2006-01-0305

Database Functionality in Engine Management System

Thomas Gustafsson
Department of Computer and Information Science, Linköping University, Linköping, Sweden

Jörgen Hansson
Department of Computer and Information Science, Linköping University, Linköping, Sweden
Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA

Anders Göras
Mecel AB, Åmål, Sweden

David Holmgren and Jouko Gäddevik
GM Powertrain Sweden, Södertälje, Sweden

ABSTRACT

Embedded systems of today need to manage more data than ever before. The main reasons for the increase in number of data items are increased functional requirements on the software. With a larger amount of data to manage comes the problems of storing data and its meta-information, sharing between programmers which data items exist, and ensuring freshness and consistency requirements of the data items.

In this work we focus on efficient data management in embedded systems, and develop a database for such systems. The database has support for transactions, snapshots, and data freshness. We argue that the software maintenance efforts can be reduced using a database, and our performance results show that the performance can be increased without affecting consistency of data values.

INTRODUCTION

With the advent of devices with the availability of larger amount of memory and more computational power, the software development has moved from assembly programming to using high-level programming languages such as C and C++. High-level programming languages can abstract data into structures and classes. Using high-level programming languages, programs may be programmed in a modular way, where data can be global and accessible from all modules, or local and only accessible by functions within a specific module. Global data can be used for communication between modules. This division of data into global and module-specific data, complicates the way one can get a view of existing data items and their usage. COMET project and other projects claim that using a database in an embedded system leads to reductions in development costs, improvements in software quality, and increased maintainability of the software (see also the appendix) [22].

In an embedded system we have studied, an engine control software, the software is divided into applications with certain responsibilities, e.g., calculating fuel and diagnosing the system. Each application consists of tasks that perform calculations. The calculations take inputs forming a subset of all available data and derive results composing a subset of the available data. Furthermore, some of the data items are subject to freshness constraints, meaning that they might be stale as the external environment changes. The freshness constraints are ensured by deriving the data items often enough, i.e., the data values never get older than what is dictated by a worst-case scenario. By storing data decentralized, it is easy to introduce duplicates of data items—since it is hard to get a picture of which data items exist—and update data items multiple times only to be sure the freshness requirements are fulfilled. Hence, excess memory and CPU resources might be needed, which is an undesirable consequence of the data management technique that is used.

An important aspect of the deterioration of data values as external environment changes is that a calculation should get a consistent view of its input data. This means that the input data must be sufficiently correlated in time. One way to achieve this is to freeze the input data and let the calculation read the frozen, unchanged, data [21]. The frozen data is denoted a snapshot. Data freshness issues can also be handled by deriving period times on calculations such that freshness of data items and timeliness of the calculations are guaranteed [13, 14, 17, 23]. However, these techniques do not change the way data is stored and fetched, and, thus, cannot help in the data maintenance effort. Also, calculations are always carried out in these techniques which might waste resources if values of data items change within some bounds, then the calculations would produce the same results that are already stored in the system. In essence, we would like a system that makes software maintenance easier and uses available resources efficiently, e.g., the CPU.

Olson describes different criteria for choosing a database for an embedded system [19]. He classifies databases into

client-server relational databases, client-server object-oriented databases, and, finally, embedded library databases. The embedded library databases are explicitly designed for embedded systems. The embedded database links directly into the software, and there is no need for a query language such as SQL. Existing client-server databases are not appropriate for real-time systems, because transactions cannot be prioritized. Nyström et al. identify that there are no commercial alternatives of embedded databases suited for embedded real-time systems [22]. Further, it is pointed out that most database systems use two-phase locking to ensure concurrent transactions do not interfere with each other [19]. Two-phase locking is an approach to concurrency control that guarantees the consistency of the data [6]. However, for some applications the consistency can be traded off for better performance [18]. This trade-off is not possible if only two-phase locking is available.

In the engine management system (EMS) software, the merits of using a database instead of storing data globally and locally in modules are that all data and data maintenance programming code are concentrated to one place in the software. In this way, the data maintenance code can easily be changed since it is hidden behind the API. Furthermore, the database can have functionality that would be hard to implement using data structures spread out in the software, e.g., the database takes care of updating data items that are needed by a calculation. Such functionality is not present in current database systems (see the surveys [19, 22]), so we have developed a database suited for embedded systems, **D**ata **I**n **E**mbedded **S**ystems ma**I**ntenance **S**ervice (DIESIS), with support for transactions, data freshness, consistent views of data, and reduced consistency requirements giving higher performance but still correct results. Our performance results achieved by using DIESIS in engine control show that the number of calculations needed to keep data items fresh with respect to changes in the external environment can decrease considerably compared to methods normally used in embedded systems, which are updating data items periodically.

The outline of this paper is as follows. In the next section we state our problem formulation. Section "Description of DIESIS" describes DIESIS and the algorithms being used. Section "Performance Evaluations of DIESIS" shows results from conducted simulations, and section "Conclusions" concludes this paper.

PROBLEM FORMULATION

Figure 1 shows a calculation using module-specific and global data. We have seen in the section "Introduction" that how the data is stored clearly affects the maintainability of the software. Having data items stored decentralized, i.e., spread out in different modules, makes it hard to overview which data items exist, what are their timing and freshness requirements, and whether these requirements are fulfilled. In figure 1, it is impossible to see if, e.g., `globalData3`, is up-to-date when the calculation reads it.

