Scheduling of Updates of Base and Derived Data Items in Real-Time Databases*

Thomas Gustafsson and Jörgen Hansson
Dept. of Computer and Information Science
Linköping University, Sweden
{thogu,jorha}@ida.liu.se

Abstract

The amount of data that is handled by embedded and real-time systems is increasing. In some applications it is important to use fresh and accurate values on data when taking decisions or controlling an external environment. 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 freshness on data values. One way to ensure freshness of data values is to use fixed updating frequencies of the data values. For many systems a fixed updating frequency is too pessimistic since the system can enter states where the need to do updates is less than what the fixed updating frequency states. Hence, many updates are unnecessarily executed. The contributions of this paper are three-fold: (i) a new notion of data freshness that uses the value domain of data items, (ii) a scheme for managing updates of data with respect to changes in data values, and (iii) two new on-demand scheduling algorithms of updates, On-Demand Depth-First Traversal and On-Demand Breadth-First Traversal, denoted ODDFT and ODBFT respectively.

Simulation results show that ODDFT and ODBFT maintains consistency of data values better than well-known on-demand algorithms. Moreover, by using the update scheme and data freshness in the value domain algorithms (ODDFT, ODBFT, and previous defined on-demand algorithms) can change the updating need to the changes in data values in the current state.

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

1 Introduction

During the last decade the software in embedded systems has increased significantly in complexity. This is due to the increase in available memory and CPU speed. In an engine control system there is a need to process a large amount of data in a timely fashion, for instance to calculate fuel injection time and fuel amount for each cylinder. To accomplish this the engine control system needs, as almost all embedded systems, to measure data from its external environment, carry out calculations, and send calculated values to actuators. Thus, sensors and actuators need to be handled in the software, and, naturally, sensor and actuator values are stored as data variables.

New law regulations and diagnosis requirements for fault detection that is put on the car industry makes the software in engine control to rely even more than before on acquired data from the environment. In this respect, it is important not to only calculate data items in a timely fashion, but also to use updated data items in the control loops and diagnosis functions. These requirements imply a need for deadlines on the calculations of data items, and a need to use fresh, i.e., updated, data items in the calculations.

The increasing amount of data items in systems of today makes it natural to use a database to handle the data. Normally, databases are used for enforcing consistency when manipulating data items. The calculation of data items can now be done in transactions. The ACID properties of a transaction [4] ensure concurrent transactions have a consistent view of the data. The freshness of data items that are used in a transaction is not guaranteed though.

In this paper, we investigate different ways of ensuring fresh data items in calculations by using the notion of expected error of a data item. The error of a data item can be defined as how much the value of the data item differs from the perfectly updated value of the data item, i.e., the value is updated immediately when it changes, and it takes zero time. One way of defining expected error of a data item, as implicitly done in other research work [3, 9, 20], is to use the notion of absolute consistency [20] where a data item is guaranteed to have a fresh value during an absolute validity interval, i.e., the error of the value of the data item is low during this interval. After this duration of time the value is not consistent anymore and the error of the data item is instead high. For some systems it is not feasible to always have the same validity interval on a data item. In an engine control system, for instance, the engine temperature increases until it reaches its working temperature. When it does, the value representing the temperature is fresh for a longer period of time, because it is not likely that the temperature diverges much from the working temperature, i.e., the validity interval of the data item is not fixed. Another way to look at it is that the error of the temperature value is low for a longer time when the engine has reached its working temperature compared to when the engine starts to heat up. Hence, the validity intervals related to engine temperature are longer in the former case.

Validity interval of the data item determines when updates should be executed [16]. The data item might not have changed at all when the time stated by the validity interval has passed. One reason can be that dependencies, such as other data items, have not changed. The update is in fact unnecessary, but it is impossible to know this since the notion of absolute consistency only considers time, not the values of data items. When the dependencies have changed and the expected error of the data item is above a given limit, then the data item need to be updated. Thus, the load of the CPU can be lowered when using values of data items instead of validity intervals.

In this work, we model the expected error of data items in a more accurate way than what is possible with the usage of validity intervals, e.g., [3, 9], and, thus, decrease the need to update data items.

The outline of the paper is as follows: preliminaries to database systems are given in section 2. The problem description is stated in section 3. A description of the database model, the data items, the data dependency graph, the transactions, and the freshness of data items is done in section 4. The scheduling algorithms ODDFT and ODBFT are also presented in section 4. The simulator, simulator parameters, and simulation results is found in section 5. Section 6 covers the related work. Conclusion and future work end the paper in section 7.

2 Preliminaries

First a short description is given of a typical engine control system. The engine control system uses two time bases, one is triggered on the angle of the crank axis, whereas the other time base is triggered every fifth msec. Here the assumption is that we have a four cylinder engine. Once every second revolution, for each cylinder, a set of tasks are generated. Two tasks are generated before the top dead center[1] (TDC) that starts calculation of ignition angle and torque, and knock detection. Three tasks start after the TDC, and they take care of updating knock diagnosis, calculation of end angles for injection, and measurement of sensor values. Each of these tasks start a set of transactions that calculates sub-results of the final result computed by the task.

The periods of the time triggered tasks are: 5, 10, 20, 50, 100, 250, and 1000 msec. Every task generates transactions that read produced results and update data values. Furthermore, when a transaction calculates a new value of a data item, it uses values of other data items in the database to calculate the new value. These values are, in turn, calculated by other transactions. To derive fresh data items, the transactions have to be executed in an order such that the data items are updated before they are read by other transactions. Hence, there are precedence constraints among the transactions that are determined by which data items the transactions read.

More formally the database consists of base items B, i.e., the sensors in the engine control system, and derived items D, such as a compensation factor for fuel amount based on engine temperature. Associated with every derived data item d is a read set, denoted $R(d)$, containing the data items used for deriving the data item. For instance, one of the items of $R(engine\_temp\_comp\_factor)$ is the value of an engine temperature sensor.

Base items can be of two types:

[1]The top position of a cylinder where the air fuel mixture is compressed to the lowest volume, i.e., the pressure on the mixture is the highest before the spark from the spark plug ignites the mixture.

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

Transactions have the properties *atomicity*, *consistency*, *isolation*, and *durability* and consist of read/write operations and calculations.

Ahmed and Vrbsky simulated and evaluated an on-demand approach to generate updates [3]: when a data item is read by a transaction, and the data item is too old, i.e., current time is not within a so called absolute validity interval of the data item, then an update is generated that updates the data item before the transaction continues to execute. The update is called a triggered update or triggered transaction. The triggering of updates can be done in one of three ways, called options in [3], depending on what is considered important in the system; consistency or throughput of produced values. The options are denoted: (i) *no option*, (ii) *optimistic option*, and (iii) *knowledge-based option* (the options are described in more detail in section 5.2). The on-demand algorithm and the options described in [3] are used as baselines in the evaluation of simulation results in section 5.

3 Problem Description

As can be seen from the example above with the engine control unit, we have two problems that have to be solved simultaneously:

P1 the requirement of usage of up-to-date data implies a need to do updates of data items before they are used in a calculation; and

P2 the updating frequency of a data item can vary during run-time. To make best use of the available computing resources the system need to adapt to the new update frequency.

The type of systems that this paper is focusing on are real-time systems, and therefore, the tasks such a system is executing have deadlines. The deadlines of the system can be a mix of hard and soft deadlines. The schedulability of a set of transactions depends on how many updates that is started and that cannot be determined before the system starts. Hence, in this paper, deadlines of transactions are firm or soft.

P1 indicates the importance to schedule and execute updates of data items in a timely fashion. Every transaction in the system is modeled as a single-writer, where the transaction either writes a base item belonging to B or a derived item belonging to D to the database. All data items in the system are ordered in a partial order given by the precedence constraints among the data items, this is described in section 4.1.1, and the updates need to be scheduled based on the partial order.

P2 is justified by that computing resources can only be bought in discrete units, e.g., a CPU with a specific frequency. In industry where costs should be kept as low as possible, it is important to use the available computing resources as efficient as possible. If an algorithm is inefficient then a more powerful CPU might be needed, which gives an extra cost. By adapting the updating frequency to the actual need, the cost of updates can be reduced compared to if static updating frequencies are used, and hence, in total, less resources might be needed.

For instance in the engine control unit mentioned above, both the efficiency (P2) and the freshness (P1) need to be addressed at the same time. Thus, a model that maintains data in an efficient way is needed.

The goal of this work is to propose and evaluate:

- A model of data items and transactions such that it is applicable to, foremost the ECU, but also other kinds of systems that handles transactions in a firm or soft setting.
- An updating scheme that takes arbitrary precedence constraints into consideration, that tries to minimize the number of generated updates, and produced results are consistent, i.e., fresh data have been used in the derivation of the results.

4 Scheduling of Triggered Transactions

When a transaction is about to execute, the data values that are used when deriving results need to have fresh values. Two algorithms that schedule the needed updates for a transaction are described in this section. The algorithmic steps of the scheduling of updates is shortly described here with references to a more elaborate description for each part of the steps.

- *Updating base items.* Base items are updated with a high frequency and derived data items are marked when they need to be recalculated. A data dependency graph is used to keep the dependencies between data items. A validity bound is introduced for each parent of a data item that indicates how much the particular parent can change before it actually affects the value of the data item. To accept different importance levels of the calculations, several queues are used to schedule transactions in. Section 4.1 contains an explanation of the database, the data items, the data dependency graph and the transaction model.
- *Scheduling of updates.* The actual scheduling of updates are done by one either On-Demand Depth-First Traversal, denoted ODDFT, described in section 4.6 (pseudo-code in figure 7) or On-Demand Breadth-First Traversal, denoted ODBFT, described in section 4.7 (pseudo-code in figure 9). The scheduling algorithms traverses the data dependency graph to find parents that need to be updated to get fresh data values on the data items that the executing transaction will use.
  - *Deciding which data items need an update.* It is worth recalculating a data item when the change of a value is noticeable. During scheduling of updates a function, called *error*, is used that gives an upper

bound of how much a data item can have changed at a given time. The function is described in section 4.2.

The latest time a data item still has to be valid for a transaction to produce a reasonable result is discussed in section 4.3. This point in time and $error$ is used during scheduling of updates to decide which data items that have to be updated.

  – *Prioritizing pending updates.* If several updates can be executed at the same time a choice has to be made. A priority is assigned to each possible update by an algorithm, denoted AssignPriority, described in section 4.4 (the pseudo-code can be found in figure 3) and the update with highest priority is scheduled first.

- *Scheduling points.* Section 4.5 discusses at which occasions updates are scheduled and when updates are executed. The algorithm, denoted Dispatch, described by pseudo-code in figure 4, is used when a transaction enters or leaves the system.

Furthermore, section 4.8 describes a feedback control model that can be used to limit the number of generated updates. Section 4 is ended with a discussion of the scheduling algorithms in subsection 4.9.

4.1 Database Model

The main purpose of the model described in this paper is to represent the handling of data in a system that has freshness requirements on the data items. Moreover, the base items in the model, e.g., sensors, are continuous data items that can be fetched by the system itself and derived data items are discrete data items.

In this subsection a model is described for data items, transactions, and the freshness of data items. A discussion of base item updates and data freshness can be found in subsection 4.9.

4.1.1 Precedence Constraints and the Updates of Base Items

The relationship between base items and derived items can be viewed as a directed acyclic graph (DAG) $G = (V, E)$. Each vertex v represents a data item and a directed edge (v, w), belonging to the set E, represents that data item v is a member of the read set of w.[2] A derived data item d resides at a particular level based on how many vertices there are in a path from a base item to d. The level of a derived item d is defined as follows.

Definition 4.1. *Each base item b has a fixed level of 1, i.e.,*

$$level(b) = 1. \tag{1}$$

[2]The data item v may be a base item or a derived item, while w always is a derived data item.

The level of a derived data item d is determined by the longest path from a base item to d. Hence, the level of d is

$$level(d) = \max_{\forall x \in R(d)} (level(x)) + 1. \tag{2}$$

Let us give an example. Figure 1 shows a DAG with seven base items and nine derived items. Here the directed edge (b_1, d_5) means that b_1 is a member of the read set of d_5. Now we describe how data freshness can be defined for continuous and discrete data items.

In [20] a data item is said to be absolutely consistent with the entity in the external environment it represents as long as the age of the value is within an interval, the so called absolute validity interval, *avi*.

Definition 4.2. *A data item x that can be either a base item or a derived item is absolutely consistent when:*

$$current\_time - timestamp(x) \leq |avi(x)|. \tag{3}$$

This definition defines data freshness for continuous data items. Note that, a discrete data item has no fixed absolute validity interval, since a discrete data item is valid until the next update arrives and that point in time is not known.

The data items in a read set are said to be relatively consistent if they are created close to each other in time. Relative consistency is defined as follows [20]:

Definition 4.3. *A set of data items V are relatively consistent if their timestamps do not differ more than an interval, the so called relative validity interval, i.e.,* $\forall y, \forall v \in V, |timestamp(v) - timestamp(y)| \leq |rvi(V)|$, *where* $rvi(V)$ *is the relative validity interval of the set of data items.*

On the other hand, it can be enough, from a consistency perspective, that the data items read by a transaction are valid at the same time. This resembles definition 4 in [15], that handles discrete data items and versions of such data items. A version of a data item is an old value that was valid during a certain validity interval (VI). The latest version is valid until a new version is installed in the database, and the end time in VI of the current version is set to ∞.

Definition 4.4. *Given a set of data items W, the data items in W are said to be relatively consistent if* $\bigcap\{VI(x_i)|x_i \in W\} \neq \emptyset$, *where* $VI(x_i)$ *is the interval* $[timestamp(x_i), timestamp(next(x_i)))$ *of time that data item* x_i *is valid in, and*

$$timestamp(next(x_i)) = \begin{cases} timestamp(x_i') & \text{if } x_i' \text{ exists and is the next version of } x_i, \\ \infty & \text{otherwise.} \end{cases}$$

In this paper, a data item has only one version, the latest, and therefore W is a set of data items as V in definition 4.3, and as can be seen from definition 4.4, the latest version of data items are always relatively consistent.

Definition 4.2 defines data freshness for a data item in the time domain. When a data item d updated at time t_1 and becomes stale at time t_2, then according to definition 4.2 its derived value at t_2, denoted $v_d^{t_2}$, can be the same or similar to the value $v_d^{t_1}$, thus, the data item is still fresh in the value domain. Hence, a data item d is fresh as long as the values of data items in the read set are within reasonable bounds from the values that were used during the derivation of d. The bounds are denoted data validity bounds and are defined as follows.

