Dynamic On-Demand Updating of Data in Real-Time Database Systems

Thomas Gustafsson
Department of Computer Science
Linköping University, Sweden
thogu@ida.liu.se

Jörgen Hansson
Department of Computer Science
Linköping University, Sweden
jorha@ida.liu.se

ABSTRACT

The amount of data handled by real-time and embedded applications is increasing. Also, applications normally have constraints with respect to freshness and timeliness of the data they use, i.e., results must be produced within a deadline using accurate data. This calls for data-centric approaches when designing embedded systems, where data and its meta-information (temporal correctness requirements etc) are stored centrally. The focus of this paper is on maintaining data freshness in soft real-time embedded systems and the target application is vehicular systems. The contributions of this paper are three-fold. We (i) define a specific notion of data freshness by adopting data similarity in the value-domain of data items using data validity bounds that express required accuracy of data, (ii) present a scheme for managing updates in response to changes in the data items; and (iii) present a new on-demand scheduling algorithm, On-Demand Depth-First Traversal denoted ODDFT, for enforcing data freshness by scheduling and executing update transactions. Performance experiments show that, by using our updating scheme and introduced notion of data freshness in the value-domain, computational work imposed by updates is reduced for both the new ODDFT and well-established on-demand algorithms. Moreover, ODDFT improves the consistency of produced results compared to well-established algorithms.

Keywords

Real-time database, triggered updates, derived data items, resource management, vehicular systems

1. INTRODUCTION

Real-time and embedded software increases in complexity due to the larger amount of available resources such as memory, CPU power, and bus bandwidth. In addition, the required number of data items that need to be handled by the software as well as the application programmer is immense in real-time applications. Data items may have requirements on freshness, i.e., such data items should be updated often enough to always have a reasonable fresh value. Furthermore, the results have to be finished in time, e.g., control loop calculations have to be sent to actuators in time. Hence, in such systems a real-time database is needed for the following reasons: (R1) to arrange the data items to ease the programmers' task, (R2) to maintain data items with freshness demands, and (R3) to support timely calculations of important data items that have deadlines.

Dynamically changing systems can enter different states, e.g., when a driver presses the gas pedal to accelerate (transient state) or when the driver drives at constant speed (steady state). In our approach, we adjust the frequency of recalculating data items (updates) due to state changes. Update frequencies are dynamically adjusted based on how rapidly sensor values change in the environment. Assigning adequate update frequencies has a positive effect on the overall available CPU (as opposed to approaches where sensor values are always updated when data is assumed to become obsolete at the end of a constant time period, representing the worst case). In resource-limited systems, like embedded systems, it becomes increasingly important that the algorithms maintaining data freshness consider adapting the update frequencies to reduce the imposed workload.

The focus of this work is on maintaining data items with freshness demands, particularly aimed at performance-critical systems such as control units in automotive vehicles. Previous work [12, 14, 17, 9] proposed fixed updating schemes of data items to maintain fresh data, but the dynamic adaptability to new states is not achieved. The best performance, when handling updates, is obtained when updates are generated on-demand [2, 11]. Furthermore, Kuo and Mok introduced a relation called similarity that reflects that small changes to data items do not affect the end result [13]. Our course of action when constructing adaptive algorithms for maintaining data freshness is to use the value-domain, as the similarity relation, to define freshness and use on-demand generation of updates. Similarity is already used by application programmers, although in an ad hoc way [13].

The contributions of this paper are three-fold: (i) a notion of data freshness in the value-domain of data items, (ii) a scheme for handling changes in data items allowing for adaptability to new states, and (iii) an on-demand scheduling algorithm of updates using the introduced data freshness notion. The performance evaluation shows that the proposed updating scheme and scheduling algorithm perform better than existing updating algorithms. Moreover,

SAC'04 March 14-17, 2004, Nicosia, Cyprus.

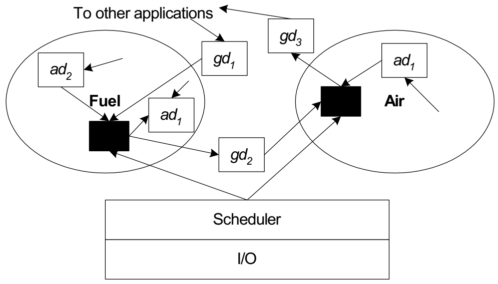


Figure 1: The EECU software is layered. Black boxes are tasks and arrows represent the way data is read and stored and the way sporadic tasks are invoked by the scheduler.

the scheme and the algorithm can also adapt the workload imposed by updates to the current state of the system.