The following bullets exemplify problems that are hard to solve using unsupervised storage of data, i.e., where data is fetched and stored uncontrollably to different memory areas.

```
/* code in a calculation */
/* local data */
int v1, v2, v3, result;
v1 = structName1.dataItem1;
v2 = globalData3;
/* perform calculations */
result = v3;
```

Figure 1: An example of a calculation accessing data from global and local data.

- For instance, let us denote the calculation in figure 1 as c_1, and assume another calculation, c_2, changes `structName1.dataItem1` to an intermediate value during its execution. Then there could be an interleaving of c_1 and c_2 such that c_1 reads the intermediate value of `structName1.dataItem1`. This, unwanted, effect is called *ghost update* (see the appendix for three other undesirable effects that can arise).
- Assume c_1 gets interrupted by some other calculations after the assignment `v1 = structName1.dataItem1`. Then if any of the calculations being executed before c_1 continues to execute updates `globalData3`, then the variables `v1` and `v2` can have values from different system states. This can affect the quality of the value of `v3`, and, thus, the end result of c_1.

Thus, the following problems exist when maintaining data in embedded systems:

- The effort of maintaining embedded software is large because data is spread out in different modules.
- Memory and CPU resources might be ineffectively used due to duplicate data items and unnecessary updating of data items.
- Calculations need to get a consistent view of its input data in order to control the external environment effectively.
- Calculations need to be controlled such that no unwanted and wrong results are stored and further used by other calculations.

We believe a wide range of embedded system software, e.g., all embedded systems doing any kind of control of the external environment, have the following properties. The embedded system:

- Makes readings of the external environment by using sensors or reading values arriving on communication links.
- Derives data items directly or indirectly from sensors or values arriving on communication links.
  - All data items are known before the system starts, i.e., functionality for adding and removing data items is not needed.

  - Data items have specified freshness requirements that are measurable in the value domain of the data item.
  - Calculations deriving data items often require values that are derived from the same system state, e.g., actuator signals.

- Starts calculations in response to external events, e.g., a clock tick of a periodic timer.
  - The calculations have soft deadlines (see the appendix).
  - Calculations can be assigned fixed priorities without affecting the intended functionality of the system.

- Has data that is volatile in the sense that all data is reinitialized at system startup. This means that calculations are independent from results produced in previous executions of the software.

We envision that using a database makes it easier to maintain the software in an embedded system. We have seen in the introduction that there are no available databases that address these problems. We believe that specializing a database for systems fulfilling the properties above can increase the performance compared to using a general database. Furthermore, by adopting an approach of central storage and maintenance of data implies that we have a global view of all data and its current status. This knowledge should be possible to exploit to effectively utilize available CPU-resources. Using a database for storing data, it would be easier to spot duplicate data items since they are stored in a central repository. Moreover, the database could have functionality that is difficult to implement using an unsupervised storage approach, e.g., concurrency control and data freshness insurance.

DESCRIPTION OF DIESIS

This section contains a description of the functionality of DIESIS. The section is divided into a description of the functionality and interface of DIESIS (section "Application Programming Interface"), and descriptions of the freshness and consistency functionality that are used in sections "Data and Transaction Model", "Data Freshness", "Updating Algorithms", and "Concurrency Control".

APPLICATION PROGRAMMING INTERFACE

DIESIS is written in the programming language C, and it is built to work with the real-time operating systems Rubus [1] and μC/OS-II [16]. DIESIS uses the notion of transactions. Transactions are atomic units of computational work, and, thus, do not see intermediate results from each other (an elaborate description can be found in the appendix). The operations that are allowed on data items are reading and writing.

Figure 2 shows a transaction. Every transaction is a C-function, since in this way the transaction is easily accessed by a function pointer. To register a transaction with DIESIS, the `BeginTransaction` function is called. The transaction is wrapped in a loop to model the possible occurrence of a transaction needing to be restarted. The only possible cause for a transaction to restart is that the operations of two concurrent transactions conflict meaning that at least one of them produce a wrong result. This is further discussed in section "Concurrency Control".

```
void TD9(s8 mode, void *arg)
{
    s8 transNr=TRANSACTION_START;
    DB_Data dbd,dbd0,dbd1,dbd2;
    while(BeginTransaction(&transNr, 10000, 200,
          LOW_PRIORITY_QUEUE, mode, D1))
    {
        UpdateDB(&transNr,D9,2500);
        ReadDB(&transNr, D6, &dbd0);
        ReadDB(&transNr, D7, &dbd1);
        ReadDB(&transNr, D8, &dbd2);
        /* Calculation of new value */

        WriteDB(&transNr,D9,dbd,&TD9);
        CommitTransaction(&transNr);
    }
}
```

Figure 2: A transaction deriving one value.

As is discussed in section "Introduction", the values of data items need to accurately reflect the state in the external environment. The function `UpdateDB` schedules data items needing to be updated before the current transaction continues deriving values (the scheduling is described in section "Updating Algorithms"). A value of a data item is read from DIESIS by using the `ReadDB` function. The values are enumerated with an enum type, e.g., `D6`, and the read value is stored in `dbd0`. A value is written to DIESIS using the `WriteDB` function. Finally, last in the loop, the `CommitTransaction` function should be called. This function checks if the values read by the transaction are consistent (this is further described in section "Concurrency Control"), and if that is the case the transaction can commit. The function registers the transaction as completed and `BeginTransaction` returns zero, which jumps out of the loop.

