# 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.

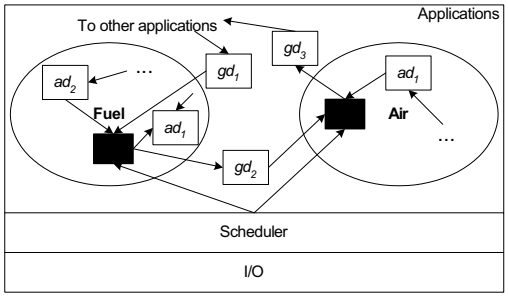



**Figure 1. The software in the EECU.**

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.

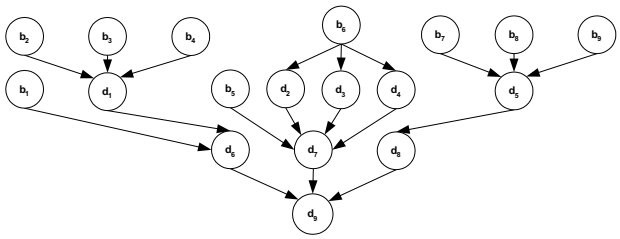
Based on the description of the EECU software above, and the experiences of our industrial partners (Mecel AB and SAAB Fiat-GM Powertrain), the following requirements on the software have been identified.

- 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, $\tau$. 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



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

* Enr. = enrichment

**Figure 2. 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 $\mu$s 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.

Every time a data item is requested condition 1 is checked. The on-demand algorithm using condition 1 is denoted OD. The triggering criterion can be changed to increase the throughput of UTs. In [4] three options of triggering criteria are presented. These are (i) *no option*, which represents OD, (ii) *optimistic option*, where an update is only triggered if it can fit in the slack time of the transaction that does the read operation (denoted ODO), and (iii) *knowledge-based option*, where an update is triggered if it can fit in the slack time when the remaining response time of the transaction has been accounted for (denoted ODKB).

### 3.2. Data Freshness in Value Domain

Data freshness of a data item $d$ can be defined in the value domain, i.e., the freshness depends upon how much data items in $R(d)$ have changed since $d$ was previously calculated and stored in the database [7].

We now define data freshness in the value domain of data items by using definitions 3.1–3.3. These definitions are used throughout the rest of the paper.

**Definition 3.1.** *Each pair $(d, x)$ where $d$ is a derived data item and $x$ is an item from $R(d)$ has a data validity bound, denoted $\delta_{d,x}$, stating how much the value of $x$ can change before the value of $d$ is affected.*

The freshness of a data item with respect to one of its read set members is defined as follows.

**Definition 3.2.** *Let $d$ be a derived data item and $x$ a data item from $R(d)$, and $v_x^t$, $v_x^{t'}$ be values of $x$ at times $t$ and $t'$ respectively. Then $d$ is fresh with respect to $x$ when $|v_x^t - v_x^{t'}| \leq \delta_{d,x}$.*

**Definition 3.3.** *Let $d$ be a derived data item derived at time $t$ using values of data items in $R(d)$. Then $d$ is fresh at time $t'$ if it is fresh with respect to all data items from $R(d)$, i.e.,*

$$\bigwedge_{\forall x \in R(d)} \left\{ |v_x^t - v_x^{t'}| \leq \delta_{d,x} \right\} \tag{2}$$

*evaluates to true.*

### 3.3. Example of Data Freshness in Value Domain

An update of a data item $d$ is only needed if the data item is stale, i.e., some of its parents have changed such that the new values of the parents result in a different value of data item $d$ compared to what is stored in the database. A data item can have several ancestors on a path to a base item in a data dependency graph $G$. For instance, one possible path denoted $P_{d_9-b_6}$ from $d_9$ to $b_6$ in figure 2 is: $d_9$, $d_7$, $d_2$, $b_6$. When a data item is updated it may make its neighbors in $G$ stale (this can be checked using equation 2). If the update makes a data item $d$ stale, then all descendants of $d$ are possibly stale since a recalculation of $d$ may result in a new value of $d$ that does not affect its descendants. Using the path $P_{d_9-b_6}$, consider an update of $b_6$ making $d_2$ stale. Data items $d_7$ and $d_9$ are possibly changed and a recalculation of $d_2$ is needed and when it has finished it is possible to determine if $d_7$ is stale.

```
ODDFT(d)
  for all x ∈ R(d) in prioritized order do
    if changed(d) ∧ error(x,t) > δ_{d,x} then
      Put τ_x in schedule
      ODDFT(x)
    end if
  end for
```

