Data Freshness and Overload Handling in Embedded Systems*

Thomas Gustafsson[a] and Jörgen Hansson[b,a]

[a]Department of Computer and Information Science, Linköping University, Sweden

[b]Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA

E-mail: `thogu@ida.liu.se, hansson@sei.cmu.edu`

Abstract

In this paper we consider data freshness and overload handling in embedded systems. The requirements on data management and overload handling are derived from an engine control software. Data items need to be up-to-date, and to achieve this data dependencies must be considered, i.e., updating a data item requires other data items are up-to-date. We also note that a correct result of a calculation can in some cases be calculated using a subset of the inputs. Hence, data dependencies can be divided into required and not required data items, e.g., only a subset of data items affecting the fuel calculation in an engine control needs to be calculated during a transient overload in order to reduce the number of calculations. Required data items must always be up-to-date, whereas not required data items can be stale. We describe an algorithm that dynamically determines which data items need to be updated taking workload, data freshness, and data relationships into consideration. Performance results show that the algorithm suppresses transient overloads better than (m, k)- and skip-over scheduling combined with established algorithms to update data items. The performance results are collected from an implementation of a real-time database on the real-time operating system μC/OS-II. To investigate whether the system is occasionally overloaded an offline analysis algorithm estimating period times of updates is presented.

1 Introduction

Embedded systems are commonplace and can be found in many different applications and domains, e.g., domestic appliances and engine control. The software in embedded systems is becoming more complex because of more functional requirements on them. Thus, the vast number of embedded systems being developed and the complexity of the software makes it important to have methods to handle specific software development issues of embedded systems. In this paper we look into data management in embedded systems, in particular maintaining data freshness and handling transient overloads.

We have found the following in a case study with two industrial partners, Mecel AB and GM Powertrain Sweden [10]:

- Development and maintenance costs of software is increasing, and one large part of this cost is data handling. This is also mentioned in [4, 20]. The main reasons are that the number of data items can be high, data storage is usually decentralized, data items must be up-to-date, and data items being used in a calculation should be derived from the same system state.
- A majority of the calculations are required to be finished before a given deadline otherwise the system suffers degraded performance, but it is acceptable to miss deadlines occasionally. Thus, we are focusing on soft real-time systems in this paper.
- These embedded systems become overloaded, and the software must be designed to cope with it, e.g., at high revolutions per minute of an engine the engine control software cannot perform all calculations. Few of the data items used in a calculation are compulsory to derive a usable result (we denote such data items as *required* and other data items as *not required*). For instance, in the engine control software, the calculation of fuel amount to inject into a cylinder consists of several variables, e.g., temperature compensation factor, and a sufficiently good result can be achieved by only calculating a result based on a few of these compensation factors. Thus, at high revolutions per minute, only a few of the compensation factors are calculated.

In our previous work we have addressed the first two bullets by introducing a real-time database, **D**ata **I**n **E**mbedded **S**ystems ma**I**ntenance **S**ervice (DIESIS), with support for

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

maintaining data freshness. We showed that updating data items on-demand and measuring data freshness in the value domain increase performance compared to measuring data freshness in time domain [9, 10]. The third bullet above describes a need to incorporate overload handling in DIESIS and in this paper we extend our previous work by an overload handler that considers workload, data freshness, and data relationships. The general problem of maintaining data freshness and considering data relationships is NP-hard in the strong sense (see section 2.3) and in this paper we describe a plausible specialization of the problem that results in an efficient algorithm. We also present an off-line algorithm that can derive the CPU utilization of a system for our algorithms.

Our contributions are:

- The **A**dmission **C**ontrol **U**pdating **A**lgorithm (ACUA) algorithm that determines which data items that should be updated to make their values up-to-date before a transaction can commence its execution. ACUA takes data freshness and data relationships into consideration and inserts into a schedule either all stale data items or only required data items. We say ACUA has two modes where the *required-mode* corresponds to scheduling required data items and the *all-mode* to scheduling all data items. Switching between all-mode and required-mode is triggered by using a utilization check. The CPU utilization of scheduled updates plus admitted transactions is compared to a bound denoted RBound [18] (see section 3 for details). This algorithm is referred to as the ACUA-RBound algorithm.

- An off-line algorithm that calculates the CPU utilization of a system consisting of data items that get updated by ACUA.

Performance results show that ACUA-RBound immediately suppresses a transient overload, and it also performs better than (m, k)-scheduling [12] and skip-over scheduling [16] of transactions updating data according to the OD algorithm [2]. Both (m, k)- and skip-over scheduling can be used to handle transient overloads.

The outline of the paper is as follows. Section 2 describes DIESIS. Section 3 describes ACUA-RBound. Section 4 describes the off-line algorithm for calculating CPU utilization. Section 5 gives the performance results, section 6 the related work, and finally, section 7 concludes the paper.

2 Real-Time Database

This section describes the data and transaction model used in DIESIS (section 2.1), a scheme for determining which data items are stale (section 2.2), and ACUA (section 2.3).

