Pessimistic Concurrency Control and Versioning to Support Database Pointers in Real-Time Databases [*]

Dag Nyström[†], Mikael Nolin[†], Aleksandra Tešanović[⋆], Christer Norström[†], and Jörgen Hansson[⋆]

[†]Mälardalen University
Mälardalen Real-Time Research Centre
Västerås, Sweden
{dag.nystrom, mikael.nolin,
christer.norstrom}@mdh.se

[⋆]Linköping University
Dept. of Computer Science
Linköping, Sweden
{alete,jorha}@ida.liu.se

Abstract

In this paper we present a concurrency control algorithm that allows co-existence of soft real-time, relational database transactions, and hard real-time database pointer transactions in real-time database management systems. The algorithm uses traditional pessimistic concurrency-control (i.e. locking) for soft transactions and versioning for hard transactions to allow them to execute regardless of any database lock. We provide formal proof that the algorithm is deadlock free and formally verify that transactions have atomic semantics. We also present an evaluation that demonstrates significant benefits for both soft and hard transactions when our algorithm is used. The proposed algorithm is suited for resource-constrained safety critical, real-time systems that have a mix of hard real-time control applications and soft real-time management, maintenance, or user-interface applications.

1. Introduction

In this paper we present a method that allow co-existence of soft real-time database transactions (denoted *soft transactions*) and hard real-time database transactions (denoted *hard transactions*) in a Real-Time Database Management System (RTDBMS). To support both types of transactions while eliminating transaction abortions caused by hard transactions and avoiding long delays for hard transactions, we propose the use of a versioning algorithm that uses traditional pessimistic concurrency control [1] for soft, relational (e.g., SQL [3]), transactions but allow hard, database pointer [8], transactions to execute regardless of any locks held by soft transactions.

Our concurrency control algorithm, called 2-Version DataBase Pointer concurrency control (2V-DBP), is suited for resource-constrained safety-critical, real-time systems that have a mix of hard real-time control applications and soft real-time management, maintenance, or user-interface applications.

We have previously studied data management in real-time control systems in an industrial case study of a vehicle control system developed at Volvo Construction Equipment Components AB, Sweden [9]. In this study a number of data management requirements were presented and it was concluded that both the system architecture, and the development and maintenance efforts could be improved by adopting a more structured approach to data management, e.g., by introducing an RTDBMS. Furthermore, it was elaborated on how to design and integrate an RTDBMS into the system. The RTDBMS can be used to ensure both logical and temporal consistency of application data within a real-time system [11]. Furthermore, RTDBMSs allow, so called, ad hoc queries that could be used by a service technician to diagnose a running system. It was concluded in the case study that the RTDBMS must have capabilities to handle both hard and soft database transactions, and that hard database transactions were issued by safety critical I/O and control-tasks running at high frequencies.

In [8] we proposed the concept of *database pointers*, as efficient means of accessing individual data elements within a RTDBMS. Database pointers have the efficiency of a shared variable combined with the advantages of using a RTDBMS. They allow a fast and predictable way of accessing data in a database without the need of consulting the RTDBMS indexing system. Furthermore, they provide an interface that uses a pointer-like syntax. This interface is suitable for control-system applications using numerous small tasks running at high frequencies. Database pointers can be used together with more flexible relational database queries without risking a violation of the database integrity.

This paper presents a concurrency control algorithm suitable for hard real-time control systems, e.g., vehicle control systems. The algorithm, which combines the concept of database pointers and relational transaction manage-

[*] This work is supported by SSF within the SAVE project, SAfety critical components for VEhicular systems.

ment, satisfies the need for predictable and time-efficient hard real-time control-applications, while allowing relational soft management transactions access to the database without being starved by the hard transactions. The algorithm uses a versioning technique for the hard transactions and *two-phase locking high priority* (2PL-HP) [1] for the soft transactions. In order to support these two concurrency control methods we introduce a simplified form of *versioning*, i.e., we maintain multiple (in our case two) versions of selected data elements. Our algorithm overcomes the widely recognized problem that transactions with low priority and long execution times are penalized due to the likeliness of data conflicts [4].

The contributions of this paper include a novel concurrency control algorithm, called 2V-DBP that: (i) provides efficient, and time-deterministic, execution of hard transactions, regardless of any database locks held by other transactions; (ii) bounds the maximum memory overhead caused by adding versions of data elements; (iii) allows soft transactions to be executed even though the database is read and updated by hard transactions.

We also present an evaluation of 2V-DBP; showing significant benefits for both hard and soft transactions.

The costs of using 2V-DBP are added (although predictable) memory overhead, since all data used by the hard transactions is stored in two versions, and a relaxation of the serialization criteria for soft management transactions.

In section 2, our system model and transaction models are presented. The paper then recapitulates the database pointer concept in section 3. The proposed concurrency algorithm is then presented, verified, and evaluated in section 4. We conclude the paper in section 5.