DATA AND TRANSACTION MODEL

This section describes the data and transaction models derived from the properties stated in section "Problem Formulation".

Data 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. In our work, we divide the data items into a set B of base items, and a set D of derived data items.

The values a calculation produce 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). The DAG is denoted *data dependency graph*, G, and an example of a data dependency graph is figure 3. Base items have zero in-degree, and nodes representing actuator values have zero out-degree. 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 $R(d_i)$. Furthermore, if a calculation derives a value of only one data item then there exists a simple mapping from precedence constraints in a DAG to calculations to data items.

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 calculation [21]. 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 items are updated.

Transaction Model

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.

The transaction in figure 2 is an update of data item `D9` in figure 3. The transaction is a user transaction when it is invoked by the software. The transaction is an update transaction if the data item `D9` has been scheduled by a user transaction, and determined to be necessary to be executed. Whichever of these two cases that apply is regulated with the argument `mode` to the function `TD9`.

DATA FRESHNESS

With the notion data freshness we mean a way to decide if a value of a data item is a correct reflection of the state of the external environment. As we saw in section "Introduction", in the EMS software, data freshness is implemented using aging of data items. When a data item d_i is too old it is considered to be stale, i.e., $|ct - t_{d_i}| > a_{d_i}$, where ct is current time, t_{d_i} is the time when the value of d_i was calculated, and a_{d_i} is the maximum allowed age on the value of d_i [20]. However, when a value of d_i is stale it may still be unchanged or close to the old value. We can reason about the *similarity* of two values instead of reason about their ages [9–12, 15]. Two values are said to be equal if they are similar according to a similarity relation. An example of a similarity relation is the one-dimensional distance between two values. If the distance is less than a specified bound then the values are similar, and, thus, equal.

In this work, the data freshness that is specified on every data item is a *data validity bound (DVB)* giving the allowed distance, e.g., using a data validity bound of 9 of data item `D3`, the two values 50 and 58 of data item `D3` are similar since the distance $|50 - 58|$ is less than 9.

UPDATING ALGORITHMS

In this section we describe an algorithm, On-Demand Top-Bottom with relevance check (ODTB), which is part of DIESIS, that creates a schedule of data items needing to be updated [11]. The algorithm is executed as a response to the start of a user transaction. The algorithm needs to find the data items that must be updated, and it needs to schedule these in the correct order, i.e., fulfilling the precedence constraints.

First, we discuss how ODTB can decide which data items need to be updated. The algorithm uses similarity to measure data freshness. 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 can be 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. Using periodic invocations of calculations, it is difficult to adapt to such changes in the external environment. For instance, consider the following.

1. Data item b_7 in the DAG in figure 3 has changed such that its new value is dissimilar to its previous value.

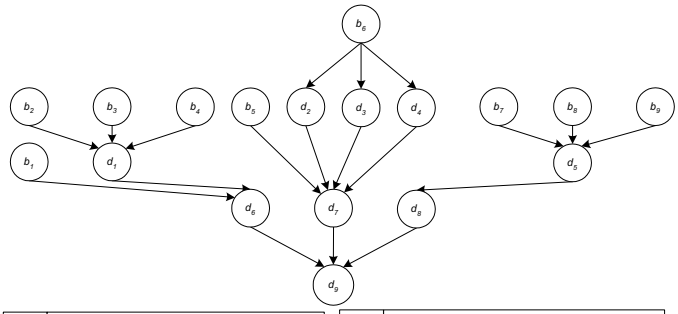
   A value becomes dissimilar when $|nv_{b_7} - ov_{b_7}| > dvb_{b_7}$, where nv_{b_7} and ov_{b_7} are the new and the old values of b_7 respectively, and dvb_{b_7} is the data validity bound of b_7.
2. Data item d_5 must be recalculated to reflect the new value of b_7.
3. Data items d_8 and d_9 might as well need recalculations, but that depends on the new value of d_5.

Using periodic updating of d_5, it is updated even though b_7 has not changed.

Recalculations of data items can be done at different moments in time. The two extremes are:

- recalculate a data item as soon as a data item the calculation reads is found dissimilar, and
- recalculate when the data item is requested by DIESIS.

The first approach is denoted *updates first*, and the latter *on-demand* [3]. The on-demand approach is found to use less CPU time than updates first [3]. In the implementation of DIESIS, recalculations are executed on-demand. Thus, there must be a way to postpone recalculations and start a subset of them in response to the starting of a user transaction. In DIESIS there is a scheme, *affected updating scheme (AUS)*, that marks a data item as affected when any of its immediate parents in the DAG changes to a dissimilar value. Returning to figure 3, when b_7 becomes dissimilar data item d_5 is marked as affected by this change. The scheme does the following:



b_1	basic fuel factor
b_2	lambda status variable
b_3	lambda status for lamda ramp
b_4	enable lambda calculations
b_5	fuel adaptation
b_6	number of combustions
b_7	airinlet pressure
b_8	engine speed
b_9	engine temperature

d_1	lambda factor
d_2	hot engine enrichment factor
d_3	enr. factor one started engine
d_4	enr. factor two started engine
d_5	temp. compensation factor
d_6	basic fuel and lambda factor
d_7	start enrichment factor
d_8	temp. compensation factor
d_9	Total multiplicative factor (TOTALMULFAC)

Figure 3: An example of a DAG.