2.1 Data and Transaction Model

Embedded systems that monitor a natural environment can represent the set of data items as *base items*, i.e., data items that are read from sensors or communication links, and *derived data items*, i.e., data items that are calculated from a set consisting of base items and derived data items. The set of base items is denoted B, and the set of derived data items is denoted D.

The values a calculation produces depend on values derived by other calculations. Hence, there are precedence constraints on the calculations, and these can be described in a directed acyclic graph (DAG), where every node represents a data item (and a calculation updating the data item) and a directed edge from node n to n' means that calculating the data item represented by n' requires reading the data item represented by n. The DAG is denoted *data dependency graph*, G, and an example of a data dependency graph is given in figure 1 and it shows a subset of data in an engine control software using DIESIS [9]. Base items have zero in-degree, and nodes representing actuator values have zero out-degree; these nodes are denoted actuator nodes. The read set of a data item d_i, i.e., the data items that are read and used when deriving d_i, is denoted $R(d_i)$ and constitute the immediate parents of d_i in G. The read set of a data item $R(d_i)$ can be divided into *required* data items, denoted $RR(d_i) \subseteq R(d_i)$, and *not required* data items, denoted $NRR(d_i) \subseteq R(d_i)$, $RR(d_i) \cap NRR(d_i) = \emptyset$. The immediate children of d_i represent the data items that are derived from d_i. The ancestors of a data item d_i are the data items that are on paths leading to d_i.

We assume the value of d_i is correct if at least all data items in $RR(d_i)$ are up-to-date when deriving d_i. We furthermore assume that values of data items in $RR(d_i)$ can be based on only up-to-date required data items. This means that the system has still a correct behavior if all transactions only use up-to-date values on required data items.

Transactions are divided into *sensor transactions (STs)* that are periodically executed updating the base items, *user transactions (UTs)* that are started by applications in the software, and *update transactions* or *triggered updates (TUs)* that are started by DIESIS to update data items before a user transaction starts to execute. Every transaction updates one data item, and, thus, there exists a mapping via the data dependency graph G and a function pointer to a transaction.

In this paper, each UT is periodically invoked requesting actuator nodes in G. This resembles an embedded system where actuator nodes represent actuator signals that are sent periodically to actuators.

2.2 Determining Stale Data

We now discuss how data items can be determined to need to be updated. Data freshness measured using similarity is in this paper defined as follows.

Definition 2.1. *Let v'_{d_i} and v''_{d_i} be two values of data item d_i. These two values are similar if the distance between them is less than a bound denoted data validity bound, δ_{d_i}. Let a data item d_j be derived by using v'_{d_i}. If $|v'_{d_i} - v''_{d_i}| \leq \delta_{d_i}$ we say d_j is up-to-date with respect to v''_{d_i}, otherwise d_j is stale.*

Since similarity uses the value domain of data items to define their freshness, the freshness of a data item, d_i, only needs to be reconsidered at an update of any of its ancestors, i.e., a member of $R(d_i)$. Hence, the execution of calculations is data-driven, meaning that during periods when the external environment is in a steady state fewer calculations need to be executed compared to during periods when the external environment is in a transient state. We show this behavior in the performance evaluations in section 5.

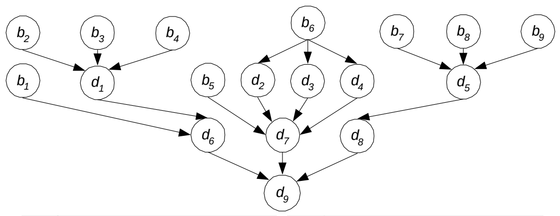
Adelberg et al. showed that executing updates on-demand uses the CPU resource most efficiently when comparing different updating algorithms [1]. We use on-demand updating of data items in this paper, so there must be a way to postpone recalculations of data items and start a subset of them in response to the starting of a user transaction. In DIESIS there is a scheme, **a**ffected **u**pdating **s**cheme (AUS), that marks a data item as affected when any of its immediate parents in the DAG changes to a dissimilar value according to definition 2.1. Looking at figure 1, when b_7 becomes dissimilar data item d_5 is marked as affected by this change. The AUS scheme does the following:

- keeps base items up-to-date by periodically updating them, and
- marks immediate children of d_i as affected when a recalculation of d_i finds the new value to be dissimilar to the old.

Another way to define data freshness is to use an absolute validity interval (AVI) and the data freshness is defined as follows [24].

Definition 2.2. *Let x be a data item. Let $timestamp(x)$ be the time when x was created and $avi(x)$, the absolute validity interval, be the allowed age of x. Data item x is absolutely consistent when:*

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



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

* Enr. = enrichment

Figure 1. An example of a DAG.