The outline of the paper is as follows. An overview of an engine control unit that requires efficient data maintenance and enforcement of data freshness is presented in section 2. We give a detailed problem description together with our objectives in section 3. Section 4 introduces the freshness notion, the scheme for managing updates, the ODDFT algorithm, and well-established on-demand algorithms. Section 5 presents a performance evaluation. Related work is presented in section 6 and the paper is concluded in section 7.

2. OVERVIEW OF ENGINE CONTROL SOFTWARE

Here we give an overview of a vehicular electronic control system (ECU), which is our target application. A vehicle control system consists of several ECUs connected through a communication link normally based on CAN.[1] A typical example of an ECU is the engine electronic control unit (denoted EECU).[2] The memory of the EECU is limited to 64kb RAM, and 512kb Flash. The 32-bit CPU runs at 16.67MHz. The EECU is responsible for controlling the engine. For example the EECU controls the fuel injection and spark plug ignition in order to have an optimal air/fuel mixture for the catalyst, and to avoid knocking of the engine (knocking is a state where the air/fuel mixture is misfired and can destroy the engine). Data that the system reacts upon are collected from the engine environment, e.g., temperature and air pressure.

The system is divided into an I/O layer, a scheduler, and applications as depicted in figure 1. The I/O handling routines read sensor and write actuator values. The scheduler executes tasks periodically based on 5 msec time ticks and sporadically based on crank angles, i.e., start times of sporadic tasks depend on the speed of the engine. The tasks are executed on a best-effort basis implying that no hard real-time deadlines are associated with the tasks. The software is divided into applications, e.g., fuel, air, and diagnosis. The applications are further divided into tasks and they communicate internally via application-wide shared memory and externally, i.e., between tasks in different applications, via global shared memory. This is depicted as gd (global data) and ad (application data) in figure 1. The total number of data items maintained by the system is in the order of thousands. Data is derived from sensors and/or global and application-wide data, i.e., a relationship between data items exists. For derived data values to be operational, a derivation has to be based on fresh and accurate data. To guarantee freshness, data values are refreshed at fixed updating frequencies or right before they are used.

[1]We use the same notation as in [15], where each subsystem is called an electronic control unit (ECU).

[2]This research project is done in collaboration with Fiat-GM Powertrain.

3. PROBLEM DESCRIPTION

Current ad hoc solutions, such as application-specific data areas and global data areas that use traditional data structures, make software maintenance a very expensive and complicated task. Embedding database functionality in ECUs for maintaining data would overcome the identified problems (see R1-R3) [15]. Furthermore, it is important, from the perspective of memory and CPU consumption that intermediate results are stored only once and calculated only when necessary.

A data-centric approach, where data is stored in a repository aware of freshness and timeliness requirements, could be the foundation of efficient data maintenance in an application. In such a case information about data items and calculations is stored in one location and freshness and timeliness requirements are handled by the storage system. In effect all three requirements (R1-R3) for central storage, freshness, and timeliness, are fulfilled.

The objective of this work is to propose and evaluate algorithms for maintaining data freshness and timeliness of transactions in a soft real-time system consisting of base and derived data. Moreover, the objective is to utilize available computing resources efficiently such that unnecessary updates are avoided.

The characteristics of the database and the system are outlined next. A database consists of data items divided into two sets: base items and derived items, denoted B and D respectively. Base items reflect an external environment and can be of two types as follows.

1. Continuous items that change continuously in the external environment, e.g., a temperature sensor. The base items need to be fetched often enough to have a current view of the environment.

2. Discrete items that change at discrete points in time in the external environment, e.g., engine on or off.

A derived data item d is derived from base items and/or other derived items, i.e., each derived data item has a read set denoted $R(d)$ that can have members from both B and D. Hence, the value of a data item d is dependent on data items in $R(d)$. Additionally each data item x from $R(d)$ has a (possibly empty) read set $R(x)$.

In this paper, base items are continuous items, i.e., the sensor values of the EECU, and derived items are discrete items as they are derived sporadically when needed. Transactions in the system are computational tasks, e.g., the derivation of a data item such as fuel amount to inject in a cylinder, has a soft real-time deadline and consists of read and write operations of data items and some computations. Each transaction has a known worst-case execution time. Base items and derived items have freshness requirements that have to be fulfilled in all calculations where these data items are used.

4. ALGORITHMS

This section describes our approach for maintaining data freshness. We define data and transaction model in section 4.1. A data freshness notion based on the value-domain is presented in section 4.2. Our scheme for adaptive handling of updates of data items and a new scheduling algorithm of needed updates are presented in section 4.3 and section 4.4, respectively. We also review previous work on an on-demand algorithm with different triggering criteria (section 4.5) and extend those with two new triggering criteria (section 4.6).