- 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.

The marking may be a binary value, marked or not marked, but on rare occasions this approach can lead to missed updates. The marking can be a timestamp indicating with a timestamp greater than zero if the data item is changed or not. See [9] for a more detailed discussion.

Using AUS there is a way to decide which data items are affected by changes in data items. Thus, to schedule the data items needing to be updated, the DAG can be searched for data items that are marked as affected by a change. However, we must take the pessimistic approach and assume that a data item with a changed value renders descendant data items stale. This means that if a user transaction deriving data item d_9 starts, and data items d_4 and d_5 are marked as changed, then d_7 and d_8 need to be scheduled for being updated. The order these updates should be executed is first d_4 and d_5, and then d_7 and d_8, since there is no meaning in executing d_8 before d_5 because it will then read an old value of d_5.

To find data items to schedule for being updated, the DAG can be traversed from base items toward the data item d' being derived by the user transaction. When a data item d is found to be marked, all other data items that are on paths[1] from d to d' in G must also be put in the schedule.

One representation of the data items and the precedence constraints that allows for easy access is a depth-first order of the DAG. The algorithm in figure 4 is run on a DAG with an added bottom element resulting in the schedule S_{total}. Executing depthfirst(S,d_9) on the DAG in figure 3 yields the following schedule: $[d_1, d_6, d_2, d_3, d_4, d_7, d_5, d_8, d_9]$.

depthfirst(schedule s, dataitem d)
  for all $x \in R(d) \backslash B$ **do**
    add x first to s
    depthfirst(s, x)
  end for

Figure 4: The depth-first algorithm.

For every data item there exists a subschedule of S_{total} that represents the data item and all its ancestors. Hence, only one schedule, S_{total}, needs to be stored together with information of where in S_{total} the subschedules, one for each data item, are located [11]. The storage requirements for the schedule and the location information is low, e.g., a graph consisting of 45 base items and 105 derived data items has 246 elements in S_{total} requiring 592 bytes of ROM.

The scheduling algorithm is executed on-demand, i.e., when a user transaction starts, and it traverses the subschedules of the read set for changed data items. The algorithm is denoted On-Demand Top-Bottom with relevance check (ODTB) and is presented in figure 5 [11]. This algorithm produces a schedule with data items that need to be updated. For each data item in the schedule a corresponding update transaction is triggered and executed.

ODTB(schedule s)
  for all $x \in R(d) \backslash B$ **do**
    for all u in top-bottom order of a depth-first subschedule of x **do**
      if data item u is marked **then**
        Copy u and the remainder of subschedule to s
        break
      end if
    end for
  end for

Figure 5: The On-Demand Top-Bottom with relevance check algorithm.

CONCURRENCY CONTROL

In a database there is normally functionality to handle concurrent transactions. DIESIS supports different concurrency control algorithms, and the algorithm is chosen at compile time. In this section we elaborate on problems of concurrently executing transactions and describe one solution.

A database has a set of integrity constraints (see the appendix)—e.g., a fuel compensation factor TOTALMULFAC that should be calculated from, say, two sensor readings that are at maximum 200 ms old—that defines what states the database can be in. Concurrency control, which maintains some of the integrity constraints, is intimately connected with the ACID properties of transactions (see the appendix). The purpose of having transactions with these properties is that committed transactions always write values that correctly reflect an environment the database models. For instance, in a banking application,

[1] Remember that there is a one to one mapping from data items to nodes in graph G.

transactions must have the ACID properties in order to have the correct balance on accounts that are touched by a money transaction.

Naturally, there are different integrity constraints for every database application. This means that some of the ACID properties can be loosened. The effects are that the throughput of the database, i.e., the number of committed transactions per time unit, increases compared to using full support for the ACID properties. Furthermore, the properties stated in section "Problem Formulation", that the system resets all values at a system startup, simplify the concurrency control algorithms, because there is no need to make sure committed results persist in the event of a system failure. Hence, the durability property is not needed.

Two well-established concurrency control algorithms that support the ACID properties are two-phase locking (2PL) and optimistic concurrency control (OCC) [6]. Two-phase locking has been extended to work in real-time databases, e.g., high-priority two-phase locking (HP2PL) [2], and OCC works correctly unchanged in real-time systems, provided fixed priority scheduling of calculations is used, i.e., the priority of the calculation never changes during the execution of it. Lee and Park present a database consistency denoted statewise consistency [18]. They claim that as long as transactions produce results equivalent to results from transactions executed one at a time, conflicting interleavings of operations do not matter. As we recall from section "Problem Formulation", data items can have freshness requirements measurable in the value domain. This implies that values on data items can take a range of values for a specific state in the external environment, and as long as conflicting operations for a data item produce results within the range of the data item, the conflicts do not matter. Hence, we can conclude that well-established concurrency control algorithms are too strict for the kind of embedded systems we focus on, and similarity provides necessary conditions wielding a loosened consistency that can improve performance.

We can infer, for databases for embedded systems, that the concurrency control algorithm should be similarity-aware since such an algorithm can increase the performance. Moreover, the database should be able to present a snapshot (section "Problem Formulation"), from a specific time, of up-to-date data items to a transaction. To achieve this, the updating algorithm and the concurrency control algorithm can be combined.