2. System model

This paper focuses on real-time applications used to control a process, e.g., critical control-functions in a vehicle such as engine or brake control. The basic flow of execution in such a system is [9]: (i) periodic scanning of sensors, (ii) execution of control algorithms, such as PID-regulators, and (iii) propagation of the result to the actuators. Typically, the application is structured into multiple tasks executed by a preemptive real-time operating system. The tasks use the RTDBMS to access and manipulate shared data. Hence, the RTDBMS needs some form of concurrency control to maintain consistency given multiple concurrent accesses. In traditional relational databases, data manipulation is performed using queries formulated in a special purpose language such as SQL. Such queries can either be created dynamically (during run-time), so called ad-hoc queries, or be created before run-time and stored in a precompiled format. The latter is the common case in real-time systems, since precompiling a query saves both time and memory during run-time.

Task type	*Hard RT*	*Soft RT*	*Frequency*	*Trans. type*	*Precompiled*	*Ad hoc*
Control tasks	x		H	U	x	
I/O tasks	x		H	RW	x	
Management tasks		x	L	RWU	(x)	(x)

Legend:
x - the property is true for the task type
(x) - the property is true for some tasks of the task type
H/L - indicates high, or low frequency
RWU - indicates read only, write only, or update transaction type

Table 1. Transaction properties for the system's task types.

2.1. Application and task model

We classify the tasks in the system into three categories, namely, I/O-tasks, control-tasks, and management-tasks [9]. The I/O-tasks are typically executed periodically, often at high frequencies. There are two types of I/O-tasks; (i) tasks that read a sensor, and write the value to the database using a write only transaction , and (ii) tasks that read a value from the database, using a read only transaction, and then write it to an actuator. Table 1 summarizes the properties of the three types of tasks. I/O-tasks touch very few, in most cases only one, data element in the database, and their transactions are always precompiled.

Control tasks take a set of data values and derive new actuator values, thus performing update transactions on the database, i.e., performing a number of read operations followed by a number of write operations. For most control tasks in a real-time control system, reading the freshest data values available is sufficient (and preferable). Note that this desire to read fresh data is not always adhered to by RT-DBMSs, since they focus on preserving transaction ordering rather than providing data freshness.

Management tasks are the only tasks running soft transactions. An example of a management task might be a task presenting statistical information about the current state of the vehicle to the user. A management transaction might also be constructed during run-time, for example by a service technician using a service tool connected to the vehicle.

2.2. Transaction models

All tasks in the system that interact with the RTDBMS do this through database transactions. A database transaction consists of a set of database operations, e.g., read and write operations. We denote transactions residing in hard real-time tasks as hard transactions, while transactions residing in soft real-time tasks are referred to as soft transactions. A task can only execute one transaction at a time, but any number of transactions in sequence.

```
1 TASK OilTempReader(void) {
2   int s;
3   DBPointer *ptr;
4   bind(&ptr, "SELECT temperature FROM engine
                WHERE  subsystem=oil;");
5   while(1) {
6     s=read_sensor();
7     write(ptr,s);
8     waitForNextPeriod();
    }
  }
```

Figure 1. An I/O task that uses a database pointer

The two different transactions types are characterized as: (i) Soft transactions utilize a relational database query interface, e.g., SQL, for database access. These transactions provide flexible and dynamic access to data in the database. (ii) Hard transactions utilize the database pointer interface. The database pointer interface only allows one operation on one data element per transaction. This operation can either be a read or a write operation. Hard transactions cannot be aborted and will always complete successfully. All transaction have atomic semantics, i.e., either they are fully executed or not executed at all.

3. Database pointers

Before addressing the concurrency control algorithm, we will recapitulate the database pointer concept presented in [8]. Noteworthy is that in that work (and, hence, in this section) database pointers use pessimistic concurrency control (i.e. locks).

Figure 1 shows an example of a I/O task that periodically reads a sensor and propagates the sensor value to the database using a database pointer, in this case the oil temperature in the `engine` relation. The task consists of two parts, an initialization part (lines 2–4) executed when the system is starting up, and a periodic part (lines 5–8) scanning the sensor. The initialization of the database pointer is first done by declaring the database pointer (line 3) and then binding it to the data element containing the oil temperature in the engine (line 4). When the initialization is completed, the task begins to periodically read the value of the sensor (line 6), then propagates the value to the RTDBMS using the database pointer (line 7), and finally awaits the next invocation of the task (line 8).

Database pointers are implemented using the data structures shown in figure 2. The binding of a database pointer to a database element is performed in the following steps:

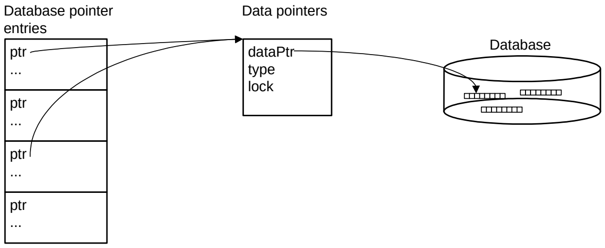

Figure 2. The data structures for database pointers.

1. A new *database pointer entry* is created in the RTDBMS.

2. The SQL query is executed. It is required that the result of the query is a single data element. If it is the first time the data element is bound to a database pointer, a new *data pointer* is created in the RTDBMS. The data pointer is initialized with the address of the data element, the data type of the element, and a pointer to the lock for the data element.