4.1 Data and Transaction Model

The relationship between data items in the database is represented in the form of a directed acyclic graph (DAG) $G = (V, E)$, with nodes V and edges E, which we denote data dependency graph. A node in G represents one data item, and a directed edge represents an item in a read set. A data item d resides in a particular level based on the path from the base items.

Definition 4.1. *Let b be a base item and d a derived item. Then the level of b is 1 and the level of d is the longest path from a base item, i.e., $level(d) = \max_{\forall x \in R(d)}(level(x)) + 1$.*

A continuous data item is said to be absolutely consistent [16] with the entity it represents as long as the age of the data item is below a predefined limit.

Definition 4.2. *Let x be a data item (base or derived). Let $timestamp(x)$ be the time when x was created and $avi(x)$, the absolute validity interval, is the allowed age of x. Data item x is absolutely consistent when:*

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

Note that a discrete data item is absolutely consistent until it is updated. Since updates can be aperiodic, there is no absolute validity interval on a discrete data item.

Relative consistency on a set of data items states the requirements on the data items to let derivations produce fresh data items. In this paper we adopt the following view of relative consistency [11].

Definition 4.3. *Let x be a data item and $VI(x)$ be the interval when x is valid. If x is a discrete data item that is currently valid, then the end time in $VI(x)$ is set to ∞. A set of data items W is relatively consistent if $\bigcap\{VI(x_i) | x_i \in W\} \neq \emptyset$.*

Definition 4.3 implies that a result derived from W is fresh as long as all data items in W are fresh at the same time, i.e., when the intersection of their validity intervals is not empty.

A transaction in this paper has zero or more read operations and one write operation writing one data item, i.e., a transaction is represented by one node in G that is the data item it writes. Furthermore, transactions are divided into: (i) write-only sensor transactions (ST) updating base items to give a current view of the environment, (ii) user transactions (UT) generated by the application and consisting of several read operations and one write operation, and (iii) triggered transactions or triggered updates (TU) that are generated by the database system to update data items read by the UTs. The distinction between the latter two is done for evaluation purposes. We denote a user transaction as τ_{UT}, and the data item it derives as d_{UT}.

4.2 Data Validity Bounds

Definition 4.2 defines freshness as an age, i.e., during a certain interval of time all possible values of a data item can be summarized by the value that is already stored in the database. When a data item gets too old, its value can still be within a tacitly accepted bound. The required update is thus unnecessary. Another way to define freshness is to use the bound on the value of a data item defined as follows.

Definition 4.4. *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 $\delta_{d,x}$, that states how much the value of x can change before the value of d is affected.*

No recalculations of derived data items are needed as long as the values of used data items are within acceptable bounds given by definition 4.4 from the used values. Hence, the values are considered to be similar [13]. The freshness of a data item with respect to one of the read set members is defined as follows.

Definition 4.5. *Let d be a derived data item, x a data item from $R(d)$, and $v_x^{t_0}$, v_x^t be values of x at times t_0 and t, respectively. Then d is fresh with respect to x when $|v_x^{t_0} - v_x^t| \leq \delta_{d,x}$.*

Building on the definitions 4.4 and 4.5, the freshness of a data item can be defined as follows.

Definition 4.6. *Let d be a data item derived at time t_0 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.,*

$$\bigwedge_{\forall x \in R(d)} \{|v_x^{t_0} - v_x^t| \leq \delta_{d,x}\} \qquad (2)$$

evaluates to true.

4.3 Scheme of Base Item Updates and Approximation of Error

In this section we describe our scheme for updating base items and annotating derived data items that need to be updated. The scheme is divided into three steps. The first step considers base items while the second and third consider derived data items.

In the first step, all base items are updated at frequencies derived from absolute validity intervals (see definition 4.2). Note that all updates of derived data items originate from changes in the base items. When a base item b is updated its new value can affect the value of its children in G, i.e., those data items in level 2 having b in their read sets. Thus, when b is updated the freshness (see definition 4.6) is checked for each child of b.

The second step is done if the freshness check of a child d of base item b evaluates to false. In this step, data item d and all its descendants are marked as *possibly* changed by setting a flag *changed* to true. For example, let us assume that the first step applied on data illustrated in figure 2 discovered that an update on b_3 makes d_2 stale. Then in the second step d_2 together with its descendants d_4–d_6 are marked as possibly changed by setting their flag *changed* to true. Although derived data items d_4, d_5, and d_6 are also marked as changed we cannot determine if d_4 and d_5 are really stale until d_2 is recalculated using the new value of b_3 (for determining staleness of d_6 a derivation of d_4 is needed with the new value of d_2). Hence, a recently derived data item is likely to still be fresh even though it is marked as changed, e.g., data items d_4–d_6. A time-dependent function and a threshold can, together with *changed*, determine if a data item is really stale. The function we use is based on the change in the value of the data item at a time t given by $error(x, t)$. Hence, a data item d is then stale when:

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

For a data item d_4 to be considered stale at time t_1 then *changed* has to be true, and the error of all parents need to be outside their data validity bounds, i.e.,

$$changed(d_4) \wedge error(d_2, t_1) > \delta_{d_4,d_2}.$$

In the third step, using equation (3) we deduce whether a data item is stale every time it is accessed.