Absolute validity intervals are, as can be seen from the definition above, defining data freshness in the time domain. AVIs are used in an updating algorithm, on-demand (OD), described by Ahmed and Vrbsky [2]. In OD, every time a data item is read by a transaction, the age of the data item's value is checked. If the value is too old, an update is triggered that is executed before the triggering transaction continues to execute. A triggered transaction can also trigger updates. Thus, OD has the effect of traversing G bottom-up, i.e., in the reverse direction of the edges. OD is used in the performance evaluations in section 5.

2.3 Admission Control Updating Algorithm

In this section we describe the **A**dmission **C**ontrol **U**pdating **A**lgorithm (ACUA) algorithm that decides which data items need to be updated when a transaction starts. The decision is based on markings by the AUS scheme and data relationships. The set-union knapsack problem (SUKP) [15] can be reduced to the problem of scheduling updates of data items to keep them up-to-date. The reduction is given in [11]. SUKP is NP-hard in the strong sense. In this paper, we reduce this complexity to make ACUA efficient to execute on-line by considering only two cases: (i) schedule all updates and (ii) schedule required updates. The computational complexity of ACUA is polynomial in the size of G.

ACUA is implemented by traversing G top-bottom in a breadth-first approach. Data structures are used to keep information necessary to put updates in a schedule containing possibly stale data items. The benefit of using a top-bottom traversal with a data structure compared to the bottom-up approach used by OD is that ACUA can be extended with different functionality, e.g., in addition to schedule stale data items it is possible to calculate probabilities that up-

```
ACUA($d$, $ancestors$, $allMode$)
 1: for all $x$ in $ancestors$ do
 2:     status($x$)
 3:     if $x.marked == true$ then
 4:         put an update for $x$ into schedule
 5:     end if
 6:     for all immediate children $c$ of $x$ do
 7:         if ($c$ is required and allMode is false) or
              (allMode is true) then
 8:             inheritstatus($c$, $x$)
 9:         end if
10:     end for
11: end for
```

Figure 2. The ACUA algorithm.

```
inheritstatus($c$,$x$)
 1: $c.parents[c.parentnum].marked = x.marked$
 2: $c.parentnum++$
status($x$)
 1: if $x$ is marked then
 2:     $x.marked = true$
 3: else
 4:     $x.marked = false$
 5: end if
 6: for all $p$ in $x.parents$ do
 7:     $x.marked = x.marked \lor p.marked$
 8: end for
```

Figure 3. Help functions.

dates get executed [11], i.e., in one traversal of G a schedule of updates and the probabilities that they get executed can be generated using ACUA.

ACUA is described in figure 2. The parameter d is the data item a user transaction requests, $ancestors$ is the set of all ancestors sorted by increasing level (see definition 2.3), and $allMode$ is true if all data items should be considered for being updated and false if only required data items should be considered. A level is defined as follows:

Definition 2.3. *Each base item b has a fixed level of 1. The level of a derived data item d is determined by the longest path in a data dependency graph G from a base item to d. Hence, the level of d is $level(d) = \max_{\forall x \in R(d)}(level(x)) + 1$, where $R(d)$ is the read set of data item d.*

The set $ancestors$ is generated off-line by depth-first traversal of G from the node representing d, after the depth-first traversal the visited nodes are sorted according to increasing level. Line 7 of ACUA checks whether an immediate child of an ancestor should be considered for being updated. A data item that is required with respect to another data item can be distinguished by marking the edge in G, e.g., with number 2, in the adjacency matrix describing G. The function status(x) (see figure 3) calculates the marking of x based on the inherited markings from ancestors of x (line 7). The inherited markings are traversed down G with the help of function inheritstatus(c,x) (see figure 3). It is easy to see that the data structure used in ACUA, status and inheritstatus can be extended to calculate different properties of the updates, e.g., probability of being executed.

In summary, a marking by AUS is traversed down the graph and updates are scheduled as they are found to be needed (line 4 in figure 2). When ACUA has constructed a schedule of updates as a response to an arrival of a user transaction, DIESIS starts to execute the updates before the UT commences. The updating scheme AUS is active whenever a data item is written to the database. This means that a data item might be in the schedule but it never becomes marked because an update in an immediate parent never resulted in a stale data item. Thus, only updates for the data items that are marked by AUS are started by DIESIS. In this way the workload is automatically adapted to how much data items change in the external environment. The experimental results presented in section 5 confirm this. Computational complexity of ACUA is polynomial in the number of ancestors of a data item, i.e., $O(|V|)$, where V is the set of vertices of G, because ACUA loops through all ancestors of a data item which, in the worst case corresponds to the whole graph G.