3. The database pointer entry is set to point at the data pointer.

4. Finally, the pointer to the database pointer entry is returned as a `DBPointer*`.

In addition to the `bind(ptr,q)` operation, the database pointer interface consists of the `remove(ptr)` operation which deallocates a database pointer, the `write(ptr,data)`, and the `read(ptr)` operations which updates, respectively reads the data element.

4. The 2-version database pointer algorithm (2V-DBP)

The 2V-DBP algorithm allows hard database transactions to execute without being blocked by soft database transactions. Furthermore, soft transactions, using the relational part of the RTDBMS are allowed to execute without being blocked or aborted by the hard database transactions. To achieve this behavior, two versions of all data elements pointed out by database pointers must exist in the database in a similar way as in the two-version priority ceiling protocol proposed by Kuo, Kao, and Shu [5].

The behavior, at a high level of abstraction, of 2V-DBP is discussed in sections 4.1 to 4.4, while the underlying versioning algorithm that ensures the desired behavior is presented in sections 4.5 to 4.6.

4.1. Soft transactions

The soft transactions utilize the relational part of the RTDBMS, and use an extended form of 2PL-HP [1]. Soft transactions pass through the following steps throughout their executions:

1. **The Begin of Transaction step (BOT)** in which the transaction becomes active.

2. **The lock-obtaining step** in which the transaction obtains all locks necessary to complete. In 2V-DBP,

the set of locks does not have to be defined prior to the BOT of a transaction, i.e., 2V-DBP allow ad hoc queries.

3. **The committing step** in which the transaction starts to write back the updated data elements to the database. Up to this step, the transaction might be aborted due to some data conflict. However, when the transaction enters the committing step it cannot be aborted any longer.

4. **The End Of Transaction (EOT) step** in which the transaction releases all locks, and the transaction is completed. When the EOT step has been executed, all changes to the database made by the transaction are made visible to other transactions.

The following rules are applicable for soft transactions:

Rule 1 *Soft transactions can read a data element from the database after having successfully obtained either a read lock or a write lock.*

Rule 2 *Soft transactions can change a value of a data element in the database after having successfully obtained a write lock.*

Rule 3 *All locks needed for completing a soft transaction must be obtained prior to the transaction entering the committing step.*

Rule 4 *Read locks on a particular data element in the database can be held by multiple soft transactions simultaneously, thus read locks are compatible with other read locks for the same data element.*

Rule 5 *A write lock on a particular data element in the database grants a soft transaction exclusive access to the data element, so that no other soft transactions can hold, or obtain, any type of lock on the data element in question during the time the write lock is held. Thus write locks are incompatible with any other lock for the same data element.*

Rule 6 *A transaction takes the database from a consistent state to a new consistent state. This means that during the execution of a soft transaction no changes to the database, caused by the transaction, are visible to other transactions until it finishes EOT.*

Rule 7 *If two soft transactions attempt to obtain a read lock or a write lock, which violate the lock compatibility stated in rule 4 and 5, result in that the transactions are considered to be in conflict with one another.*

4.2. Hard transactions

All hard transactions use database pointers. Even though hard transactions can access the same data elements as soft transactions, hard transactions are never blocked by database locks. However, hard transactions take database locks in consideration and access the database differently if the data element is locked, see section 4.5.

The following rules are applicable to all hard transactions:

Rule 8 *A hard transaction can either read or write a data element, even if the data element is locked by a soft transaction.*

Rule 9 *A hard transaction can never come in conflict with any other transaction.*

Rule 9 is enforced by making hard transactions non-preemptable, see section 4.5.

4.3. Transaction conflicts

Since soft transactions might be in conflict with other soft transactions, as stated in rule 7, a policy on how to resolve these conflicts is needed. The following rules are applicable to solve transaction conflicts:

Rule 10 *A soft transaction that has not yet entered the committing step will be aborted by the concurrency control algorithm iff it is in conflict, according to rule 7, with a soft transaction executing with a higher priority.*

Rule 11 *A soft transaction that is in conflict, according to rule 7, with a soft transaction executing at a lower priority which has entered the committing step, will be blocked from execution until the committing transaction has finished its execution.*

Theorem 1 *A database transaction can never enter a state of deadlock caused by conflicts with any other database transaction.*

Proof Since a hard database transaction can never be in conflict with any other transaction (rule 9), conflicts can thus only occur among soft transactions. Transaction conflicts among soft transactions are resolved in two ways; (i) If the conflicting transaction is executing at a lower priority than any other conflicting transaction, and has not yet entered the committing step it is aborted (rule 10), thus resolving the conflict. (ii) If the conflicting transaction is executing at a lower priority than any other conflicting transaction, and has entered the committing step, any conflicting transaction will be blocked until the transaction is completed (rule 11), and thus releasing all it locks. Since a transaction, which has entered the committing step, cannot obtain any further locks (rule 3), it cannot cause any further conflicts with any other transaction. $\therefore$

4.4. Transaction serialization and relaxation

The goal of a concurrency control algorithm is to resolve data conflicts between concurrent transactions so that it appears that they are run in sequence, hence transactions are serialized. The traditional notion of serialization is to serialize transactions in the order that they commit, i.e., in the order their updates are visible to other transactions. However, it has been recognized that this notion of serialization is not ideal for accessing real-time data [6], since freshness of data often is more important than traditional serialization. Hence, relaxing this serialization criteria, in a controlled way, might increase the freshness of the accessed data.