When a derived data item d is updated the data freshness is checked for all its children, and if necessary, *changed* on a child and its descendants is set to true as described above for base items.

4.4 Data Validity Bound Aware Algorithms for Scheduling of Updates

We now introduce a new algorithm On-Demand Depth-First Traversal, denoted ODDFT, which schedules triggered updates based on data validity bounds. The main idea is to update a data item only when it is necessary to do so. Hence, the algorithm is able to adapt the updating frequency to the current changes in the external environment.

We use the scheme presented in section 4.3 to keep base items fresh and sets *changed* of derived data items to true when a base or derived item changes. Moreover, a user transaction arrives in the system in one of two queues: high or low; high for important transactions, such as spark ignition, and low for all other transactions. The UT creates a derived data item that is sent to an actuator. On the arrival of a UT a schedule of triggered updates is generated based on the data dependency graph, flag *changed*, and the approximated error $error(d, t)$.

In this paper we set t of $error(x, t)$ in equation (3) to the deadline of τ_{UT} to make sure that all data items are fresh during the execution of τ_{UT}. This time is called the $freshness\_deadline$. The relative consistency is according to definition 4.3, since all members of $R(d_{UT})$ have to be valid at the same time and until deadline of τ_{UT}. Thus, the intersection of the validity intervals spans the execution of τ_{UT} (see [7] for an extensive discussion).

Let us now describe the algorithmic steps of ODDFT used to schedule triggered updates. ODDFT is a recursive algorithm that works as follows.[3]

1. Construct a graph G' by reversing the directions of all edges in G.

[3]Presentation of pseudo-code and a timing analysis see [7].

2. Assign $deadline(\tau_{UT}) - releasetime(\tau_{UT})$ time units to the schedule.
3. Traverse G' in a depth-first manner as follows starting from $d_i = d_{UT}$.
   (a) Determine stale data items from the set $R(d_i)$ using equation (3). Prioritize[4] stale data items.
   (b) If there is time left in the schedule, then put the stale data item $d_j \in R(d_i)$ with highest priority in the schedule; otherwise stop the algorithm for this branch. The schedule is a last in first out (LIFO) queue and the latest possible release time of an update is stored in the schedule.
   (c) If there are no stale data items, then stop traversal of this branch.
   (d) Go to step 3 and use d_j as the reached node.

Triggered updates are picked from the schedule and execute with the same priority as τ_{UT}. If current time is larger than the latest possible release time of a triggered update in the schedule, then skip the update.

To simplify the scheduling algorithm there is no check to see if a triggered update for a data item makes the data item fresh until $freshness\_deadline$. Further, any duplicates of triggered updates are removed assuming the remaining triggered update makes the data item fresh until $freshness\_deadline$.

An example is given in figure 2. A UT deriving and using data item d_6 starts to execute. Assume all ancestors need to be updated. Triggered updates are added at the head of the schedule when a needed update is found. G' is traversed in a depth-first manner. Note that a triggered update for d_1 is put twice in the queue, because d_1 is a parent both for d_3 and d_4. During scheduling, duplicates are removed and the final schedule looks as the bottom schedule in figure 2.

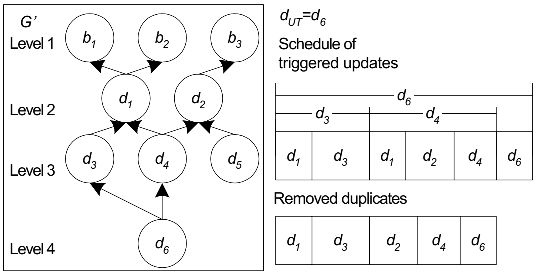


Figure 2: Example of an ODDFT schedule.

4.5 On-Demand Algorithms

Here we review an on-demand algorithm for triggered updates presented in [3]. Generally, an on-demand algorithm checks a triggering criterion when a particular resource is accessed and an action is taken if the criterion is fulfilled. For ODDFT the resource is the derivation of a data item and the triggering criterion is equation (3).

[4]Can be done by calculating a normalized error for each stale value by taking $error(d, freshness\_deadline)/v_d^{t_0}$. Highest normalized error has highest priority.

