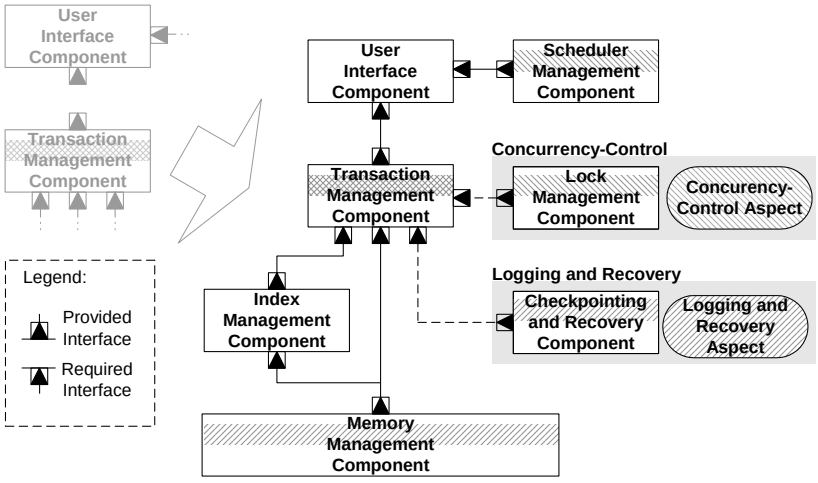

Figure 5.4: The Architecture of COMET

In the third step, concurrency controlling and logging/recovery management were identified as aspects, because both activities crosscut several components.

However, in both the case of concurrency control and in the case of logging/recovery, there are parts of these activities that can be viewed as isolated activities. In the concurrency control management, a lock-manager is an isolated activity that was considered best implemented as a component. In the case of the logging and recovery management, making periodic checkpoints and recover a restarted system are also considered isolated activities, and was therefore implemented as a component, e.g., the checkpointing and recovery component. These two components however, are optional, since they are not needed if the system does not use the corresponding aspect.

The resulting architecture of COMET, is shown in figure 5.4, which shows all the components and their interrelation in the architecture. The concurrency control, and logging/recovery aspects, as well as the components affected by each aspect are also shown. Since it is possible to have multiple transaction-types in a DBMS simultaneously, it is possible to have multiple types of user interface components and transaction management components in the architecture.

This architecture will provide COMET with a high degree of flexibility, and tailorability. All components in the system are exchangeable, so that different algorithms can be used, e.g., data indexing, transaction management, user interface management, memory management, etc. It is also possible to extend the functionality of the DBMS by applying an aspect, such as concurrency control, or logging and recovery. This means that an indvidual instance of COMET need only consist of the functionality needed for the particular application, thereby minimizing the DBMS footprint.

5.6 The Components in COMET

5.6.1 User Interface Component

The User Interface Component (UIC) provides an interface towards the user, or application. All operations to the DBMS are received by the UIC. Examples of requests are; (i) Begin of transaction, which indicates that a new transaction is initiated, (ii) Commit/Rollback, which ends a transaction, either by completing it or aborting it, (iii) A DML/DDL statement, which can be written in the supported language, e.g., structured query language, SQL.

The main activities of the UIC are to:

- receive and parse incoming requests from users and applications.
- maintain the list of all active transactions, i.e., all initiated transactions that are not yet committed or roll backed, and
- parse incoming DML/DDL statements, and convert these into execution plans.

The user interface component will perform syntactical checking (parsing) of the incoming DML/DDL statements. The UIC knows nothing about the structure of the database, and therefore cannot perform semantical checks. This approach has both advantages and disadvantages. One disadvantage is that the execution plans cannot be optimized. Unoptimized execution of queries might take significantly longer time than optimized. An advantage is that the UIC might be executed on an external computer, thus performing the parsing on the external CPU. Consider the following example:

The COMET DBMS is installed in an electronic control unit (ECU) that resides in a vehicle. A real-time control application is continuously updating data elements within the database. Consider a service computer connected to

the vehicle during run-time. If the UIC, and thus the parser, is located on the service computer, the ECU will not have to handle the parsing.

The interfaces of the user interface component are presented in table 5.1.