**Figure 4. Simplified version of ODDFT.**

### 3.4. Updating Algorithm Based on Data Freshness in Value Domain

This section describes the ODDFT algorithm and the updating scheme used with ODDFT [7].

When a UT starts to execute the data items it uses need to be made fresh before it continues. Only data items that can affect $d_{UT}$ need to be considered. A top-down traversal of $G$ cannot decide if a data item affects $d_{UT}$, thus, a bottom-up traversal is needed. Possibly changed data items need to be considered for being updated since it is not possible to conclude if a data item is fresh or stale in a bottom-up approach. The data items are investigated in a depth-first order (see figure 4) if an update is needed, and an update of a data item is put as late as possible in the schedule of updates. In this way, precedence constraints are obeyed. In order to know which data items that are possibly changed the following update scheme that consists of three steps is used.

In the first step (S1) all base items are updated with fixed frequencies such that the base items are always fresh. When a base item $b$ is updated, the freshness according to definition 3.2 is checked for each child of $b$ in $G$. Thus, in our example, base items $b_1$–$b_9$ from figure 2 are updated with fixed frequencies, e.g., base item $b_3$ is updated, then $d_1$ is checked if it is still fresh.

The second step (S2) is performed when a data item $d$ is found to be stale due to the new value of base item $b$. The child $d$ of $b$ is marked as changed and all derivatives of $d$ in $G$ are also marked as changed. A derivative of $d$ is actually possibly changed (see section 3.3). An error function was introduced in [7] to estimate the error of a data item at a given time $t$. This time is the deadline of the UT. The estimated error can together with the stored value and $changed$ determine if a particular data item makes one of its children stale at a given future time $t$.

The third step (S3) occurs every time a UT starts to execute. The freshness of a data item $d$ that can be directly

or indirectly read by the UT is deduced using the following equation:

$$\exists x \in R(d)(changed(d) = true \wedge error(x,t) > \delta_{d,x}) \quad (3)$$

Data item $d$ is stale if equation 3 evaluates to true. A schedule of updates is built with ODDFT during this step using the algorithm in figure 4. For an ODDFT scheduling example we refer to [7] due to space limitations.

Every update is tagged with the latest possible release time and deadline by accounting for WCETs of added updates in the schedule. When no more updates can be placed in the schedule the algorithm terminates and the database system starts triggering updates from the schedule. The updates are executed with the same priority as the UT. If the calculated release time of an update is earlier than the current time, then the update is not executed since it is considered not to finish within its deadline. When a transaction commits (user or triggered) the $changed$ flag of the installed data item is set to false and a freshness check is done for its children. If a child is stale then $changed$ flags are set as in S2. It is possible that duplicates of an update are put in the schedule. Such duplicates are checked for every time an update is put in the schedule, and the ones closest to $deadline(\tau_{d_{UT}})$ are removed.

The algorithms OD, ODO, and ODKB can use freshness in the value domain as defined in section 3.2. The updating scheme, steps S1–S3, is used and S3 occurs for every read operation in a UT or TU. The triggering criterion becomes equation 3 and the chosen option (no option, optimistic option or knowledge-based option). These versions of the algorithms are denoted OD_V, ODO_V, and ODKB_V where _V indicates that the value domain is used for data freshness.

# 4. On-Demand Updating Algorithms With Relevance Check

This section introduces a check before the triggering of an update that makes it possible for ODDFT, OD_V, ODO_V, and ODKB_V to skip updates that would not produce a value different from the value stored in the database. The new algorithm On-Demand Top-Bottom with relevance check, ODTB, is described and a scheduling example is given.