3 Admission Control using ACUA

The load represented by the admitted updates can be expressed as $U = \sum_{\forall i \in ActiveUT} \frac{C_i}{P_i}$, where $ActiveUT$ is the set of active user transactions, C_i is the sum of execution times of updates and UT τ_i, and P_i is the period time of UT τ_i. In order to successfully execute all UTs, U always needs to be below a specific bound. In this paper we choose to use RBound [18] since it gives a bound tighter than RMA. RBound says that if

$$\sum_{i \in ActiveUT} \frac{C_i}{P_i} \leq (m-1)(r^{1/(m-1)} - 1) + \frac{2}{r} - 1, \quad (2)$$

where m is the number of active UTs and r is the ratio $P_{smallest}^{\log_2 \lfloor \frac{P_{smallest}}{P_{highest}} \rfloor} / P_{highest}$, where $P_{smallest}$ is the smallest period time of active UTs and $P_{highest}$ is the highest [18]. As with the well-known Liu and Layland bound [21], RBound is sufficient but not necessary.

Admission control of updates in DIESIS using RBound is done as follows. When a UT arrives to DIESIS, ACUA using $allMode$ set to true, i.e., all-mode of ACUA, is used to generate a schedule of updates, i.e., required-mode is used. If (2) is false, then a new schedule using ACUA with

$allMode$ set to false is generated. The execution time of a UT is estimated to the sum of execution times in the schedule.

In practice only one execution of ACUA is needed, because the not required data items can be marked, and removed from the schedule if (2) is false. Using ACUA to schedule updates and the feasibility test RBound is denoted ACUA-RBound.

4 Analyzing CPU Utilization

Since DIESIS executes only the updates of data items that need to be updated, there is a need to determine off-line the mean time between invocations of updates of data items because the mean time between invocations of an update (of a data item d_i) can be used to calculate the CPU utilization by taking $\frac{C_{d_i}}{MTBI_{d_i}}$. Here C_{d_i} is the worst-case execution time of the update of d_i and $MTBI_{d_i}$ is the mean time between invocations of the update of d_i. There are two things that determine the mean time between invocations of an update of d_i: (i) the period times of UTs, and (ii) the probability that a recalculation of a member of the read set $R(d_i)$ results in a change in the value of d_i. See figure 5 for an example. Timeline 1 (the timeline to the left of the encircled 1) shows the timepoints where sensors are updated. We assume every time any of the sensors is updated, data item d_k is affected by the change. Timeline 2 shows the time instances where an update of d_k is called and these time instances come from the actuator nodes representing periodic UTs. Timeline 3 shows which calls of the update of d_k result in an execution of the update. An algorithm, MTBIOfflineAnalysis, determining the mean time between invocations of updates of data items in the system is given below. We see that the algorithm above traverses G top-bottom. In this paper, timelines have a length of 400000 time units which give accurate values on mean time between invocations. In order to get an accurate mean time between invocations, the length of the timelines needs to be equal to the hyperperiod of period times of the read set and tasks. To shorten the execution time of MTBIOfflineAnalysis, the length of timelines can be fixed, but the length must be order of magnitudes longer than the period times of elements in the read set and of tasks in order to capture the arrival pattern of execution of updates. Line 10 determines whether an occurrence of an update of a read set member will make the value of d_i stale. The CPU utilization can easily be calculated by calculating timeline $T3$ for each data item and then derive the mean time between invocations on that timeline followed by calculating $C_{d_i}/MTBI_{d_i}$. The total CPU utilization is the sum of $C_{d_i}/MTBI_{d_i}$ for each data item d_i. If the CPU utilization is below a threshold given by the scheduling algorithm being used, then there should be no deadline misses. A sensor network can be described as a graph and a sensor network database measuring data freshness in the value domain [25] can thus be analyzed using MTBIOfflineAnalysis.

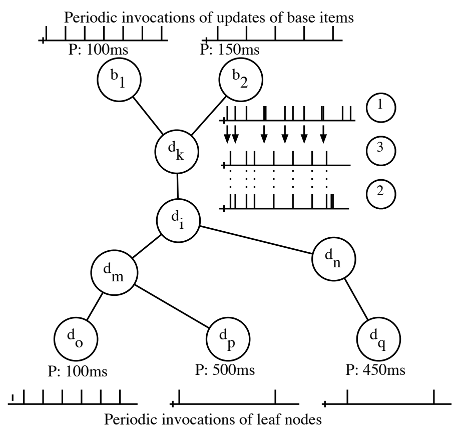

Periodic invocations of updates of base items
P: 100ms
P: 150ms
b1
b2
dk
1
3
2
di
dm
dn
do
dp
dq
P: 100ms
P: 500ms
P: 450ms
Periodic invocations of leaf nodes

Figure 5. An example of invocation times of updates.

5 Performance Evaluations

5.1 Evaluated Algorithms

In the evaluations the deadline miss ratio is used as a performance metric. We compare AUS, ACUA using all-mode (denoted ACUA-All), ACUA-RBound to OD [2] in three different settings: OD-All, OD-(m, k) and OD-Skipover. When a UT arrives to the system, OD traverses G bottom-up from the data items written by the UT and visited data items are updated if they are stale according to AVIs (definition 2.2). Thus, using OD, data freshness is measured in the time domain.