Interface	Type	Description
`IDbOperation`	Provided	Provides all operations reachable for the user
`IParse`	Provided	Provides the Parse Operations.
`ISchedule`	Required	Interface for scheduling new transactions.
`IExecute`	Required	Interface to execute transactions.

Table 5.1: The interfaces of the user interface component

5.6.2 Scheduling Management Component

The scheduling management Component (SMC) is responsible for scheduling the active transactions for execution, as well as maintaining a ready-queue containing all active unscheduled transactions. The SMC will schedule transactions based on the scheduling policy chosen. A scheduled transaction will be assigned to one instance of the transaction management component, e.g., one transaction manager from the transaction process pool, see section 5.6.3.

The interface of the scheduling management component is presented in table 5.2.

Interface	Type	Description
`ISchedule`	Required	Interface for scheduling new transactions.

Table 5.2: The interfaces of the scheduling management component

5.6.3 Transaction Management Component

The transaction management component (TMC) executes execution-plans received from the UIC. It ensures transaction serialization by using the lock management component; see section 5.6.5. Which concurrency and lock policy to use is defined by the applied concurrency-control aspect, described in more detail in section 5.7.1. The tuples needed for the transaction are then located and fetched, using the index management component and the memory management component, see sections 5.6.4 and 5.6.6.

The flow of events in the TMC is described most easy by using an example. Consider a TMC that executes relational transactions using a pessimistic concurrency control algorithm. For such a configuration, the flow of events could be as follows:

1. The execution-plan is received from the user interface component.
2. The execution of the transaction is begun.
3. At some point during the execution, a relation must be fetched from the database. This is done by:
   - Issuing a request that the appropriate locks should be obtained. This request is sent to the lock management component.
   - Requiring the address of the tuples by calling the index management component.
   - Fetching the tuples from the database by calling the memory management component.
4. When all tuples needed by the transaction have been fetched, the data elements in the tuples can be manipulated according to the specification in the execution-plan.
5. After the execution of the transaction, the user interface component is notified.
6. The TMC will now wait until a command to commit (or abort) the transaction is received from the user interface.
7. If a commit-request is received, any updated tuples are permanently written to the database.
8. The locks obtained for the transaction are released.
9. The user interface component is notified that the transaction is completed.

In order to process multiple transactions concurrently, several instances of the transaction manager can exist in the database. Each instance of the TMC is then assigned to a dedicated task, i.e., a transaction process. These processes make up the transaction process pool.

Interface	Type	Description
`IExecute`	Provided	Interface to execute transactions.
`IIndex`	Required	Interface to access the tuple index.
`ILock`	Required[3]	Interface to access database locks.
`IMemory`	Required	Interface to access the tuples and dynamic memory.
`ICheckpoint`	Required[4]	Interface to perform a checkpoint or recovery operation.

Table 5.3: The interfaces of the transaction management component

The interfaces of the transaction management component are presented in table 5.3.

5.6.4 Index Management Component

The index management component (IMC) is responsible for the indexing of tuples. It matches an alphanumerical key, i.e., a database key, with its corresponding tuple identifier (TID). A TID contains information about the location of a tuple in the memory.

The interfaces of the index management component are presented in table 5.4.

Interface	Type	Description
`IIndex`	Provided	Interface to access the tuple index.
`IMemory`	Required	Interface to access the database.

Table 5.4: The interfaces of the index management component

5.6.5 Lock Management Component

The lock management component (LMC) administers the database locks. Database locks are used to prevent data access conflicts, such as read write conflicts in which a transaction has read a data element and a second transaction assigns a new value to it immediately after. . At this point in time the two, still active, transactions have different views of the current value of the data element.

[3]This interface is required only if concurrency control is used.

[4]This interface is required only if logging/recovery is used.

How database locks are used is determined by the concurrency control algorithm applied to the system by the concurrency control aspect, described in section 5.7.1.

The interface of the lock management component is presented in table 5.5

Interface	Type	Description
`ILock`	Provided	Interface to access database locks

Table 5.5: The interface of the lock management component

5.6.6 Memory Management Component

The memory management component (MMC) administers the database, as well as the meta-data, such as database indexes and lock information.

The interface of the memory management component is presented in table 5.6.

Interface	Type	Description
`IMemory`	Provided	Interface to access the tuples and dynamic memory.