## 4.1. Enhancement to Existing Algorithms

Requirement R2 states that the CPU should be efficiently utilized, i.e., unnecessary updates should be avoided. All the updates that ODDFT puts in the schedule are executed (exceptions are described in section 3.4). The assumption of deterministic calculations is valid for an EECU since all data items are derived from sensor values and these are time-invariant. Some of the updates scheduled by ODDFT might produce the same result already stored in the database, i.e., the CPU could be better utilized by skipping such updates and using the value already stored in the database. When an update is about to start, the data freshness of the data item can be checked with equation 2. If the data item is fresh then the update is not needed. ODDFT and ODKB_V with such a check are denoted ODDFT_C and ODKB_C.

| Schedule | $d_5$ | $d_8$ | $d_2$ | $d_3$ | $d_4$ | $d_7$ | $d_1$ | $d_6$ | $d_9$ |
|---|---|---|---|---|---|---|---|---|---|
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

**Figure 5. Schedule $S$ for $G$ in figure 2.**

## 4.2. ODTB algorithm

The ODTB algorithm is a top-bottom traversal of a schedule generated by ODDFT. By traversing the data dependency graph top-bottom it is possible to decide if needed updates in a branch can fit in the schedule. Moreover, in ODTB the $changed$ flag indicates a stale data item and not a possibly changed data item.

In order to traverse updates top-bottom and decide which ones to use, the schedule has to be already generated or be generated every time updates need to be scheduled. The proposed ODTB algorithm uses a pregenerated schedule by ODDFT. The reason is that in an EECU the data items are fixed, i.e., no data items are added or removed, and, thus, the schedule is also fixed.

To obtain a pregenerated schedule that can be used by ODTB, we add a bottom node, denoted $bottom$, to $V$ and connect all leaf nodes to it by adding edges to $E$. Now we can generate an ODDFT schedule for the added node and denote it $S$. Branches are chosen by the following order $b_1 < b_i < d_1 < d_j$, where $i, j > 1$. If some data items are highly important then weights on the edges can be used.

**Theorem 4.1.** *It is always possible to find a sub-schedule of $S$ that is identical with respect to elements and order of the elements to a schedule $S_d$ generated by ODDFT, and which starts in node $d$.*

*Proof.* Assume the generation of $S$ by ODDFT has reached node $d$. Start a generation of a schedule at $d$ and denote it $S_d$. ODDFT only considers outgoing edges from a node. Assume two invocations of ODDFT, which origin from the same node, always pick branches in the same order. ODDFT has no memory of which nodes that have already been visited. Hence, the outgoing edge that is picked by ODDFT generating $S$ is the same as ODDFT generating $S_d$ and, thus, there exists a sub-schedule of $S$ that has the same elements and the same order as $S_d$. □

For data dependency graph in figure 2, the pregenerated schedule is shown in figure 5.

```
ODTB(d_UT)
  at = deadline(τ_UT) − release_time(τ_UT)
       −WCET(τ_{d_UT})
  for all x ∈ R(d_UT) do
    Get schedule for x, S_x, from S
    for all u ∈ S_x do
      if changed(u) = true then
        wcet_u_x =WCET of path from u to x
        if wcet_u_x ≤ at then
          Add data items u to x to schedule Updates
          at = at − wcet_u_x
        else
          Break
        end if
      end if
    end for
  end for
```

**Figure 6. ODTB algorithm.**

**Corollary 4.2.** *A schedule $S_d$ generated by ODDFT for data item $d$ with $l$ number of updates can be found in $S$ from index $start_d$ to index $stop_d$ where $l = |start_d - stop_d|$.*

*Proof.* Follows immediately from theorem 4.1. □

By corollary 4.2 it is always possible to fetch from $S$ a sub-schedule of all possibly needed updates for data item $d_{UT}$ that a UT derives. Every data item has start and stop indexes indicating where its ODDFT schedule starts and stops within $S$. Every data item also knows about its neighbors (parents and children) in $G$.

