# A similarity-aware multiversion concurrency control and updating algorithm for up-to-date snapshots of data*

Thomas Gustafsson, Hugo Hallqvist, and Jörgen Hansson

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

## Abstract

*Real-time databases handle reading and writing of data with time constraints on transactions. Normally, data items in a real-time system have freshness requirements which need to be guaranteed, and for many transactions it is important that accessed data items origin from the same system state, which can be ensured by letting the transactions read a snapshot of the database. In this context, a snapshot at a specific time represents values on data items that were stored in the database at this time. Furthermore, similar values can be considered equal because values within given bounds do not affect the results from calculations. Previous work shows that using similarity among values of data items greatly increases the performance because there is a possibility to skip calculations. In this paper we present the MVTO-S concurrency control algorithm, which supports similarity and multiple versions of data and ensures that transactions read an up-to-date snapshot of a database. Performance evaluations show that MVTO-S increases the performance considerably compared to well-established single-version concurrency control algorithms.*

## 1. Introduction

The number of data items that are used in real-time embedded systems has increased over the years. The reasons are the availability of more powerful CPUs and larger amounts of memory, and more advanced functions in the software. A data item can have freshness requirements and a calculation can have a deadline, i.e., there is meta-information on data, e.g., its freshness and the maximum time allowed for calculating it. One way to organize data and its meta-information is to use a database system, where data is stored centrally, which is in contrast to ad hoc solutions where storage of data is spread out in the software. The benefit of having data stored centrally is that the software becomes easier to maintain. Using a database it is possible to determine which data items are used in the software and their meta-information. This can be difficult to determine if the data is spread out in the software. Also, the database system can be extended with algorithms that automatically maintain data, e.g., the freshness of data. This is possible since the database system has a global knowledge of all data.

For applications the required precision for input data might vary and where the maximum allowed tolerated deviation between, e.g., two consecutive readings of a data value, can be specified. If some tolerance is accepted, then exact values of data items are not important. This can be formalized by assuming an upper bound on how old the value of a data item may be. Ramamritham gives a definition of data freshness using the time domain [16]. Another way to define data freshness of a data item is to use its values. Similarity is a formalization by Kuo and Mok on how to determine if two values of a data item can be considered equal [11]. A value needs only be recalculated if at least one of the values it depends on are dissimilar to the values used deriving the value. Thus, data freshness can be defined in the value domain of data items. Performance evaluations have shown that using similarity can improve the performance of the system [5, 6, 8, 15, 21].

In this paper, we use an engine electronic control unit (EECU) in a car as an example (this is one of our target applications). Control loop calculations need to base their results on up-to-date data values. Also, the EECU has a diagnosis subsystem that executes with the lowest priority. Hence, it is likely that the diagnosis task in the worst case gets interrupted frequently. These interruptions cause the diagnosis task to read values from different states of the external environment giving untrustworthy diagnosis results. By a snapshot at time $t$ we mean that the values of data items in the database are frozen at $t$, and these values are read by a transaction. By an up-to-date snapshot at time $t$ we mean a snapshot at $t$ where all values are up-to-date, i.e., all data items that are affected by changes in values of other data

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

items are updated. Sundell and Tsigas have developed wait-free snapshot algorithms that guarantee a calculation uses values on data items from the same state of the system [20]. However, current snapshot algorithms do not consider similarity or that values need to be updated when a snapshot of data items is derived. Single-version concurrency control algorithms, e.g., high-priority two-phase locking (HP2PL) and optimistic concurrency control (OCC), need restarts to guarantee to give a snapshot to a transaction [4].

We have built a real-time database that can be used in the EECU software. The data in an EECU is derived from sensor readings or from derived data items. This means that an algorithm deriving snapshots must take into consideration that additional data items, to those read by a transaction, need to be updated to have an up-to-date snapshot.

In this paper we contribute by introducing a new multiversion timestamp ordering with similarity (MVTO-S) concurrency control algorithm that ensures transactions see an up-to-date snapshot of the database. By using multiple versions of data items it is possible to drastically decrease the number of conflicts between transactions. In single-version concurrency control, every conflict results in a restart of a transaction.

Performance evaluations are conducted using a real-time database and well-established single-version concurrency control algorithms: HP2PL and OCC, a similarity-aware implementation of OCC denoted OCC-S, and three different implementations of MVTO-S (each implementation uses different techniques to store data versions). The evaluations show that the number of transactions committing within their deadlines compared to single-version concurrency control algorithms can significantly increase using MVTO-S. Transactions read an up-to-date snapshot of the database whereas this cannot be guaranteed using a single-version concurrency control.

The outline of the paper is as follows. A data and transaction model is given in section 2. The multiversion timestamp ordering using similarity (MVTO-S) algorithms and three implementations of it are described in section 3. Section 4 contains performance evaluations. Section 5 contains related work, and section 6 concludes the paper.

# 2. Data and Transaction Model

In this section we shortly describe a real-time system, namely EECU which is our target application, a data model suited for such a system, a transaction model, and an updating algorithm.

## 2.1. Engine Electronic Control Unit

The task of the EECU is to control the engine such that the fuel consumption and the pollution are low, and that the engine is not knocking. The EECU has sensors to monitor the external environment and values are sent to actuators to respond to changes in it. The software consists of tasks that are triggered either by crank angles or clock ticks. The tasks perform calculations based on read data, and store the result in main memory. Control loop calculations are the most important calculations; they should finish before a given time and use data that is fresh at the same time, thus, giving a snapshot of the external environment. The snapshot should be from a time that is sufficiently close to the calculation, e.g., the start time of the calculation.