Table 5.6: The interface of the memory management component

5.6.7 Checkpointing and Recovery Component

The checkpointing and recovery component (CRC) is responsible for making checkpoints of the database and its meta-data. This component can be invoked periodically, or on request. It is responsible for restoring the system after a system failure.

The interface of the checkpointing and recovery component is presented in table 5.7.

Interface	Type	Description
`ICheckpoint`	Provided	Interface to perform a checkpoint or recovery operation..

Table 5.7: The interface of the memory management component

5.7 The Aspects in COMET

5.7.1 Concurrency-Control Aspect

The concurrency control aspect (CCA) ensures that concurrent transactions are serialized. This is enforced using (i) a serialization policy, such as optimistic or pessimistic concurrency control, (ii) the lock management component, (iii) the scheduling management component, and (iv) the transaction management component.

The CCA consists of the following:

- Aspect-code that is weaved into the transaction management component. This code is responsible for obtaining and releasing the appropriate locks, in accordance with the concurrency control policy used.
- Aspect-code that is weaved into the lock management component. This code instructs the component what to do if a lock conflict occurs.
- Aspect-code that is weaved into the scheduling management component. This code decides what scheduling policy to use.

5.7.2 Logging and Recovery Aspect

The logging and recovery aspect (LRA) is responsible for ensuring that volatile data can be safely stored and recovered in case of a system breakdown. The LRA uses (i) a logging and recovery policy, such as roll-forward recovery, (ii) the transaction management component, and (iii) the memory management component.

The LRA consists of the following:

- Aspect-code that is weaved into the memory management component. This code instructs the component to issue a log entry in a log, when a tuple is created, erased or modified.
- Aspect-code that is weaved into the transaction management component. This code will issue log entries whenever a transaction is started, committed, or aborted.
- Aspect-code that is weaved into the checkpointing and recovery component to perform the correct type of checkpointing and recovery.

5.8 COMET BaseLine

The COMET BaseLine is the first implementation instance of COMET DBMS. It implements a database management system suited to the control system described in [2].

The COMET BaseLine provides two types of database transactions:

- **Soft real-time transactions**. Non-critical tasks can use this transaction type, which provides a flexible and generic way to manipulate the database. The interface supports a subset of the SQL commands, such as `SELECT`, `INSERT` and `UPDATE`. A set of SQL statements can be grouped together into a transaction.

- **Hard real-time transactions**. Time-critical tasks can use this transaction type, which provides a deterministic and efficient way of manipulating individual data elements. The interface implements the concept of Database Pointers discussed in [11].

Since the Volvo control-system is non-preemptive, there is no need for concurrency control in COMET BaseLine. Therefore the concurrency control aspect is not implemented in COMET BaseLine. The transaction manager component is, however, prepared for the implementation of future aspects, e.g., the concurrency-control aspect, and the logging/recover aspect.

The logging/recovery management aspect has not yet been implemented in COMET BaseLine.

The current version of COMET BaseLine compiles down to an executable with the size of under 20kb.

5.8.1 Relational Processing

The relational transaction processing provides a generic and flexible way of accessing the database. It supports a subset of the commands in SQL; see table 5.8. These commands allow the user to retrieve any view of the data in the database, as well as add, delete and modify data elements. Finally the structure of the data can be changed, i.e., relations can be added and removed.

A set of queries can be bundled together to form a transaction. A transaction is considered an indivisible, e.g., atomic, operation, with a well-defined beginning and end. Table 5.9 shows the four API functions required to use a relational transaction in COMET BaseLine.

SQL command	Description
`SELECT ... FROM`	Selects a subset of the tuples and attributes from a number of relations.
`CREATE TABLE`	Creates a new relation.
`DROP TABLE`	Removes a relation.
`INSERT ... INTO`	Inserts a new tuple into a relation.
`UPDATE ... WHERE`	Updates tuples in a relation.
`DELETE ... WHERE`	Deletes tuples from a relation

Table 5.8: The SQL-commands supported in COMET BaseLine

Relational API functions	Description
`BeginTransaction()`	Initializes a new transaction returns a transaction ID.
`Query()`	Takes a SQL query, parses it and prepares it for execution.
`CommitTransaction()`	Executes the SQL queries in the transaction, updates the database and closes the transaction.
`RollbackTransaction()`	Aborts the transaction.