In DIESIS there is a multiversion timestamp ordering concurrency control algorithm using similarity (MVTO-S) [10], that ensures that transactions read an up-to-date snapshot of data items.

Next is an outline of the MVTO-S algorithm given (more details are in [10]). Every transaction gets a unique timestamp, and data items can have several versions where the versions are annotated with a timestamp. The timestamp on a version written by a transaction is the maximum of the timestamps on read versions by the transaction. The rationale behind using the maximum is that a version is valid when all members of the read set are valid, and they are all valid from the mentioned time. A transaction reads the latest version of a data item that is older than or as old as the transaction. Such a version is denoted a *proper version*. Hence, DIESIS is restricted to read values that were valid when a transaction starts. There are no guarantees that these values are valid when the transaction commits. We think this is not a drawback, since this is the functionality that is normally requested in the EMS software we have studied. Furthermore, when there are active but preempted transactions in the system, then written values result in new versions, since the old values might be needed by preempted transactions.

To ensure up-to-date values the ODTB algorithm (section "Updating Algorithms") is used to schedule needed updates. The scheduling of updates is done atomically when a user transaction starts, which gives a view of which updates might be needed. Scheduled updates need to be executed if input data to the update is different from the input data an existing version of the data item used. Thus, updates can be skipped due to that the value already exists in a version. This check is done by comparing the values of the read set members of the update to the corresponding values of a proper version. If all values are similar, then the update is not needed.

The MVTO-S algorithm is an abstract algorithm that can be implemented in different ways. The next subsection describes three implementations.

Description of Implementations of MVTO-S

The MVTO-S algorithm uses a memory pool to store versions of the data items. The amount of memory available for versions is limited, and therefore there are three implementations of MVTO-S, which each requires different amount of memory for storing a version. The algorithm MVTO-S$^{\text{UV}}$ stores the value of each member of $R(d_i)$ when deriving d_i in the version. This strengthens the possibilities to make a similarity check. The MVTO-S$^{\text{UP}}$ implementation, on the other hand, stores no values from members of $R(d_i)$, but these values are instead looked-up in the pool. The version size gets smaller, but the possibility to do a similarity check depends on if the values of members of $R(d_i)$ can be found in the pool. When the pool becomes full, versions are removed from it, and affected transactions are restarted. The implementation MVTO-S$^{\text{CRC}}$ constructs versions with the same information as MVTO-S$^{\text{UV}}$ but they are reduced in size because the values of members of $R(d_i)$ are condensed into a cyclic redundancy check (CRC) value. The CRC is calculated by taking an interval number (IN) for the value of each member of $R(d_i)$ by, e.g., dividing the value with 64, and calculating a CRC from these interval numbers. We assume these interval numbers can be represented as an 8-bit or 16-bit integer, and that the CRC of two versions are equal if and only if the used values are similar. For instance, let us denote two versions that the transaction in figure 2 has produced as v_1 and v_2, and the values the versions used are:

Data item	Used values				Data validity bounds
	v_1	IN	v_2	IN	
D6	11	2	13	2	5
D7	10	2	10	2	5
D8	5	1	6	1	5

From the table above we can easily see that used values on D6, D7, and D8 have the same interval numbers, and, thus, the same CRCs, which means that v_1 and v_2 are similar.

PERFORMANCE EVALUATIONS OF DIESIS

This section describes performance evaluations of DIESIS in two settings. One setting using DIESIS in an EMS connected to an engine simulator is described in section "Performance Evaluation using EMS". The other setting using DIESIS together with the real-time operating system μC/OS-II on top of Windows 2000 is described in section "Performance Evaluation of MVTO-S".

PERFORMANCE EVALUATION USING EMS

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

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

The derived data item TOTALMULFAC is requested periodically by a task in the EMS software (see data item d_9 in figure 3). The request is transformed into a user transaction that arrives to DIESIS (see figure 2). ODTB is used in the EMS software, i.e., relevance checks are done if calculations are needed based on specified data freshness. This means that the data items d_1–d_8 in figure 3 are updated when needed, but not more often than the periodicity of TOTALMULFAC. In this experiment, the HP2PL concurrency control algorithm is used.

Recalculations of TOTALMULFAC are needed when the engine speed changes. Figure 6 shows how the requests for calculations are serviced only when the system is in a transient state, i.e., when the engine speed is changing. The plots in the bottom graph show the cumulative numbers of requests. The number of requests (dashed line) is increasing linearly since the requests are periodic and in the original EMS software each such request is processed. However, when using ODTB only some of the requests need to be processed. The number of serviced requests (solid line) shows how many of the requests need to be processed. In steady states, none of the requests need to be processed, and the stored value in DIESIS can be used immediately (e.g., the steady state in the time interval 2–7). Hence, during a steady state a considerable amount of requests can be skipped. Notice also that the data validity bounds allow DIESIS to accept a stored value if changes to the engine speed are small (in this case ± 50 rpm). This can be seen in the time interval