Definition 4.5. *A data validity bound $\delta_{d,x}$ states how much data item $x \in R(d)$ can change before it affects the value of d, requiring d to be recomputed.*

The data freshness in the value domain is defined as follows.

Definition 4.6. *A derived data item d that was derived at time t_0 is derived from $R(d)$ by using values $\forall d' \in R(d), v_{d'}^{t_0}$. d is fresh as long as all data items in $R(d)$ have values that are within acceptable bounds from $\forall d' \in R(d), v_{d'}^{t_0}$, i.e., d is fresh at times t when:*

$$\bigwedge_{\forall d' \in R(d)} \{|v_{d'}^{t_0} - v_{d'}^{t}| \leq \delta_{d,d'}\} \tag{4}$$

evaluates to true.

All possible changes of data items are originated from the base items. To capture changes of data items, the system updates base items frequently and data freshness according to definition 4.6 is checked for every derived item d in level 2 (see figure 1). If equation (4) evaluates to false, then a flag denoted *changed* is set to true for all data items, including the derived item d in level 2, that are derived from d.

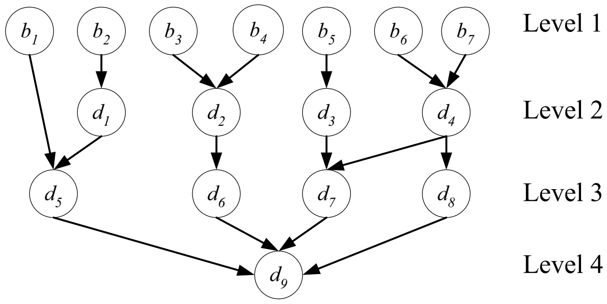


Figure 1. The base items at level 1 and the derived items.

4.1.2 Data Items

A data item is represented by the following tuple

$d : (value, timestamp, R(d), changed, weight, priority, avi, error, WCET)$ where:

- *value* is the value of the data item.
- *timestamp* is the timestamp when the data item is updated.
- $R(d)$ is the read set where each element is a tuple $(x, \delta_{d,x})$ where x is a data item that is needed when deriving d.
- *changed* is set to true if any of the ancestors of d have been changed according to definition 4.6.
- *weight* is the importance of the data item (high weight means that the data item is highly important). The weight is fixed and determined before the system starts.
- *priority* is the priority assigned by the system during run-time. How the priority is assigned among the data items is further explained in section 4.3.
- *avi* is the absolute validity interval as given in definition 4.2, and is used on base data items to decide the update frequency. All data items have an *avi*. For derived data items the *avi* is not necessary, since the validity of a data item depends on the data validity bound δ, but the *avi* is used in section 5 to compare the baseline algorithms with the proposed algorithms.
- $error(d, t) \rightarrow \mathbb{R}$ is a function that is used to approximate how much the value of a data item d has deviated, at time t, from the stored value in the database. The function is used to assign priorities to data items when updates are scheduled.
- $WCET$ is the worst-case execution time, without updates, for deriving the data item d.

4.1.3 Transactions

Base items are kept fresh by sensor transactions (ST) that only consists of one write operation writing data item b_{ST}. A user transaction, denoted τ_{UT}, consists of one or several read operations reading the members in the read set, calculations, and one write operation writing data item d_{UT}. τ_{UT} has to read fresh data items belonging to the read set of d_{UT}. If the data items in the read set are not fresh enough, they have to be made fresh by updating them. Transaction τ_{UT} triggers the necessary transactions that updates the data items. These transactions are denoted triggered transactions (or triggered updates (TU)) and τ_{UT} denotes the triggering transaction. The triggering transaction and the triggered transactions are under a precedence constraint, due to the dependency of data items. The triggered transactions need to be executed before the triggering transaction since the result of the triggered transactions are used by the triggering transaction.

A triggered transaction must not delay triggering transactions other than the one that generated it, because an update is considered a part of the triggering transaction.[3] This means that triggered transactions belonging to the triggering transaction with the earliest deadline can be executed.

The database system has two priority levels for transactions: high, which the most important transactions are scheduled at, and low, which all other transactions are scheduled at. Each priority level is implemented as a queue. If we return to the ECU as an example the high priority queue are used for those transactions that are generated by angle triggered tasks, since these transactions handles ignition of the air fuel mixture, which can be considered as an important real-time task, even though occasional deadline misses of the calculation of ignition time can accepted; the previously calculated value is used. Transactions generated by the time triggered tasks are put in the low priority queue. The earliest deadline first scheduling algorithm (EDF) [6] is used to schedule transactions in both queues. Tasks in the low priority queue are only executed when there are no tasks with high priority ready to execute. Base item updates are scheduled in the high priority queue. These updates are periodic and have a high frequency.

It is not possible to guarantee the execution of low-priority transactions since a higher priority transaction can arrive and interrupt the low-priority transaction. A transaction is typically part of the execution flow of a task—as mentioned in section 2—and therefore the produced data item of a transaction is needed in subsequent calculations in the task. Hence, the transaction need to update the data item, and therefore the deadlines on low priority transactions are soft.

Concurrency control in the database in our model can be either pessimistic or optimistic concurrency control. If pessimistic concurrency control is used the locking mechanism has to be aware of the two priority queues since a lock should be given to a transaction in the high-priority queue before the lock is given to a transaction in the low-priority queue even though the transaction in the low-priority queue has an earlier deadline. One way to solve the problem is to use 2PL-HP [1] and set priorities to the transactions in the high-priority queue to a higher value than the transactions in the low-priority queue.

4.2 Validity

It is impossible to know when a data item d is not fresh anymore. Remember that a data item is considered fresh as long as the data items in its read set have not deviated more than the data validity bound $\delta_{d,x}$, where x is a data item in $R(d)$. The current value of any x that belongs to $R(d)$ can only be determined by recalculating x, and, thus, all parents of d in the data dependency graph need to be recalculated.

A data item that is updated to be used in a derivation of a new data item can change in the middle of the

[3] Assume transactions are scheduled with EDF. A triggered transaction is generated with deadline t_1 by a user transaction with deadline t_2, and now a new user transaction arrives to the system with deadline t_3. If $t_1 < t_3 < t_2$ then the triggered transaction delays the new user transaction.

derivation of the new data item. There are two possible ways to handle this: (i) update the changed value and restart the derivation, and (ii) accept the fact that the value has changed and continue the derivation. If the maximum deviation of the value of the data item is known before a transaction that uses it starts and the deviation is within acceptable bounds, then the data item need no updates during the derivation of a new value, i.e., (i) above is not necessary.

If the error is acceptable then the data item can be used without a need for updating it. Examples of functions that could be used as the *error* function are:

1. $error(x, t) = c(t - timestamp(x))^z$, where $0 < z < \infty$; or

2. $error(x, t) = c \log(t - timestamp(x))$.

The constant c, which is an individual constant for each d, in the above examples is set to a value that models the current state of the system. The constant c can also be fixed during the run-time of the system.

To be able to compare the values returned by error functions for different data items, the return values need to be normalized. This is done by dividing the returned value by the value of the data item, e.g., $error(d_1, 10)/v_{d_1}^t$ Now the normalized value represents the maximum deviation in percent.

For instance, d_4 in figure 1 is fresh as long as $|v_{b_6}^{t_0} - v_{b_6}^{t_1}| \leq \delta_{d_4,b_6} \wedge |v_{b_7}^{t_0} - v_{b_7}^{t_1}|\delta_{d_4,b_7}$ evaluates to true, i.e., as long as the read set members of d_4 have not deviated more than the acceptable validity bounds δ_{d_4,b_6} and δ_{d_4,b_7}.

The implication of the choice of the value of avi is discussed in section 4.9

4.3 Freshness of Data Items During Derivation of New Data Items

When a transaction is executing it updates one data item, say d. In both ODDFT and ODBFT, $freshness\_deadline$ indicates the point in time where all the members of $R(d)$ have to be fresh. Here fresh means that either the data item x in $R(d)$ is absolutely consistent or is within its data validity bound $\delta_{d,x}$ at the given time. Both algorithms schedule the updates, i.e., the triggered transactions—one triggered transaction for each update—starting from the deadline of the triggering transaction towards the release time of it, see figure 2. The reason is that (see the algorithm PREC1 in [17]) it is easier to maintain the precedence constraints in this way. The triggering transaction has to be scheduled last since it uses the results of all the updates. The closer an update is to the deadline, the less time is available to diverge from the stored value.

In figure 2 the circles represent possible points in time for $freshness\_deadline$, and each box with a data item represents the worst-case execution time (WCET) it takes to calculate the data item. Each possible $freshness\_deadline$ occurs twice, once with a dashed boundary representing a $freshness\_deadline$ in the schedule that starts to execute at the release time of the transaction, and a solid boundary representing $freshness\_deadline$ in the schedule when the data items are as close as possible to the deadline of the triggering transaction. The

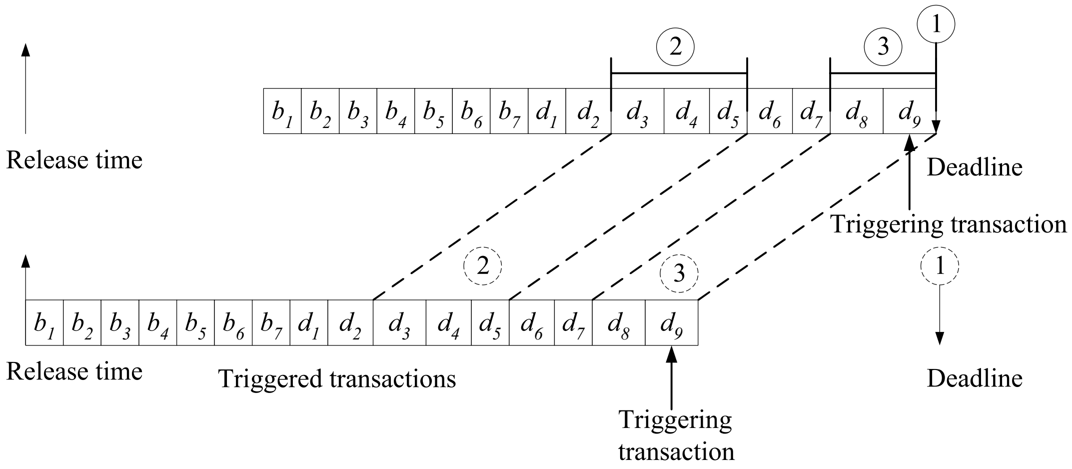


Figure 2. Different possible values of $freshness\_deadline$. The transactions are taken from figure 1 in a top to bottom, left to right order.

number it represents in the circle indicates which item in the numbered list below. When the schedule is generated, the updates are moved to the release time (see PREC1 [17]), i.e., the EDF scheduling algorithm is used for transactions. It is possible to set $freshness\_deadline$ to one of the following values:

1. Equal to the deadline of the triggering transaction τ_{UT} writing data item d, i.e., all the members of $R(d_{UT})$ need to be valid throughout the execution of τ_{UT}. As can be seen in figure 2 the $freshness\_deadline$ does not change when the schedule is moved towards the release time of the triggering transaction.

2. Equal to the release time of a transaction τ_{UT}, triggered or triggering, that reads data item d_{UT}. Thus, d_{UT} has to be valid at least until τ_{UT} is started. For an example see figure 2. The $freshness\_deadline$ of d_2 is the release time of d_6, because d_2 is a parent of d_6 in figure 1. The actual release time of the triggering transaction updating d_6 might not, during the execution of the schedule, be the same as in the generated schedule. One reason is that the schedule is moved if it does not reach the release time of the triggering transaction (the move of the schedule as done from the upper part to the bottom part of figure 2), another reason is that the triggered updates can be finished before their WCET and, thus, all successor transactions will be started earlier.

3. Equal to the commit time of the triggering transaction τ_{UT}. The main problem is that the commit time of the arriving transaction is not known when the scheduling takes place, see figure 2. If transactions in the schedule are completed before their worst-case execution time, then the remaining transactions in the schedule will be started earlier, advancing the commit time of τ in the schedule. Here the $freshness\_deadline$ of d_7 is the commit time of d_9 (the data dependency graph can be found in figure 1).

One could argue that the $freshness\_deadline$ as in the top of figure 2 is the latest possible $freshness\_deadline$ and that this value could be used, but for both ODDFT and ODBFT case 1 is used. We assume that the freshness of a data item cannot be higher when all data items it is derived from are valid until the deadline of the transaction.

4.4 Priority on Update Transactions

Triggered transactions are prioritized according to algorithm AssignPriority described in figure 3. The idea is that transactions updating the most important data items are scheduled first when a large part of the interval $[release\_time, deadline)$ is available to schedule them in. Updating a data item implies it is considered fresh enough when it is later used to derive other data.

AssignPriority uses the expected error of data given by the function called *error* to set the priority among the triggered transactions. The total error among those transactions that need to be updated is calculated and the more a triggered transaction contributes to reducing the total error the higher priority it gets. If a transaction cannot be scheduled due to no available execution time, then its priority is higher the next time AssignPriority is used since the error function returns a higher value due to increase in deviation from the stored value over time. A weight is also defined for each data item. The weight is used together with the priority given by AssignPriority to determine which update to schedule first. The weight is multiplied with the priority, and the product is used as the priority of the data item. The purpose to use a weight for each data item is to capture the need to force updates of some particularly important data items. In the engine control system, for instance, it is important to check if the engine starts knocking. The data items that are used for monitoring this have to be fresh and have priority over other data items. Since each read set of a data item d has a data validity bound $\delta_{d,x}$, where x is a member of $R(d)$, it is possible to assign a small value to $\delta_{d,x}$ such that the update is always scheduled. If the weight is not needed, then it can safely be set to 1 for all data items.

The algorithm is presented in figure 3. The computation complexity of the two for-loops are $O(n)$ and $O(n \log n)$, where n is the size of $R(d)$. It computation complexity for adding an element to a sorted list is $O(\log n)$.