Table 5.9: The relational API functions supported by COMET BaseLine

DB pointer API functions	Description
`bind()`	Binds the database pointer to the query specified.
`remove()`	Deletes the database pointer from the database pointer table.
`write()`	Writes a value to the data element pointed out by the database pointer.
`read()`	Reads the value from the data element pointed out by the database pointer.

Table 5.10: The database pointer API functions supported by COMET BaseLine

5.8.2 Database Pointer Processing

The database pointer processing provides a way to manipulate individual data elements within the database, in an efficient and predictable way. The database pointer processing engine is implemented in accordance with the concept presented in [11].

The database pointer processor supports four interface operations, see table 5.10.

5.8.3 COMET BaseLine Component-Model

The component-model used in COMET BaseLine differs somewhat from the original RTCOM component-model described in [10]. The component-model used in COMET BaseLine is slightly relaxed since it allow mechanisms to call operations in other components. This relaxation affects how certain properties, such as worst case execution-time, are calculated. However, it would have been difficult to implement COMET BaseLine with this restriction.

Definition 1. *T is the set of **component-types**, where t is a tuple* $< O, S, M, P, R >$ *in which*

1. *O is a set of operations* $O = \{o_1, o_2, \ldots, o_n\}$
2. *S is a set of state variables* $S = \{s_1, s_2, \ldots, s_m\}$
3. *M is a set of mechanisms* $M = \{m_1, m_2, \ldots, m_p\}$
4. *P is a set of provided interfaces* $P = \{p_1, p_2, \ldots, p_q\}$

5. *R is a set of required interfaces* $R = \{r_1, r_2, \ldots, r_r\}$

Definition 2. *C is the set of* ***components****, where c is a tuple* $< S, P, R >$*, in which*

1. *S is a set of state variables* $S = \{s_1, s_2, \ldots, s_m\}$
2. *P is a set of provided interfaces* $P = \{p_1, p_2, \ldots, p_q\}$
3. *R is a set of required interfaces* $R = \{r_1, r_2, \ldots, r_r\}$

Definition 3. *A* ***component*** c *is an instance of a component-type* t*.*

$$c = \texttt{inst}(t)$$

where `inst` *is the instantiation function.*

Intuitively, we have defined component-types. A component-type, can be instantiated into a component, which is a run-time artifact. A component-type is partially a white-box component, since its operations can be modified by aspects, as we will se later. The instantiated component, however, is a black box artifact which can only be accessed through its interfaces. For the remainder of this section, we will use $t.O$, $t.S$, etc. to denote the sets O, S, etc. in the tuple t.

Definition 4. *An* ***operation*** o *in a component-type t is a tuple* $< M_o, O_o, S_o, g >$*, in which*

1. M_o *is a set of mechanisms* $M_o \subseteq t.M$*.*
2. O_o *is a set of operations* $O_o \subseteq t.O \cup O_{required}$*, where* $O_{required}$ *is a set of operations provided by other component-types.*
3. S_o *is a set of state variables* $S_o \subseteq t.S$*, that can be manipulated by* o*.*
4. g *is glue code used in the operation.*

Definition 5. *A* ***mechanism*** m *in a component-type* t *is a tuple* $< M_m, O_m, S_m, g >$*, in which*

1. M_m *is a set of mechanisms* $M_m \subseteq t.M$*.*
2. O_m *is a set of operations* $O_m \subseteq O_{required}$*, where* $O_{required}$ *is a set of operations provided by other component-types.*

3. S_m *is a set of state variables* $S_m \subseteq t.S$*, that can be manipulated by* m*.*

4. g *is glue code used in the operation.*

Definition 6. *A* ***provided interface*** p *in a component-type* t *is defined as* $p \subseteq t.O$*.*

Definition 7. *A* ***required interface*** r *in a component-type* t *is defined as a provided interface of a component-type* $t' \neq t$*, where* t' *is a component-type other than t,*

Intuitively, we have defined the constituents of a component-type, and its instantiation. The implementation of a component is divided into two parts, a mechanism part, which contains the basic building-blocks of the component, and an operation part, which contains operations that makes the top level functions of the component. These operations can be exported to a provided interface.