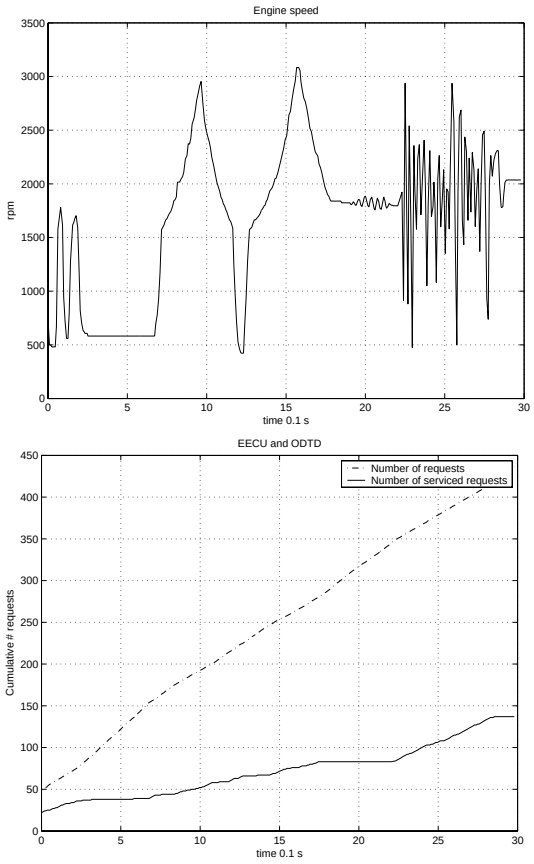


Figure 6: Number of requests of calculation of fuel compensation factor in EMS.

17-22, where the small changes in engine speed do not result in recalculations of the TOTALMULFAC variable. The number of serviced requests does not increase in this interval.

PERFORMANCE EVALUATION OF MVTO-S

The performance evaluations are conducted using DIESIS running on a real-time operating system that can be used in an embedded system. The operating system is μC/OS-II [16], and it provides the same functionality as Rubus we used in the EMS software in the experiments described in previous section. The simulations simulate periodic tasks invoking transactions from a database with similarity-aware updating and concurrency control algorithms. The base items are changing rapidly to model a transient state, where, as we saw in the previous section, the system needs to execute the most updates, i.e., the workload used in the simulator typifies a worst-case workload in a real-life system. 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.

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.

Simulation Setup

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 [7]. 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 uniform 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 ($|R(d)|$), ratio of $R(d)$ being base items, and ratio being derived items with only base item parents. The cardinality of $R(d)$ is set randomly for each d 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 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 EMS 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 uniform distribution U(0,350) to the previous most recent version. The data validity bounds are set to 400 for all data items. 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 where periodic updates of data items is used. Table 1 shows the concurrency control algorithms that are used in the simulations. The algorithm MVTO is MVTO-S without a relevance check. OCC-S is OCC where the validation phase has been extended with a check whether the conflict involves similar values or not [9]. This extension of OCC is in line with the stepwise consistency (see section "Concurrency Control") that considers that as long as transactions produce

Table 1: Concurrency control algorithms used in simulations.

Algorithms		
MVTO-S^{UV}	MVTO-S^{UP}	MVTO-S^{CRC}
OCC-S	HP2PL	OCC
OD-HP2PL	RCR-OCC	RCR-OCC-S
NOCC	RCR-NOCC	

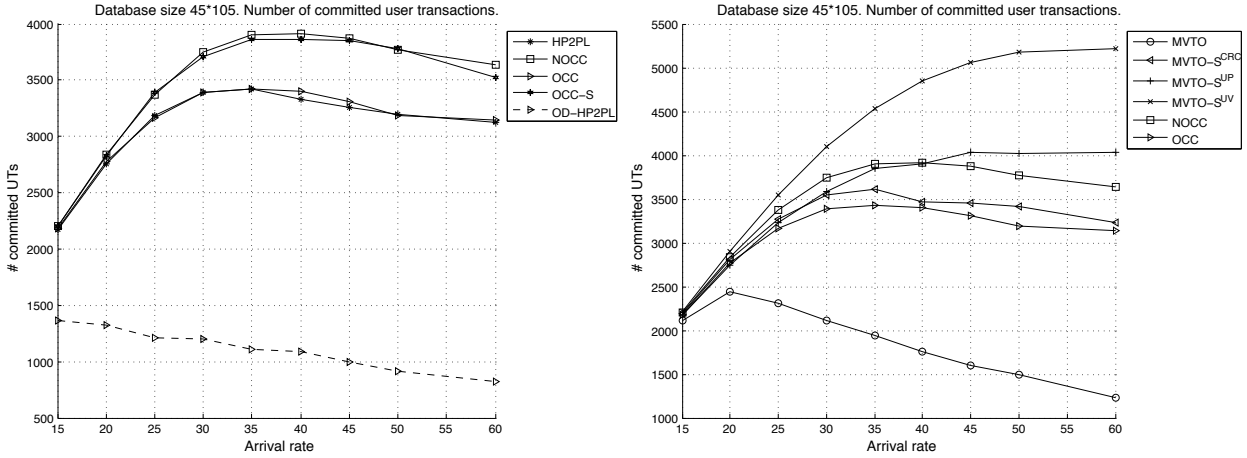
end results correctly reflecting the external environment, then all conflicts are acceptable. By using a similarity check in the validation phase of OCC, it is possible to allow small errors in values that do not affect end results. The RCR implementations of OCC and OCC-S are restarting transactions until they are able to read a snapshot of DIESIS [9]. The on-demand updating and using HP2PL for concurrency control (OD-HP2PL) algorithm [4] 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. The algorithm no concurrency control (NOCC) does not use any concurrency control, and is used as a baseline.

Simulation Results

The experiment investigates the number of committed user transactions within their deadlines. The no concurrency control scheme is used as a baseline. Figure 7 shows the performance of the algorithms.

