Data Management Issues in Vehicle Control Systems: a Case Study*

D. Nyström[†], A. Tešanović[⋆], C. Norström[†], J. Hansson[⋆], and N-E. Bånkestad[‡]

[†]Mälardalen University
Dept. of Computer Engineering
Västerås, Sweden
{dnm,cen}@mdh.se

[⋆]Linköping University
Dept. of Computer Science
Linköping, Sweden
{alete,jorha}@ida.liu.se

[‡]Volvo Construction
Equipment Components AB
Eskilstuna, Sweden
nils-erik.bankestad@volvo.com

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.

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 [5]. 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 [8] 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 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

*This work is supported by ARTES, a network for real-time and graduate education in Sweden.

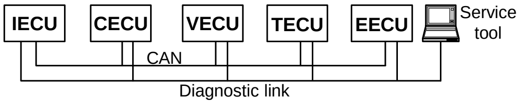

Figure 1. The overall architecture of the vehicle controlling system.

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.

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 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 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 TECU.

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 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).

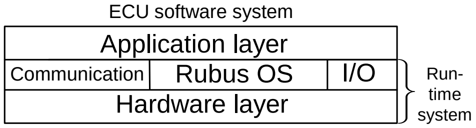

Figure 2. The structure of an ECU.

2.1 Rubus

Rubus is a real-time operating system designed to be used in systems with limited resources [1]. 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.

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

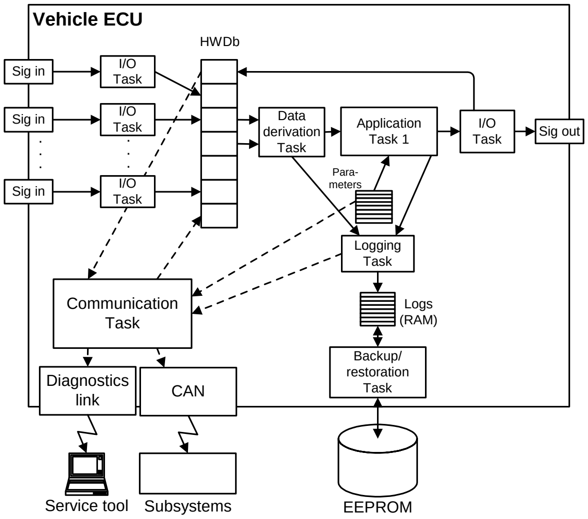

Figure 3. The original architecture of the VECU.

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 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. 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.

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

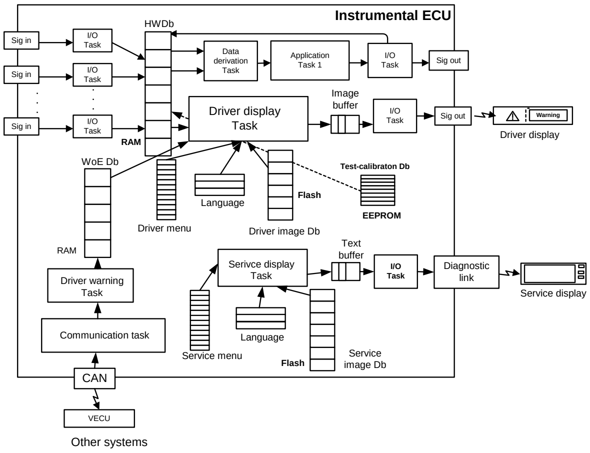

Figure 4. The original architecture of the IECU.

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 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 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 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.

2.4 Data Management Requirements

The table gives an overview of data management characteristics in the VECU and IECU systems. The following symbols are used in the table:

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 1: Data management characteristics for the systems.

As can be seen from the table 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 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

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

by the data derivation task, as explained in section 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 1).

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 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 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 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 2.2 and shown in figure 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 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 3.1. 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,

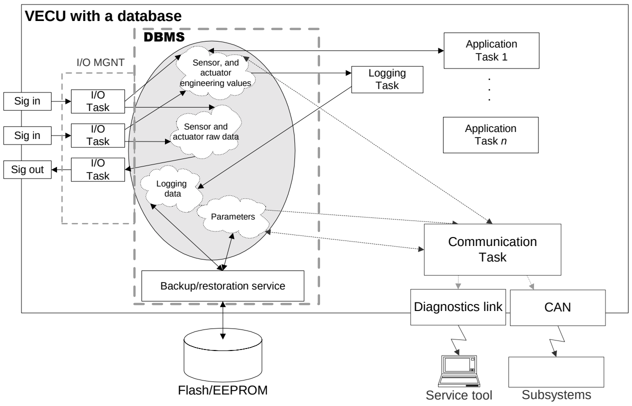

Figure 5. The new architecture of the VECU.

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 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 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 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.

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 [10]. 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 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 [13].

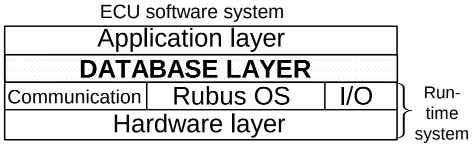

Figure 6. The structure of an ECU with an embedded database.

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 [7] and hashing algorithms [6].

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.

3.2 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 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 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 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).

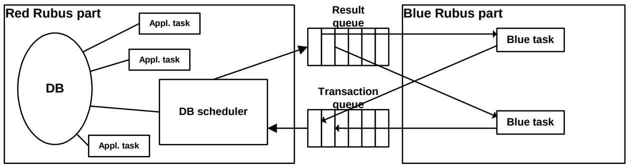

Figure 7. A database that supports non-periodic transactions via an external scheduler.

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 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 [3] 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 [2] a main memory database that would suit this application. The Beehive [11] system implements the concept of temporal validity, that would ensure that temporal consistency always ex-

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

ists 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 [4] 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 [9]. 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.

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 [12]. 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.

References

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

[2] B. Adelberg, B. Kao, and H. Garcia-Molina. Overview of the STanford Real-time Information Processor (STRIP). *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(1), 1996.

[3] 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*, Volume 25, 1996.

[4] 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, 1995.

[5] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano - a revolution in on-board communications. Technical report, Volvo Technology Report, 1998.

[6] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th International Conference on Very Large Databases*, 1980.

[7] H. Lu, Y. Ng, and Z. Tian. T-tree or b-tree: Main memory database index structure revisited. *11th Australasian Database Conference*, 2000.

[8] K. Ramamritham. Real-time databases. *International Journal of distributed and Parallel Databases*, pages 199–226, 1993.

[9] X. Song and J. Liu. Maintaining temporal consistency: Pessimistic vs. optimistic concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, October 1995.

[10] J. Stankovic, S. Son, and J. Hansson. Misconceptions about real-time databases. *IEEE Computer*, 32, June 1999.

[11] 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.

[12] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Embedded databases for embedded real-time systems: A component-based approach. Technical report, Dept. of Computer Science, Linköping University, and Dept. of Computer Engineering, Mälardalen University, 2002.

[13] 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.