In 2V-DBP, the following serialization rules apply to transactions:

Rule 12 *A set of executing soft transactions are serialized in the order they perform EOT, thus making their changes visible to other transactions.*

Rule 13 *A hard transaction, reading or writing the value of a data element x, is serialized before all hard transactions reading or writing the value x at a later time. Furthermore, the transaction is serialized before any soft database transaction obtaining a lock on x at a later time.*

Rule 14 *A hard transaction, updating the value of a data element currently locked by a soft transaction, is serialized after that transaction.*

Intuitively, rule 14 implies that if, during the execution of a soft transaction, a hard transaction updating the database is serialized after the soft transaction, the soft transaction must not update the data element in question. This is because the hard transaction is serialized after the soft transaction, and thus the value produced by the soft transaction is already overwritten by the hard transaction, since logically it was executed after the soft transaction.

Rule 13 and 14 imply a relaxation of the serialization order. Consider the example given in figure 3 where transactions T_1 to T_3 execute in the following way:

	Event	Data State	Comment
T_1	BOT	$\{x, y\}$	T_1 starts
T_1	$W\_lock(x)$		T_1 obtains write lock on x, thus obtained a local copy of x.
T_2	$Write(x) \rightarrow x'$	$\{x', y\}$	T_2 pre-empts T_1 and updates x. Since x is write locked by T_1, T_2 is serialized after T_1, according to rule 14.
T_3	$Write(y) \rightarrow y'$	$\{x', y'\}$	T_3 updates y. Since y is not yet write-locked by T_1, T_3 is serialized before T_1, according to rule 13.
T_1	$W\_lock(y')$		T_1 obtains write lock in y, thus obtaining a local copy of x'.
T_1	Execute query		T_1 derives x'' and y''.
T_1	committing		T_1 enters the committing step.
T_1	$\neg(Upd(x''))$		T_1 does not update $x- > x''$, according to rule 14.
T_1	$Upd(y') \rightarrow y''$		T_1 updates y, however this update is not yet visible to other transactions.
T_1	EOT	$\{x', y''\}$	T_1 ends and releases its locks. y'' is now visible to other transactions.

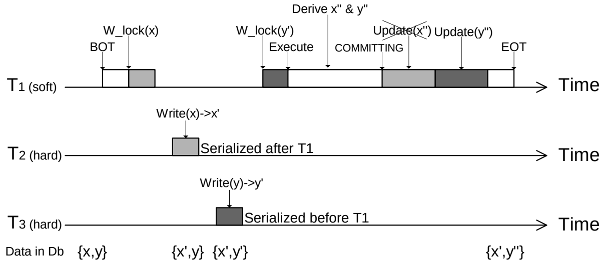

Figure 3. An execution-trace of three transactions

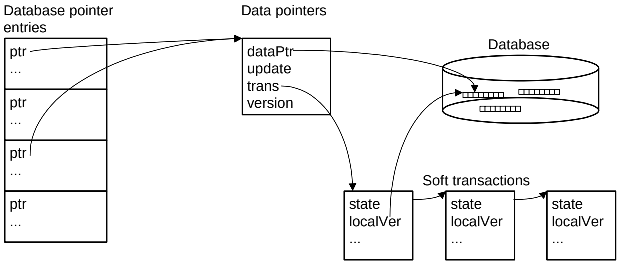

Figure 4. The data structures used for versioning

From the example we see that the resulting serialization order is T_3, T_1, and T_2, even though the actual order of commit is T_2 , T_3 , and T_1 . This relaxation of the serialization does, however, not imply that soft transactions read inconsistent data since all transactions, according to rule 6, take the database from one consistent state to another, see section 4.6. This serialization approach trades a relaxation of serialization for freshness of data.

4.5. Realizing 2V-DBP using versioning

As stated earlier, the RTDBMS maintains two versions of data items that have data pointers to them. These versions are used to realize 2V-DBP, as described by rules 8, 9, and 12–14. The data structures from figure 2 are modified as follows (depicted in figure 4):

- A list of the active soft transactions, where each entry consists of, among other information, the current state (`state`) of the transaction, and a local working copy of the data element (`localVer`).

- A second version of the data element (`version`) as well as a flag (`update`) are added to the data pointer. The `update` flag can have the values `clean` and `dirty`, where the latter indicates that the data has been updated by a hard transaction since it was previously write locked by a soft transaction. Since hard transactions do not use database locks, the `lock` entry presented in figure 2 is removed.