ODTB is shown in figure 6. Every time a UT, $\tau_{UT}$, is started the algorithm ODTB is executed. Updates are placed in the queue $Updates$. When ODTB has finished then the first update from $Updates$ is executed. The second step S2 of the updating scheme is changed to mark the children that are stale due to the newly stored value. The third step S3 occurs every time an update is started. If the data item it derives has $changed$ set to true then the update is executed, otherwise it is skipped. Hence, by using ODTB and the update scheme it is possible to skip unnecessary updates and instead read the value directly from the database. If $Updates$ is empty this means there are no updates to execute because all ancestors of $d_{UT}$ are fresh or the given time is too small to be able to execute the first necessary update and the descendants of this data item. In either case, there is no meaning in executing the user transaction.

Next we give an example of using ODTB. A UT $\tau_{d_7}$ arrives to a system having data dependency graph in figure 2. The fixed schedule $S$ is given in figure 5. Indexes for finding $S_{d_7}$ in $S$ are 2 and 5, i.e., schedule $S_{d_7}$ is the sub-schedule that spans indexes 2 through 5 in $S$. For every parent $x$ of $d_7$ ($d_2$, $d_3$, and $d_4$) the schedule $S_{d_x}$ is investigated from the top for a data item with $changed$ set to true (see figure 6). If such a data item is found, the WCET for the data item $u$ and the remaining data items in $S_x$, denoted $wcet\_u\_x$, has to fit in the available time $at$ of $\tau_{d_7}$. For each update, the cumulative execution time of all updates up to that point is stored. By taking the difference between two updates from $S_d$ the cumulative part of the update for $d$ is cancelled and the result is the execution time between the updates. When ODTB is finished the schedule $Updates$ contains updates that can be executed between the current time and the deadline of the UT.

## 5. Evaluation

Section 5.1 contains simulations conducted on a simulator to show the performance of OD, ODKB, ODKB_V, ODDFT, ODKB_C, ODDFT_C, and ODTB. The engine simulator and the EECU is used in section 5.2 to show that it is possible to use the database system in a real-life system. The section finishes with a discussion on some issues related to overhead for ODTB.

### 5.1. Consistency and Throughput

This experiment measures consistency and throughput of committed UTs. The number of committed UTs within their deadlines is affected by the number of updates generated by the system. If it is important to have transactions that only use fresh data items then the consistency is more important than transaction throughput. A balance between consistency and throughput must be found. This balance is primarily system specific and the constructor of the system has to choose the updating algorithm that gives the desired behavior. This section shows how the different algorithms behave with respect to these properties.

We now describe the simulator setup of the discrete event simulator RADEx++ that is also used in [7]. RADEx++ is set up to simulate a main-memory real-time database. A data dependency graph $G$ is generated randomly. The size of $G$ is $|B| \times |D|$ where $B$ is the set of base items and $D$ the set of derived items. A derived data item has one to six parents and the likelihood that a parent is a base item is $60\%$. All experiments using the same database size are using the same data dependency graph.

Transactions arrive aperiodically with exponentially distributed arrival times and every simulation is executed for 100 s of simulated time. The transactions are scheduled by EDF. The arrival rates range from 0 to 100 UTs per second with steps of 5 and the simulator runs 5 times for each arrival rate. Confidence intervals for experiments are given in figure captions. The data item an arriving UT derives is uniformly distributed among data items in $D$. The value of a data item $x$ is changed with a value from U(0,max_change), where max_change is 800, and this change only happens if $x$ is stale due to a change of any of

its parents. The absolute validity interval of a data item is taken from the distribution U(0,800) and the unit is ms. Periodic STs update base items based on their $avi$. STs have higher priority than UTs.

The execution time of an operation of a transaction is determined during run-time by randomly picking a value from the distribution N($\bar{e}$,2.5) until a value in the interval (0,10] is achieved for UTs and TUs; STs always have an execution time of 1 ms. The average execution time $\bar{e}$ of an operation is randomly determined at initialization of the database and is in the range (0,10] ms for UTs and TUs.