In [3] the basic triggering criterion is inequality (1) from definition 4.2, i.e., the age of a data item. A triggering criterion is checked in each read operation, and if it is evaluated to false a triggering update is generated that updates the data item before τ_{UT} continues to execute.

Ahmed and Vbrsky introduce three different triggering criteria (referred to as options in their paper) of how the triggering criterion could work. These focus on either being consistency-centric or throughput-centric. They are [3]:

- *No option.* This is the same as the basic triggering criterion. This option is consistency-centric as a triggered update is always generated when a stale data item is encountered.
- *Optimistic option.* This checks to decide if the conceivable triggering update can fit in the available slack time. This option is throughput-centric as triggered updates are stopped if they cannot fit in the available time, and the user transaction is executed even though stale values are read. The user transaction is more important than the updates.
- *Knowledge-based option.* The check now also includes remaining response time of the user transaction, and the triggering update has to fit in the time the remaining response time has been accounted for. This check is more accurate than the one in the optimistic option and, thus, user transactions have the chance of being committed before the deadline.

The three proposed triggering algorithms are abbreviated as shown in the column "Age at current time" in table 1.

4.6 Extension of On-Demand Algorithms

The triggering criteria of the on-demand algorithm (see 4.5) triggers updates based on current age and a validity interval. With freshness based on data validity bounds, it is possible to trigger updates based on value instead. Furthermore, the triggering criterion can base the age of a data item, not at the current time, but at the deadline of τ_{UT} as ODDFT does. Thus, we have two new types of triggering criteria: (i) the data validity bound criterion triggers an update of d if equation (3) is true, and (ii) the age criterion triggers an update of d if $deadline(\tau_{UT}) - timestamp(d) > avi(d)$. The on-demand algorithms are denoted as in table 1. The columns "Age at deadline" and "Value" are our extensions.

Table 1: Abbreviations of triggering algorithms.

Option	Domain		
	Time		Value
	Age at current time	Age at deadline	
No	OD	OD_AD	OD_V
Optimistic	ODO	ODO_AD	ODO_V
Knowledge-based	ODKB	ODKB_AD	ODKB_V

AD: Age at Deadline.
V: invalid based on Value.

5. PERFORMANCE EVALUATION

This section describes the simulator and the simulator setup used in our experiments. The following simulations are conducted and evaluated:

- **Consistency of user transactions:** Number of committed UTs, number of valid committed UTs, and number of generated triggered updates are used to investigate the throughput and consistency of UTs.
- **Transient and steady states:** Two different states are modeled. A transient state that lasts for 15–20 sec, where the sensor values change significantly and a steady state that lasts for 60 sec where the sensor values do not change much. This test for adaptability to state changes. The number of generated triggered updates for OD, ODKB_V, and ODDFT is measured.

5.1 Simulator Setup

The simulator used is a discrete-event simulator called RADEx++ [8], which is setup to function as a real-time main-memory database. Two queues are used: STs in the high priority queue, and UTs in the low priority queue. HP-2PL [1] is used as concurrency control protocol and transactions are scheduled based on earliest deadline first (EDF) [4]. The updating frequency of base items is determined by their *avi*. An *avi* is also assigned to each derived data item to determine freshness for on-demand algorithms with triggering criteria using the time-domain. UTs are aperiodic and the arrival times of UTs are exponentially distributed. A ST only consists of a write operation writing data item b_{ST}. UTs consist of read operations, computational work, and finally a write operation of data item d_{UT}. The number of read operations is the cardinality of read set $R(d_{UT})$. The WCET of a transaction is determined by the number of operations and the maximum execution time of these. For ST the single write operation always takes `STProcCPU` time. The maximum execution time for operations in a UT is `UTProcCPU`. During simulation each operation in a UT takes a uniform time to execute, which has an average determined during initialization of the database. This randomness models caches, pipelines, but also the usage of different branches of an algorithm. The deadline of a transaction is its WCET times a uniformly chosen value in the interval [1,7]. The derived data item a transaction updates is randomly chosen from the set of all derived data items.

Values of the data items are simulated with the parameter `max_change`, which is individual for each data item and it says how much a value can change during its *avi*. When a new value for a data item is written to the database, the stored value is increased with an amount that is proportional to U(0,`max_change`), where `max_change` and *avi* are derived from the same distribution U(0,800). $\delta_{i,j}$, where j is a parent of i, is given by $avi(j)$ times $factor$; $factor$ equal to one implies that the *avi*s give a good reflection of the changes of values, whereas if $factor$ is greater than one, the *avi*s are pessimistic, i.e., the values of data items are generally fresh for a longer time than the *avi*s indicate. The database parameters and the settings are given in table 2.