- A pointer (`trans`) to any soft transaction holding a

```
1  trans.state=EXECUTING; //BOT
2  For each tuple loop
3    obtainLock(tuple);
4    For each element in tuple loop
5      if (HasDbP(element) and isWriteLocked(tuple))
6         beginATOM();
7           DbP.version=Database.element;
8           DbP.trans=currentTrans();
9           DbP.update=CLEAN;
10          localVer=Database.element;
11        endATOM();
12     else
13        localVer=Database.element;
14     End if
15   End loop
16 End loop
17 //Manipulate tuples
18 trans.state=COMMITTING
19 For each manipulated tuple loop
20   For each element in tuple loop
21     if (HasDbP(element))
22       beginATOM();
23         if (DbP.update==CLEAN)
24           Database.element=localVer;
25         End if
26       endATOM();
27     else
28       Database.element=localVer;
29     End if
30   End loop
31 End loop
32 releaseAllLocks(trans); //EOT
33 trans.state=NO_TRANS;  //EOT
```

Figure 5. A soft transaction

write lock on the data element is added to the data pointer.

The implementation of a soft transaction is presented in figure 5. First the BOT step is executed (line 1), by setting the state of the transaction to `executing`. The next step, the lock obtaining step, is then executed (lines 2-16) by obtaining a lock for each tuple (line 3). When the lock is granted, the data element in the tuple is fetched to the local version. If a write locked data element is also pointed out by a database pointer, lines 6-11 are atomically executed, i.e., without being preempted. This atomicity is ensured by the `beginATOM()` and `endATOM()` functions, e.g., by temporarily disabling all interrupts. When fetching the data element, the `version` and the `trans` in the data pointer are also updated. Furthermore, the `update` flag is set to `clean` (line 9), to indicate that no hard transaction has altered the data element since the locking of the tuple. Finally, the data element is read from the database (line 10).

When all data elements of all tuples needed by the transactions are locked and copied to local versions, the transaction executes the query, in which all local versions of the write locked data elements can be manipulated.

The next step, the committing step, is entered (line 18) by changing the state of the transaction. Now the transaction can write all manipulated data elements back to the database (line 19-31) The data elements pointed out by database pointers are only updated if the `update` flag still indicates `clean`. Note that the second version (`version`) is not updated.

```
1  beginATOM();  //BOT
2    if (DbP.trans->state!=NO_TRANS)
3      version=NEW_VALUE;
4      update=DIRTY;
5    End if
6    *(DbP.dataPtr)=NEW_VALUE;
7  endATOM();    //EOT
```

Figure 6. A hard write transaction

```
1  beginATOM();  //BOT
2    if (DbP.trans->state!=NO_TRANS)
3      localVer=version;
4    else
5      localVer=*(DbP.dataPtr);
6    End if
7  endATOM();    //EOT
```

Figure 7. A hard read transaction

In the last step, the EOT step, the transaction releases all its obtained locks (line 32), and changes state to `no_trans` (line 33). The algorithm ensures that no data produced by the transaction is visible to any other transaction prior to this final step, see section 4.6 for the verification of this property.

Hard transactions execute entirely in one atomic operation, this implies that BOT and EOT coincide in time. This atomicity is, again, provided by `beginATOM()` and `endATOM()`, see lines 1 and 7 in figures 6 and 7. A hard transaction will read the data element in the database if the data element is not write locked by a transaction. Otherwise, it will read the value from the `version` in the data pointer. A hard transaction always writes directly to the data element in the database. However, if it is locked by a write lock, it will also update the `version` in the data pointer, as well as setting the `update` flag to `dirty` to indicate that the data element is now updated after it was write locked by the transaction.

4.6. Formal verification of the versioning algorithm

In order to formally show the correctness of the versioning algorithm in section 4.5, we have chosen to use the tool UPPAAL [7] to verify important properties of the algorithm.

One important property to verify is whether or not transactions always read the correct version of a data element, i.e., the value produced by the last serialized transaction updating that particular data element. We refer to this as the durability of transactions. Another equally important property is to verify that no intermediate results produced by executing transactions are visible to other transactions, e.g., verify the consistency of transactions. Finally, we verify that the versioning algorithm is deadlock-free.

UPPAAL is a toolbox for modeling, verification and validation of real-time systems. It is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels and (or) shared variables. UPPAAL is designed mainly to check invariant and reachability properties by exploring the state space of a system [7]. UPPAAL provides a graphical interface in which the user graphically models the system using timed automatas. Each transition in the automatas can have guards assigned to them, preventing the system to perform the transition if the condition stated in the guard is not fulfilled. Guards use the same syntax as conditions in `c`, e.g., `op == op`, `op < op`, and `op != op`. If a transition is performed, a (possibly empty) set of assignments is executed, e.g., `op := op`. All operations, e.g., guards, assignments and synchronization, performed during one transition are considered to be one atomic operation which cannot be preempted by other transitions. It is also possible to use communication channels in which two automatas can perform synchronized transitions. When the real-time system has been modeled, the system can be model-checked using requirement specification queries.

The automatas depicted in figures 8 to 11 show the behavior of transactions (for a data element pointed out by a data pointer), as modeled in UPPAAL. In the modeling of the system, two states (`AFTERREAD_VERIFICATION_STATE` in figure 8 and 10) and one variable (`lastCommittedTransaction`) have been added. These do not affect the behavior of the model, but are added for verification purposes. Three of the states are marked with the letter "C", which, in UPPAAL, indicates that the state is committed, i.e., the automata must immediately move to the next state.