4.5 Scheduling Points

Algorithm Dispatch described in figure 4 is used every time a transaction, triggering or triggered, finishes. The algorithm checks whether the transaction finishes ahead of time, if it does, then the extra free time can be used to schedule transactions that could not be fitted into the original schedule. The only transactions that can be scheduled are those updating data items at the same level or lower[4] levels as the finished transaction and whose corresponding children in the data dependency graph are not yet executed. The reason is that all transactions are

[4]Data item d is closer than data item d' to the base items in the data dependency graph if $level(d) < level(d')$, and, thus, resides in a lower level.

AssignPriority(d,$freshness\_deadline$)

for all $x \in R(d)$ **do**

  if $error(x, freshness\_deadline) \geq \delta_{d,x}$ **then**

    $total\_error = total\_error + error(x, freshness\_deadline)$

    Put x in queue Q_1

  end if

end for

for all $x \in Q_1$ **do**

  $prio(x) = error(x, freshness\_deadline)/total\_error$

  Multiply $prio(x)$ with $weight(x)$

  Put x in queue Q_2 sorted by priority

end for

Return Q_2

Figure 3. The AssignPriority algorithm that assign priorities to the data items in a read set $R(d)$.

under a precedence constraint and, thus, it makes no sense to execute a triggering transaction whose parents have not been updated. Line 4 in figure 4 can be replaced with the following pseudo-code that searches for updates to execute in available free slots due to early finishing of updates:

if $current\_time < deadline(\tau)$ **then**

  Let $freshness\_deadline$ be equal to the $freshness\_deadline$ used for τ_t

  Let $schedule$ be schedule of triggered updates for τ_t

  for all $\tau_x \in schedule$ with $release\_time(\tau_x) < release\_time(\tau)$ and not scheduled **do**

    if $(level(\tau_x) = level(\tau)) \vee (level(\tau_x) < level(\tau) \wedge \tau_x$ parent of not yet scheduled transaction) **then**

      $deadline(\tau_x) = deadline(\tau)$

      Call ODDFT or ODBFT

    end if

  end for

end if

When a transaction arrives to the system, algorithm Dispatch is called. Triggered updates for the transaction are scheduled in line 10 in figure 4.

Triggered transactions are picked from a schedule belonging to the triggering transaction with the earliest deadline. In this way a triggered transaction τ cannot delay triggering transactions with earlier deadlines than the triggering transaction that triggered τ. A check can be done that a triggered transaction has a chance to commit.

Dispatch(τ)

if τ is triggered transaction **then**
    Let τ_t be the triggering transaction of τ
    Remove τ from the scheduling queue of τ_t
    Pick the next triggered transaction of τ_t
else if τ is triggering transaction **then**
    if τ already in one of the two transaction queues **then**
        Let τ_t be the triggering transaction with closest deadline in the high priority queue or in the low priority queue if the high priority queue is empty
        Pick the first transaction in the scheduling queue of τ_t
    else
        Call ODDFT(τ, $deadline(\tau)$) or ODBFT(τ, $deadline(\tau)$)
        Let τ_t be the triggering transaction with closest deadline in the high priority queue or in the low priority queue if the high priority queue is empty
        Pick the first triggered transaction of τ_t
    end if
end if

Figure 4. The Dispatch algorithm that is used every time a transaction finishes and when a new triggering transaction arrives to one of the two queues.

The check is not part of the pseudo-code in figure 4. The release times that algorithms ODDFT and ODBFT calculate are the latest possible release times of a triggered transaction, since the release times are calculated based on the deadline of the triggering transaction. Lines 4, 8, and 12 can have such a test:

Calculate $available\_time = current\_time - deadline(\tau_t)$
for all $\tau_x \in$ schedule of τ_t in reverse order **do**
  $available\_time = available\_time - WCET(\tau_x)$
  if $available\_time < 0$ **then**
    break
  end if
  if τ_x =preempted triggered transaction **then**
    break
  end if
end for
Continue executing triggered transactions from τ_x

The test checks which triggered transaction to continue executing from in the schedule of updates.

4.6 On-Demand Depth-First Traversal Scheduling

The rationale with the depth-first scheduling algorithm is that the read set members have to be fully updated before they are used. To fully update a read set member, all its parents in the data dependency graph need to be updated. The read set member x of data item d, can be seen as a triggering transaction updating x before the transaction deriving d reads the value of x. For instance, in figure 1 transactions for d_5, d_6, d_7, and d_8 need to be executed before the transaction that writes d_9. This is a recursive algorithm that updates one branch of the data dependency graph before it continues with the next. The algorithm is described in figure 7. The parameter τ is the transaction that needs a data item to be updated, and $freshness\_deadline$ is the time at which a read data item (or any of its predecessors) need to still be fresh. As pointed out above, $freshness\_deadline$ is set to the deadline of the triggering transaction. The transaction is put in a queue with release time and deadline as late as possible. When all triggered transactions are put in the scheduling queue, they are all moved[5] as close to the original release time of the triggering transaction as possible. This effect is achieved by the algorithm Dispatch that always schedules the first triggered transaction in the schedule of a triggering transaction, irrespective of the calculated latest release time of the triggered transaction. If the schedule is not moved to the release time, i.e., triggering transactions are postponed, then it is possible that another transaction arrives and, due to the

[5]The release time of the transactions in the schedule are set such that the precedence constraints of the updates are still fulfilled and the transactions cannot be closer to the arrival time of the transaction generating the updates.

postponement, one or several of the transactions cannot finish within their deadlines.

As an example of how the scheduling works see figure 5. Here the data dependency graph from figure 1 is used and a triggering transaction uses data item d_9. The read set members of τ_{UT} is prioritized in the following order, highest first, d_8, d_7, d_6, and d_5.

In figure 5, the triggered transaction that writes to data item d_4 and its parents are scheduled twice. Once because it is a parent of d_8, and once because it is a parent of d_7. Normally, the value of d_4 generated by the triggered transaction associated with data item d_7, denoted $\tau(d_7)$, can be used by $\tau(d_8)$. Line 8 of ODDFT removes such already scheduled transactions. An update can be removed if the value produced by the earlier update is valid when the value is later used. If the value is not valid the duplicate transaction cannot be removed. In this paper, it is assumed that all duplicates can be removed.

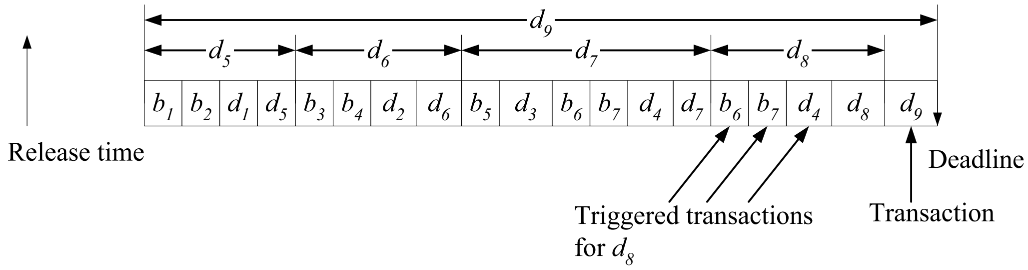


Figure 5. The schedule before moved as close as possible to the release time.

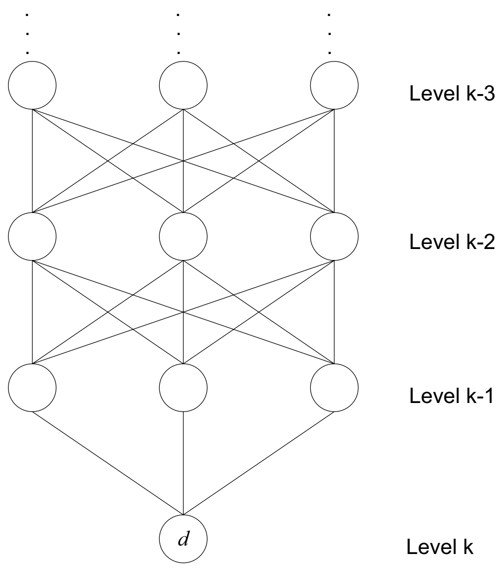


Figure 6. A graph that gives the worst-case running time of algorithm ODDFT.

The complexity of the ODDFT algorithm is exponential in the size of the graph. Regard the graph in figure

6. In this graph, every node is connected to all nodes in one lower level, i.e., a node in level k is connected to all nodes in level $k-1$. There are a maximum of m nodes in one level. The recurrence relation $T(n)$, where $T(n)$ denotes the computation time spent in one node in level $n \leq k$, can be stated as:

$$\begin{cases} T(n) = mT(n+1) + O(m \log m) \\ T(k) = 1 \end{cases}$$

where $O(m \log m)$ is the running time of AssignPriority and m is the maximum out-degree of a node, i.e., the maximum size of a read set. Note that the recurrence relation is defined for the bottom level, k, which is the data item d to update, to the top level. The total running time of algorithm ODDFT is $O(m^n m \log m)$, where m is the maximum in-degree of a node in graph G, and n is the number of levels in the graph. Hence, the computation time for ODDFT is exponential in the size of the graph in the worst case, since a graph with n levels can be constructed from $m \times n$ nodes, which is a polynomial in the number of levels, and hence a polynomial in the number of nodes, i.e., the computation time is, in the worst case, exponential in the size of the graph. Note that this only shows that the running time is exponential for certain graphs. Note, ODDFT stops traversing the graph when the release time is reached. The number of recursive calls can be calculated by taking $\frac{deadline - release\_time}{WCET}$.

ODDFT(τ, $freshness\_deadline$, $schedule$)

$release\_time(\tau) = deadline(\tau) - WCET(\tau)$
Put transaction τ first in $schedule$
Set d=data item updated by τ
$priority\_queue = AssignPriority(d, freshness\_deadline)$
for all items x in $priority\_queue$ in priority order **do**
    if $changed(x) = true$ **then**
        Let τ_x be the transaction that updates x
        Remove duplicates of τ_x from the schedule
        $deadline(\tau_x) = release\_time(\tau)$
        ODDFT(τ_x, $freshness\_deadline$, $schedule$)
    end if
end for

Figure 7. The ODDFT algorithm that schedules updates in a depth-first manner.

4.7 On-Demand Breadth-First Traversal Scheduling

When there is not enough execution time to schedule all necessary updates, the depth-first algorithm focuses on the most important read set members according to AssignPriority. One of the read set members d_i might not be updated, even though all of the parents of d_i already are up to date. For instance, the branch, from top to bottom, b_1, b_2, d_1, d_5 in figure 5 cannot be executed if the release time of the triggering transaction would have been closer to the deadline. The idea of the breadth-first scheduling algorithm is that a data item is updated based on the level it belongs to, not on which branch it belongs to.

Normally, the breadth-first algorithm (see [8]) is implemented by using a FIFO queue for determining from which node to continue to expand the frontier between discovered and undiscovered nodes. Here, this is not sufficient, we instead want the nodes to be picked in both level and priority order. Level order is used to obey the precedence constraints, and priority order is used to pick the most important update first. The relation $\sqsupset$ is introduced, and $x \sqsupset y$, where x and y are data items in the database, is defined as;

$$\begin{aligned} x \sqsupset y \text{ iff } & \mathit{level}(x) > \mathit{level}(y) \vee \\ & (\mathit{level}(x) = \mathit{level}(y) \wedge \mathit{prio}(x) > \mathit{prio}(y)) \vee \\ & (\mathit{level}(x) = \mathit{level}(y) \wedge \mathit{prio}(x) = \mathit{prio}(y) \wedge \mathit{id}(x) > \mathit{id}(y)), \end{aligned}$$

where *prio* is the product of the priority of the data item and the weight, *level* is the level in the data dependency graph the data item resides at, and *id* is a unique identifier associated with the data item. The use of an identifier of a data item gives a total order of the data items. If data item d_4 has the integer 4 as an identifier and data item d_5 the integer 5, then $d_5 \sqsupset d_4$ if they reside in the same level and is assigned the same priority.

The queue of nodes to use as a frontier in the ODBFT algorithm is sorted by the relation $\sqsupset$. Every time a node of the data dependency graph is inserted into the queue, the queue is inserted in the right position based on relation $\sqsupset$. The inserted node is a read set member that later is put in the schedule and also is the origin of a new search for updates. The first element in the queue is put in the schedule every time the algorithm iterates and a new node is used to search for new updates. The first element is also used by the algorithm to start the new search for undiscovered nodes in the graph. Since $\sqsupset$ orders the data items according to level, ODBFT behaves as a breadth-first search. Only those transactions that updates a data item that might change are scheduled, i.e., those data items whose *changed* flag were set to true.

Initially, all nodes are colored white. A node can only be inserted in the queue when it is white. ODBFT cannot schedule the same transaction more than once, since the first time a node in the data dependency graph is visited it is colored gray and can, thus, not be included again in the queue. Hence, the problem of the occurrence of duplicates of a transaction is not possible.

Figure 8 shows how the transactions for the data item in figure 1 would be scheduled by ODBFT. As can be seen, the triggered transactions are scheduled by level and priority.

The total running time of algorithm ODBFT is $O(V + E)$ [8] if the operations for enqueuing and dequeuing Q take $O(1)$ time. In algorithm ODBFT, the enqueuing takes in the worst case $O(\log V)$ since the queue can be kept sorted and elements are inserted in the sorted queue. The total running time of algorithm AssignPriority called

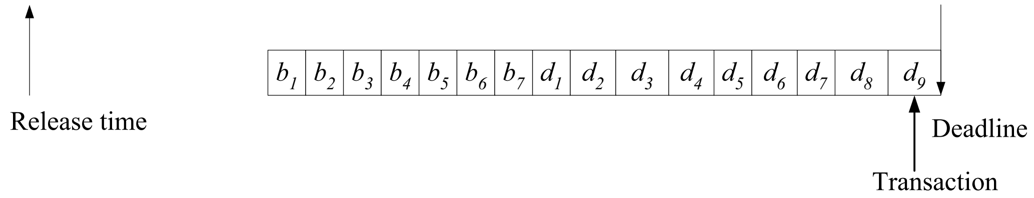


Figure 8. A triggering transaction and triggered transaction scheduled by breadth-first.

by ODBFT is the same as the for-loop adding elements to a sorted queue, i.e, $O(E \log p)$, where p is the maximum size of the read set of a data item. Thus, the algorithm has a total running time of $O(V \log V + E \log p)$.