The engine control software handles real-life entities, such as air pressure, engine temperature, fuel amount, and injection times. Changes within known bounds can be considered negligible to the derived results. For instance, an engine temperature of 80°C or 81°C does not matter for calculating control variables as the temperatures are close enough, implying that a value which has been calculated using 80°C can also be used when the temperature is 81°C.

| Symbol | Denotes |
|---|---|
| $d_i^j$ | Version $j$ of data item $i$ |
| $V(d_i)$ | Set of all versions of $d_i$ |
| $R(d_i)$ | Set of data items read when deriving $d_i$ |
| $G$ | Data dependency graph |
| $dl(\tau)$ | Relative deadline of transaction $\tau$ |
| $ts(\tau)$ | Unique timestamp of transaction $\tau$ |
| $wt(d_i^j)$ | Write timestamp of version $j$ of $d_i$ |
| $v_{d_i}^t$ | Value of $d_i$ at time $t$ |
| $pa(d_i)$ | Timestamp when latest version of $d_i$ was affected by any $d_k \in R(d_i)$ |
| $fixedint_{d_i}(v_{d_i}^t)$ | Interval number of $d_i$ at time $t$ |

**Table 1. Notation.**

## 2.2. Data Model

Data items are divided into base items $B$ and derived items $D$, where the base items are given by sensors and the derived items are derived from base items or other derived items. The read set of data item $d_i$ is denoted $R(d_i)$, and contains all data items that are read when updating $d_i$. The relationships among data items are described in a data dependency graph $G = (V, E)$, where $V$ is the set of all data items and $E$ is a set of edges where every edge connects a read set member in $R(d_i)$ with $d_i$. The notation used in this paper is summarized in table 1.

Further, a data item has one or several versions and the set of versions of data item $d_i$ is denoted $V(d_i)$. Each version $d_i^j$ of data item $d_i$ has a write timestamp $wt(d_i^j)$. A version is said to be valid during a time interval starting from its timestamp until the timestamp of the following version, i.e., $[wt(d_i^j), wt(d_i^{j+1})]$. If $d_i^j$ is the newest version it is assumed to be valid until a newer version is installed. Hence, the time

interval is $[wt(d_i^j), \infty]$. A proper version with respect to a timestamp $t$ is the latest version with a write timestamp less than or equal to $t$, i.e., a proper version of $d_i$ at $t$ has the following timestamp: $\max\{wt(d_i^j) | \forall d_i^j \in V(d_i), wt(d_i^j) \leq t\}$.

As described above, a real-time system can benefit from handling data items that have similarities. The following definitions give two ways of reasoning about similar values. Definitions 2.1 and 2.2 define the freshness of a data item with respect to one read set member, and the value of a data item is fresh if the value of the read set member is similar to the previously used value.

**Definition 2.1** (Data Freshness using Flexible Data Validity Interval)**.** *Each pair $(d_i, d_k)$, where $d_i$ is a derived data item and $d_k$ is an item from $R(d_i)$, has a data validity interval, denoted $\delta_{d_i,d_k}$, that states how much the value of $d_k$ can change before the value of $d_i$ is affected. Let $v^t_{d_k}$ and $v^{t'}_{d_k}$ be values of $d_k$ at times t and $t'$ respectively. A version $j$ of $d_i$ reading $v^t_{d_k}$ is fresh, with respect to the version of $d_k$ valid at t, for all $t'$ fulfilling $|v^t_{d_k} - v^{t'}_{d_k}| \leq \delta_{d_i,d_k}$.*

The interval is flexible because the value of $d_k$ is the origin of a data validity interval. The value of $d_i$ is unaffected by changes in $d_k$ when they are within the interval. Using fixed validity intervals, the freshness of a data item with respect to one of its read set members is as follows:

**Definition 2.2** (Data Freshness using Fixed Data Validity Interval)**.** *Let $fixedint_{d_k}$ be a function mapping values of a data item $d_k$ to natural integers, i.e., $fixedint_{d_k} : D \rightarrow \mathbf{N}$, where $D$ is the domain of values of data item $d_k$. All values of $d_k$ mapping to the same interval are similar. Let $v^t_{d_k}$ and $v^{t'}_{d_k}$ be values of $d_k$ at times t and $t'$ respectively. A version $j$ of $d_i$ reading $v^t_{d_k}$ is fresh, with respect to the version of $d_k$ valid at t, for all $t'$ fulfilling $fixedint_{d_k}(v^t_{d_k}) = fixedint_{d_k}(v^{t'}_{d_k})$.*

One example of the function $fixedint$ is: $fixedint_{d_k}(v^t_{d_k}) = \left\lfloor \frac{v^t_{d_k}}{64} \right\rfloor$, where the value domain of data item $d_k$ is divided into intervals of size 64. As long as the value of $d_k$ maps to the same number as the value of $d_k$ being used to derive $d_i^j$, the value changes of $d_k$ do not affect the value of $d_i^j$.

As discussed earlier it is important for many calculations that all used values are derived from the same state of the external environment. Values that are correlated in time, i.e., the values are derived from the same state, are said to be relatively consistent. In this paper we adopt the following view of relative consistency [10].

**Definition 2.3** (Relative Consistency)**.** *Let the time interval when version $j$ of data item $d_i$ is valid be defined as $VI(d_i^j) = [start, stop] \subseteq \mathbf{R}$, and $VI(d_i^j) = [start, \infty]$ if $d_i^j$ is currently the latest version. Then, a set of versions of data items, denoted $RS$, is defined to be relatively consistent if*

$$\bigcap \{VI(d_i^j) | \forall d_i^j \in RS\} \neq \emptyset. \tag{1}$$

The definition of relative consistency implies a derived value from $RS$ is valid in the interval when all versions in the set $RS$ are valid.

### 2.3. Transaction Model and Updating Algorithm

Every data item is associated with a transaction that derives the value of the data item. A transaction updating a base item is denoted a sensor transaction (ST). A transaction issued by a task is denoted user transaction (UT), and a transaction issued by the database system to update a data item is denoted triggered update (TU). Transactions have a begin of transaction operation (BOT), read operations (sensor transactions do not have any read operations), one write operation, an end of transaction operation (EOT), a unique timestamp, $ts(\tau)$, a release time $rt(\tau)$, a priority, and a relative deadline $dl(\tau)$. The active transaction with the highest priority is executing. The timestamp is monotonically increasing for every new base and user transaction. Triggered updates inherit the timestamp of the UT that generated the update. The timestamps on transactions relate the order they were started, i.e., if transaction $\tau_1$ started before $\tau_2$ then $ts(\tau_1) < ts(\tau_2)$. The most current version of every data item has a timestamp, denoted $pa$, set to the latest logical time a transaction found the data item being potentially affected by a change in any of its read set members.