The algorithm OD-(m, k) executes updates of data items according to OD and the priorities of UTs are set according to the (m, k) algorithm where $m = 1$ and $k = 3$, thus, four distances are possible (see [12] for details). The dynamic priorities of (m, k) are implemented in μC/OS-II by priority switches. Five priorities are set aside for each distance. When a UT starts its distance is calculated and its priority is switched to the first free priority within the set for that distance. OD-All uses OD from [2] and the UTs' priorities are fixed. OD-Skipover uses OD to update data items and the skip-over algorithm is red tasks only where, in this paper, every third instance of UTs are skipped [16]. ACUA-RBound is the algorithm described in section 3.

The evaluations show that using ACUA-RBound a transient overload is suppressed immediately. OD-(m, k) and OD-Skipover cannot reduce the overload to the same ex-

```
1: Assign period of tasks that can use an update of a data item
2: Draw a timeline $T3$ for each base item with each occurrence of an update of it
3: for all levels of $G$ starting with level 2 do
4:     for all data items $d_i$ in the level do
5:         Merge all $T3$ timelines of $x \in R(d_i)$ and call the timeline $T1$
6:         Create $T2$ with possible updates of $d_i$, i.e., when derivatives of $d_i$ are called.
7:         $p = 0$
8:         for all Occurrences $ot2_i$ in $T2$ do
9:             for all Occurrences $ot1_i$ in $T1$ in the interval $]ot2_i, ot2_{i+1}]$ do
10:                if $r \in U(0, 1) \leq p$ then
11:                    put $ot2_{i+1}$ into a timeline $T3$
12:                    $p = 0$
13:                    break
14:                else
15:                    increase $p$ with probability that an update of a read set member affects the value of $d_i$.
16:                end if
17:            end for
18:        end for
19:    end for
20: end for
```

Figure 4. MTBIOfflineAnalysis algorithm.

tent as ACUA-RBound. Thus, constructing the contents of transactions dynamically taking workload, data freshness, and data relationships into consideration is a good approach to overload handling.

5.2 Simulator Setup

DIESIS is implemented on top of the real-time operating system μC/OS-II [17].[1] Five tasks are executing periodically to model five time-based tasks in an engine control unit, and they invoke UTs that execute with the same priority as the task. Other tasks in an engine control unit can be tasks that are triggered based on the speed of the engine. The tasks are prioritized according to rate monotonic (RM [21]), i.e., priorities are proportional to the inverse of task frequencies. The base period times are: 60 ms, 120 ms, 250 ms, 500 ms, and 1000 ms. These period times are multiplied with a ratio to get a specific arrival rate of tasks. All tasks start user transactions that derive actuator nodes and the tasks can use approximately the same number of actuator nodes.

In the experiments, a database with 45 base items and 105 derived items has been used. A database of 150 data items represents update intensive data items in an embedded system, e.g., in the engine management system 128 data items are used to represent the external environment and actuator signals. Tasks have specialized functionality so data items tend to seldom be shared between tasks, thus, the data dependency graph G is broad (in contrast to deep). 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 immediate parents consisting of only base items. 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 a read set consisting of only base items, and the remaining 10% are other derived items. These figures are rounded to nearest integer. The required data items are chosen by iteratively going through every member of $R(d_i)$ and set the member to be required with the probability $1/|R(d_i)|$. The iteration continues as long as $|RR(d_i)| = 0$. The number of derived items with only base item parents is set to 30% of the total number of derived items.

Every sensor transaction executes for 0.2 ms and has a period time of 100 ms. 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 $[0, 10]$. Every simulation is executed 5 times and the results shown 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.

To model changing data items, every write operation is taking a value from the distribution U(0,350) and divides it with a variable, $sensorspeed$, and then adds the value to the previous most recent version. To get a load of the system at an arrival rate of 20 UTs per second that shows the perfor-

[1]The simulations run 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.

Table 1. CPU utilizations.

Mode	$sensorspeed$	p base	p derived	U
all-mode	1	0.20	0.20	1.38
all-mode	10	0.02	0.20	0.72
required-mode	1	0.20	0.20	0.28

mance of the algorithms the data validity intervals are set to 900 for all data items, i.e., $\delta_{d_i} = 900$, and the absolute validity intervals are set to 500 because with a mean change of 175 and a period time of 100 ms on base items a base item's value is, on average, valid for at least 500 ms. The probability that an update must execute is 175/900=0.2 where 175 is the mean value change. Table 1 shows the CPU utilization, calculated using MTBIOfflineAnalysis in section 4, where $sensorspeed$ is 1 and 10. We see that the system should be overloaded when ACUA-All is used in a transient state ($sensorspeed = 1$, i.e., sensors change much) and not overloaded when required-mode is used. In a steady state, i.e., $sensorspeed = 10$, the system is not overloaded.

The concurrency control algorithm that is used is High-Priority Two-Phase Locking (HP2PL). We have shown in a previous work that using OCC gives the same performance as using HP2PL [8].