4.8 Feedback Control

As simulation results show in section 5.4, ODDFT and ODBFT have a higher miss ratio of user transactions at high arrival rates than compared to on-demand algorithms with knowledge-based option. The main reason for this behavior is that the actual user transaction is delayed by the generated updates and is therefore executed closer to its deadline. The available slack of the user transaction is decreased so the available time for blocking and restarts is also decreased. What is needed is a way to determining how much of the accessible slack that can be used by the updates. Next is a solution presented based on basic control theory. An application of this solution is presented in section 5.7.

The size of the slack available for updates is measured as the ratio of the available slack for updates and remaining execution time when the WCET of the user transaction has been accounted for, i.e., the fraction of execution times of S_2 and S_1 in figure 10. When the arrival rate is high, then the ratio should be set high and stop updates to be executed. Thus, the load on the system is lowered and the user transactions have a greater chance to be executed.

One way to control the size of the slack is to use a feedback control scheduling algorithm. The monitored variable is the miss ratio of user transactions. We model the controlled process as shown in [19]. The system is pictured in figure 11. The manipulated variable is the slack ratio, SR. The controlled variable is the miss ratio of user transactions, M. The performance reference of the miss ratio, M_r is set to 20%.

A P-controller is used to change the slack ratio in order to increase/decrease the miss ratio. The controlled process, i.e., the database, contains an integration part which contains the sum of the old estimated slack ratio and the new output from the controller. The model is tuned by taking the derivative

$$\frac{dM(k)}{dSR(k)}$$

at the vicinity of M_r. The performance error in period k is given by $E(k) = M(k) - M_r$. This error in miss ratio is the input to the controller which estimates the needed change in slack ratio. The database adds the needed change in slack ratio to the current slack ratio and uses the new slack ratio for the next sampling period.

```
ODBFT(τ, freshness_deadline)
  Assign WHITE to all nodes in the data dependency graph
  Let d be the data item updated by τ
  Put d in queue Q
  while Q ≠ ∅ do
    Let u be the top element from Q, remove u from the queue
    Let τ_u be the transaction associated with u
    deadline(τ_u) = release_time(previous scheduled transaction)
    release_time(τ_u) = deadline(τ_u) − WCET(τ_u)
    priority_queue = AssignPriority(u, freshness_deadline)
    for all v ∈ priority_queue in priority order do
      if color(v) =WHITE then
        color(v) =GRAY
        if changed(v) = true then
          Put v in Q sorted by relation ⊐
        end if
      end if
    end for
    color(u) =BLACK
    Put τ_u in the scheduling queue
  end while
```

Figure 9. The ODBFT algorithm schedules updates in a breadth-first manner.

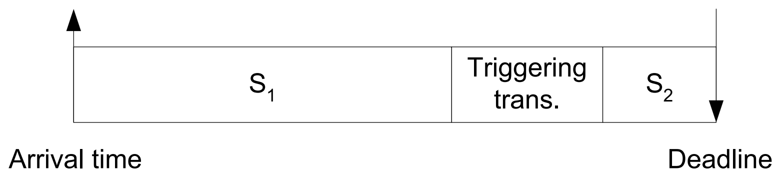


Figure 10. Division of slack. S_1 is for updates and S_2 for restarts and blockings of the triggering transaction.

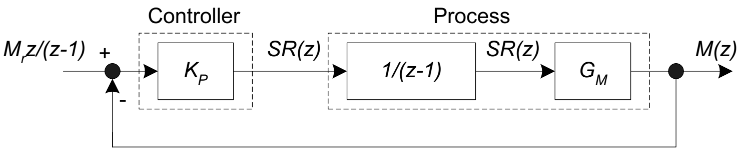


Figure 11. The controller and the controlled system.

The z-transform is used and the transfer function of the system is calculated as:

$$H(z) = \frac{C(z)P(z)}{1 + C(z)P(z)},$$

where $C(z) = K_p$, and $P(z) = \frac{G_M}{z-1}$. The system is stable if the poles of the denominator reside inside the unit circle. By applying a step function to the input we get the following expression of $SR(z) = H(z)\frac{z}{z-1}$. The denominator is now

$$z - 1 + K_p G_M.$$

Set the denominator equal to zero and solve for z. We have that $z = 1 - K_p G_M$, which implies that z is always less than 1 if $K_p G_M < 0$. Now we have to make sure z is greater than -1, i.e., $1 - K_p G_M > -1$, and, thus, $2 > K_p G_M$.

The value of G_M is chosen in section 5.7 by doing simulations with different values of the slack ratio. K_p is then determined based on the chosen G_M.

4.9 Discussion

The base item updates can be handled in two different ways. Either every base item update is one single transaction scheduled in the high priority queue as a normal transaction, or a single process takes care of updating all base items periodically. A similar problem is discussed in [9], where it is found that there is not a significant difference between the two approaches, at least in the setup with a disk-resident database. In [2] it is argued that the updates should be handled by one single process, because the overhead of one single transaction for each base item update would be too great. In the simulations reported in section 5, the base item updates are modeled as transactions.

If the difference $deadline(\tau) - release\_time(\tau)$ is large compared to the duration the data items are fresh, then it is not optimal to move the transactions as close to $release\_time(\tau)$ as possible, since the time from commit time to deadline of triggering transactions is large. As mentioned above it is difficult to know the actual commit time at scheduling time. The commit time can be approximated and used as the freshness deadline, but in this work the deadline is used as the freshness deadline of data items.

For soft transactions, i.e., triggering transactions with low priority, both the commit time and the deadline are not fixed; they depend on the arrivals of higher priority transactions. At commit time the transaction has to check if the expected error of the derived data item is acceptable. The freshness of the values used in the transaction can have been degraded considerably due to interleavings of the arrivals of higher priority transactions. If the expected error is not acceptable the transaction has to be restarted.

5 Performance Evaluation

This section describes the simulator, possible parameter settings, and the setup that has been used during performance evaluation. The setup of the simulator can be found in subsection 5.1. Two different families of baseline algorithms are described in subsection 5.2. The two families differ in how the freshness guarantee is handled.

The simulations are described in sections 5.3–5.8. The simulations that are performed and evaluated are:

- **Sensor transactions and user transactions**. This simulation investigates the effect the size of the database has on the throughput of user transactions at different loads. The simulation can be found in section 5.3.

- **Throughput of user transactions**. The miss ratio of user transactions is investigated. Simulations for different arrival rates are conducted and the baseline algorithms are compared to ODDFT and ODBFT. The simulation can be found in section 5.4.

- **Weights**. Every data item can be annotated with a weight symbolizing how important the data item is for the derivations. ODDFT and ODBFT take the weights into consideration during scheduling. This experiment checks if there is a noticeable difference between the proposed algorithms and the baselines. The simulation can be found in section 5.5.

- **Transient and steady state**. Triggered transactions constitute excess load on the system. This experiment examines how many triggered transactions the algorithms generate during different system states. The simulations can be found in section 5.6.

- **Feedback control scheduling of slack**. As simulations in section 5.4 show, ODDFT and ODBFT has a higher miss ratio of user transactions than some of the baseline algorithms. One way to address this problem is to use a controller that allocates a certain amount of the available slack time to the triggering transaction. This experiment tests whether the controller can decrease the miss ratio or not. The simulation can be found in section 5.7.

- **Overload**. The proposed algorithms add CPU overhead due to added complexity of data structures, such as traversal of the data dependency graph, the use of the error function, and maintenance of the change flag. The simulation can be found in section 5.8.

5.1 Simulator Setup

The simulator is a discrete-event simulator called RADEx that is used for instance in [12, 24]. The simulator setup is to simulate a soft real-time main-memory database. The transaction model in section 4.1.3 uses two transaction queues. Two queues are used in the simulator, base item updates in the high priority queue and user transactions in the low priority queue. The sensor transactions execute with higher priority than the user transactions. The updating frequency of sensor transactions are determined by their avi. The arrival times of user transactions are determined by the arrival rate and an exponential distribution. The database consists of data items taken from two sets. Either a data item is a base item b, that holds a sensor value, or a data item is a derived item d that holds the result of a calculation using one or more other data items. The updating of a data item takes time and this is modeled with the constants `STProcCPU` and `ProcCPU` that denote the maximum time it takes to write a sensor value (a write operation), and to read or write (a read/write operation) a derived value respectively. The average execution time of an operation on a data item is randomly set during initialization of the simulator. The average execution time for an operation in a sensor transaction is in the interval [0,`STProcCPU`] and in the interval [0,`ProcCPU`] for a user transaction. The actual execution time of an operation is drawn from a standard distribution $N(\mu, \sigma)$ with the expectation μ set to the average execution time decided during initialization. The standard deviation σ is set to one fourth of the maximum possible execution time for user transactions and zero for sensor transactions. The relative deadline of a transaction is calculated by taking the WCET of the transaction times a random value in the interval [2,8].

A sensor transaction is executed periodically based on the avi of the base item the sensor transaction updates. User transactions arrive aperiodically from an exponential distribution and the derived data item a user transaction updates is randomly chosen from the set of all derived data items. This scenario, with the same likelihood for all derived data items to be in a user transaction, might not be found in a real-life application, since it might be more likely that a leaf node in the data dependency graph is needed by the system, e.g., an actuator value. Different probabilities for data items membership of

No real calculations are made as a data item is updated. To be able to simulate changing values, all data items have an upper bound on how much the value can change during an avi. When a data item is updated its timestamp is set to the current time and its value is increased with an amount that is randomly picked in the interval [0,`max_change`] and multiplied with the fraction $(timestamp - previous\_timestamp)/avi$, i.e., the value of a data item can at maximum change `max_change` during its avi. The new value of the data item is the sum of the previous value plus the change.

A user transaction updating data item d_i only writes a new value to the database when at least one of the data items d_j that d_i is derived from have changed.

When a data item d_i gets updated its old value is stored as an outdated version, v, in the database. Associated with version v is the versions of all d_j that were used during the derivation of d_i. When a user transaction commits, it is checked whether its parents have been updated or not, and if their values are still within the data validity bounds. Also during commit, it is checked whether all d_j are consistent according to their avis. Two different ratios are constructed in the simulations from these values. The first ratio is the number of committed transactions that are valid, either based on data validity bounds or avi, divided by the maximum possible number of committed transactions. The maximum possible number of committed transactions is taken from the simulation without any updates at all, since more committed transactions than in this case cannot be achieved with triggering of updates enabled. The second ratio is the number of committed valid transactions divided by the number of committed transactions.

Table 1 summarizes the parameters used in the simulator. In table 1, UT denotes a user transaction and ST denotes a sensor transaction. The $factor$ in the entry $\delta_{i,j}$ is described in section 5.6. The setting of the sensor transaction parameters make sensor transactions to always take `STProcCPU` time to execute.

Table 1. Parameters for the database simulator.

Parameter	Value
avi	uniform(200,800) msec
arrival rate user trans.	[0,100] exponential distr.
$\delta_{i,j}$	factor×avi
`max_change`	uniform(200,800)
slack (UT)	uniform(1,7)×WCET msec
`ProcCPU`	10 msec
`STProcCPU`	1 msec
mean processing time ($\bar{e}$) (UT)	uniform(0,`ProcCPU`) ms
std deviation processing time (UT)	2.5
processing time (UT)	$N(\bar{e}, 2.5)$ msec
mean processing time (ST)	`STProcCPU` msec
std deviation processing time (ST)	0
processing time (ST)	`STProcCPU` msec

The dependencies among the data items can be viewed as a directed acyclic graph (DAG). A random DAG, i.e., the layout of the database, is generated by a program. The parameters that can be set for the DAG generator are:

- Number of data items in the database.
- The ratio of base items and derived items.

- The cardinality of the read set $R(d)$ of a derived item d, i.e., the maximum number of data items that have to be read in order to derive a new value of the data item. The cardinality of a derived data item is taken from a uniform distribution. The same distribution is used for all derived data items d.

- The likelihood that a member of $R(d)$ is a base item.

The values of the parameters for generating a DAG is summarized in table 2.

Table 2. Parameters for the DAG generator.

Parameter	DB 20×20	DB 20×60	DB 45×105
Number of data items	40	80	150
Ratio sensor items	0.5	0.25	0.3
Ratio derived items	0.5	0.75	0.7
Max number items in read set	6	6	6
Probability read set member is sensor item	0.6	0.6	0.6

In the simulations performed in the simulator, the database size is given as $|B| \times |D|$, where $|B|$ is the number of base items, and $|D|$ is the number of derived items. When a database is given, such as 45×105, the DAG has been generated once by the DAG generator, i.e., it is the same database that is used in all experiments.

5.2 Algorithms for Generating Updates

The algorithms proposed in section 4, ODDFT and ODBFT, are on-demand based; an update of a data item is put in the schedule when the *change* of the data item is true and the value of the data item might change if it is recalculated, i.e, the change approximated by *error* is above the data validity bounds of the children. The schedule is generated when it is needed, i.e., when a data item is updated by a transaction, all the parents of the data item are checked for possible change. Simulation results for the proposed algorithms should, thus, be compared to other on-demand algorithms.

The on-demand algorithms described in [3] are used as baselines. In [3] a triggered update is generated as soon as the age of the data item that a transaction accesses is found to be greater than the largest age of its read-set members. Three strategies for generating triggered updates are evaluated:

- No check. A triggered update is always generated. This option is consistency-centric since an update is generated as soon as a data item is found to be stale.

- Optimistic. The summation of past waiting time and the execution time of a triggered transaction is compared to the slack time. If the slack time is greater there is enough free time to execute the triggered update. This option is throughput-centric since updates are not generated when there is not enough available time for them to execute, i.e., a user transaction has priority over updates even though the user transaction produces a value from stale data.

- Knowledge-based. Uses knowledge of past waiting time to predict the remaining response time. If the sum of the past waiting time, the execution time of the triggered update, and the remaining response time is less than the slack, the triggered update is executed. This option is also throughput-centric.

If a high number of committed transactions is important then the knowledge-based option should be used, since available slack time, waiting time, and remaining waiting time is used to decide if the triggered update can finish its execution. On the other hand, if valid transactions are important, then the no option should be used, since an update for a too old data item will always be generated, and therefore a high degree of the transactions that are able to commit are valid.

Moreover, a family of on-demand algorithms, with the same options as in [3] (no check option, optimistic, and knowledge-based), but the triggering of updates is decided based on the data validity bounds, *changed* and approximated error instead of time (*avi*) is also used as baseline algorithms. A triggered transaction updating data item d_i is generated when d_i is requested and at least one of the parents d_j of d_i has *changed* set and *error* outside δ_{d_i,d_j}, $d_\in R(d_i)$.