First the benefit of using similarity can be seen in figure 7(a) studying the difference in performance between HP2PL and OD-HP2PL. In figure 7(b), MVTO-S^{UV} outperforms the single-version concurrency control algorithms at all arrival rates. The difference is most notable at higher loads where MVTO-S^{UV} performs significantly better than HP2PL, OCC, and NOCC. MVTO-S^{UP} cannot perform as good as MVTO-S^{UV} because less transactions are skipped [10]. MVTO-S^{UP} performs better than single-version algorithms at high arrival rates, but for small arrival rates OCC-S performs 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 other 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 DIESIS, 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^{UV}, MVTO-S^{UP}, and MVTO-S^{CRC} is due to a full memory pool, whereas for single-version algorithms restarts are due to conflicts among concurrent transactions.



(a) Single-version algorithms.

(b) Multiversion algorithms.

Figure 7: Experiment 1: Performance evaluations of single-version and multiversion concurrency control algorithms.

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

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%

In figure 7(a), we see that using OCC-S gives 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 DIESIS, 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 [8].

The implementations of MVTO-S guarantee that a transaction reads an up-to-date snapshot of DIESIS. 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 8 shows the performance using restarts to enforce relatively consistent read sets. The MVTO-S implementations 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.

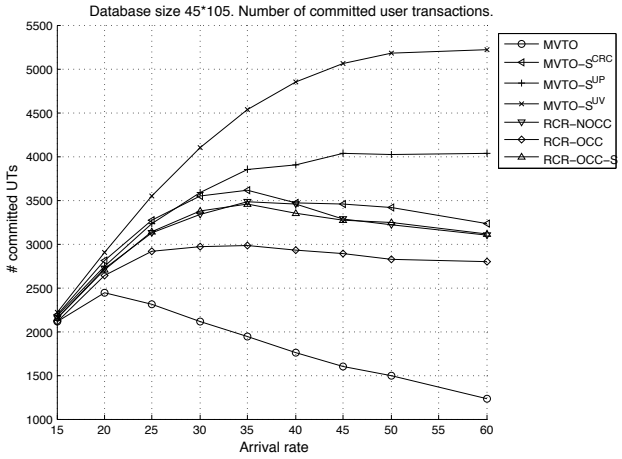


Figure 8: Experiment 1: Performance of relative consistency restarts algorithms.

CONCLUSIONS

In this paper we have described a database implementation aimed at being used in embedded systems—**D**ata **I**n **E**mbedded **S**ystems ma**I**ntenance **S**ervice (DIESIS)—and specifically in an electronic engine control unit. The following is found to justify the usage of a real-time database in an embedded system:

- Software is built as modules that naturally divide data into different areas, but this makes it difficult to overview existing data and their freshness and time requirements. Using a database, data is instead stored in one central repository which is easier to overview. Moreover, it is possible to add meta-information of data items to the database.
- Data read by a transaction should correlate to the same state of the external environment. Using a database, the internal functions of the database can ensure that transactions read such data. This is hard (or even impossible) to guarantee if not having a global knowledge of all data.

The contribution of this work is incorporating functionality in DIESIS that:

- ensures data items have a specified data freshness,
- automatically adapts the frequency of executing transactions to the current state of the system, and
- guarantees a transaction reads values of data items that were valid when the transaction started and they correlate to the same state of the external environment.

These functionalities are transparent to the user which means it is easier to use the system for the programmer since he can focus on other things than data freshness and consistent values. Moreover, these functionalities are evaluated in both a real-life system, an electronic engine control unit, and in a simulator setup. The performance evaluations show that using an on-demand updating algorithm that is similarity-aware, i.e., defines data freshness in the value domain of data values, can greatly reduce the number of transaction invocations when the system enters a steady state compared to periodically recalculating data. Moreover, using a similarity-aware snapshot algorithm not only guarantees transactions see a consistent view of data values, but it also improves performance since calculations can be skipped to a larger extent compared to well-established concurrency control such as HP2PL and OCC.

For future work we intend to study an admission control of transactions in DIESIS to reduce the workload in the event of an overload.

REFERENCES

[1] Arcticus AB homepage. http://www.arcticus.se.

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

[3] 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.

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

[5] Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, and Riccardo Torlone. *Database Systems Concepts, Languages and Architectures*. The McGraw-Hill Companies, 1999.

[6] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Publishing Company, 1987.

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

[8] Marc 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.

[9] Thomas 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.

[10] Thomas Gustafsson, Hugo Hallqvist, and Jörgen Hansson. A similarity-aware multiversion concurrency control and updating algorithm for up-to-date snapshots of data. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*. IEEE Computer Society Press, 2005.

[11] Thomas Gustafsson and Jörgen 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.

[12] Thomas Gustafsson and Jörgen 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.

[13] Shao-Juen Ho, Tei-Wei Kuo, and Aloysius 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.

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

[15] Tei-Wei Kuo and Aloysius K. Mok. Application semantics and concurrency control of real-time data-intensive applications. In *Proceedings of IEEE 13th Real-Time Systems Symposium*, pages 35–45. IEEE Computer Society Press, 1992.

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

[17] Chang-Gun Lee, Young-Kuk Kim, S.H. Son, Sang Lyul Min, and Chong Sang 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.