As we will see below, aspects can only be weaved into the operations of the component, and not into the mechanisms. Therefore, the mechanisms can be viewed as the fixed part of the component.

Definition 8. *An* ***application aspect*** a *is a tuple* $< T_{asp}, J, D, W >$*, in which*

1. $< T_{asp}$ *is a set of component-types* $T_{asp} = \{t_1, t_2, \ldots, t_s\}$ *which* a *crosscuts.*

2. J *is a set of join points* $J = \{j_1, j_2, \ldots, j_t\}$*, where each* j_x *is a location in a specific operation* o *in a component-type* $t \in T_{asp}$*.*

3. D *is a set of advices* $D = \{d_1, d_2, \ldots, d_u\}$*, where each* d_x *is a piece of code that can be inserted before, after or instead of the location pointed out by a join point* j*.*

4. W *is a set of code weavings* $W \in \mathcal{P}(J \times D)$[5]

Definition 9. *A set of component-types* $\mathcal{T}$ *weaved with a set of aspects* $\mathcal{A}$ *will result in a new set of component-types* $\mathcal{T}'$*, i.e., there is a function* weave*, such that*

$$\texttt{weave} : \mathcal{T} \times \mathcal{A} \rightarrow \mathcal{T}'$$

Intuitively, we have defined application aspects which can be weaved into the operations of components. As we can see, weaving an aspect into a component-type, result in a new component-type is created.

[5]The symbol $\mathcal{P}$ represents the powerset. The powerset of a set $\mathcal{A}$ is the set of all subsets of $\mathcal{A}$.

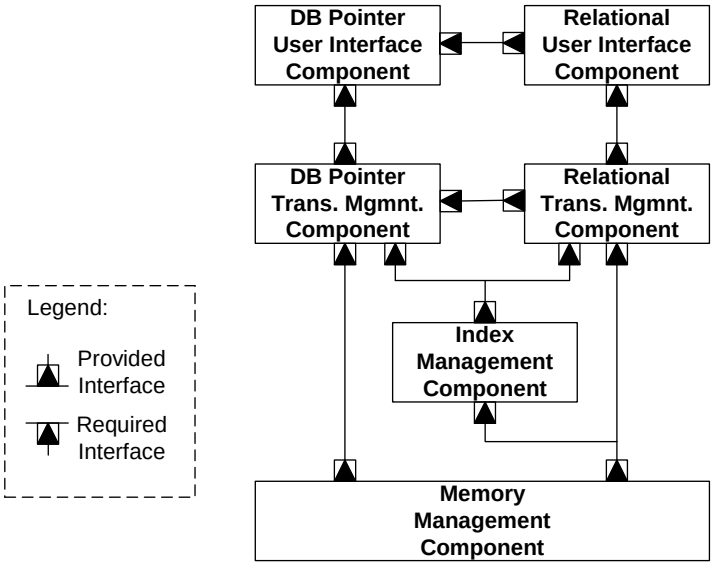

Figure 5.5: The architecture of COMET BaseLine

5.9 COMET BaseLine Architecture

5.9.1 Relational User Interface Component

The relational user interface component (RUIC) is responsible for administering the transactions as well as converting the SQL queries into execution-plans. The RUIC creates a tree that represents the execution-plan. An example of an execution-plan is shown in figure 5.6.

The execution-tree is stored until the `commitTransaction` operation is issued. At this stage, all execution-plans belonging to the committing transaction are sent to the relational transaction management component (RTMC) to be executed.

The nodes in the execution-tree can be of 5 different classes, namely:

1. **Relational Operations.** Originally C.F. Codd formulated 8 different relational operations, `select`, `project`, `join`, `product`, `union`, `intersect`, `difference`, and `divide`. In the subset of SQL supported in COMET BaseLine, only the `select`, `project`, `join`, and

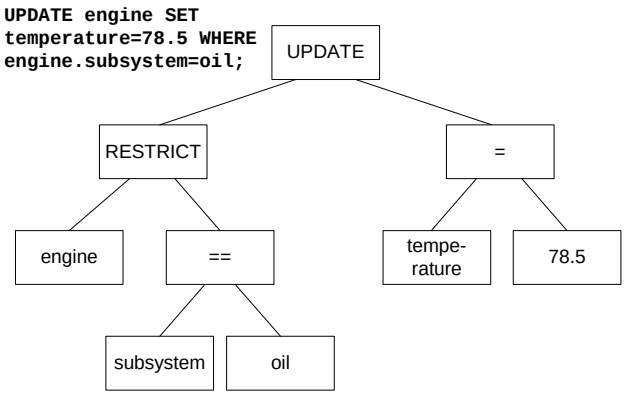

Figure 5.6: Example of execution-plan

`product` operations are necessary. The relational operations create new relations by modifying existing, and are used to create new views of the data.