5.3 Experiments

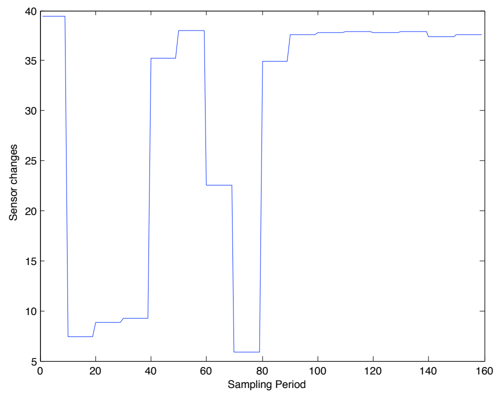
Figure 6(b) shows the performance of the algorithms where sampling periods are 500 ms. We show the mean deadline miss ratio for the intervals where $sensorspeed$ is set to the same value, which is periods of 5 seconds, i.e., 10 sampling periods. The max mean deadline miss ratio is shown in table 2. The sensors change as showed in figure 6(a) . The deadline miss ratio of OD-All, OD-(m, k), and OD-Skipover is unaffected of the sensor changes which is expected because using AVIs for data freshness makes updating unaware of values of data items. The miss ratio drops using ACUA-All when the number of sensor changes per time unit is small as in the interval 15–40 sampling periods and 70–80 sampling periods. This is also expected since the entry $sensorspeed = 10$ in table 1 says the system should be not overloaded.

The data consistency achieved by skip-over scheduling is worse than the consistency achieved by ACUA-All, ACUA-RBound, OD-All, and OD-(m, k), because using skip-over scheduling every third instance of a task never updates any data items. For ACUA-All and ACUA-RBound data items are always updated such that transactions use up-to-date values on required data items. OD-All and OD-(m, k) also use up-to-date values on data items.

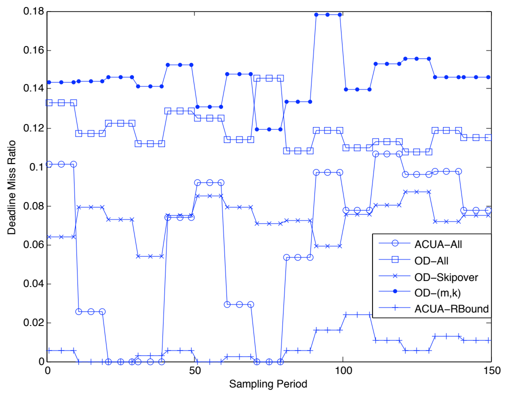
Using skip-over scheduling improves the performance compared to OD-All. However, ACUA-All has similar per-

Table 2. Max mean deadline miss ratio (MMDMR) for transient and steady states.

Algorithm	MMDMR transient; steady
OD-All	0.146;0.146
OD-(m, k)	0.178;0.146
OD-Skipover	0.090;0.080
ACUA-All	0.109;0.02
ACUA-RBound	0.029;0.002



(a) Sensor changes per time unit



(b) On-demand updating algorithms

Figure 6. Performance of overload handling algorithms.

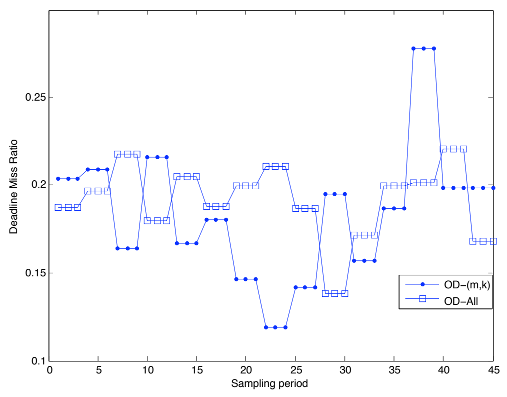


Figure 7. Deadline miss ratio for task with second highest priority.

formance as OD-Skipover. Thus, ACUA-All has similar deadline miss ratio compared to OD-Skipover and the data consistency is higher. OD-(m,k) does not perform, overall, better than OD-All and that is because the task having the highest priority according to RM gets a dynamic priority that might be lower than other running tasks with the same distance. Thus, the task with shortest period time misses more deadlines but other tasks meet more deadlines, and this is for instance showed in figure 7 where the deadline miss ratio for tasks with second highest priority is lower for OD-(m,k) compared to OD-All. However, the performance of OD-(m,k) cannot be better than OD-Skipover because task instances are skipped using OD-Skipover which they are not using OD-(m,k).