All algorithms that trigger updates based on the *changed*-flag, approximated error, and data validity bounds, i.e., data freshness as defined in definition 4.6, are said to be value-aware, whereas algorithms that trigger updates based on age are age-aware.

Table 3. Notation of algorithms.

	On-demand			Proposed algorithms	
	no option	optimistic	knowledge-based	ODDFT	ODBFT
time based	OD	ODO	ODKB		
value based	OD_V	OD_V	ODKB_V	ODDFT	ODBFT

The on-demand algorithms, OD, ODO, and OD_V, described above, only update data items when necessary. These algorithms do not take how long the data item is valid into consideration. The proposed algorithms, ODDFT and ODBFT, both try to make the used data items valid at least until the deadline of the transaction. The commit time cannot be used since it is not known at the time of scheduling the updates (see section 4.3). To be able to compare the on-demand algorithms and the proposed algorithms, one additional category of on-demand algorithms is implemented in the simulator. Now, an update is triggered when a read data item will be outdated at the deadline of the transaction. The string _AD is appended to the algorithm name from table 3 to denote such an algorithm, e.g., ODKB_AD for on-demand updating for valid data items at deadline of transactions, and knowledge-based option. In the following sections, the algorithms are denoted as in table 3.

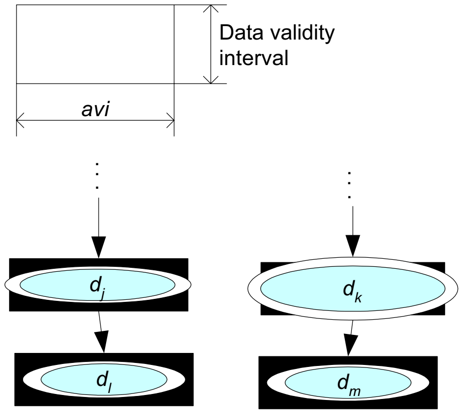


Figure 12. The x-axis of the ellipses is the age of the data item and the y-axis is the value from the error function. A black rectangle gives the avi and the data validity bound (here we assume it is the same for all children).

Two examples are given from figure 12. Colored ellipses gives the age in x-axis and error in y-axis at $t = 7$ and white ellipses at $t = 10$. Black rectangles give in x-axis the avi and in y-axis the data validity bound for the children. All data items in figure 12 have *changed* set to true, i.e., any of the parents have been updated. Suppose a user transaction arrives and uses d_m, OD, ODO, and ODKB would not generate an update for the parent d_3 since the x-axis of the colored ellipsis for d_k is inside the black rectangle's x-axis which is the avi of the data item. OD_AD, ODO_AD, and ODKB_AD on the other hand would generate an update of d_k since the white ellipsis is outside the rectangle (x-axis is only considered), which is the age of d_k at the deadline of the transaction. ODDFT and ODBFT would both generate updates of d_k since the y-axis of the white ellipsis is outside the y-axis of the rectangles, which is the δ_{d_m,d_k}. If the transaction instead updates d_l, then as before, OD, ODO, and ODKB would not generate an update of d_j since its colored ellipsis is within the avi. The _AD versions of the on-demand algorithms would generate an update of d_j, but ODDFT and ODBFT would not since the error of d_j as calculated by *error* is within the limit δ_{d_l,d_j}, which is pictured by the white d_j ellipsis is within the rectangle on the y-axis.

The release time test in Dispatch described in section 4.5 is used. The concurrency control algorithm is 2PL-HP, and the transactions are scheduled according to EDF. It is shown in [21] that optimistic concurrency control is better than lock-based concurrency control in a setting with periodic transactions. It is also found that optimistic concurrency control is poor at maintaining temporal consistency. Here we only use 2PL-HP, but how concurrency control algorithms are affected by the introduction of data validity bounds need to be investigated. Sensor transactions always execute before user transactions. Where otherwise stated, the weights on all data items are set to

1. All data items have the same error function. It is defined as:

$$error(x,t) = (t - timestamp(x)) \times c,$$

where c is set to 1 for all data items, i.e., the data items age, in the worst case scenario, linearly with time. This makes *error* behave as the *avi* of a data item, because the change in value of a data item (`max_change`) is in the same range as the *avi*, uniform(200,800) in both cases, and the value of a data item is the sum of all changes. The purpose with *error* is to get an upper bound of how much the value of a data item has changed at a given future time. The *avi* of a data item is a pessimistic approximation, based on time, on how long time the value of a data item is valid, i.e., the same as *error*.

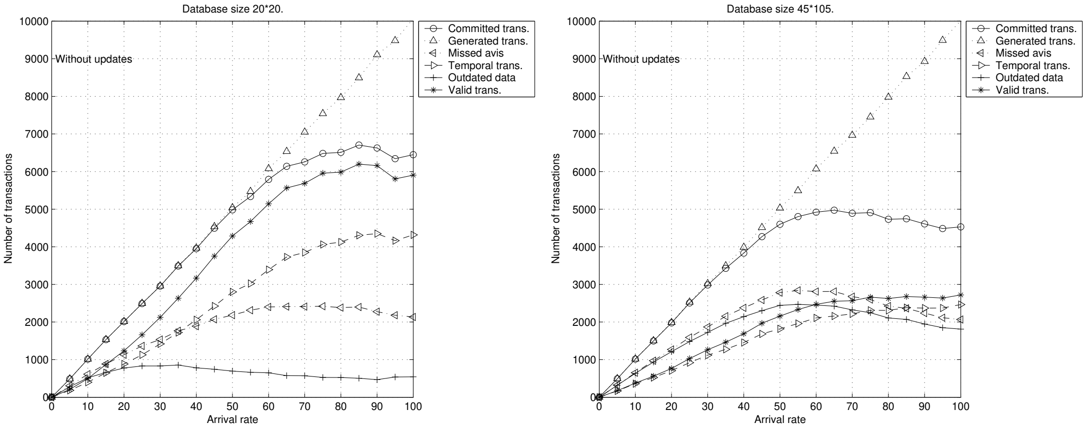
The simulations are conducted 5 times for each arrival rate, and this gives a confidence interval of ...

5.3 Sensor Transactions and User Transactions

The update transactions for the base items have the highest priority and, thus, always interrupt executing user transactions. The load put on the system by update transactions for base items depends on the number of base items in the database. Figure 13 shows two different sizes of a database, 40 data items in total where 20 are base items in figure 13(a), and 150 data items among which 45 are base items in figure 13(b). Stale time domain is the number of transactions where at least one of the read set members is invalid based on *avi* at commit time. Temporal transactions is the number of committed transactions that have not missed any *avi*. Stale value domain refers to the number of user transactions where at least one of the read data items has a new value that is outside the data validity bound at commit time. Valid transactions is the number of committed transactions that use valid data according to definition 4.6. The data validity bound is set to match the *avi* of the data items, i.e., the factor for the $\delta_{d,x}$ entry in table 1 is set to one since the `max_change` value is derived from the same interval as the *avi*.

The number of generated base item update transactions for each arrival rate is 8095 for the database with size 40 and 18897 for the database with size 150. The number of generated user transactions depends on the arrival rate, and, as can be seen in the figures, increases linearly with increasing arrival rate. Above, roughly, an arrival rate of 50 user transactions per second, there is not enough time to execute all user transactions, i.e., the number of committed transactions diverges from the number of arriving user transactions. The higher number of generated base item update transactions in the larger database makes less number of user transactions to commit. The top number of committed user transactions are 6000 in the small database, and 5000 in the large database.

For a small database, as in figure 13(a), the database is kept up to date at high arrival rates. The number of outdated committed user transactions is almost constant at arrival rates above 30 transactions per second. When a user transaction arrives to the system it is randomly determined which derived data item it updates. When the arrival rate is high, the number of arriving user transactions is sufficient to keep the database up to date. As can be seen in figure 13(b) the outdated committed user transactions increase as the arriving user transaction increase.



(a) 20 base item

(b) 45 base item

Figure 13. Load generated by base item update transactions and user transactions.

An upper limit of outdated transactions as for the small database is not reached. Remember that in this setup, the number of transactions are the sum of sensor and user transactions. No triggered updates are generated. Therefore, the number of committed user transactions cannot be higher than in this simulation for any size of the database.

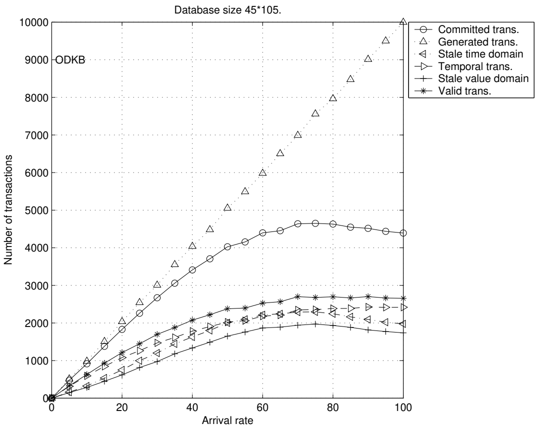
5.4 Throughput of User Transactions

In this set of experiments the goal is to investigate what the number of valid committed user transactions for each algorithm is. The simulator runs the simulated database for 100000 msec at a specific arrival rate. The performance of ODBFT is not showed since it behaves as ODDFT (see figure 15(a)).

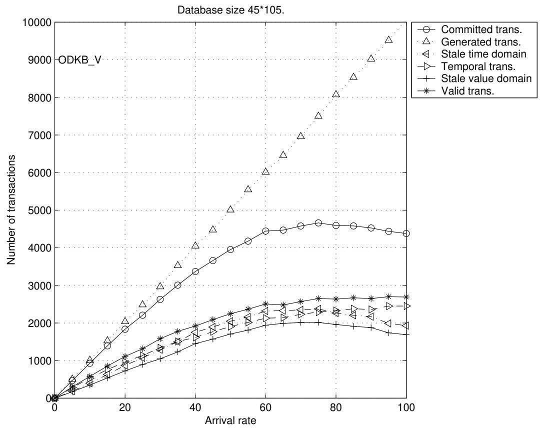
5.4.1 Optimistic and Knowledge-Based Option

Figure 14 shows how the algorithms perform at arrival rates from 0 to 100 user transactions per second. At arrival rates above 25 user transactions per second the ODDFT algorithm falls behind the other algorithms when it comes to committed user transactions see figure 14(d). This is discussed and solved in section 5.7. Note, since the knowledge-based option is used it is hard to have more committed user transactions than ODKB and ODKB_V, because the knowledge-based algorithms skip updates if the user transaction cannot finish in time, i.e., a user transaction is prioritized over an update even though the user transaction will be invalid when it commits.

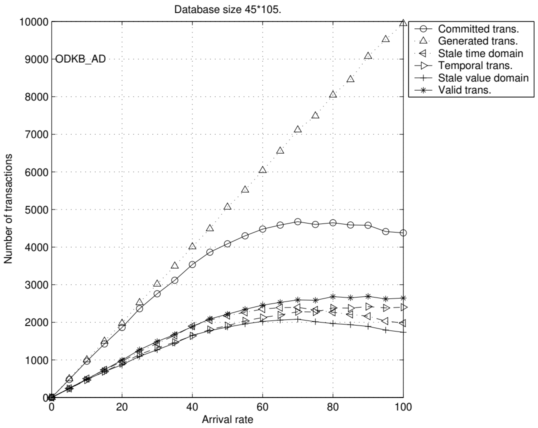
Figure 15 shows in 15(a) and 15(c) the ratio of committed user transactions and possible committed user transactions. The number of possible committed user transactions are taken from a simulation with no updates, and this is the highest possible number of committed user transactions for an arrival rate of committed user transactions. Figures 15(b) and 15(d) show the ratio of valid committed user transactions and committed user



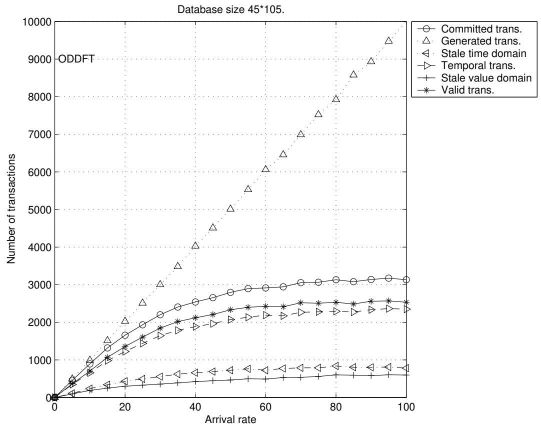
(a) On-demand Knowledge-based option and *avi* (ODKB). 95% confidence interval for all plots ± 172.9.



(b) On-demand Knowledge-based option and validity bound (ODKB_V). 95% confidence interval for all plots ± 238.1.

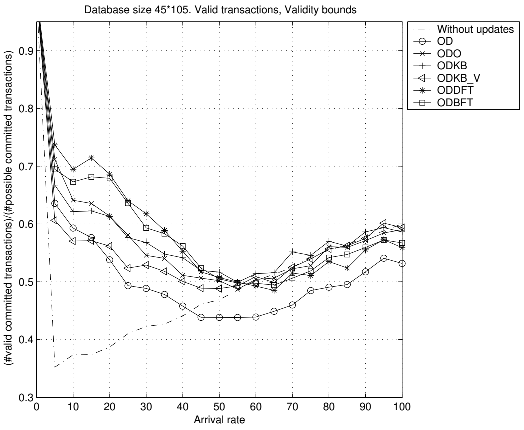


(c) On-demand Knowledge-based and *avi* at deadline of user transaction (ODKB_AD). 95% confidence interval for all plots ± 165.9.

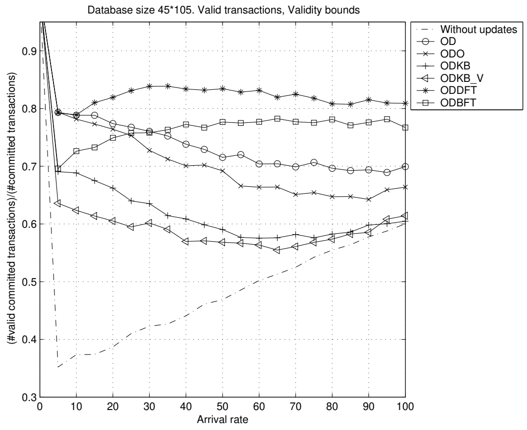


(d) Depth-first scheduling (ODDFT). 95% confidence interval for all plots ± 134.4.