2. **Data Manipulating Operations.** These operations, namely `delete`, `insert`, and `update`, modify tuples in a relation.

3. **Data Definition Operations.** These operations are used to modify the structure of the database, i.e., `create table` and `drop table`, in which the former creates a new relation, while the latter deletes a relation.

4. **Operators.** Typical operators are conditions and assignments.

5. **Operands.** Operand can be values, tuples, or relations.

5.9.2 Relational Transaction Management Component

The relational transaction management component (RTMC) executes the execution-plans created by the RUIC.

A central part of the RTMC is the buffer manager. The buffer manager is used to store intermediate, i.e., non-materialized, relations used during the execution of a query. The buffer manager, which handles the relational structure,

the relational meta-data and the actual data, is of such a considerable complexity that it is implemented as an internal component inside the RTMC.

An example, using the query and execution plan in figure 5.6, is given below to illustrate how the execution of a query is performed in COMET BaseLine.

1. The execution begins using a bottom-up approach.
2. The actual relation `engine` is loaded. This is performed in the following steps:
   - **a)** The meta-data for the relation is fetched from the index management component (IMC).
   - **b)** From the meta-data, the size of the relation, i.e., the length of the tuples and the number of tuples, is determined.
   - **c)** Using this information, a buffer is allocated.
   - **d)** The location of the first tuple is fetched using the `search()` operation in the IMC.
   - **e)** The tuple is loaded from the database using the memory management component (MMC), and is stored in the buffer.
   - **f)** The location of the next tuple is fetched using the `searchGT()` operation in the IMC
   - **g)** The steps e) and f) are repeated until the whole relation is loaded.
3. The relation is now restricted, i.e., all tuples not fulfilling the `WHERE subsystem=oil` are removed from the buffer.
4. The remaining tuples in the buffer are now updated according to the assignment `SET temperature=78.5`.
5. The last step is to write the updated tuples back to the database.

5.9.3 Database Pointer User Interface Component

The database pointer user interface component (DPUIC) provides the database pointer interface. The DPUIC uses the RUIC to create the execution-plan needed for the `bind` operation.

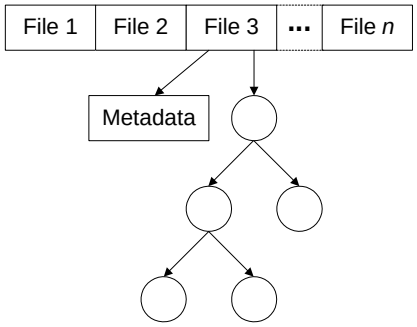

Figure 5.7: The architecture of the interface management component

5.9.4 Database Pointer Transaction Management Component

The database pointer transaction management component (DPTMC) executes the incoming database pointer operations. It administers the database pointer table, in which information about all the database pointers is stored.

The DPTMC uses the RTMC to execute the execution-plans received from the DPUIC. The difference, compared with relational query processing, is that the query does not return the value of the data element, but the address of the tuple in which the data element resides, and an offset.

Since the COMET BaseLine is a non-preemptive DBMS, the database pointer table does not include locking information. However it is implemented in such a way that this can easily be extended, or aspectualized.

The DPTMC can also receive requests from the RTMC to update the database pointer table. This is typically done, when tuples are moved or deleted from the relational query processor.

5.9.5 Index Management Component

The index management component (IMC) in COMET BaseLine implements the t-tree indexing algorithm. The information in the nodes in the tree has been extended to contain the database pointer flag.

The architecture of the IMC is shown in figure 5.7. The IMC consists of two layers of indexes, an upper layer, that indexes the different database files, and the lower layer that indexes the tuples in each database file. This

approach was suggested in [12]. The architecture allows the index manager to be configurable to a high degree.