Skip-over gave the best effects on deadline miss ratio using the OD algorithm. Figure 8 shows the performance of ACUA-All using skip-over to skip every third task instance. The deadline miss ratio drops by introducing skip-over, but it is not affected much by the skips. Hence, to reduce workload in an overloaded system other means must be used than skipping invocations of tasks. The ACUA algorithm can generate schedules containing data items that might need to be updated, which can be seen in figure 6(b). To improve the performance of ACUA-All, the schedules' lengths must be varied depending on the workload. However, data relationships must still be considered. One way to shorten the length of a schedule is to use the required-mode of ACUA. Switching to required-mode when the RBound feasibility test fails gives the performance denoted ACUA-RBound in figure 6(b). As can be seen ACUA-RBound decreases the deadline miss ratio better than any of the other algorithms and suppresses the deadline miss ratio when the system goes from a steady to a transient state, e.g., sampling period 80, where number of sensor changes from low to high. The mean deadline miss ratio is at maximum 0.029 in the interval 100 to 110 where sensors change much, i.e., the system is in a transient state, compared to OD-Skipover that has its maximum mean at 0.09. Using ACUA-RBound, the deadline miss ratio can be above zero because if the utilization bound (2) (section 3) is false, required-mode of ACUA is used, but (2) can still be false due to admitted UTs that have used all-mode. One way to resolve this is to reschedule updates of active UTs, and this is our future work.

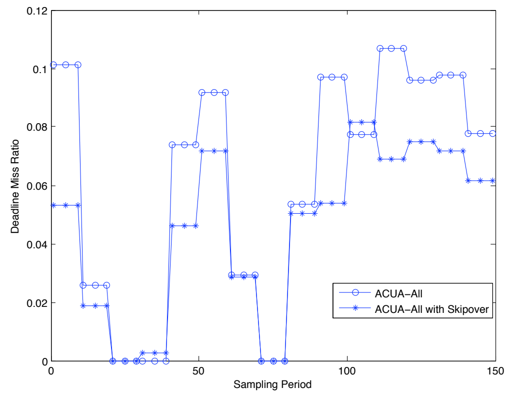


Figure 8. ACUA-All using skip-over.

6 Related Work

The research on admission control in real-time systems has been extensive [5,6,12–14,16]. However, the work that has been done primarily focuses on admission control of independent tasks, whereas we in this paper focus on admission control where data has relationships.

Work on maintaining data freshness can be classified into (i) off-line algorithms determining period times on tasks [19,26,27], and (ii) on-line algorithms [2,7,9,10,14,23]. In our previous work we showed that maintaining data freshness on-line measuring data freshness in the value domain can use the CPU resource more efficient compared to if data freshness is measured in the time domain [9,10]. However the on-line algorithms might not work properly in the case of overloads, because they cannot guarantee that data items are updated, before a transaction starts, in such a way that the transaction can produce an acceptable result. The reason is that the general problem of choosing updates and considering data relationships is NP-hard in the strong sense (see section 2.3) and previous on-line algorithms are simplified to reduce computational complexity in such a way that they reject updates when the updates cannot be fitted within the available time. Let us assume d_9 in figure 1 is about to be read in a transaction and d_3 and d_5 are marked as potentially affected by changes in the external environment. An on-demand updating algorithm traverses d_7 and d_3 followed by d_8 and d_5. Let us assume there is time available for d_7 but not d_3 and for d_8 but not d_5. Thus, updates are executed for d_7 and d_8 and they will read old values on d_3 and d_5 since they were not updated. In this paper we choose to divide data items into required and not required data items which is justified by examples in industrial applications, and this gives a polynomial time algorithm without the exemplified problem.

Tasks in the imprecise computation model can be described with one of the following approaches [22]. *Milestone approach:* The result of a task is refined as its execution progresses. A task can be divided into a mandatory and an optional part, where the result after executing the mandatory part is acceptable, and the result after also executing the optional part is perfect, i.e., the error of the calculation is zero. *Sieve approach:* A task consists of operations where not all of them are compulsory [3]. A typical example is when a data item's value can be updated or used as is, i.e., the update is skipped. *Primary/alternative approach:* The task can be divided into a primary task containing functionality to produce a perfect result. The alternative task takes less time to execute and the result is acceptable. One of the primary and the alternative task is executed.

ACUA is consistent with the imprecise computation model because ACUA can construct a schedule that gives an imprecise but acceptable result. The result is acceptable because required data items are always updated. The approach of dividing data items into required and not required data items has industrial application as was discussed in section 1, e.g., a subset of fuel compensation factors must be updated in order to get an acceptable result. In this paper, we have focused on on-line constructing the content of user transactions by considering workload, data freshness, and data relationships to get acceptable results. To the best of our knowledge this is the first time such an approach is evaluated.

Kang et al. describe a flexible data freshness scheme that can reduce the workload by increasing the period times on updates of data items [14]. A feedback approach is used where a monitoring of changes in deadline miss ratio results in changing period times of updates within given bounds. The work in [14] does not consider data relationships nor data freshness measured in the value domain. Moreover, using a feedback approach introduces a settling time, i.e., it takes a time before the system stabilizes after a workload change. Some systems need fast reactions, and our evaluations show that using ACUA with a feasibility test lets the system react immediately on a workload change.