The random execution times of operations (read and write) models the EECU software well, since different branches in a calculation can give rise to different execution times. The randomly generated data dependency graph might not fully resemble a data dependency graph in an EECU software. Further, the uniform probability for a UT to derive any derived data item might not be correct for an EECU software, but we think the simulator gives an accurate enough simulation platform for results to also apply to an EECU. This is confirmed by results from two transient and steady state experiments. One in section 5.2 for an EECU, and one in [7] simulating using the same simulator and settings as in this work. Both results show a clear reduction of number of generated triggered updates at a steady state.

Figure 7(a) shows the total number of committed UTs within their deadlines for value domain based updating algorithms (ODKB_V, ODDFT, and ODTB), time domain based updating algorithms (OD and ODKB), and without updates. In this experiment, the algorithms OD, ODKB, ODKB_V, and ODDFT have no relevance check and, thus, try to execute as many of the updates as possible even though some of them might be unnecessary. The total number of transactions executed by the system is the lowest possible when no triggered updates are generated. At around 45 UTs per second the system becomes overloaded since the number of committed UTs stagnates when no updates are generated. ODDFT and OD are consistency-centric and generate more updates than the throughput-centric ODKB and ODKB_V. That can be seen in figure 7(a) where ODKB and ODKB_V have nearly as many committed UTs as when no updates are generated while ODDFT and OD let a smaller amount of UTs to commit. The load on the system can be decreased by using ODTB since it lets unnecessary updates and UTs to be skipped. The value stored in the database can be used without recalculating it. Thus, this enables resources to be reallocated to other tasks, e.g., the diagnosis application of an EECU. Figure 7(b) shows the number of committed UTs that are valid, i.e., the data item that the transaction writes has no updated ancestor at commit time that affects the value of the data item. ODDFT lets more valid UTs to commit up to 45 UTs per second compared to throughput-centric algorithms. OD has the worst performance under all arrival rates. ODTB excels the other updating algorithms in this respect as well. From 15 UTs per second ODTB lets the most valid UTs to commit and during overload (above 45 UTs per second) the difference is in the order of thousands committed UTs or more than a 50% increase in number of committed UTs. The consistency, i.e., the ratio of number of valid committed UTs and committed UTs (the ratio of values in figure 7(b) and figure 7(a)), is highest for ODDFT with a ratio of 0.85. Executing UTs without any updates give the worst consistency and for throughput-centric algorithms (ODKB and ODKB_V) the ratio is 0.60–0.70, whereas for ODTB the ratio is 0.70.

The results of comparing ODTB to ODDFT_C and ODKB_C are in figure 8. Both value domain based updating algorithms (ODDFT_C and ODKB_C) can now let more UTs commit at high load. This is because many of the updates can be skipped because executing them produces only the same result as the one already stored in the database, i.e., unnecessary updates are skipped. The total load on the system is thus decreased.

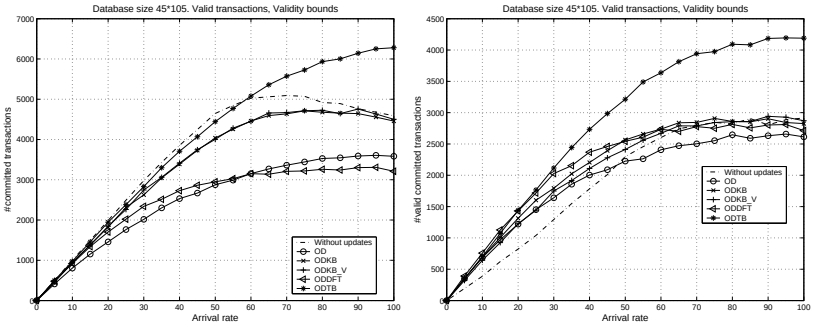
From figure 8 we see that ODKB_C lets slightly more UTs commit than ODTB, but more UTs are valid for ODTB. ODTB also has more valid committed UTs than ODDFT_C. This is possible because ODTB checks for updates top-bottom and can therefore distinguish needed updates from unnecessary updates.