A database is described by $|B| \times |D|$. The directed acyclic graph giving the relationships among data items is randomly generated once for each database, i.e., the same relationships are used during all simulations. In our experiments we use a 45×105 database, implying that we have 150 data items

Table 2: **Parameter settings for database simulator.**

Parameter	Explanation	Setting
avi	absolute validity interval	U(200,800) msec
$\delta_{i,j}$	data validity bound for i	$factor \times avi(j)$
`max_change`	max change of a data item during its *avi*	U(200,800)
`STProcCPU`	max execution time of ST operation	1 msec
`UTProcCPU`	max execution time of UT operation	10 msec

in the database; the ratio of base items and derived items is 0.3. Moreover, cardinality of a read set $R(d)$ is 6, and the likelihood that a member of $R(d)$ is a base item is 0.6. The error function is defined as: $error(x, t) = t - timestamp(x)$, since `max_change` and *avi* are taken from the same distribution.

5.2 Experiment 1: Consistency of User Transactions

The goal of this experiment is to investigate how many valid user transactions there are for different triggering criteria. Validity is measured either as age (see definition 4.2), or data validity bound (see definition 4.6). The simulator is executed for 100 sec of simulated time. Each triggering criterion is simulated five times for arrival rates from 0 to 100 transactions per second with steps of 5 transactions.

At an arrival rate of 50 UTs/sec, the number of committed transactions can be found in table 3. At higher arrival rates, the number of committed and valid committed UTs do not increase (or increase only slightly) see [7], due to high load; there is not enough time to execute all transactions. This can be seen in figure 4 where the throughput-centric algorithms skip triggered updates to a greater extent than the consistency-centric algorithms.

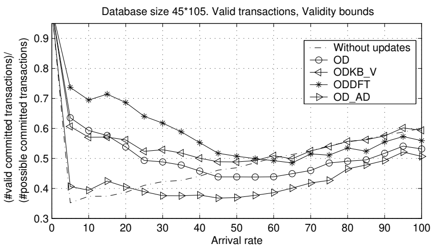


Figure 3: Ratio of valid committed user transactions. 95% confidence interval is ± 0.059.

Table 3: Ratio of valid committed user transactions and committed user transactions at arrival rate of 50 UTs/sec. 95% confidence interval is given.

Alg.	comm UTs	valid comm. UTs	ratio
OD	2818.0 ± 26.9	2015.0 ± 56.6	0.72
ODKB_V	3950.8 ± 66.7	2245.6 ± 37.2	0.57
ODDFT	2796.2 ± 78.3	2332.6 ± 73.4	0.83

When triggered updates are skipped the consistency is degraded. Figure 3 shows the ratio of number of valid committed UT and number of possible committed UT, which is taken from a simulation where no updates are generated. Validity is measured by definition 4.6. ODDFT has the largest number of valid committed UT up to an arrival rate of 50 UTs per second. Above 50 UTs per second no updating scheme does better than the no updates scheme, because there is no time to execute the updates. OD has less valid committed UTs than throughput-centric options, but from table 3 we can see that the ratios of number of valid committed UTs and committed UTs are higher for consistency-centric algorithms, i.e., OD and ODDFT, than for throughput-centric, i.e., ODBK_V. This shows that OD has a better consistency among committed UTs than throughput-centric options and ODDFT has the best consistency among all updating algorithms.

5.3 Experiment 2: Transient and Steady States

At transient states the base items might change frequently which implies more frequent updates of the derived data items. At steady states the changes of base items are less frequent, and, thus, the derived items need to be updated less frequent. This experiment shows that value-aware[5] triggering mechanisms can adapt to these changes, but *avi*-aware[6] triggering mechanisms, at least for static *avi*s, cannot.

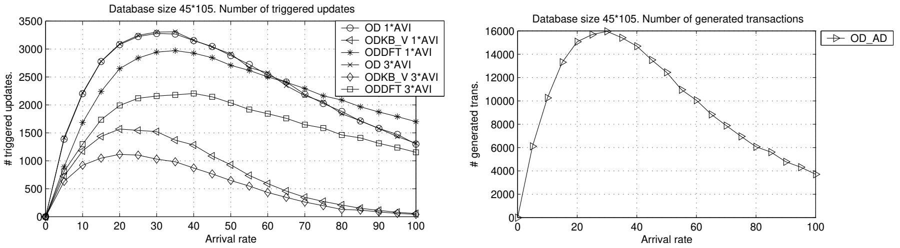
The number of generated triggered updates during a simulation is counted. The simulation is conducted as follows: the arrival rate is 30 UTs/second, the size of the database is 45×105, and 100 sec is simulated. Two parameters are introduced: `change_speed_of_sensors` and `change_speed_of_user_trans`. Data items change with the following speed: N(`max_change`/`change_speed_of_X`, `max_change`/(2×`change_speed_of_X`), where `X` is substituted with `sensors` or `user_trans`. For the first 15 sec, `change_speed_of_sensors` is set to 1.2 which gives rapid changes (transient state), from 15 sec to 75 sec `change_speed_of_sensors` is set to 50 (steady state), and from 75 sec the system enters again a transient state where `change_speed_of_sensors` is set to 2.0. During the simulation `change_speed_of_user_trans` is set to 2.0.