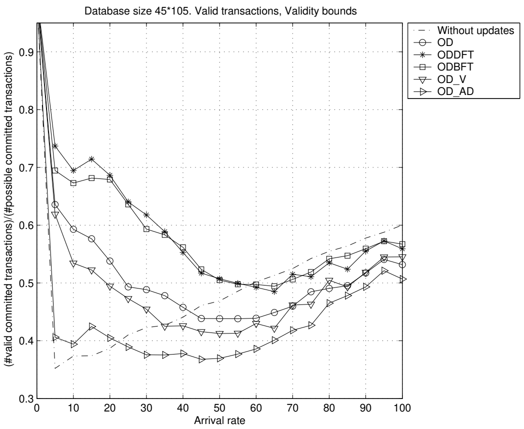
Figure 14. Performance of algorithms at different arrival rates for a database of size 45$\times$105. The data validity bounds are set to match the avis.



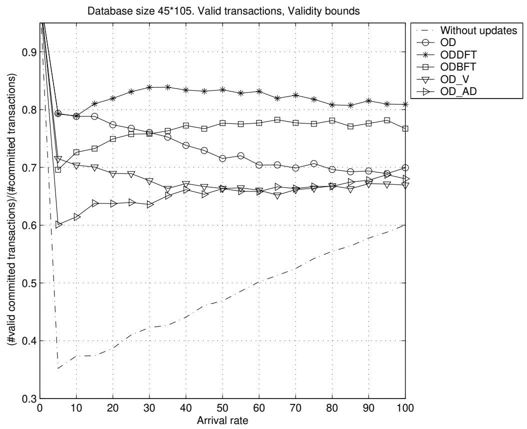
(a) Ratio of valid committed user transaction out of possible number of committed user transactions. 95% confidence interval for all plots ± 0.051.



(b) Ratio of valid committed user transactions. 95% confidence interval for all plots ± 0.026.



(c) Ratio of valid committed user transaction out of possible number of committed user transactions. 95% confidence interval for all plots ± 0.051.



(d) Ratio of valid committed user transactions. 95% confidence interval for all plots ± 0.026.

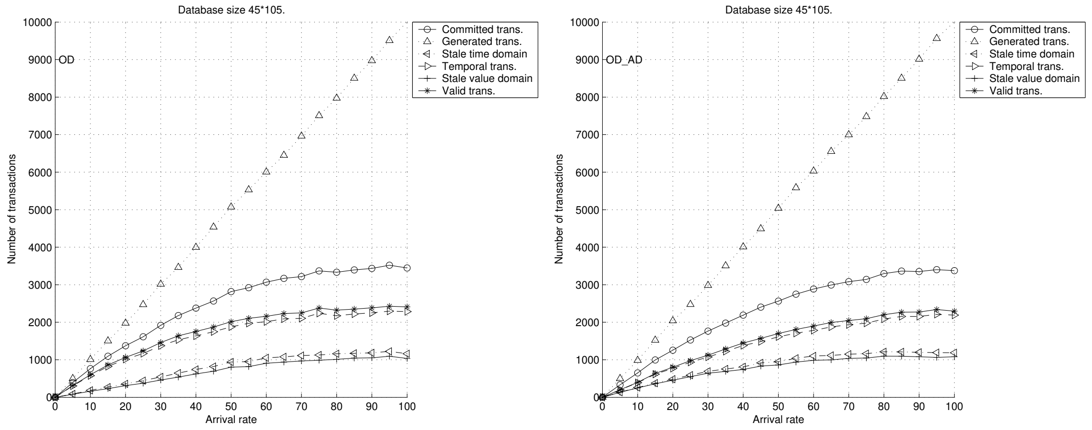
Figure 15. Ratio of valid committed user transactions.

transactions for specific algorithms.

Studying figure 15, we see that for arrival rates in the interval [0,45], ODDFT and ODBFT make the highest number of valid user transactions committed (see figure 15(a)). Figure 15(b) shows that out of the transactions that commit for each algorithm, ODDFT and ODBFT have the most valid transactions.

5.4.2 No Option

The no option presented in [3] generates a triggered update as soon as a requested data item is found to be too old. This option can be used if temporal consistency is of prime importance, i.e., it is more important to have valid committed user transactions than a high throughput of user transactions. However, as can be seen in figure 15(a), OD has worse consistency than ODKB and ODKB_V, since the ratio is lower than for ODKB and ODKB_V. This is because of the larger number of generated triggered updates. A higher ratio of the committed transactions are valid for no option, figure 15(d), compared to knowledge-based, figure 15(b).

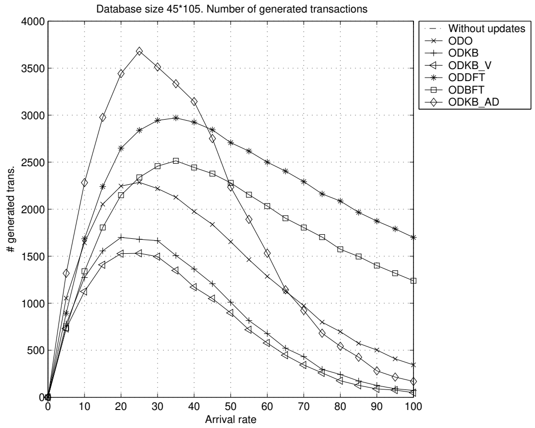


(a) On-demand no option. 95% confidence interval for all plots ±159.3.

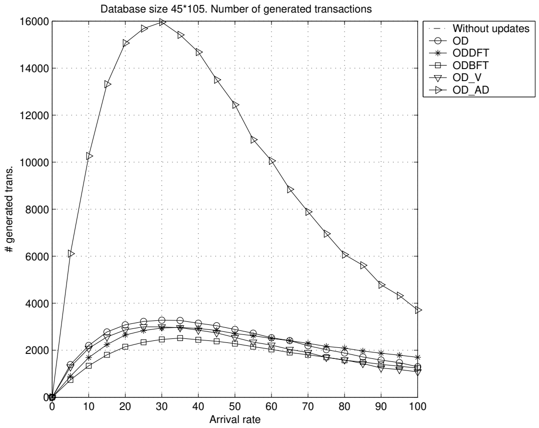
(b) On-demand no option and age at deadline. 95% confidence interval for all plots ±164.8.

Figure 16. On-demand no option algorithms.

Figure 16 shows that the performance of ODDFT (figure 14(d)) and the on-demand no option algorithms are roughly the same at high arrival rates. More committed user transactions are valid under ODDFT and ODBFT compared to on-demand no option (figure 15(c)). Out of the committed user transactions a higher degree of them are valid under ODDFT and ODBFT (figure 15(d)), even though a smaller number of triggered updates are generated (figure 17). This shows that ODDFT and ODBFT perform better than on-demand algorithms with no option.



(a) Number of generated updates for optimistic, knowledge-based and proposed algorithms. 95% confidence interval for plots are ±172.9.



(b) Number of generated updates for no option, and proposed algorithms. 95% confidence interval for plots are ±200.0.

Figure 17. Number of generated triggered transactions.

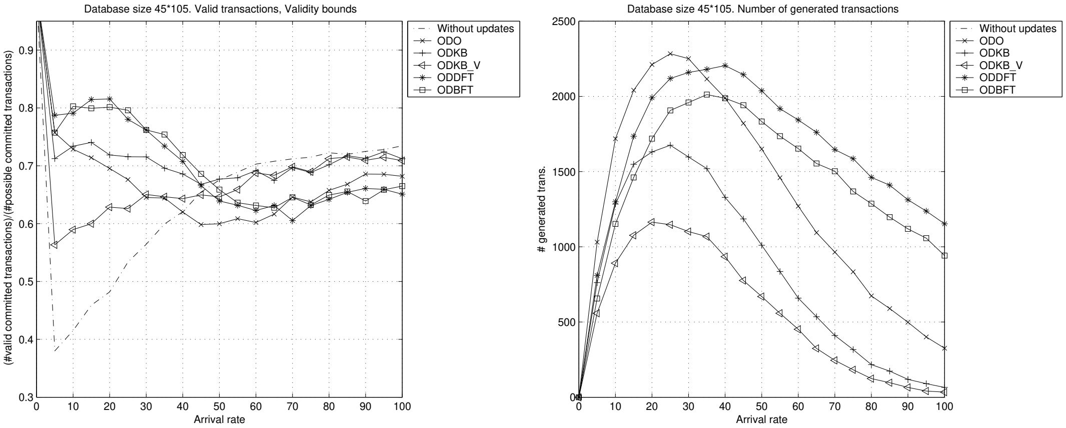
5.4.3 Pessimistic Absolute Validity Interval

The on-demand algorithm that triggers on outdated parents performs in the setup with $factor$ set to 1 slightly better than an on-demand algorithm triggering on too old data. The reason is that the data validity bounds on the data items matches the *avi*. If the *avi* is pessimistic, i.e., the data items lives generally longer than the *avi*, then the difference between the two types of on-demand algorithm should have been larger. This is shown below.

Figure 18 shows the impact of pessimistic *avi*s. The data validity bounds are set to 3×`max_change`, and since `max_change` and *avi* are derived from the same distribution, uniform(200,800), the *avi*s are now pessimistic. By comparing figure 18(a) with 15(a) and 15(c), we see that all algorithms perform better than before; the ratios are higher. This is because of the larger data validity bounds that make a data item live longer. The age-aware algorithms cannot take advantage of that the data items live longer. Figure 18(b) and figure 17 show that the number of generated triggered transactions has decreased for the data validity bound algorithms.

5.4.4 Results

ODDFT and ODBFT perform better than the no option of the on-demand algorithm, which is the consistency-centric option. Less number of triggered transactions are generated and a higher number of valid user transactions commit. Compared to the knowledge-based option of the on-demand algorithm, ODDFT and ODBFT generate more triggered transactions, and at high arrival rates a less number of valid user transactions are committed. However, at moderate arrival rates, ODDFT and ODBFT have more valid transactions than the knowledge-based



(a) Ratio of committed valid user transactions. 95% confidence for plots are ± 0.059.

(b) Generated triggered transactions. 95% confidence interval for plots are ± 79.2.

Figure 18. avi**s are pessimistic.**

option. When the avis on data items are too pessimistic, ODDFT and ODBFT are not affected.

5.5 Weights

In this experiment the weights of data items are set randomly by taking a number n from a normal distribution, $N(2.0, 3.0)$. If the picked number n is less than 1, then the weight is set to 1, otherwise stick with n.

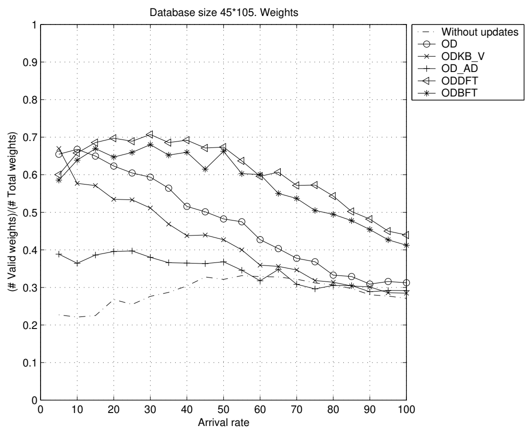


Figure 19. Weights other than 1.0 on data items. One simulation run for each arrival rate.

ODDFT and ODBFT take weights into consideration when prioritizing updates. The AssignPriority algorithm that both ODDFT and ODBFT use, multiply the priority returned by the error function with the weight on the data item. The on-demand algorithm with any option does not use a weight on the data items.

The metric in this experiment is the weight ratio:

$$WR(d) = \frac{\sum_{\{x:\forall x\in R(d), |x-x'|\leq \delta_{d,x}\}} weight(x)}{\sum_{\forall x\in R(d)} weight(x)},$$

where x' is the value the previous derivation of d used, i.e., the sum of the weights for all valid parents $x \in R(d)$ divided by the sum of the weights for all parents.

Figure 19 shows that for a low arrival rate, ODDFT, ODBFT, OD, and ODKB_V all have the same ratio. At this low arrival rate, there is time available to execute updates in, and almost all updates have time to complete. Thus the same number of updates are executed, and therefore there is no difference between the algorithms. As the arrival rate increases and the available time for updates decreases, ODDFT and ODBFT can keep the ratio at the initial level for higher arrival rates than the other algorithms. For instance, at an arrival rate of 40 user transactions per second, the difference between ODDFT and OD is 17 percentage points, but in figure 15(d) the difference is 10 percentage points. The performance in number of committed user transactions is the same for ODDFT and OD, see figure 14(d) and 16(a) respectively. The 7 percentage points that ODDFT performs better than OD comes from the fact that ODDFT uses weights when scheduling updates, whereas OD does not.

5.6 Transient and Steady States

The idea with the data validity bounds on the data items is to capture the fact that values of the base items can change differently during the execution of the application. At transient states the base items might change rapidly and much, then the derived items need to be recalculated often. At steady state the base items change less frequently. Hence, the derived data items need to be updated less often. When *avi*s are used on the data items, then it is difficult to capture these dynamic changes. One way, of course, is to change the *avi* of the data items when the state of the external environment changes. The application then needs to monitor the changes from one state to the other, which is not needed when data validity bounds are used.

One measure of load put on the system is the number of generated triggered updates. The number of generated sensor transactions and user transactions are constant for all algorithms at a specific arrival rate. To investigate if the value-aware algorithms generate less number of triggered transactions during steady state than the age-aware algorithms, the following simulation was conducted.

- The arrival rate is kept at 30 user transactions per second throughout the simulation. The size of the database is 45×105.
- The simulation is executed for 100000 msec.
- Two parameters are introduced: `change_speed_of_sensors`, and `change_speed_of_user_trans`. These parameters are used to set a point in the interval [0, `max_change`] that a data item increases its value with once the data item is updated. The increase in the value of a data item is determined by

N(max_change/change_speed_X, max_change/(2×change_speed_X))

The first parameter is the average the distribution has, and the second parameter is the standard deviation of the distribution. The _X in change_speed_X is substituted either for _sensors or _user_trans.