### 5.2. Transient and Steady States in EECU

In many cases an embedded and real-time system is installed in a dynamically changing environment meaning that the system has to respond to these changes. Since tasks use data that should be fresh, state changes in the environment also affects the need to update data. This experiment treats steady and transient states and the number of required updates in each state. The number of updates is contrasted between an updating algorithm using value domain for data freshness and periodic updates, i.e., time domain for data freshness.

We evaluate the algorithms on the EECU using the implementation of a database system. In the performance evaluations, we use the engine simulator to adjust engine speed. The EECU reacts upon the sensor signals as if it controlled a real engine. The performance evaluation shows how the updating algorithms react on state changes (transient and steady states).

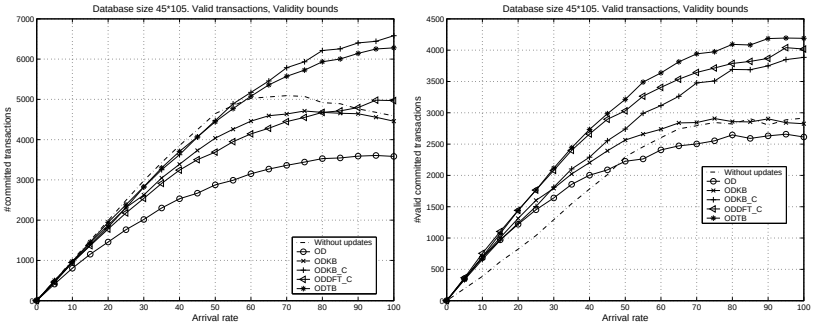
The derived data item TOTALMULFAC is requested periodically by a task in the EECU software. The request is transformed into a UT that arrives to the database system (see figure 3). All calculations on data items in the database produce deterministic results, i.e., the same result is always produced given the same input. Moreover, small changes in values of data items do not affect the result on derived data




(a) Number of committed UTs. 95% confidence interval is $\pm 321.5$

(b) Number of valid committed UTs. 95% confidence interval is $\pm 231.9$

**Figure 7. Consistency and throughput of UTs with no relevancy control on ODDFT and ODKB_V.**




(a) Number of committed UTs. 95% confidence interval is $\pm 321.5$

(b) Number of valid committed UTs. 95% confidence interval is $\pm 254.1$

**Figure 8. Consistency and throughput of UTs with a relevancy control on ODDFT and ODKB_V.**

items. Hence, one of the updating algorithms using value domain for data freshness should be feasible to use in this setting. ODTB is used in the EECU software, i.e., relevance checks are done if calculations are needed.

Recalculations of TOTALMULFAC are needed when the engine speed changes. Figure 9 shows how the requests for calculations are serviced only when the system is in a transient state, i.e., when the engine speed is changing. The plots in the bottom graph are cumulative number of requests. The number of requests is increasing linearly since the requests are periodic (remember that all time-based tasks are executed with fixed periods) and in the original EECU software each such request is processed. However, when using ODTB only some of the requests need to be processed. The number of serviced requests shows how many of the requests need to be processed. In steady states, none of the requests need to be processed, and the stored value in the database can be used immediately (e.g., the steady state in the time interval 2–7). Hence, during a steady state a considerable amount of requests can be skipped. Notice also that the data validity bounds allow the database system to accept a stored value if changes to the engine speed are small (in this case $\pm 50$ rpm). This can be seen in the time interval 17-22, where the small changes in engine speed do not result in recalculations of the TOTALMULFAC variable. The number of serviced requests does not increase in this interval.

