Data Management in Real-Time Systems: a Case of On-Demand Updates in Vehicle Control Systems*

Thomas Gustafsson and Jörgen Hansson

Department of Computer Science, Linköping University, Sweden
E-mail: {thogu,jorha}@ida.liu.se

Abstract

Real-time and embedded applications normally have constraints both with respect to timeliness and freshness of data they use. At the same time it is important that the resources are utilized as efficient as possible, e.g., for CPU resources unnecessary calculations should be lowered as much as possible. This is especially true for vehicle control systems, which are our targeting application area. The contribution of this paper is a new algorithm (ODTB) for updating data items that can skip unnecessary updates allowing for better utilization of the CPU. Performance evaluations on an engine electronic control unit for automobiles show that a database system using the new updating algorithm reduces the number of recalculations to zero in steady states. We also evaluate the algorithm using a simulator and show that the ODTB performs better than well-established updating algorithms (up to 50% more committed transactions).

1. Introduction

In a vehicle (in this particular case a car), computing units are used to control several functional parts of the car. Every such unit is denoted an electronic control unit (ECU). The software in the units is becoming more complex due to increasing functionality that is possible because of additional available resources such as memory and computing power. This functionality is also needed because of stricter law regulations that are put on the car industry. Examples are lower pollution, detection of evaporation of gas through a hole in the gas hose, and detection of malfunctioning components within a limited time. Since such functionality requires additional data, the amount of data handled by the ECUs is constantly increasing. Moreover, the data has to be fresh when used to make correct calculations of control variables and accurate diagnosis of the system. Data freshness in an ECU is guaranteed by updating data items with fixed frequencies. Previous work [12, 16, 9] proposes ways of determining fixed updating frequencies on data items to fulfill freshness requirements. This means that a data item is recalculated when it is about to be stale, even though the new value of the data item is exactly the same as before. Hence, the recalculation is unnecessary. We collected statistical data from an engine ECU (EECU) that shows most of such periodic recalculations are unnecessary at steady states, i.e., when sensor values are not changing. To avoid doing unnecessary updates another updating mechanism than periodic updates is needed. Adelberg et al. [2] found that on-demand installation of updates of data items gives the best performance with respect to meeting deadlines and usage of fresh data. Data freshness is defined in the time domain by setting a maximum allowed age before which the data item is considered to be fresh [14].

On-demand updating algorithms — On-Demand Depth-First Traversal (ODDFT) [7], On-Demand Optimistic option (ODO) [4], and On-Demand Knowledge-Based option (ODKB) [4] — decide on necessary updates based on the order data items are read, i.e., bottom-up in a precedence graph of relationships of data items. Furthermore, the updating algorithms also assume that a recalculation of a data item always gives a new result. However, if calculations are deterministic, i.e., given the same inputs the same output is always produced, a more intelligent decision can be made of which updates are needed. Based on this we propose a new algorithm On-Demand Top-Bottom traversal with relevance check (ODTB) that is based on the on-demand strategy and is able to skip unnecessary calculations and, thus, utilize the CPU better. A freshness check that checks if an update is needed is added to ODDFT and ODKB resulting in the algorithms ODDFT_C and ODKB_C, where _C denotes that the freshness check is done in the value domain of data items.

In this paper, a description of a database system implementation in an EECU software is presented. Performance evaluations of ODTB, using the database system on

* This work was funded by ISIS (Information Systems for Industrial Control and Supervision) and CENIIT (Center for Industrial Information Technology) under contract 01.07.

Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)
1080-1812/04 $ 20.00 © 2004 IEEE

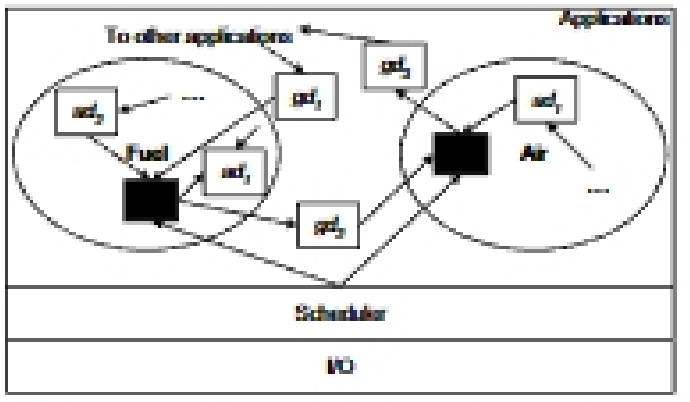


Figure 1. The software in the EECU.

an EECU, show that ODTB reduces the number of times periodic calculations need to be executed to zero when the system enters a steady state. Performance evaluations on a simulator show that ODTB, ODDFT_C, and ODKB_C give improved performance (up to 50% more committed transactions) than ODDFT and ODKB_V (_V is data freshness in value domain without freshness check).