The hard write, the hard read, and the soft read operations are a direct translation of the pseudo programs presented in section 4.5. The soft write transactions presented in figure 8, however, deserve further explanation:

1. The transition from `BOT` to `EXECUTING` (The `AFTERREAD_VERIFICATION_STATE` is disregarded for now) corresponds to the lock obtaining step of the transaction.

2. The transition from `EXECUTING` to `COMMITTING_NOT_WRITTEN_DATA` correspond with the transaction entering the committing state.

3. In the `COMMITTING_NOT_WRITTEN_DATA` the decision whether or not to update the database is taken based upon the value of the `update` flag.

4. The transition from `COMMITTING_HAS_WRITTEN_DATA` to `EOT` corresponds to lines 32-33 in figure 5.

In our verification we parallel compose the four automatons. This configuration is minimalistic but sufficient in order to capture all possible types of interactions where hard and soft transactions can interfere with each other (i.e., soft read vs. hard write, soft write vs. hard write, soft write vs. hard read, and all possible orderings of these pairs of interactions).

In this verification, three properties are verified, (i) that the algorithm is deadlock-free, (ii) that the value written by the last committed transaction is the value read by other transactions, i.e., the durability of transactions, and (iii) the consistency of transactions. The syntax of the queries in UPPAAL are not be explained in this paper, instead we refer to [7] for a detailed description of the UPPAAL syntax.

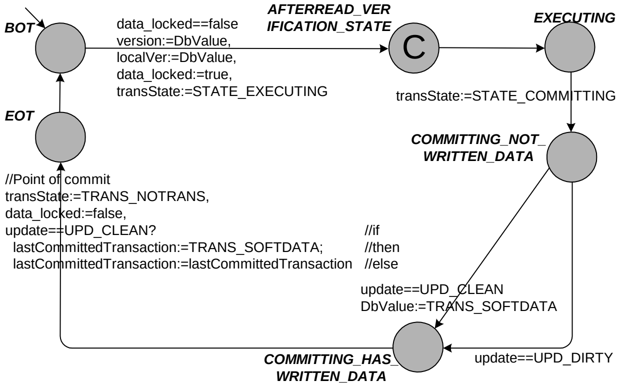

Figure 8. Automata for a soft write transaction

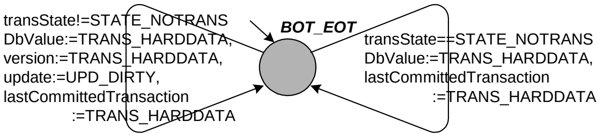

Figure 9. Automata for a hard write transaction

Verification that the versioning algorithm is deadlock-free This property is trivial to check, since UPPAAL already has a mechanism to verify this. The result of the query `A[] not deadlock` showed that the algorithm is deadlock-free.

Verification of the durability of transactions To verify this property, four queries are used, namely:

1. `A[] ((SR.AFTERREAD_VERIFICATION_STATE`
   `and SR.localVer==TRANS_HARDDATA) imply`
   `(lastCommittedTransaction==TRANS_HARDDATA))`
   where SR is a soft read transaction. The query can be interpreted as "Is it always so that if a soft read transaction has read a value produced by a hard transaction, the latest serialized transaction was a hard transaction?"

2. `A[] ((SR.AFTERREAD_VERIFICATION_STATE`
   `and SR.localVer==TRANS_SOFTDATA) imply`
   `(lastCommittedTransaction==TRANS_SOFTDATA))`
   where SR is a soft read transaction. The query can be interpreted as "Is it always so that if a soft read transaction has read a value produced by a soft transaction, the latest serialized transaction was a soft transaction?"

3. `A[] ((HR.EOT and SR.localVer==TRANS_HARDDATA)`
   `imply (lastCommittedTransaction==TRANS_HARDDATA))`

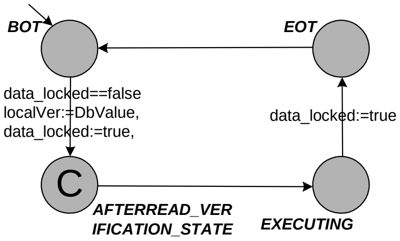

Figure 10. Automata for a soft read transaction

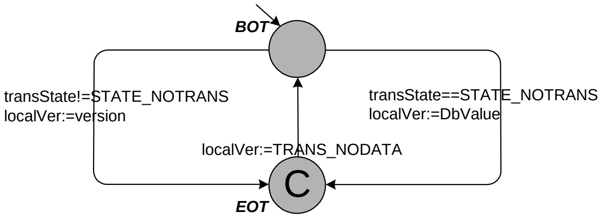

Figure 11. Automata for a hard read transaction

where HR is a hard read transaction. The query can be interpreted as "Is it always so that if a hard read transaction has read a value produced by a hard transaction, the latest serialized transaction was a hard transaction?"

4. `A[] ((HR.EOT and SR.localVer==TRANS_SOFTDATA) imply (lastCommittedTransaction==TRANS_SOFTDATA))` where HR is a hard read transaction. The query can be interpreted as "Is it always so that if a hard read transaction has read a value produced by a soft transaction, the latest serialized transaction was a soft transaction?"