The database system has a global knowledge of all data and its meta-information. This allows for an updating algorithm deciding which data items that need to be updated. A schedule of data items required to be updated is constructed as a response to the arrival of a user transaction. There are two problems to address: (i) which data items should be scheduled, and (ii) which scheduled updates need to be executed. These problems are addressed by using an updating scheme, denoted affected updating scheme (AUS), which is described next.

AUS consists of three steps: AUS_S1, AUS_S2, and AUS_S3. The first step, AUS_S1, keeps base items up-to-date. Base item updates are executed periodically.

Step AUS_S2 determines if a change in a data item affects the values on any of its children in $G$. The freshness of a child $d_i$ is checked by checking its data freshness with respect to one parent—the one being updated. If $d_i$ is found to be stale it is marked as affected with the timestamp of the latest transaction $\tau$ producing a value that affected the data item, i.e., $pa(d_i) = \max(pa(d_i), ts(\tau))$.

Step AUS_S3 is an on-demand step, and $G$ is traversed top-bottom meaning that only those data items $d_i$ with

$pa(d_i) > 0$ are affected and need to be updated. In AUS_S3, when the latest version of a data item $d_i$ is updated, $pa(d_i)$ is set according to the following equation

$$pa(d_i) = \begin{cases} 0 & \text{if } ts(\tau) \geq pa(d_i) \\ pa(d_i) & \text{otherwise,} \end{cases} \tag{2}$$

where $\tau$ is the transaction updating data item $d_i$. Note, data items affected by $\tau$ are marked according to step AUS_S2.

The ODTB algorithm is an implementation of step AUS_S3. The details are covered in [5]. When a UT using data item $d_i$ arrives, pregenerated schedules of all members of $R(d_i)$ are checked for a data item $d_k$ with $pa(d_k) > 0$ and the remainder of the schedule is copied to a schedule of needed updates. The pregenerated schedule of $d_m \in R(d_i)$ contains the depth-first order of data items in the branch originating in $d_m$. The pregenerated schedule is traversed from base items towards $d_m$. Hence, the check $pa(d_k) > 0$ finds the first data item that is stale and its descendants are potentially affected by the stale data item.

**Definition 2.4** (Staleness of a data item)**.** *Let a value of data item $d_i$ be derived at time $t$ using values of data items in $R(d_i)$. The value of $d_i$ is denoted $v^t_{d_i}$. The value $v^t_{d_i}$ is stale at time $t'$ if there exists at least one element $d_k$ in $R(d_i)$ such that $|v^t_{d_k} - v^{t'}_{d_k}| > \delta_{d_i,d_k}$ or $fixedint_{d_k}(v^t_{d_k}) \neq fixedint_{d_k}(v^{t'}_{d_k})$ depending on which definition of freshness is used.*

**Proposition 2.1.** *Let $d_i$ be a data item and $pa(d_i)$ the timestamp set in steps AUS_S2 and AUS_S3. If data item $d_i$ is stale according to definition 2.4 then its timestamp is larger than zero, i.e., $pa(d_i) > 0$.*

*Proof.* Proof by contradiction. Assume a data item $d_i$ is stale. The $pa(d_i)$ timestamp has been set by AUS_S2 otherwise $d_i$ is not stale. The $pa(d_i)$ timestamp is determined by taking $pa(d_i) = \max(pa(d_i), ts(\tau_1))$; further, assume $\tau_1$ is the latest update affecting $d_i$, thus, $pa(d_i) = ts(\tau_1)$. If $pa(d_i) = 0$, then $d_i$ has been updated by a transaction $\tau_2$, implying $ts(\tau_2) \geq pa(d_i)$ and $ts(\tau_2) > ts(\tau_1)$. Hence, $\tau_2$ arrived after $\tau_1$ since timestamps on transactions increase monotonically, and $d_i$ is up-to-date which is a contradiction. Thus, a stale data item $d_i$ implies $pa(d_i) > 0$. □

Proposition 2.1 shows that it is possible to find stale data items, and, thus, ODTB schedules the necessary updates.

# 3. Multiversion Concurrency Control using Similarity

In this section we describe the MVTO-S concurrency control algorithm that stores several versions of each data item, and makes versions up-to-date before they are read by transactions.

## 3.1. The MVTO-S Algorithm

We first discuss the outline of the MVTO-S algorithm in the context of one UT. Assume one transaction, $\tau$, is about to start, and its read operations should perceive values as originating from the same system state. The read operations must then read correct versions of data items, and these versions must be up-to-date. Hence, there should be a way of mapping the readings by read operations in $\tau$ to updated versions.

The mapping from transaction to versions is done via logical time. It is sufficient to read versions that were valid when $\tau$ started, because $\tau$ then perceives versions from the same state that also are sufficiently close in time to the calculation. A proper version of a data item is the version with latest timestamp less than or equal to $ts(\tau)$. If the ODTB algorithm atomically generates a schedule of updates when $\tau$ starts, then we know which updates are needed to make data items up-to-date. Due to similarities some of the updates might be possible to skip. MVTO-S is divided into two sub-algorithms: arriving transaction (AT) that creates a schedule, and executing transaction (ET) that checks similarities and writes new versions.

The AT sub-algorithm executes when a transaction $\tau$ arrives. The steps are:

**AT1:** A global virtual timestamp $gvts$ is assigned the timestamp of the oldest active transaction, i.e., $gvts = \min_{\forall i, \tau_i \in activeT}\{ts(\tau_i)\}$, where $activeT$ is the set of all active transactions.

**AT2:** If $\tau$ is a UT then a schedule of needed updates is constructed atomically, i.e., uninterrupted by other transactions, by ODTB.

The steps of the ET sub-algorithm are:

**ET1:** When a transaction $\tau$ enters its BOT operation the following steps are taken:

**ET1.1:** Calculate the write timestamp of version $j$ of data item $d_i$ that $\tau$ derives, $\forall d_m \in R(d_i)$:

$$wt(d_i^j) = \max\left\{\max\{wt(d_k^l) | \forall d_k^l \in V(d_m)\}\right\} \tag{3}$$

**ET1.2:** Find a proper version at time $wt(d_i^j)$ and denote it $d_i^n$. If $wt(d_i^j) = wt(d_i^n)$, then the update can be skipped since the version already exists. Otherwise continue with ET1.3.