### 5.3. Overhead in EECU

The memory overhead for storing the pregenerated schedule is low. For instance, the schedule in figure 5 consists of 9 elements and a schedule for 150 data items (the graph used in section 5.1) consists of 246 elements. Including execution times for each data item in the sched-

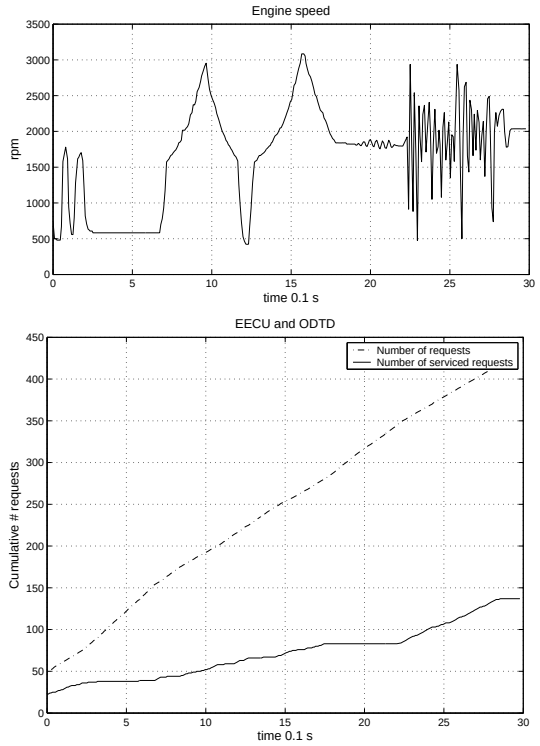



**Figure 9. Number of requests of calculation of fuel compensation factor in EECU.**

ule, 18 and 492 bytes of memory is needed respectively to store the schedules. However, in the worst case, the schedule takes exponential amount of memory since a graph can be constructed that contains a node that ODDFT visits an exponential amount of times, but for practical examples the size is linear in the size of the graph. In the database implementation, 500 bytes of flash memory is used to represent the schedule and data relationships, and 1.5 Kb of RAM for data items, meta-information, queues for EDF and CC, and statistics. The code size is 10 Kb and 8 Kb for the database system and Rubus respectively. Memory pools for database functions, mutexes, stacks for the periodic tasks take 19 Kb of RAM.

The algorithm ODTB traverses a pregenerated schedule top-bottom and if a stale data item is found the remaining part of the schedule is put in a schedule of updates. Some of these items might be fresh and unrelated to the found stale data item, i.e., they are unnecessary updates. Duplicates of a data item can be placed in the schedule. Checks for detecting these two issues can be added to the algorithm but this is not done in this work since the overhead should be kept as low as possible in an EECU.

ODTB takes the longest time to execute when none of the data items in the schedule is stale. One way to address this is to have two $changed$ flags, one that indicates a stale data item and one that indicates that none of the parents are changed. These two markings are a combination of the second step of the updating scheme from section 3.4 and 4.2. Hence, more time is spent marking data items when they change, but when data items do not change a fresh data item can immediately be detected.

The performance evaluation on the EECU also indicates that the extra processing time due to the added functionality imposed on the system is acceptably low, as the operational envelope of the system is equal to that of the original EECU software. This is confirmed as the EECU with a database is able to process transactions, meeting their deadlines, during transient states at the required peak load of the system. During overloads, i.e., beyond required peak load, the performance of the system decays similarly to the original EECU.

## 6. Related Work

In order to utilize the CPU resource as efficient as possible unnecessary updates must be avoided. Fixed updating schedules as in [12, 16, 9] cannot achieve this. Hamdaoui and Ramanathan introduced $(m, k)$-firm deadlines in [8], where $m$ deadlines out of $k$ consecutive invocations of a task have to be met. A task invocation can be skipped during an overload to increase the possibility for tasks to meet $m$ out of $k$ invocations. However, tasks that update data cannot be skipped as is possible with ODDFT_C, ODKB_C, and ODTB when data values are unchanged.

Another technique for skipping calculations is imprecise computations [13], where computations are split into a mandatory part and an optional part. The optional part can be skipped and the task produces an approximate result. In the case of overload, optional parts can be skipped and freed CPU resources are used to complete as many mandatory tasks as possible. ODKB_C, ODDFT_C, and ODTB skip calculations based on the staleness of a data value. No approximate results are produced, and calculations can be skipped at steady states.