For systems where the number of relations is fixed, or bounded, the upper layer index can be implemented as an array of files, while for a more flexible system a dynamic tree index might be more suitable. The lower layer index can implement any index structure, e.g., tree or hash index. Different index structures can be used simultaneously for different relations.

For the upper layer index, COMET BaseLine uses a static array with a pre-defined length. For the lower layer, the t-tree algorithm is used.

COMET BaseLine uses one file for each relation. Therefore a search for a tuple in a relation begins with the location of the relation in the upper layer index. When the relation is located, the lower layer index is used to locate the tuple.

5.9.6 Memory Management Component

The memory manager component (MMC) manages the stored tuples, and acts as a dynamic memory manager. It delivers a hardware independent memory management interface to the database, even for systems that do not support dynamic memory allocations.

The MMC provides two different types of dynamic memory:

- Tuple ID's, data structures that contain address information about one tuple in the database. The MMC provides four operations on tuple IDs, namely `allocate`, `delete`, `write`, and `read`.

- Dynamic memory used for buffering and temporary data structures used during the execution of a transaction. The MMC provides two operations for dynamic memory, `malloc` and `free`.

5.10 Conclusions and Future Work

In this report, we have described the architecture of the COMET database management system, which is configurable to suit different kinds of embedded control-systems. Furthermore, we have described the first instance of COMET, designated COMET BaseLine, a version of COMET suited to a vehicle control system developed at Volvo Construction Equipment Components in Eskilstuna, Sweden.

COMET has been developed using a design method called ACCORD [7], which combines component-based software development and aspect-oriented software development. COMET BaseLine is implemented using a component-model called RTCOM [8], which also is a part of ACCORD.

Since the application for which COMET BaseLine is developed is non-preemptive, there is no need for transaction synchronization and conflict resolution. This means that there is no need for the transaction scheduling component, the locking management, or the concurrency control aspect.

The next step is to evaluate the performance of COMET BaseLine, by integrating it into the Volvo Construction Equipment Component control-system. Implementation of the scheduling management component, the locking management component, the concurrency control aspect and backup/recovery aspect is also planned.

Bibliography

[1] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano - a Revolution in On-Board Communications. Technical report, Volvo Technology Report, 1998.

[2] Dag Nyström, Aleksandra Tešanović, Christer Norström, Jörgen Hansson, and Nils-Erik Bånkestad. Data Management Issues in Vehicle Control Systems: a Case Study. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pages 249–256. IEEE Computer Society, June 2002.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[4] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, 2002.

[5] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 2002. Australian Computer Society. AspectC++ can be downloaded from: http://www.aspectc.org.

[6] The AspectJ programming guide. Xerox Corporation, September 2002. Available at: http://aspectj.org/doc/dist/progguide/index.html.

[7] Aleksandra Tešanović, Dag Nyström, Jörgen Hansson, and Christer Norström. Towards Aspectual Component-Based Development of Real-Time Systems. In *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications*, pages 278–298, February 2003.

[8] Aleksandra Tešanović, Dag Nyström, Jörgen Hansson, and Christer Norström. COMET: a COMponent-based Embedded real-Time database. Technical report, Dept. of Computer Science, Linköping University, and Dept. of Computer Engineering, Mälardalen University, 2002.

[9] Aleksandra Tešanović, Dag Nyström, Jörgen Hansson, and Christer Norström. Integrating Symbolic Worst-Case Execution Time Analysis with Aspect-Oriented System Development. In *Proceedings of OOPSLA 2002 Workshop on Tools for Aspect-Oriented Software Development*, November 2002.

[10] Aleksandra Tešanović. Towards Aspectual Component-Based Real-Time System Development. Licentiate thesis, Department of Computer and Information Science, Linköping University, June 2003.

[11] Dag Nyström, Aleksandra Tešanović, Christer Norström, and Jörgen Hansson. Database Pointers: a Predictable Way of Manipulating Hot Data in Hard Real-Time Systems. In *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications*, pages 623–634, February 2003.

[12] Tony Bertilsson. Implementation Analysis for Databases in Embedded Systems. Master's thesis, Mälardalen University, 2002.