**ET1.3:** Check the relevance of executing transaction $\tau$ by using similarity. The value of read set members of $d_i^j$ is compared to values of read set members of $d_i^n$. A read set member is denoted $d_m$. The check is done as follows using flexible validity intervals:

$$\forall d_m \in R(d_i), |v_{d_m}^{wt(d_i^j)} - v_{d_m}^{wt(d_i^n)}| \leq \delta_{d_i,d_m}, \tag{4}$$

and as follows using fixed validity intervals, $\forall d_m \in R(d_i)$:

$$fixedint_{d_m}\left(v_{d_m}^{wt(d_i^j)}\right) = fixedint_{d_m}\left(v_{d_m}^{wt(d_i^n)}\right). \tag{5}$$

If all checks in equations 4 or 5 evaluate to true this means that $\tau$ can be skipped. Otherwise start executing $\tau$.

**ET2:** Every read operation of $\tau$ reading a data item $d_i$ reads a proper version $n$ of $d_i$.

**ET3:** Handling of write operations of $\tau$.

**ET3.1:** If $ts(\tau) > gvts$, then an operation writing data item $d_i$ creates a new version if enough space can be accommodated for such a version (otherwise go to step ET3.2). If $ts(\tau) = gtvs$ then no transaction is interrupted and might need the old version, and, thus, $\tau$ overwrites the current version of the data item. The timestamp of the new version is the maximum of the write timestamp of read values, i.e., $wt(d_i^j) = \max\{wt(d_k^n) | \forall d_k^n \in RS\}$. Also in this step, all versions older than $gvts$ are pruned from the memory pool to free memory.

**ET3.2:** If there is not enough space for a new version, the transaction with timestamp equal to $gvts$ is restarted and $gvts$ is recalculated. Versions with a write timestamp less than the new $gvts$ are purged to free memory. In this way the oldest active transaction gets restarted, and this is also the transaction with the lowest priority (note that transactions are executed according to priority). Thus, MVTO-S is aware of transaction priorities and restarts low priority transactions before high priority transactions.

Next an example is given on how the MVTO-S algorithm works.

**Example 3.1:** Consider that an arriving UT, $\tau_1$, using data item $d_5$ is assigned timestamp 8. Step AT1 assigns 8 to $gvts$. Step AT2 creates a schedule of needed updates, e.g., $[\tau_{d_1}, \tau_{d_3}, \tau_{d_2}, \tau_{d_4}]$, where $d_5$ directly depends on $d_3$ and $d_4$ and indirectly on $d_1$ and $d_2$. Assume two STs arrive updating base items $d_8$ (that $d_1$ reads) with timestamp 9 and $d_9$ (that $d_2$ reads) with timestamp 10. Step ET3.1 creates new versions of $d_8$ and $d_9$ since both STs had larger timestamps than $gvts$.

Next arrives $\tau_2$ with timestamp 11 using data item $d_6$. It has higher priority than $\tau_1$ since it is not yet finished. Thus, $gvts$ is 8, and step AT2 creates the following schedule $[\tau_{d_2}, \tau_{d_4}]$. The TUs $\tau_{d_2}$ and $\tau_{d_4}$ are executed with timestamp 11. In step ET1.1 of $\tau_{d_2}$, the write timestamp of a possibly new version of $d_2$ is calculated by looking at read set members of $d_2$. In this case it is 10 since a ST with timestamp 10 updated $d_9$. Step ET1.2 finds a proper version of $d_2$, say with timestamp 5. In step ET1.3 a similarity check is done for each read set member. Hence, a similarity check is done between a version of $d_9$ with timestamp 5 and the version with timestamp 10. If these two versions are similar, then transaction $\tau_{d_2}$ can be skipped, and transaction $\tau_{d_4}$ would read the version of $d_2$ with timestamp 5. $\square$

Next we give theorems and proofs on the behavior of MVTO-S.

**Lemma 3.1.** *Using MVTO-S, a proper version of a data item $d_i$ at time $t = ts(\tau)$ represents an up-to-date value.*

*Proof.* Assume $d_i^n$ is a proper version but it is stale. Now assume step ET3.1 of a TU installs a version since an update was scheduled in step AT2 which schedules all needed updates. Denote the new version $d_i^{n+1}$. The timestamps are ordered as follows $wt(d_i^n) < wt(d_i^{n+1}) \leq t$ since by step ET1.1 and ET2 the write timestamp of $d_i^{n+1}$ is the maximum of all accessed read set members but limited by $t$, i.e., $\forall d_m \in R(d_i)$, $wt(d_i^{n+1}) = \max\left\{\max\{wt(d_k^j) | \forall d_k^j \in V(d_m), wt(d_k^j) \leq t\}\right\} \leq t$, and $wt(d_i^n) < wt(d_i^{n+1})$ since $d_i^n$ was taken for a proper version and is stale. Version $d_i^{n+1}$ is an up-to-date proper version, and it is valid at time $t$.

When versions are removed from the pool by step ET3.2, they are removed according to earliest timestamp first. Thus, if version $d_i^{n+1}$ is removed, version $d_i^n$ has been removed before $d_i^{n+1}$ and therefore a proper version $d_i^{n+2}$ of data item $d_i$ at time $t$ is up-to-date. $\square$

**Theorem 3.2.** *MVTO-S ensures that a transaction $\tau$ reads up-to-date versions of read set members (ET2) such that the start time of $\tau$ is in the time interval $I = \bigcap\{VI(d_i^j) | \forall d_i^j \in RS\}$.*

*Proof.* We only consider transactions that commit. An interval $I$ is built iteratively for each read operation. We have that for any version $j$ of data item $d_i$, $VI(d_i^j) = [wt(d_i^j), wt(d_i^{j+1})]$. A proper version $n$ has by definition $wt(d_i^n) \leq ts(\tau)$. For every read version $d_i^n$ (ET2) it holds that $wt(d_i^n) \leq ts(\tau)$ since by lemma 3.1 there cannot exist a not yet updated version in the interval $[wt(d_i^n), ts(\tau)]$.