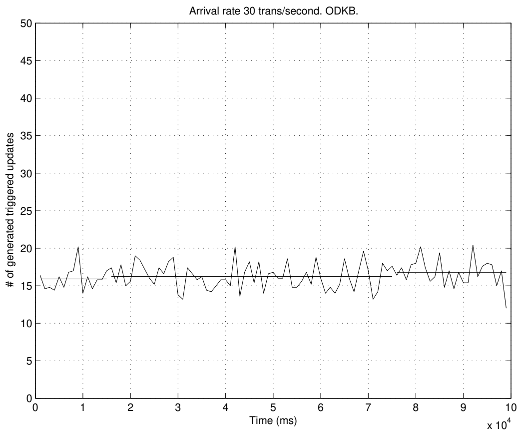
- The change_speed_of_sensors has initially, for the first 15000 msec, a low value (it is set to 1.2) which implies rapid changes.
- At 15000 msec the change_speed_of_sensors parameter is changed to 50.0 which implies small changes in the base items, i.e., a steady state is reached. This should be reflected in the number of generated triggered updates for the value-aware algorithms, because the values of the base items are kept within the validity bounds a longer time.
- At 75000 msec the change_speed_of_sensors parameter is changed to 2.0. A transient state is reached again. The changes are not as rapid as during the first 15000 msec, but the changes are much more frequent than in the steady state.
- During the simulation the parameter change_speed_of_user_trans has a value of 2.0.

Figure 20 contains the simulation results from the abovementioned simulation. The horizontal lines are the average number of generated triggered transactions during the interval. The value-aware algorithms (figure 20(b) (ODKB_V) and 20(d) (ODDFT)) clearly generate less number of triggered updates during the interval 15000–75000 msec than compared to the number of triggered updates that is generated during the transient states.[6] No such difference can be seen for the age-aware algorithms (figure 20(a) (ODKB) and 20(c) (ODKB_AD)). Two of the four algorithms, ODKB_AD and ODDFT, try to make sure that the read data items are valid at least until the deadline of the user transaction. The other two algorithms, ODKB and ODKB_V, only make sure a read data item is up to date, but not until the deadline of the user transaction. From the number of generated triggered updates in figures 20(c) and 20(d), compared to those in figures 20(a) and 20(b), it notably requires more updates for the former algorithms. This is due to the larger probability that a data item is outdated in a future point in time, since there is more time where the data item can change. It is hard to compare the algorithms since they focus on different things.

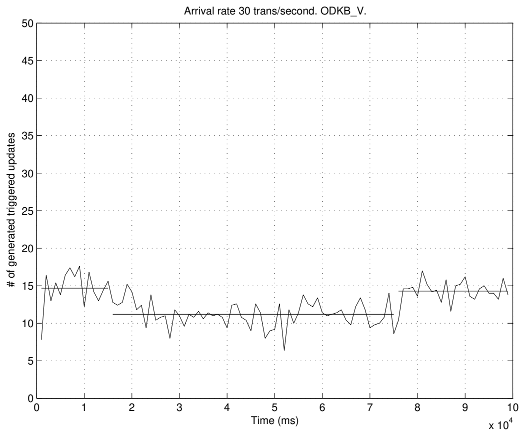
Table 4 lists the overall statistics for the total simulation time 0–100000 msec. At the slightly low arrival rate, 30 user transactions per second, there is time available for updates. This fact can be seen in figure 13 since without updates at an arrival rate of 30 user transactions per second almost all arrived user transactions are able to meet their deadlines (see figure 13(b)).

ODDFT has the lowest number of committed transactions, but on the other hand it has the highest number of committed transactions that are valid based on the data validity bound. If it is important to have a high number

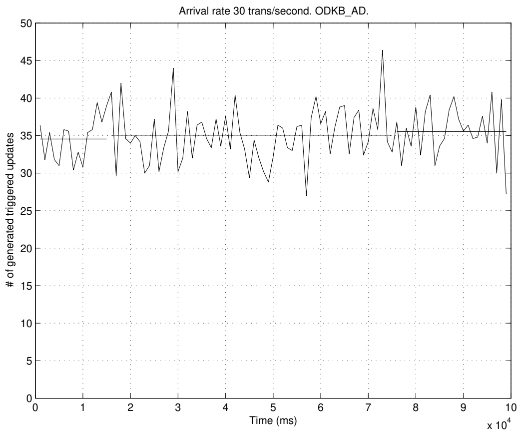
[6] The difference in mean is due to statistical differences.



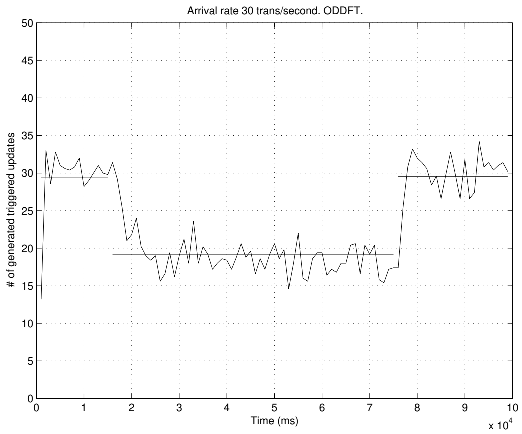
(a) On-demand remaining waiting time and *avi*



(b) On-demand remaining waiting time and data validity bound



(c) On-demand remaining waiting time and *avi* at deadline of user transaction



(d) Depth-first scheduling

Figure 20. Number of generated triggered updates for four different algorithms. One simulation run for each arrival rate.

of committed user transactions, then ODDFT is not a good algorithm. If it is important to, at all times, have valid user transactions, then ODDFT is the best algorithm. Table 5 shows the percentage of valid committed transactions for the different algorithms.

Table 4. Overall statistics for transient and steady state simulation. Arrival rate 30 transactions per second. Database size 45×105.

	ODKB	ODKB_V	ODKB_AD
# generated user trans.	2997.2 ± 23.9	2995.4 ± 54.4	3013.0 ± 53.0
# committed user trans.	2650.8 ± 46.4	2666.2 ± 39.7	2758.6 ± 70.5
# valid trans. validity b	1999.8 ± 31.0	2028.0 ± 9.3	1876.0 ± 69.9
# valid trans. *avi*	1452.8 ± 25.0	1255.6 ± 19.9	1310.0 ± 33.4
	ODDFT	Without updates	
# generated user trans.	3005.0 ± 47.7	3037.8 ± 20.8	
# committed user trans.	2330.8 ± 79.1	3004.4 ± 21.4	
# valid trans. validity b	2092.2 ± 97.0	1811.0 ± 30.9	
# valid trans. *avi*	1448.2 ± 44.8	1094.6 ± 22.3	

Table 5. Percentage of committed transactions that are valid. Average values are used.

	ODDFT	ODKB	ODKB_V	ODKB_AD	Without updates
percentage of valid committed tr.	90%	75%	76%	68%	60%

Simulations were performed with the same setup of the simulator where an arrival rate of 60 user transactions per second was used. At this arrival rate, as can be seen in figure 13, there is not enough time to execute all arriving user transactions. From figure 14 we can see that at high arrival rates ODDFT performs badly. The overall simulation results are presented in table 6. The number of valid committed transactions is among the lowest for the ODDFT algorithm. Without updates has among the highest number of valid committed transactions.

The knowledge-based option stops updates to be executed if there is not enough time available. Table 6 shows that ODKB and ODKB_V is much closer to the performance Without updates has than ODDFT. ODKB_AD also has the remaining waiting time estimation, but it cannot perform the same as ODKB and ODKB_V because it generates more triggered updates due to the requirement that data items are valid by *avi* at commit time.

The reason ODDFT performs badly at high arrival rate is that all updates are executed before the user transaction starts to execute (see section 4). This makes it more likely that the most important transaction is executed close to its deadline. Only one interruption from a sensor update and the user transaction might miss its deadline.

In section 5.7, slack is put in the, by ODDFT, generated schedule between the user transaction and the deadline. Now the user transaction has more time to execute in, and it should also meet its deadline more often.

Table 6. Overall statistics for transient and steady state simulation. Arrival rate 60 transactions per second. Database size 45×105.

	ODKB	ODKB_V	ODKB_AD
# generated user trans.	6039.4 ± 71.8	5998.6 ± 119.4	6011.6 ± 29.3
# committed user trans.	4368.4 ± 98.4	4423.2 ± 94.7	4493.8 ± 37.4
# valid trans. validity b	3107.2 ± 85.0	3252.8 ± 35.4	3087.4 ± 44.2
# valid trans. *avi*	2192.4 ± 25.2	2053.0 ± 53.2	2112.2 ± 24.7
	ODDFT	Without updates	
# generated user trans.	6031.0 ± 134.7	5978.0 ± 57.0	
# committed user trans.	3127.8 ± 53.7	4896.0 ± 87.0	
# valid trans. validity b	2749.6 ± 43.3	3210.8 ± 83.5	
# valid trans. *avi*	2054.6 ± 47.3	2052.8 ± 36.4	

The test in the knowledge-based option stops almost all updates. Thus, at high arrival rates it might be better to not run any updates at all. Still ODDFT has the highest percentage of valid transactions of those that commit, 89% in this case. Table 7 shows the results for the different algorithms.

Table 7. Percentage of valid committed transactions. Average values are used.

	ODDFT	ODKB	ODKB_V	ODKB_AD	Without updates
percentage of valid committed tr.	88%	71%	71%	69%	66%

5.7 Feedback Control Scheduling of Slack

As mentioned in section 5.6 the ODDFT algorithm performs bad at high arrival rates, both when it comes to number of committed user transactions and number of valid committed user transactions. One way to resolve this issue is to allot a certain part of the slack time to the generated schedule to make room for interruptions of the user transaction. These interruptions are likely to occur at high arrival rates because a user transaction can arrive with closer deadline than the currently executing user transaction. Triggered updates can also delay the start of execution of an user transaction.

The parameters of the controller described in section 4.8 have to be set. First we start with setting a value on G_M, which represents the database.

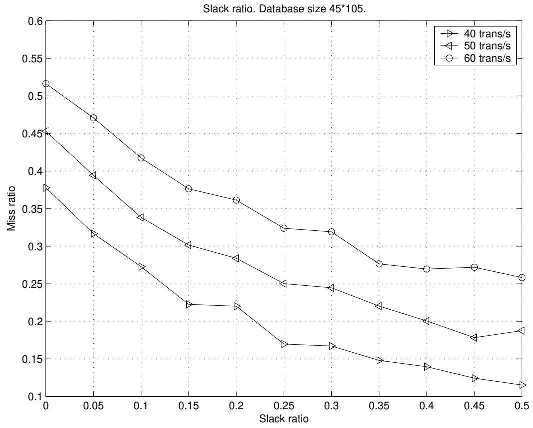


Figure 21. Miss ratio as a function of slack ratio for a database with the size 45×105 and the arrival rate 40, 50, and 60 user transactions per second. One simulation run for each arrival rate.

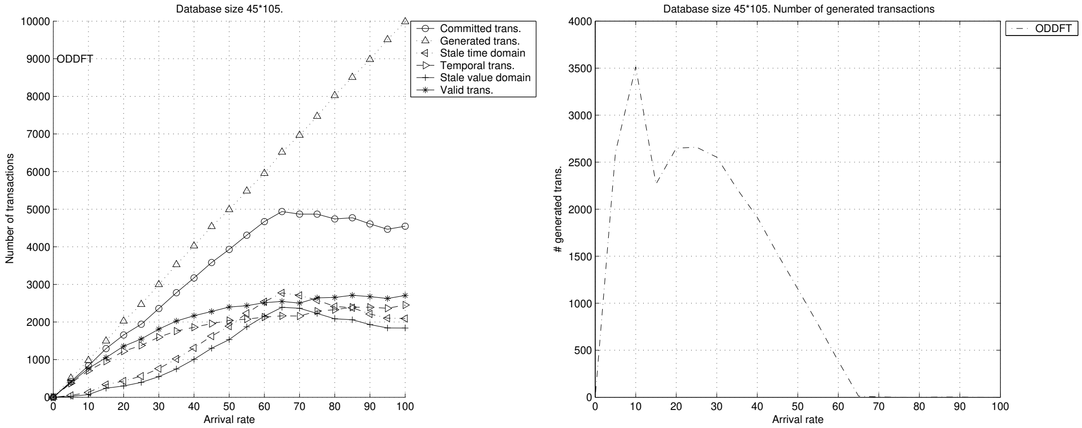
Figure 21 contains three plots at different arrival rates. G_M is calculated by taking the derivative at the chosen miss ratio, 20% in this case, for each arrival rate:

Arrival rate	G_M
40	-0.56
50	-0.5
60	N.A.

Now it is possible to calculate a value for the controller K_p. For the system to be stable K_p should be in the interval $-3.56 > K_p > 0$. $G_M = -0.56$ has been used which is the largest slope in the table above. This gives the lowest $|K_p|$, and, thus, for other values on G_M the poles still reside inside the unit circle. Hence, the system is better designed for systems with other arrival rates than those listed above.

When the feedback control manipulation of the slack ratio is used, the plots in figure 14(d) look now as in figure 22.

Comparing figure 22(a) to 13(b) gives that the number of committed user transactions now is as high as Without updates for high arrival rates. The reason can be seen in figure 22(b), where there are almost no triggered updates at arrival rates above 60 user transactions per second. The miss ratio is above 20% at these arrival rates so the slack ratio is set to a high value which implies no available time for triggered updates. Furthermore, the transient and steady state simulation for the ODDFT algorithm was rerun with the feedback control enabled. For an arrival rate of 30 user transactions per second, the numbers now look as follows: generated user transactions: 3005, committed user transactions: 2391, valid committed user transactions based on *avi*: 1470, valid committed user transactions based on data validity bound: 2149. The numbers are almost the same as in table 4, i.e., for low arrival rates and, thus, low miss ratios the controller do not need to increase the slack ratio because the miss ratio is kept below



(a) Behavior of feedback control at different arrival rates

(b) Number of generated triggered updates

Figure 22. Feedback controlled manipulation of slack ratio for the ODDFT algorithm.

the reference M_r. For an arrival rate of 60 user transactions per second, the numbers are as follows: generated user transactions: 5851, committed user transactions: 4537, valid committed user transactions based on *avi*: 1999, valid committed user transactions based on data validity bound: 3149. The miss ratio is always above the reference M_r and the controller sets a high slack ratio. The numbers are now as the remaining waiting time on-demand algorithms, i.e., almost all updates are denied execution on the system.