Ramamritham et al. developed algorithms for data dissemination of data on the web and data freshness is measured in the value domain of data items [7, 23]. Their work considers the problem of refreshing data values on clients when values dynamically change on a server, which can be mapped to when derived data items in level 2, i.e., immediately below base items, should be refreshed. Push and pull techniques are combined meaning that on-demand and time triggering of updates of data are combined. In our work, we consider, on a single CPU embedded system, data relationships in several levels.

7 Conclusions and Future Work

This paper has described an algorithm that handles overloads by determining, on-line, which calculations should be performed such that up-to-date data items are used. Data relationships are taken into consideration when deciding which calculations to perform. An overload is due to too many concurrent calculations updating data items, and a response to the overload is to reduce the amount of concurrent calculations. The updating algorithm has two modes. The required-mode considers a subset of the data items when determining which should be updated. This subset consists of *required* data items that have to be up-to-date. The all-mode considers required and *not required* data items. This is the default behavior of the updating algorithm. A utilization check denoted RBound [18], which is an enhancement to the Liu and Layland RM bound, is used to decide when to switch between all-mode and required-mode. Simulation results show that the algorithm can successfully be used for reducing the effects of overloads. Overloads are immediately suppressed using the updating algorithm and they are suppressed to a larger extent compared to using (m, k)- and skip-over scheduling with an established algorithm that updates data items. Moreover, we also present an off-line algorithm to derive the CPU utilization of a system using our overload handling algorithm.

Future work is to develop an on-line version of the CPU utilization algorithm where it can be used as an estimator in a feed-forward control-loop.

References

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

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

[3] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Incorporating unbounded algorithms into predictable real-time systems. Technical report, Real-Time Systems Research Group, Department of Computer Science, University of York, 1993.

[4] E. Barry, S. Slaughter, and C. F. Kemerer. An empirical analysis of software evolution profiles and outcomes. In *ICIS '99: Proceeding of the 20th international conference on Information Systems*, pages 453–458, Atlanta, GA, USA, 1999. Association for Information Systems.

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

[6] A. Datta, S. Mukherjee, P. Konana, I. R. Viguier, and A. Bajaj. Multiclass transaction scheduling and overload management in firm real-time database systems. *Information Systems*, 21(1):29–54, 1996.

[7] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: disseminating dynamic web data. In *Proceedings of the tenth international conference on World Wide Web*, pages 265–274. ACM Press, 2001.

[8] T. Gustafsson, H. Hallqvist, and J. Hansson. A similarity-aware multiversion concurrency control and updating algorithm for up-to-date snapshots of data. In *ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 229–238, Washington, DC, USA, 2005. IEEE Computer Society.

[9] 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.

[10] 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.

[11] T. Gustafsson and J. Hansson. Data freshness and overload handling in embedded systems. Technical report, `http://www.ida.liu.se/~thogu/gustafsson06admission.pdf`, 2006.

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

[13] J. R. Haritsa, M. Livny, and M. J. Carey. Earliest deadline scheduling for real-time database systems. In *IEEE Real-Time Systems Symposium*, pages 232–243. IEEE Computer Society Press, 1991.

[14] K.-D. Kang, S. H. Son, and J. A. Stankovic. Managing deadline miss ratio and sensor data freshness in real-time databases. *IEEE Transactions on Knowledge and Data Engineering*, 2003.

[15] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.

[16] G. Koren and D. Shasha. Skip-over: algorithms and complexity for overloaded systems that allow skips. In *RTSS '95: Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*, page 110, Washington, DC, USA, 1995. IEEE Computer Society.

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

[18] S. Lauzac, R. Melhem, and D. Mossé. An improved rate-monotonic admission control and its applications. *IEEE Transactions on Computers*, 52(3):337–350, 2003.

[19] C.-G. Lee, Y.-K. Kim, S. Son, S. L. Min, and C. S. Kim. Efficiently supporting hard/soft deadline transactions in real-time database systems. In *Third International Workshop on Real-Time Computing Systems and Applications, 1996.*, pages 74–80, 1996.

[20] E. A. Lee. What's ahead for embedded software? *Computer*, pages 18–26, 2000.

[21] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[22] J. W. S. Liu, K.-J. Lin, W.-K. Shih, A. C. shi Yu, J.-Y. Chung, and W. Zhao. Algorithms for scheduling imprecise computations. *Computer*, 24(5):58–68, 1991.

[23] 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.

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

[25] M. A. Sharaf, J. Beaver, A. Labrinidis, and P. K. Chrysanthis. Tina: A scheme for temporal coherency-aware in-network aggregation. In *Proceedings of 2003 International Workshop on Mobile Data*, 2003.

[26] M. Xiong and K. Ramamritham. Deriving deadlines and periods for real-time update transactions. In *Proceedings of the 20th Real-Time Systems Symposium*, pages 32–43. IEEE Computer Society Press, 1999.

[27] M. Xiong, R. Sivasankaran, J. Stankovic, K. Ramamritham, and D. Towsley. *Scheduling access to temporal data in real-time databases*, chapter 1, pages 167–192. Kluwer Academic Publishers, 1997.