The verification showed that all four queries were fulfilled.

Verification of the consistency of transactions This property was implicitly verified when the durability property was verified, since if the value visible to transactions always originate from the last committed transaction, no data can be visible from uncommitted transactions.

4.7. Performance evaluation

We have performed a performance evaluation of 2V-DBP. The goal of the evaluation is to illustrate, for a synthetic but realistic scenario, the positive impact 2V-DBP has, comparing it to traditional pessimistic concurrency control. Specifically, the goal is to demonstrate that 2V-DBP provides significant benefits for *both* soft and hard transactions, illustrating that 2V-DBP do not represent a tradeoff between good service for either soft or hard transactions.

To evaluate the performance of the 2V-DBP algorithm, we compared it with using the 2PL-HP algorithm for both soft and hard transactions. 2PL-HP is a well-known pessimistic concurrency control algorithm which can be implemented on all priority-based operating systems. To achieve this evaluation, a real-time system executing soft relational transactions and hard database pointer transactions was implemented on the Asterix real-time kernel [14]. These tests were then executed on a standard PC with an Intel Pentium 350MHz processor.

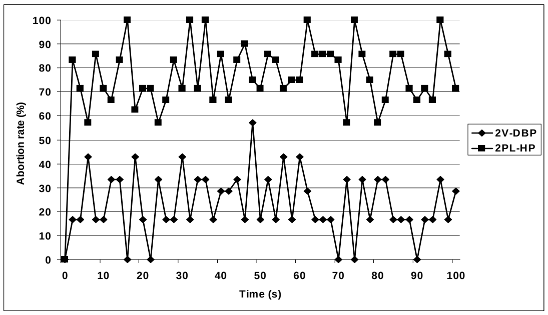

Figure 12. Abortion-ratios for soft transactions

The Asterix real-time kernel is an operating system, using fixed priority scheduling, intended for small embedded applications. Since Asterix supports pre-emptable scheduling, semaphores are needed to ensure task synchronization. In Asterix, semaphores are implemented using the immediate inheritance protocol [2]. The interrupt latency of the kernel, executing on the computer used in these tests, is in the order of 20μs.

The RTDBMS in the test consists of 300 tuples with four data elements each. Also 300 randomly selected data elements are pointed out by database pointers. Every 400ms a soft transaction is launched into the system. Each soft transaction randomly write- and/or read locks up to 200 tuples. Also, every 20ms, a hard transaction is launched. The hard transaction executes either a read or a write operation on a randomly selected database pointer.

This transaction schedule mimics a hard real-time vehicle control system with numerous hard I/O and control tasks, as well as a number of management tasks executing data intensive soft transactions. It is, furthermore, fair to assume that for RTDBMSs residing in vehicle control systems, a significant part of the database is accessed by hard transactions, since most execution in these systems would involve the controlling of the vehicle, hence the high amount of database pointers.

In figure 12 the mean abortion ratios for both 2V-DBP and 2PL-HP is presented. The system's abortion rate is sampled with an interval of two seconds, and the samples (indicated by boxes and diamonds in figure 12) show the mean abortion ratio for each interval. The comparison shows that the total mean abortion ratio for 2V-DBP is 25%, compared to approximately 75% for 2PL-HP. Furthermore, it is noticeable that all abortions for 2V-DBP are induced by soft transactions aborting other soft transactions, and that

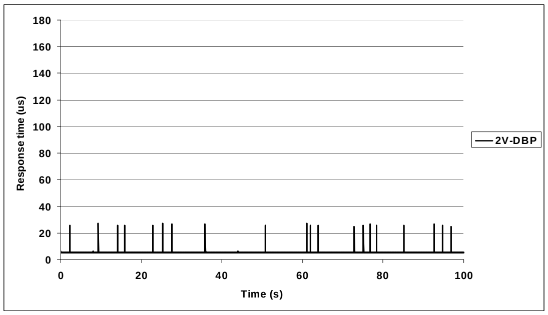

Figure 13. Response-times for hard transactions using 2V-DBP

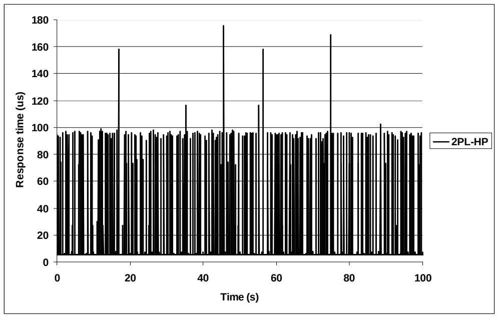

Figure 14. Response-times for hard transactions using 2PL-HP

no transactions are aborted because of transaction conflict with a hard transaction.

Figure 13 shows the response times for the hard transactions executing under 2V-DBP. The figure shows that all 5000 transactions launched during the 100 second test interval, but a handful (∼20 transactions) have constant execution time (5 μs). The remaining transactions have suffered from latency caused by the kernel, i.e., a timer interrupt has occurred during the execution of the transaction.