Figure 5 contains the simulation results. The horizontal lines represent the average number of generated triggered updates during the indicated interval. ODDFT clearly generates less number of triggered updates during the interval 15–75 sec than OD, which is not aware of that base items live longer in this interval. ODKB_V, which uses a value-aware triggering criterion also has less generated triggered updates in steady state. The load on the CPU is thus lower for ODDFT during a steady state than OD, and the extra load for OD consists of unnecessary triggered updates, which can be seen from the fact, with a 95% confidence interval, that ODDFT has 2092.2 ± 97.0 valid committed UTs and OD has 1514.2 ± 24.6 even though OD generates more updates., ODKB_V has 2028 ± 9.3 valid committed UTs, which on average is less than for ODDFT.

5.4 Summary and Discussion

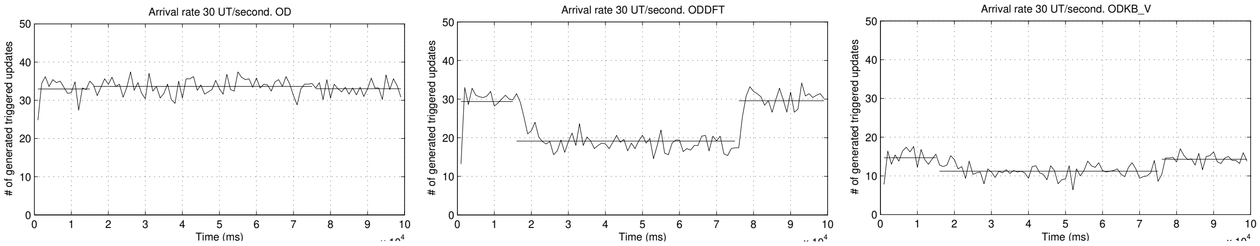
It has been shown that value-aware triggering criteria can adapt to state changes of the system (ODDFT and ODKB_V in the simulations). ODDFT lets the highest number of valid user transactions to commit up to an arrival rate of 50 user transactions per second, i.e., the consistency is high

[5]Using data freshness in the value-domain.
[6]Using data freshness in the time-domain.



(a) OD, ODKB_V, and ODDFT (b) OD_AD

Figure 4: Number of generated triggered updates. 95% confidence interval is ± 119.



(a) OD (b) ODDFT (c) ODKB_V

Figure 5: Number of triggered updates for different states.

among committed user transactions, but the total number of committed user transactions is considerable less than for throughput-centric criteria, e.g., the knowledge-based option. If it is acceptable to derive values that use stale data, then optimistic and knowledge-based options are better suited than ODDFT, but if it is important to always use fresh values then ODDFT gives the best results. One way to increase the throughput of UT at high arrival rates for ODDFT is to use a feedback control process to adjust the permissible time for updates (that is initially investigated in [7]). With this scheme it should be possible to have the benefits of a consistency-centric triggering mechanism as ODDFT at low arrival rates and the benefits of throughput-centric triggering mechanisms at high arrival rates.

6. RELATED WORK

For hard real-time systems static period times for updates and calculations are derived to guarantee freshness and timeliness of produced results [12, 14, 17, 9]. If unnecessary updates should be avoided, static period times are not feasible. In this paper we are considering a soft real-time system and the objective is to achieve a high effective utilization of the CPU. Hence, static period times are not appropriate. Mode changes could be used with different updating frequencies of data items in each mode (see [16]). The proposed updating scheme for freshness maintenance does not need mode changes since all modes are covered by the data validity bounds, which represents the updating frequencies in each mode.

Kuo and Mok [13] introduced a binary relation called similarity to capture the fact that small changes in a value of a data item might not affect the end result. Their work on similarity found different applications in real-time systems [13, 9]. In these applications a fixed time interval called a similarity bound is used, i.e., similarity is applied to the time-domain much like an absolute validity interval (see section 4.1). The similarity bound states that writes to the same data item within the prescribed time interval are similar. However, in our work we use a data similarity such that data freshness can be stated and used in the value-domain (in contrast to time-domain), i.e., we use data validity bounds in the value-domain (see section 4.2). By using data validity bounds we are able to construct an updating scheme that automatically adapts the required number of updates to the current state of the system, which is not possible with fixed time intervals such as similarity bounds and absolute validity intervals (avi).