A freshness concept is introduced in [11], where a similarity relation is used to define freshness in the value domain. The similarity relation states that a read operation can use a stored value as long as a concurrent write of the data item writes a similar value. A time interval, similarity bound, is introduced where all accesses to a data item within this similarity bound are similar. Hence, the data freshness is in practice defined in the time domain. Wedde et al. uses validity bounds in [15]. Base items are updated continuously and a data manager marks transactions based on validity bounds in a fixed schedule. Marked transactions are executed. However, a decision whether a UT can be serviced within its deadline cannot be made as in this work.

Kao et al. define data freshness in the time domain for discrete data items [10]. Absolute and relative systems are

introduced, where, in an absolute system, all data items used by a transaction need to be fresh when the transaction commits, whereas in a relative system, the data items need to be relatively consistent (including an allowance for slightly outdated data items) at the start of the transaction. Relative systems perform better than absolute systems. However, updates cannot be skipped.

Adelberg et al. [3] found that it is possible to delay recomputations and thereby allow more updates of data items to arrive before the recomputation is started. The data validity bounds work like a delay since several small changes do not trigger updates of data items. When the change is large enough an update is triggered.

Blakely et al. show how it is possible to decide which updates to data objects affect views, i.e., derived data objects, and which updates do not [5]. They assume that a data object is a relation, i.e., contains several columns of data. In this paper, however, a data item is a scalar value and the freshness of a data item can easily be tested with an inequality. By doing a top-bottom traversal of a graph it is possible to determine stale data items.

## 7. Conclusions and Future Work

In this paper we have studied different updating algorithms that maintain data freshness such that transactions read fresh data when deriving new values on data items. There are two families of updating algorithms, those that use the time domain and those that use the value domain on data items to measure data freshness. We show that the algorithms that use the value domain can adapt the required number of updates, when the system changes state, without introducing state-changes in the database system. Moreover, by assuming deterministic calculations a new updating algorithm with a relevance check of updates (ODTB) has been defined that outperforms well-established updating algorithms. A simple check on the necessity of an update generated by well-established updating algorithms (ODDFT and ODKB_V) improves the performance of these algorithms (with up to 50%), but ODTB has the best performance overall. ODTB is tested on an engine electronic control unit (EECU). We found that under a steady state no updates of data items are needed whereas in the original implementation of the EECU software these data items are recalculated periodically resulting in many unnecessary recalculations during a steady state. The CPU resources can be reallocated to, for instance, more detailed diagnosis of the system.

For future work we plan to look into concurrency control issues in electronic control units for vehicles.

## Acknowledgments

The authors would like to thank Aleksandra Tešanović and Mehdi Amirijoo for valuable comments on the paper.

## References

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

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

[3] B. Adelberg, B. Kao, and H. Garcia-Molina. Database support for efficiently maintaining derived data. In *Proc. of Extending Database Technology '96*, pages 223–240, 1996.

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

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

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

[7] T. Gustafsson and J. Hansson. Dynamic on-demand updating of data in real-time database systems. In *Proc. of ACM SAC '04*. ACM, March 2004.

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

[9] S.-J. Ho, T.-W. Kuo, and A. K. Mok. Similarity-based load adjustment for real-time data-intensive applications. In *Proc. of RTSS '97*, pages 144–154. IEEE Computer Society Press, 1997.

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

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

[12] C.-G. Lee, Y.-K. Kim, S. Son, S. L. Min, and C. S. Kim. Efficiently supporting hard/soft deadline transactions in real-time database systems. In *Proc. of Third International Workshop on RTCSA'96.*, pages 74–80, 1996.

[13] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. *Proceedings of the IEEE*, 82(1):83–94, January 1994.

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

[15] H. F. Wedde, S. Böhm, and W. Freund. Adaptive concurrency control in distributed real-time systems. Technical report, University of Dortmund, Lehrstuhl Informatik 3, 2000.

[16] M. Xiong and K. Ramamritham. Deriving deadlines and periods for real-time update transactions. In *Proc. of the 20th IEEE RTSS'99*, pages 32–43, 1999.