We must show that $ts(\tau) < wt(d_i^{n+1})$ for all read versions $d_i^n$, i.e., an up-to-date proper version is alway chosen. Since a version to read is chosen such that $wt(d_i^n) \leq ts(\tau)$ and $wt(d_i^{n+1}) > wt(d_i^n)$ as step ET1.2 forces unique timestamps on versions, then $wt(d_i^{n+1}) > ts(\tau)$ otherwise $d_i^{n+1}$ would have been chosen in step ET2. Thus, we have shown that read operations executed by $\tau$ choose versions such that they are relative consistent (definition 2.3) and $ts(\tau)$ is included in the interval where these versions are valid. $\square$

The effect of theorem 3.2 is that MVTO-S guarantees that transactions read an up-to-date snapshot of the database that was valid when the transaction started. This is an important property of the algorithm. Some transactions need to read values of data items that are correlated in time, e.g., diagnosis transactions. Next we describe three versions of MVTO-S that differ in the amount of meta-data every version has.

### 3.2. MVTO-S$^{UV}$

This implementation is denoted use values (UV), because each version stored in the database also holds the values of the read-set members. Hence, the relevance check in step ET1.3 of MVTO-S is easy to implement. First, the

proper version needs to be found, then the value of each read set member in the proper version is compared to the corresponding value a new version would read. If all comparisons involve similar values, the new version is not needed and the update can be skipped.

### 3.3. MVTO-$S^{UP}$

This implementation is denoted use pool (UP), because finding the values used for deriving a version might be found in the memory pool. The proper version needs to be found (step ET1.2), and based on its timestamp the read set members might be found in the memory pool. If any of these versions cannot be found, it is impossible to succeed in step ET1.3. If all read set member versions can be found, and values are similar, then the update can be skipped.

The MVTO-$S^{UP}$ is an implementation where the old similar version is duplicated but with a new timestamp calculated from the read set members. The reason is that in step ET3.1, old versions can be removed, but installing a new version makes it more likely to find it in the relevance check of step ET1.

### 3.4. MVTO-$S^{CRC}$

Clearly we want a combination of MVTO-$S^{UV}$, where values of read set members can always be found, and MVTO-$S^{UP}$, where each version has less memory overhead than using MVTO-$S^{UV}$. Thus, the same information stored in the values of read set members in MVTO-$S^{UV}$ versions should be stored in less space.

If values are represented as fixed validity intervals, a value can uniquely be represented as a number given by the $fixedint$ function. Since similar values are always mapped to the same number, the read set member values in MVTO-$S^{UV}$ could instead be interval numbers. Let us denote a number from the $fixedint$ function as an interval number. It is possible to combine the interval numbers of read set members into one single value by using a checksum or CRC. We have done an investigation on how robust checksums and CRCs are in mapping a few numbers to a unique number [4]. The CRC algorithm (a CRC-32 algorithm) produces unique values in our evaluations, meaning that two equal CRCs have similar read set members, and, thus, the two versions are similar.

In summary, MVTO-$S^{CRC}$ works in the following way. The proper version of a data item is fetched. This version has a CRC attached to it which is the CRC of the interval numbers of its read set members. A relevance check of an update is done, and interval numbers on all read set members are derived and a CRC is calculated. If the CRC of the proper version is the same as for the update, then these two versions would use similar read set members. Thus, the update can be skipped. If the CRCs are unequal then some read set members are dissimilar and the update is needed.

### 3.5. Memory Consumption

Versions are allocated from a memory pool with a predefined size. Assume the following sizes: four bytes for a pointer, four bytes for a data value, two bytes for the timestamp, four bytes for a CRC, and a data item has on average three parents. The storage requirements for a memory pool holding 200 versions are $200 \times (4+2+3\times 4+4) = 4400$ bytes for MVTO-$S^{UV}$, 2000 bytes for MVTO-$S^{UP}$, and 2800 bytes for MVTO-$S^{CRC}$. We now compare these results to the HP2PL algorithm, where every data item need to have a semaphore. In $\mu$C/OS-II it takes 10 bytes [12]. For 150 data items, the storage requirements are $150 \times (10+2+4) = 2400$ bytes. Hence, depending on the needed pool size the MVTO-S implementations are not using considerably more memory than other concurrency control algorithms.

## 4. Performance Evaluations

The performance evaluations are conducted using a database system running on a real-time operating system. The database can be configured for using one updating algorithm and one concurrency control algorithm from a range of updating algorithms and concurrency control algorithms.[1] The database system can, out of the box, be compiled together with the engine control software.

The results show that using implementations of MVTO-S greatly improves performance and the number of transactions that commit within their deadlines, compared to using single version concurrency control algorithms.

Five tasks are executing periodically, and they invoke UTs that execute with the same priority as the task. The tasks are prioritized according to rate monotonic (RM), where a task gets a priority proportional to the inverse of its frequency. The base period times are: 60 ms, 120 ms, 250 ms, 500 ms, and 1000 ms. These period times are multiplied with the ratio $32/arrival_rate$, where 32 is the number of invoked tasks using the base period times, and $arrival_rate$ is the arrival rate of UTs. The data item a UT derives is randomly determined by taking a number from the distribution U(0,$|D|$). In the experiments, a database with 45 base items and 105 derived items has been used. The graph is constructed by setting the following parameters: cardinality of the read set ($|R(d_i)|$), ratio of $R(d_i)$ being base items, and ratio being derived items with only base item parents. The cardinality of $R(d_i)$ is set randomly for each $d_i$ in the interval 1–8, and 30% of these are base items, 60% are derived items with only base item parents, and the remaining 10% are other derived items. These figures are rounded to nearest integer. The number of derived

1 The database system is running on a simplistic real-time operating system $\mu$C/OS-II [12] that runs in a DOS command window in Windows 2000 Professional with servicepack 4. The computer is an IBM T23 with 512 Mb of RAM and a Pentium 3 running with 1.1 GHz.

items with only base item parents is set to 30% of the total number of derived items. We believe a database of 150 data items represents the storage requirements of a hotspot of an embedded system, e.g., in the EECU 128 data items are used to represent the external environment and actuator signals. Further, we believe the data dependency graph $G$ is broad (in contrast to deep), and that a data item does not depend on many other data items.