One problem that has been noted during the simulations is that it is difficult to measure the miss ratio. A sampling interval of 1 second has been used and still the measured miss ratio is oscillating. The reason is shown in figure 20. Here the number of generated triggered updates is oscillating, suggesting that we have bursty arrivals at workload and this influences the miss ratio, since, as mentioned above, a triggered update can delay the starting of an user transaction. If there are many triggered updates then it is more likely that the user transaction is scheduled close to its deadline, and then it is more likely that the user transaction misses its deadline. A moving average can stabilize the miss ratio signal, but on the other hand, the controller becomes slower. Initial tests with a moving average has been done, but this issue need to be further investigated.

5.8 Overhead

Figure 23 shows the total scheduling time over the whole simulation of several different algorithms. ODDFT and ODBFT have the largest scheduling times. The graph complexity is calculated as the number of edges in all paths from all leaves to the base items. The database size is 20×20 for all graphs, but the graphs are generated with different random seeds to the DAG generator. A highly complex graph contains more and longer paths than less complex graphs. The ODDFT and ODBFT scheduling algorithms traverses the graph and the more complex the graph is the longer time it takes to traverse the graph and construct the update schedule. The overhead time is bounded, though, since the update schedule generation can be stopped when the start time of an update is less than the arrival time of the user transaction.

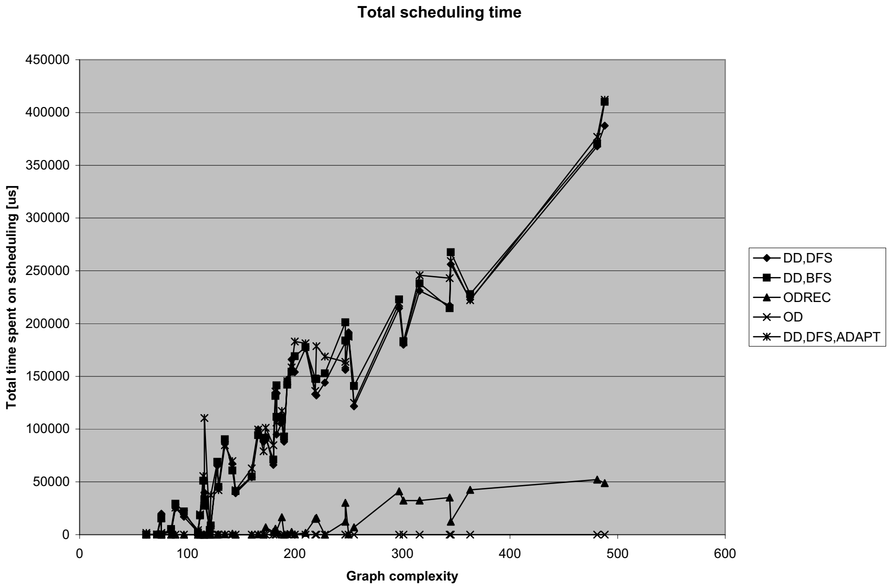


Figure 23. Total scheduling time for different algorithms and database sizes.

No investigations have been done to find out if the ODDFT and ODBFT algorithms still perform better when the overhead is accounted for in RADEx. The simulation in figure 23 is performed in a database that is implemented on top of a real-time operating system [11]. In that particular setup the ODDFT and ODBFT algorithms perform better than the on-demand algorithms, even though the overhead is larger.

The memory overhead of the proposed algorithms in section 4 consists of the following: say that each element in the tuple in section 4.1.2 needs 4 bytes of memory, except $R(d)$ that requires memory proportional to the read set size. Moreover, the previous used value of the parent has to be stored, because it is needed for the data freshness in the value domain (see definition 4.6). Each member of the read set then takes 12 bytes of memory, 4 bytes for x, four bytes for $\delta_{d,x}$, and 4 bytes for the old value. Say that a read set has at maximum a cardinality of 6, then the over head is $6 \times 12 + 32 = 104$ bytes. This is a significant overhead, but it can be reduced. For instance, in the simulator all data items the same error function. This field is thus not necessary. Furthermore, the storing of old values in the read set might not be an overhead, since the data item can request old values and these then have to be stored elsewhere.

6 Related Work

Freshness in the form of intervals of time (absolute validity interval and relative validity interval) has been used in research, e.g., [5, 9, 10, 16, 23–25]. Freshness in the form of value has been used in Kuo and Mok's work on similarity [18], where similarity mainly is used for concurrency control. A similarity bound is an interval of value for a data item where two writes within the same interval are interchangeable. The notion of similarity is thus mapped to an interval in time, much like the absolute validity interval. Similarity is also used for skipping unnecessary transactions in [13]. The frequency of periodic transactions are determined such that they do not produce similar values, i.e., the frequency is determined by the similarity bound. In this paper, similarity is given by the data validity bound $\delta_{d,x}$. Transactions producing similar results, i.e, the distance of the new and the old values are within $\delta_{d,x}$, do not set the change flag of its children in the data dependency graph. Thus, a transaction that uses one of d's children, d', can skip all updates on the path from d' to d. This scheme adapts the frequency of transactions updating a data item to the current state of the system.

In [9] a system with rapid changes in the environment is considered, i.e., updates arrive to the system with a high frequency. It is pointed out that even though a data item remains constant for a period longer than the absolute validity interval (called Maximum Tolerable Delay, MTD, in [9]), the data item is not valid anymore, i.e., only recency (absolute consistency) is considered in [9]. It is assumed that a new update is reported at least once every MTD unit of time. The update reports the same value, it only refreshes the timestamp of the data item, i.e., the data item is fresh with respect to MTD. In this paper updates of data items can be skipped if the parents have not changed. On-demand algorithms for installing the updates are not taken into consideration in [9] due to difficulties in maintaining relative consistency with an on-demand updating scheme. In this paper, data items are relatively consistent as long as all data items in the read set are fresh at the same time.

Two algorithms are presented in [14] that uses the notion of freshness in a feedback control scheduler [19]. The system is divided into periodic update transactions[7] and user transactions. The user transactions use base items to derive a result that is not used by other transactions. The deadlines on the user transactions are firm. The algorithm QMF-1 (a QoS management architecture for Miss ratio and Freshness guarantees) uses two metrics: miss ratio and perceived freshness. QMF-1 divides base items into those that are immediately updated and those that are updated on-demand. If a base item is accessed at least as often as it is updated it is called hot and should be updated immediately, otherwise it is called cold and can be updated on-demand when the system is overloaded. When the system reaches an overload state some immediate updates are degraded to on-demand updates in order to lessen the load. The feedback control scheduler tries not to overshoot miss ratio and perceived freshness requirements that are specified by a database manager.

The other algorithm, QMF-2, uses flexible validity intervals on the sensor updates. That is, definition 4.2 is

[7] Here an update transaction updates a base item, for instance reading a sensor and installing the acquired value in the database.

reformulated to allow for changing validity intervals. Quality of data is defined as the sum of the ratio between the minimum period and the new period for all update transactions. The quality of data is managed by defining which validity intervals of the update transactions that can be changed, how much they can be changed, and by how much each time. When the system becomes overloaded the update transactions whose period can be changed are changed to keep the miss ratio within a given bound.

Derived data items are not used in user transactions in [14] as is allowed in our work. Furthermore, in our work, the validity interval of a data item has no upper bound, since it is solely determined by the data validity bound $\delta_{d,x}$.

Triggered updates are used in [3] to update data items that are too old. A triggered update is generated on-demand when such a data item is accessed. The basis for an update is only the age, defined as the difference of version numbers of a data item, whereas in this paper the expected error is the triggering factor.

If several data items in a read-set need to be updated the triggered update strategies as described in [3] do not consider any priority among the data items. In this paper, the data item with the highest contribution to the total error is given a high priority and is given access to the slack time first. Even if a data item is old, i.e., its *avi* has been passed, it can still reflect the true value reasonably well, this is captured by the function *error* in this paper.

As pointed out in section 4, the scheduling algorithms schedule the transactions as close to the deadline of the arriving transaction as possible and then move the schedule as close to the release time as possible. Actually, the freshness of data items would be better if the triggering transactions could be executed as close to the deadline as possible, since the amount of time the data items can diverge from the stored value is less. There is an algorithm, EDL [7] (Earliest Deadline Last), for this purpose, but it requires lists with tasks start times during a hyperperiod, which in turn requires complete knowledge of tasks period times. In [7], EDL is used for scheduling arriving a-periodic tasks. The amount of idle processor time for executing a-periodic tasks is maximized under EDL. The periods of all tasks are known before hand. Lists with task finishing times and amount of available processing time are constructed and used during run-time. The system we try to model in this work is not known before hand to the extent needed for the EDL scheduling algorithm to work. The transactions with high priority are sporadic, because their frequency depends on the rpm the engine is running at. Thus, the lists in [7] need to be recalculated every time the periods change, and this take considerable amount of time. Note that the length of the lists are proportional to the number of tasks starting within one hyperperiod. By using these pre-calculated lists it is possible to do an acceptance test for a-periodic tasks during run-time in $O(N)$ time where N is the number of distinct requests in the hyperperiod. Another test was presented in [22] which runs in $\Theta(n)$, n is the number of periodic tasks. In our work, the scheduler tries to guarantee the execution of an arriving transaction. This is done by using two scheduling queues and hard real-time transactions are scheduled only in the queue with highest priority.

7 Conclusions

In this paper, the scheduling of updates of derived data items for maintaining data freshness are discussed. The data items that are used during the derivation of a data item need to be reasonable fresh. In this paper, a scheme for updates based on marking of possibly changed data items, data freshness based on current and previous values of a data item, and approximation of error of values of data items. Data items that might need to be updated are marked as possibly changed. The error in the value of a data item is approximated and used as a factor when determining, together with marking of a possibly changed data item, which data items to update and the order of the updates. Such an order is important, since the available time for executing updates is limited. Two ways of scheduling the updates are presented; On-Demand Depth-First Traversal (ODDFT) that focuses on updating one data item fully before continuing with the next, and On-Demand Breadth-First Traversal (ODBFT) that first updates data items close to the deriving data item with respect to distance in the data dependency graph. Moreover, in order to evaluate the proposed algorithms, their simulation results are compared to a set of baseline algorithms. The baseline algorithms are taken from [3], but they have been extended with new triggering criteria. The criteria are, except for triggering an update when a too old data item is encountered as in [3]: (i) when one of the parents of a data item is invalid based in data freshness in the value domain, and (ii) when a data cannot fulfill the avi at the deadline of the transaction.

Simulation results show that at moderate arrival rates, ODDFT and ODBFT perform the best concerning valid committed user transactions. The ODDFT and ODBFT algorithms also have the highest ratio of valid committed transactions. At higher arrival rates the ability to make user transactions to commit drops significantly for ODDFT and ODBFT. By applying a feedback control loop the amount of slack used by the generated schedule of updates can be controlled. The ratio of slack allocated for updates can be used to prepare for more interruptions and delays of the user transactions. This shows to be enough to let more user transactions to commit and not generate as many triggered updates as before. ODDFT and ODBFT still has the good performance at low arrival rates, but now perform as the on-demand with knowledge-based option at high arrival rates.

The value-aware algorithms generate less triggered updates at states where base items are not changed often (steady state) than in states where base items change much and often (transient state). This holds both for the proposed algorithms and well-known on-demand algorithms changed to use the proposed scheme of updates and the notion of data freshness. The age-aware algorithms generate the same number of triggered updates in both states. This means that the age-aware algorithms generates too many triggered updates during a steady state.

The prize that has to be paid for the ability to automatically adapt to the changes in states in the environment is larger overhead for both CPU and memory.

The concurrency control protocol used in the simulations is 2PL-HP [1]. A value-aware concurrency control protocol might give better results, and this is an issue for future work.

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] P. Atzeni, S. Ceri, S. Paraboschi, and R. Torlone. *Database System - Concepts, Languages and Architectures*. McGraw-Hill, 1999.

[5] N. Audsley, A. Burns, M. Richardson, and A. Wellings. Absolute and relative temporal constraints in hard real-time databases. In *Fourth Euromicro workshop on Real-Time Systems, 1992. Proceedings.*, pages 148–153. IEEE, 1992.

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

[7] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15:1261–1269, oct 1989.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. The MIT Press, 2 edition, 2001.

[9] 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.

[10] L. B. C. DiPippo and V. F. Wolfe. Object-based semantic real-time concurrency control. In IEEE RealTime Systems Symposium, 12 1993.

[11] 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.

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

[13] S.-J. Ho, T.-W. Kuo, and A. K. Mok. Similarity-based load adjustment for real-time data-intensive applications. 1997.

[14] K.-D. Kang, S. H. Son, and J. A. Stankovic. Specifying and managing quality of real-time data services.

[15] 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.

[16] Y.-K. Kim and S. H. Son. Supporting predictability in real-time database systems. pages 38–48.

[17] C. M. Krishna and K. G. Shin. *Real-Time Systems*. McGraw-Hill, 1997.

[18] 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.

[19] C. Lu, J. A. Stankovic, and S. H. Son. Feedback control real-time scheduling: Framwork, modeling, and algorithms. *Real-Time Systems*, 23(1–2):86–126, 2002.

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

[21] X. Song and J. W. S. Liu. Maintaining temporal consistency: Pessimistic vs. optimistic concurrency control. In *Proceedings of the IEEE Symposium on Computer-Aided Control System Design*, Napa, California, Mar. 1992.

[22] M. E. Thomadakis and J. C. Liu. Linear time on-line feasibility testing algorithms for fixed-priority, hard real-time systems. Technical Report TR00-006, Department of Computer Science, Texas A&M University, College Station, TX 77843-3112, Jan 2000.

[23] M. Xiong, R. Sivasankaran, J. Stankovic, K. Ramamritham, and D. Towsley. Scheduling transactions with temporal constraints: Exploiting data semantics. pages 240–253.

[24] M. Xiong, J. A. Stankovic, K. Ramamritham, D. F. Towsley, and R. M. Sivasankaran. Maintaining temporal consistency: Issues and algorithms. In *RTDB*, pages 1–6, 1996.

[25] D. Zöbel. Schedulability analysis for real-time processes with age constraints. In *Proceedings of the 24th IFAC/IFIP Workshop on Real-Time Programming (WRTP'99)*, June 1999.