The outline of the paper is as follows. An EECU software description, the data and transaction model, and a database implementation are given in section 2. Section 3 covers ODDFT and algorithms from [4]. The new algorithm ODTB and extensions to ODDFT are described in section 4. Performance evaluations, related work, and finally conclusions are given in sections 5, 6, and 7, respectively.

2. A Real-Time Embedded System

This section describes an EECU and the requirements of the software on the data and transaction model of the EECU. Here we also discuss the implementation of a database system in the EECU.

2.1. Electronic Engine Control Unit (EECU)

An EECU is used in vehicles to control the engine such that the air/fuel mixture is optimal for the catalyst, the engine is not knocking, and the fuel consumption is as low as possible. To achieve these goals the EECU consists of software that monitors the engine environment by reading sensors, e.g., air pressure sensor, lambda sensor in the catalyst, and engine temperature sensor. Control loops in the EECU software derive values that are sent to actuators, which are the means to control the engine. Examples of actuators are fuel injection times that determine the amount of fuel injected into a cylinder and ignition time that determines when the air/fuel mixture should be ignited. Moreover, the calculations have to be finished within a given time, i.e., they have deadlines.

The EECU software is layered, which is depicted in figure 1. Black boxes represent tasks, labeled boxes represent data items, and arrows indicate inter-task communication. The bottom layer consists of I/O functions such as reading raw sensor values and transforming raw sensor values to engineering quantities, and writing actuator values. On top of the I/O layer is a scheduler that schedules tasks both periodically and sporadically based on crank angles. The tasks are organized into applications that constitute the top layer. Each application is responsible for maintaining one particular part of the engine. Examples of applications are air, fuel, ignition and diagnosis of the system, e.g., check if sensors are working. Tasks communicate results by storing them either in an application-wide data area (*ad*, application data in figure 1) or in a global data area (*gd* in figure 1). The total number of data items in the EECU software is in the order of thousands.

Data items have freshness requirements and these are guaranteed by invoking the task that derives the data item often enough. This way of maintaining data results in unnecessary updates of data items, thus leading to worse performance for the overall system.

Based on the description of the EECU software above, and the experiences of our industrial partners (Mecel AB and SAAB Fiat-GM Powertrain), the following requirements on the software have been identified.

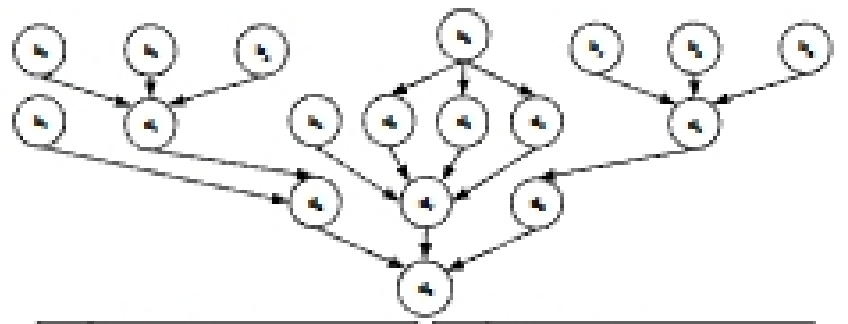
- R1 A way to maintain and organize data is needed because the storing of data using global and application-wide data areas makes it complex and expensive to maintain the software. Due to the vast amount of data items stored in different places, it is easy to introduce the same data item again, duplicating memory and CPU consumption.
- R2 Utilize available CPU resources efficiently in order to be able to choose as cheap as possible CPU and also extend the lifetime of the CPU.
- R3 Calculations have to be finished before a deadline and data items have freshness requirements.

A real-time database system divided into a central storage of data with meta-information and a data management system, making data items up-to-date when they are used, solves requirements R1–R3.