Every sensor transaction executes for 1 ms and every user transaction and triggered update executes for a time repeatedly taken from a normal distribution with mean 5 and standard deviation 3 until it is within $[0, 10]$. A simulation runs for 150 s with a specified arrival rate. Every simulation is executed 5 times and the showed results are the averages from these 5 runs. The user transactions are not started if they have passed their deadlines, but if a transaction gets started it executes until it is finished.

Every write operation creating the most recent version is adding a value from the distribution U(0,350) to the previous most recent version. The data validity intervals are set to 400 for all data items, i.e., $\delta_{d_i,d_k} = 400$, and $fixedint(v^t_{d_k}) = \left\lfloor \frac{v^t_{d_k}}{400} \right\rfloor$. The creation of versions by the multiversion concurrency control algorithm involves taking values of the two closest versions, one older and one newer, and then randomly derive a value that is between the versions. The memory pool for storing versions holds 300 versions where 150 versions are reserved for the current value of each data item. Base item updates have a priority higher than UTs and execute on average every 100 ms, i.e., the period time is 50 ms and for every base item there is a 50% chance that the item is updated. The updating algorithm used is the ODTB algorithm except in the OD-HP2PL simulation.

Next we describe the concurrency control algorithms that are used in the evaluations. MVTO is MVTO-S where step ET1.3 is not used, i.e., no relevance check is done. OCC-S is OCC where the validation phase has been extended with a check whether the conflict involves similar values or not [4]. The RCR versions of OCC and OCC-S are restarting transactions until they are able to read a snapshot of the database [4]. The on-demand updating and using HP2PL for concurrency control (OD-HP2PL) algorithm [1] triggers updates based on time instead of similarity, and in our test platform setup the allowed age on data items is set to 400 ms which is a good estimate on how long time a sensor value lives, since the average period time of sensors are 100 ms and it requires, on average, 3 updates to change outside a validity bound giving that values live 300-400 ms.

### 4.1. Experiment 1: Committed User Transactions

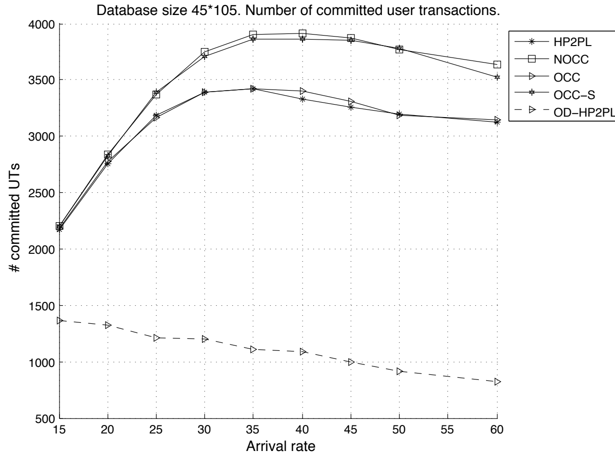
In this experiment, the number of committed user transactions within their deadlines is evaluated. The no concurrency control scheme is used as a baseline. Figure 1 shows the performance of the algorithms.

First the benefit of using similarity can be seen in figure 1(a) studying the difference in performance between HP2PL and OD-HP2PL. In figure 1(b), MVTO-S$^{\text{UV}}$ outperforms the single-version concurrency control algorithms at all arrival rates. The difference is most notable at higher loads where MVTO-S$^{\text{UV}}$ performs significantly better than HP2PL, OCC, and NOCC. MVTO-S$^{\text{UP}}$ cannot perform as good as MVTO-S$^{\text{UV}}$ because less transactions are skipped in step ET1.3. MVTO-S$^{\text{UP}}$ performs better than single-version algorithms at high arrival rates, but for small arrival rates, OCC-S, HP2PL, and OCC perform better.

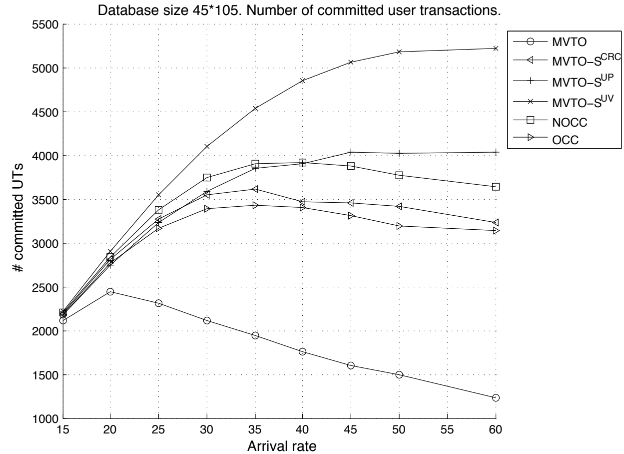
Using multiple versions there are in total fewer restarts of transactions, i.e., less unnecessary work of transactions is done, compared to algorithms using restarts as resolving conflicts. Moreover, more transactions can be skipped since updates do not overwrite each others results because old values are stored in new versions. A transaction that derives an old version of a data item can benefit from versions from almost the same point in time that might exist in the database, because a relevance check checks against such a version and not the most recent version as in single-version concurrency control algorithms. Table 2 shows the percentage of the total number of UTs and TUs that restart and can be skipped. The multiversion timestamp ordering algorithms have considerably fewer restarts than single-version algorithms. Every restart for MVTO, MVTO-S$^{\text{UV}}$, MVTO-S$^{\text{UP}}$, and MVTO-S$^{\text{CRC}}$ is due to a full memory pool, whereas for single-version algorithms restarts are due to conflicts among concurrent transactions. The multiversion algorithms that use step ET1.3 are successful in skipping transactions.

In figure 1(a), we see that using OCC-S gives that the same number of UTs can commit as for NOCC. However, NOCC cannot guarantee the consistency of results produced by transactions since transactions read and write uncontrollably to the database, but OCC-S produces consistent results. Table 2 shows that using similarity in OCC-S compared to using no similarity as in OCC, the percentage of restarts drops from 9.03% (OCC) to 0.96% (OCC-S). This indicates that conflicts among transactions often involve similar values since many of these conflicts cause restarts in OCC but not in OCC-S. Thus, a considerable amount of conflicts that do occur do not need any concurrency control. This is in line with observations made by Graham [3].