[18] Kyuwoong Lee and Seog Park. Classification of weak correctness criteria for real-time database applications. In *Proceedings of the 20th International Computer Software and Applications Conference 1996 (COMPSAC'96)*, pages 199–204, 1996.

[19] Michael A. Olson. Selecting and implementing an embedded database system. *IEEE Computer*, 2000.

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

[21] Håkan Sundell and Philippas 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.

[22] Aleksandra Tešanović, Dag Nyström, Jörgen Hansson, and Christer Norström. Embedded databases for embedded real-time systems: a component-based approach. Technical report, Department of Computer Science, Linköping University, 2002.

[23] Ming Xiong and Krithi 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.

CONTACT

Thomas Gustafsson
e-mail: thogu@ida.liu.se
phone: +46 (0)13281426
address: Thomas Gustafsson
Department of Computer and Information Science
Linköping University
SE-581 83 Linköping
Sweden

Jörgen Hansson
e-mail: jorha@ida.liu.se
e-mail: hansson@sei.cmu.edu
phone: +1 412 268 6733

Anders Göras
e-mail: anders.goras@mecel.se
phone: +46 (0)53262120

Jouko Gäddevik
e-mail: jouko.gaddevik@se.gm.com
phone: +46 (0)855377078

David Holmberg
e-mail: david.holmberg@se.gm.com
phone: +46 (0)855377086

APPENDIX

REAL-TIME SYSTEMS

A real-time system consists of tasks, where some/all have time constraints on their execution. It is important to finish a task with a time constraint before its deadline, i.e., it is important to react to an event in the environment before a predefined time.

The correct behavior of a real-time system depends not only on the values produced by tasks but also on the time when the values are produced [7]. A value that is produced too late can be useless to the system or even have dangerous consequences. A task is, thus, associated with a deadline relative to the start time of the task. Note that a task has an arrival time when the system is notified of the existence of the ready task, and a start time when the task starts to execute. Tasks are generally divided into three types:

- *Hard real-time tasks*. The missing of a deadline of a task with a hard requirement on meeting the deadline has fatal consequences on the environment under control. For instance, the landing gear of an aeroplane needs to be ejected at a specific altitude in order for the pilot to be able to complete the landing.
- *Soft real-time tasks*. If the deadline is missed the environment is not severely damaged and the overall system behavior is not at risk but the performance of the system degrades.
- *Firm real-time tasks*. The deadline is soft, i.e., if the deadline is missed it does not result in any damages to the environment, but the value the task produces has no meaning after the deadline of the task. Thus, tasks that do not complete in time should be aborted as late results are of no use.

Scheduling algorithms determine the order tasks are executed. Two well-known real-time scheduling algorithms are rate-monotonic (RM) and earliest deadline first (EDF). Using RM means that a task is assigned a priority that is proportional to the inverse of its invocation frequency. This means that tasks with high frequency get higher priority than tasks with low frequency. The priority of a task, using EDF, is commensurate with the distance to the deadline, and the shorter the distance the higher the priority.

DATABASE

The following benefits of using a database for data storage are given in database textbooks [5]: (i) data storage is centralized, (ii) data storage is hidden behind an API or query language, and the data storage details are hidden for the users of the database, (iii) the database can seamlessly handle data freshness and concurrency control, (iv) meta-information, e.g., timestamps and deadlines, can be stored with the data, and (v) the relationships among data items can be stored in the database.

TRANSACTIONS

We define a transaction as computational work carried out in the database, e.g., a calculation. The database has integrity constraints on the data, and transactions must be executed such that the constraints are always fulfilled. In some applications, e.g., in bank applications moving money between accounts, the execution of transactions must conform to the following properties [5]:

- *Atomicity.* All operations of a transaction are reflected in the database, or none of them are.
- *Consistency.* The execution of transactions preserves the consistency of the database.
- *Isolation.* Each transaction is unaware of other transactions executing in the database system.
- *Durability.* The results of a committed transaction persist even in the course of system failure.

Transactions should avoid the four effects of conflicting operations that are described below.

- *Lost update*, where a transaction overwrites the result of another transaction, and, hence, the result from the overwritten transaction is lost.
- *Dirty read*, where a transaction reads and uses a result of a transaction that is aborted later on, i.e., the transaction should not have used the results.
- *Inconsistent read*, a transaction reading the same data item several times gets, because of the effects of concurrent transactions, different values.
- *Ghost update*, where a transaction only sees some of the effects of another transaction, and, thus, consistency constraints do not hold any longer. For example [5], consider two transactions, τ_1 and τ_2, and the constraint $s = x + y + z = 1000$. The operations are executed in an order such that the value of s in τ_1 at commit time is 1100 since τ_1 has seen intermediate results from τ_2.

Consistency constraints can be constructed for the following types of consistency requirements: internal consistency, external consistency, temporal consistency, and dynamic consistency. Below each type of consistency is described [15].

- *Internal consistency* means that the consistency of data items is based on other items in the database. For instance, a data item Total is the sum of all accounts in a bank, and an internal consistency constraint for Total is true if, and only if, Total represents the total sum.
- *External consistency* means that the consistency of a data item depends on values in the external environment that the system is running in.
- *Temporal consistency* means that the values of data items read by a transaction are sufficiently correlated in time.
- *Dynamic consistency* refers to several states of the database. For instance, if the value of a data item was higher than a threshold then some action is taken that affects values on other data items.