Datta and Viguier [5] describe transaction processing for rapidly changing systems, where base item updates are committed even though the updates do not affect other data items, i.e., unnecessary updates are executed. Moreover, in their work calculations only depend on base items, i.e., intermediate results are not considered, whereas in our work arbitrary dependencies among data items are allowed.

Flexible period times are used in [10] to adjust period times between two limits by a feedback controller such that the workload of updates is lessened when the miss ratio of transactions is above a specific reference. Here all produced values are valid if the updating frequency lies within the bounds, but similarity of data values is not considered at all. Thus, unnecessary updates can still happen if the value is similar between updates.

Data-deadline and forced wait is used in [18] to achieve freshness at the deadline of a user transaction as in our work indicated by $freshness\_deadline$, but no derived data is used, only base data that is updated periodically, i.e., data-deadline and forced wait cannot deal with derived data. Our work considers derived data where a derived data item can be derived from other derived data items.

7. CONCLUSIONS AND FUTURE WORK

This paper addresses the maintenance of data in an embedded real-time system using a central storage. In particular, algorithms for keeping data fresh are proposed and evaluated. A scheme for generating necessary updates that is based on data validity bounds and that does not allow updates as long as values are considered similar. Similarity is defined elsewhere as a relation in the value-domain of data [13]. However, in the shown applications of the similarity relation a fixed time interval, similarity bound, is introduced meaning that data freshness is moved to the time-domain. In this work the updating scheme works in the value-domain of data giving good performance when it comes to number of needed updates compared to static time intervals.

Furthermore, well-known on-demand algorithms are extended with two new triggering criteria: (i) to use data validity bounds, and (ii) to measure the age of a data item from the deadline of the user transaction.

Simulations are conducted and the proposed updating scheme and a new scheduling algorithm (ODDFT) are compared to well-known on-demand algorithms. It is found that ODDFT maintains consistency among committed user transactions well. ODDFT also adapts updating frequencies to the current state of the system. Hence, available computing resources are better utilized compared to when static updating frequencies are used. It is also shown that the well-known on-demand algorithms can adapt updating frequencies to the current state by using data validity bounds. Consistency among committed user transactions is highest for ODDFT.

Our current work focuses on integrating the ODDFT triggering mechanism together with an existing database [6] in an engine electronic control unit. This should increase the understanding of the overhead costs of CPU and memory. It also seems that developing validity-aware concurrency control protocol should have merits in increasing data consistency and concurrency of transactions.

8. ACKNOWLEDGEMENTS

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

9. REFERENCES

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

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

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

[4] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1997.

[5] A. Datta and I. R. Viguier. Providing real-time response, state recency and temporal consistency in databases for rapidly changing environments. *Information Systems*, 22(4):171–198, 1997.

[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. Scheduling of updates of base and derived data items in real-time databases. Technical report, Department of computer and information science, Linköping University, Sweden, 2003.

[8] J. Hansson. *Value-Driven Multi-Class Overload Management in Real-Time Database Systems*. PhD thesis, Institute of technology, Linköping University, 1999.

[9] S.-J. Ho, T.-W. Kuo, and A. K. Mok. Similarity-based load adjustment for real-time data-intensive applications. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, pages 144–154. IEEE Computer Society Press, 1997.

[10] K.-D. Kang, S. H. Son, and J. A. Stankovic. Specifying and managing quality of real-time data services. Technical Report CS-2002-28, Computer Science Department at University of Virginia, 2002.

[11] 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 Transactions on Computers*, 2002.

[12] Y.-K. Kim and S. H. Son. Supporting predictability in real-time database systems. In *2nd IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, pages 38–48. IEEE Computer Society Press, 1996.

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

[14] 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 *Third International Workshop on Real-Time Computing Systems and Applications, 1996.*, pages 74–80, 1996.

[15] D. Nyström, A. Tešanović, C. Norström, J. Hansson, and N.-E. B. nkestad. Data management issues in vehicle control systems: a case study. In *Proceedings of the 14th Euromicro International Conference on Real-Time Systems*, pages 249–256, Vienna, Austria, June 2002. IEEE Computer Society Press.

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

[17] M. Xiong and K. Ramamritham. Deriving deadlines and periods for real-time update transactions. In *IEEE Real-Time Systems Symposium*, pages 32–43, 1999.

[18] M. Xiong, R. Sivasankaran, J. Stankovic, K. Ramamritham, and D. Towsley. *Real–Time Database Systems: Issues and Applications*, chapter Scheduling access to temporal data in real-time databases, pages 167–192. Kluwer Academic Publisher, 1997.