2.2. Data and Transaction Model

Figure 1 shows that a calculation in a task uses one or several data items to derive a new value of a data item. Hence, every date item is associated with a calculation that produces the value of the data item. The calculation is denoted a transaction, τ. Hence, a data item is associated with one value, the most recently stored, and a transaction that produces a value of the data item. The set of all data items in the EECU software can be classified as base items (B) and derived items (D). The base items are sensor values, e.g., engine temperature, and the derived data items are actuator values or intermediate values used by several calculations, e.g., a fuel compensation factor based on temperature. The relationship between data items can be described in a

Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)
1080-1812/04 $ 20.00 © 2004 IEEE



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

* Enr. = enrichment

Figure 2. Data dependency graph.

directed acyclic graph $G = (V, E)$, where nodes (V) are the data items, and an edge from node x to y shows that x is used by the transaction that derives values of data item y. In this paper we refer to G as the data dependency graph. Figure 2 shows the data dependency graph on a subset of data items in the EECU software. This graph is used throughout the paper. All data items read by a transaction to derive a data item d are denoted the read set $R(d)$ of d. The value of a data item x stored in the database at time t is given by v_x^t.

There are three types of transactions: (i) sensor transactions (ST) that only write monitored sensor values, (ii) user transactions (UT) that are requests of calculations from the system, and (iii) triggered updates (TU) that are generated by the database system to make data items up-to-date. A user transaction derives data item d_{UT}, a sensor transaction derives b_{ST}, and a triggered update derives d_{TU}.

2.3. Implementation of Database System

This section contains a description of our implementation of a database system in the EECU software. In the implementation of the database system, a real-time operating system, Rubus, is used as means for scheduling and communication between tasks. The database system including a concurrency control (CC) algorithm and the earliest deadline first (EDF) scheduling algorithm is implemented on top of Rubus. The concurrency control algorithm is Optimistic Concurrency Control Broadcast Commit (OCC-BC) [1]. The implementations of OCC-BC and EDF in the EECU software are presented in [6]. We are using the periodic tasks of the EECU software.

All the functionality of the original EECU software is kept. The database system is added to the EECU software and it runs in parallel to the tasks of the original EECU software. Hence, it is possible to compare the number of needed updates of data items between the existing EECU software and the added database system. All data items are stored in one data area and access to the data items is possible through a well-defined interface.

```
void TotalMulFac(s8 mode) {
  s8 transNr = TRANSACTION_START;
  while(BeginTransaction(&transNr,
        10000, 10, HIGH_PRIORITY_QUEUE,
        mode, TOTALMULFAC)) {
    ReadDB(&transNr, FAC12_5, &fac12_5);
    /* Do calculations */
    WriteDB(&transNr, TOTALMULFAC,
        local_fac, &TotalMulFac);
    CommitTransaction(&transNr);
}}
```

Figure 3. Example of a transaction.

An example of a transaction in this system is given in figure 3. BeginTransaction starts a transaction with a relative deadline of 10000 μs that derives data item TOTALMULFAC, d_9 in figure 2. Read and write operations are handled by ReadDB and WriteDB, and CommitTransaction notifies the database system that the transaction commits. The next invocation of BeginTransaction either breaks the loop due to a successful commit or a deadline miss, or restarts the transaction due to a lock-conflict. Detailed elaboration of the interface is presented in [6].

3. Existing On-Demand Updating Algorithms

This section describes existing on-demand algorithms. Section 3.1 covers previous work on on-demand algorithms using time domain for data freshness. Section 3.2 introduces data freshness in the value domain and section 3.3 gives a discussion of staleness of data items. Section 3.4 describes ODDFT, and how on-demand algorithms using time domain for data freshness can use value domain for data freshness.

3.1. Updating Algorithms and Data Freshness in Time Domain

An on-demand updating algorithm checks a triggering criterion every time a resource is requested (e.g., a data item). If the triggering criterion evaluates to true, a certain action is taken. When a read operation accesses a stale data item, an update updating the data item is triggered. Staleness can be decided in the time domain by using a maximum allowed age given by the absolute validity interval (avi) [14], i.e.,

$$current\_time - timestamp(x) \leq avi(x), \qquad (1)$$

where x is a data item, and $timestamp(x)$ is the time when x was last written to the database.