The implementations of MVTO-S guarantee that a transaction reads an up-to-date snapshot of the database. The single-version concurrency control algorithms can guarantee this by restarting transactions until they read data items that are from the same external state. These algorithms are prefixed with RCR for relative consistency restarts. Figure 2 shows the performance using restarts to enforce relatively consistent read sets. The MVTO-S implementations




(a) Single-version algorithms.




(b) Multiversion algorithms.

**Figure 1. Experiment 1: Performance evaluations of single-version and multiversion concurrency control algorithms.**

| Alg. | Restarts | Skipped transactions |
|---|---|---|
| HP2PL | 9.32% | 16.2% |
| OD-HP2PL | 9.46% | 0% |
| OCC | 9.03% | 16.0% |
| OCC-S | 0.96% | 15.0% |
| NOCC | 0% | 14.5% |
| MVTO | 0.24% | 7.03% |
| MVTO-S$^{UV}$ | 0.039% | 55.7% |
| MVTO-S$^{UP}$ | 0.15% | 38.5% |
| MVTO-S$^{CRC}$ | 0.23% | 39.6% |

**Table 2. Experiment 1: Percentage of total number of UTs and TUs that restarts, and percentage of skipped transactions.**

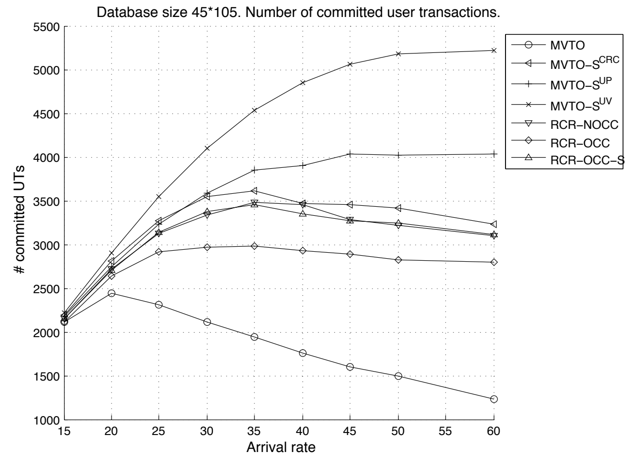



**Figure 2. Experiment 1: Performance of relative consistency restarts algorithms.**

are performing better than the single-version algorithms with restarts since values needed for deriving snapshots for transactions are stored in memory, therefore new updates to data items cannot destroy a snapshot for a transaction. Using a RCR algorithm, an update to a data item can be read by a preempted transaction which may destroy the derivation of a snapshot.

### 4.2. Experiment 2: Memory Pool Sizes

In this experiment, the number of committed user transactions within their deadlines is investigated for different sizes of the memory pool holding versions. The memory pool size affects the performance of the system since transactions are restarted only when the memory pool is full. The smaller the memory pool the higher the probability that it is full. Figure 3 shows the performance for different pool sizes, and the number after the name of the algorithm is the pool size.

The plots show that all implementations of MVTO-S suffer when using smaller pool sizes since the number of restarts is increasing. However, MVTO-S$^{UP}$ suffers the most of a small pool size since the ability to check similarity depends on how often old versions are purged from the pool, and they are purged more often when the pool size is small.

MVTO-S$^{CRC}$ has worse performance than MVTO-S$^{UV}$ because values on data items are, in this experiment, always

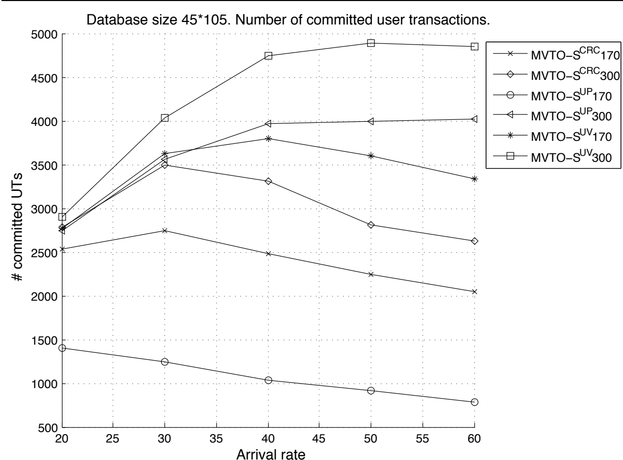



**Figure 3. Experiment 2: Investigation on how the pool size affects performance.**

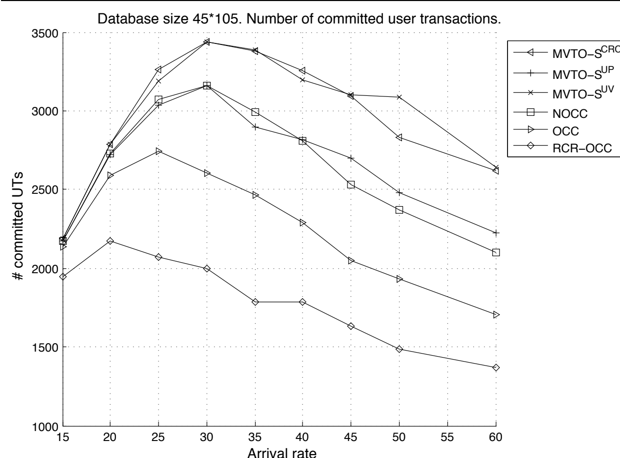



**Figure 4. Experiment 2: Performance using fixed validity intervals as data freshness.**

increasing and fixed validity intervals are used. For fixed intervals, a new value on a data item might be located somewhere inside a new interval, meaning that, on average, a smaller number of updates is required to land outside an interval compared to flexible intervals where a new value is the origin of a new interval. Thus, fewer updates can be skipped, which affects the performance of the system. Figure 4 shows the performance of the implementations using fixed intervals. We can see that MVTO-S$^{CRC}$ has the same performance as MVTO-S$^{UV}$, and they perform better than all the other algorithms. This means that when fixed validity intervals are used then MVTO-S$^{CRC}$ is as good as MVTO-S$^{UV}$.

## 5. Related Work

In this section we discuss related work on concurrency control algorithms for real-time databases in embedded systems [2, 7, 9, 13, 14, 17–19, 21].

Two-phase locking and optimistic concurrency control have been evaluated for real-time systems [2, 7, 9]. We have found that HP2PL and OCC give similar performance when they are executed in a database system on a simplistic real-time operating system, where transactions execute with fixed unchangeable priority, and it is impossible to restart a currently executing transaction. To the best of our knowledge, no evaluation of the performance of HP2PL and OCC on such system has been documented elsewhere.

Multiversion concurrency control algorithms have also been evaluated [17–19]. It has been found that 2PL performs better than MVTO and the single-version timestamp ordering concurrency control algorithm [19]. Song and Liu evaluate the 2PL and OCC multiversion algorithms in a hard real-time system [18]. In their work, a set of data items is said to be temporally consistent when they are absolute and relative consistent. The evaluation results show that temporal consistency is highly affected by the transaction conflict patterns and also, OCC is poor in maintaining temporal consistency in systems consisting of periodic activities. The implementations of MVTO-S are free of restarts until the memory pool gets full, and, thus, the conflict patterns should not affect these algorithms at all or to a much smaller extent compared to algorithms using restarts for conflict resolution.

The proposed multiversion concurrency control algorithms, MVTO-S$^{UV}$, MVTO-S$^{UP}$, and MVTO-S$^{CRC}$ use similarity. To the best of our knowledge, using multiversion concurrency control and similarity is a novel approach. The main reason to use multiversion concurrency control is to be able to guarantee relative consistency. This can also be guaranteed by using a snapshot technique using wait-free locks [20], but these algorithms are not using similarity and, thus, cannot skip transactions.

## 6. Conclusions and Future Work

Real-time databases have many applications, and for some systems, e.g., diagnosis and control applications in vehicular systems, it is important to have up-to-date snapshots of data. Further, the exact values of data items may be unimportant and, thus, small deviations in values do not require updates on dependent data items.

Our contribution is a multiversion concurrency control algorithm (MVTO-S) that gives an up-to-date snapshot of the database to transactions. The algorithm uses similarity, which gives conditions for skipping unnecessary calculations. Using a history of versions allows more trans-

actions to be skipped compared to single-version concurrency control algorithms without sacrificing data quality. Furthermore, the number of restarts of transactions can be reduced since conflicts among concurrent transactions are not resolved using restarts, but are instead avoided by storing new versions of data items. Thus, the overall performance of the system is greatly improved using implementations of MVTO-S compared to single-version algorithms. Moreover, transactions are guaranteed to read an up-to-date snapshot of the database using MVTO-S, and this is not guaranteed using single-version concurrency control algorithms. Hence, using MVTO-S the performance is increased and valid snapshots are given to transactions.

We have implemented three variants of the MVTO-S algorithm. In memory-constrained systems, MVTO-$S^{UP}$ and MVTO-$S^{CRC}$ are the best choices since the memory overhead for each version of a data item is small. MVTO-$S^{UV}$ has the best performance, but its memory requirement is also larger than MVTO-$S^{UP}$ and MVTO-$S^{CRC}$.

The performance evaluations show that many of the conflicts between concurrent transactions involve similar values, i.e., using optimistic concurrency control with similarity (OCC-S) give similar performance as not using concurrency control.

For future work, we plan to investigate quality of data and how overloads can be handled by changing the quality of data transactions perceive.

## References


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

[2] A. Chiu, B. Kao, and K.-Y. Lam. Comparing two-phase locking and optimistic concurrency control protocols in multiprocessor real-time databases. In *Proceedings of the Joint Workshop on Parallel and Distributed Real-Time Systems*, 1997.

[3] M. H. Graham. How to get serializability for real-time transactions without having to pay for it. In *Proceedings of the Real-Time Systems Symposium 1993*, pages 56–65, 1993.

[4] T. Gustafsson. Maintaining data consistency in embedded databases for vehicular systems. Linköping Studies in Science and Technology Thesis No. 1138. Linköping University. ISBN 91-85297-02-X.

[5] T. Gustafsson and J. Hansson. Data management in real-time systems: a case of on-demand updates in vehicle control systems. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, pages 182–191. IEEE Computer Society Press, 2004.

[6] T. Gustafsson and J. Hansson. Dynamic on-demand updating of data in real-time database systems. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 846–853. ACM Press, 2004.

[7] J. R. Haritsa, M. J. Carey, and M. Livny. On being optimistic about real-time constraints. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 331–343. ACM Press, 1990.

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

[9] J. Huang, J. A. Stankovic, and K. Ramamritham. Experimental evaluation of real-time optimistic concurrency control schemes. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 35–46, September 1991.

[10] B. Kao, K.-Y. Lam, B. Adelberg, R. Cheng, and T. Lee. Maintaining temporal consistency of discrete objects in soft real-time database systems. *IEEE Transactions on Computers*, 52(3):373–389, March 2003.

[11] 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.

[12] J. J. Labrosse. *MicroC/OS-II The Real-Time Kernel Second Edition*. CMPBooks, 2002.

[13] K.-Y. Lam, T.-W. Kuo, B. Kao, T. S. Lee, and R. Cheng. Evaluation of concurrency control strategies for mixed soft real-time database systems. *Information Systems*, 27:123–149, 2002.

[14] K.-Y. Lam and W.-C. Yau. On using similarity for concurrency control in real-time database systems. *The Journal of Systems and Software*, (43):223–232, 2000.

[15] R. Majumdar, K. Ramamritham, R. Banavar, and K. Moudgalya. Disseminating dynamic data with qos guarantee in a wide area network: A practical control theoretic approach. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Synmposium (RTAS'04)*, pages 510–517. IEEE Computer Society Press, May 2004.

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

[17] L. Shu and M. Young. Versioning concurrency control for hard real-time systems. *The Journal of Systems and Software*, (63):201–218, 2002.

[18] X. Song and J. W. Liu. Maintaining temporal consistency: Pessimistic vs. optimistic concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):786–796, 1995.

[19] R. Sun and G. Thomas. Performance results on multiversion timestamp concurrency control with predeclared writesets. In *Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 177–184. ACM Press, 1987.

[20] H. Sundell and P. Tsigas. Simple wait-free snapshots for real-time systems with sporadic tasks. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applicatins (RTCSA04)*, 2004.

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