Figure 14 shows the corresponding response times for hard transactions executing under 2PL-HP. Roughly, the response times can be grouped into three classes, namely:

- Transactions executed without interference from other transactions. More than 95% of the hard transactions falls into this class. These transactions have a response time similar to the 2V-DBP case, i.e., 5-8μs. However, a fraction of these have also suffered from the kernel timer interrupt.
- Transactions causing soft transactions to be aborted. This implies that the hard transaction must execute the abort transaction procedure before continuing. These transactions, which are just above 4% of all hard transactions, have an execution time of 95μs.
- Transactions suffering from priority inversion. Due to the use of a common semaphore to administrate the 2PL-HP lock tables these transactions have been blocked by soft transactions. The execution times of these transactions range up to ∼180μs. Only a small fraction (0.1%) of all transactions fall into this class.

The measurements taken from these two execution cases show that 2V-DBP outperforms 2PL-HP, both with respect to a minimized amount of aborted transactions, as well as constant execution times for hard transactions. This shows that 2V-DBP is a suitable approach to manage hard transactions in real-time control systems, since it provide high throughput of soft transaction, as well as short constant execution times for hard transactions.

4.8. Memory overhead of 2V-DBP

Another important issue for embedded systems is the memory overhead. In most cases there is a clear trade-off between functionality and resource allocation; 2V-DBP is no exception. However, the memory consumption of 2V-DBP is bounded and predictable. Consider the example database presented in 4.7.

This database would, for an average length of each data element of 2 bytes, add up to 2,3kb (300 tuples * 4 data elements * 2 bytes). To structure the data, a RTDBMS needs to use additional memory, e.g., overhead used to index data and to store relation information. The commercial Polyhedra embedded database management system [10] has an overhead of 28 bytes/tuple [13]. Using this RTDBMS for the database would add 8,2 kb of overhead. Adding 300 database pointers that uses locking would imply an extra overhead of 2,7kb (if pointers use 4 bytes, integers use 2 bytes, and the lock information is stored in an integer). On the other hand, e.g., if 300 database pointers use 2V-DBP instead of locking, implying an overhead of no more than 3,6kb.

This implies that for a system that uses 2PL-HP the total memory consumption would be 13,2 kb. The memory consumption for the same system using 2V-DBP would be 14,1 kb, i.e., 2V-DBP increases the overhead by 6,8 %.

5. Conclusions and future work

We have presented a concurrency control algorithm that allows co-existence of soft real-time, relational database transactions (soft transactions) and hard real-time database pointer transactions (hard transactions) [8] in a Real-Time Database Management System (RTDBMS). The method, called 2-Version DataBase Pointer concurrency control (2V-DBP) uses, traditional, pessimistic concurrency control for soft transactions and a simplified form of versioning, i.e., we maintain multiple (in our case two) versions of data accessed by database pointers [8].

2V-DBP supports soft transactions with long execution times without risking that soft transactions are aborted by high priority hard transactions. Thus, 2V-DBP overcomes the recognized problem that transactions with low priority and long execution times are penalized due to the likeliness of data conflicts, resulting in frequent aborts [4].

2V-DBP supports hard transactions without risking hard transactions being delayed by long-running soft transactions. Such delays could otherwise be the case even if *high priority abort* is employed, since (i) abort of soft transactions is itself a time-consuming procedure, and (ii) once a soft transaction reaches the commit state it can no longer be aborted. Also, database pointers ensure fast and deterministic access to data elements, allowing access to the database without consulting the RTDBMS indexing system.

We have proved that 2V-DBP is free of deadlocks and formally verified that the versioning algorithm provides consistency and durability of transactions. Unlike traditional versioning algorithms for databases [12], the 2V-DBP uses a bounded number of versions: two versions for data that are accessed by database pointers, other data uses one single version.

We have implemented 2V-DBP and compared it to the pessimistic concurrency-control algorithm *two-phase locking high priority* (2PL-HP). Our comparison scenario shows that the abortion ratio was significantly decreased, from an average of 75% using 2PL-HP to 25% using 2V-DBP. Furthermore, the worst observed response-time for hard transactions was greatly reduced from about 175μs to 27μs (of which 20μ is the system interrupt latency; the actual execution times for hard transactions were always in the range 5-7μs). Thus, we conclude that both hard and soft transactions benefit from the 2V-DBP algorithm. The cost for introducing 2V-DBP is slightly increased memory overhead for maintaining internal data structures and one extra version of the data elements used by hard transactions. For an example database, the overhead of using 2V-DVP instead 2PL-HP was calculated to 6,8%.

In our future work we will derive response time analysis for both hard and soft transactions. Response times for hard transactions will be trivial to derive since they no longer interfere with soft transactions and, thus, their accesses to the database can be modeled as accesses to shared variables. However, for soft transactions the response time analysis will be more advanced, and a suitable modeling technique has been deployed. Also, further performance evaluations, comparing 2V-DBP to other concurrency-control algorithms under various workloads are planned.

Some hard real-time applications may have consistency requirements when reading multiple values, i.e. they are not necessarily interested in the most recent values. For this reason we plan to introduce a *snapshot* functionality in 2V-DBP where also hard transactions way be able to read a consistent set of values.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database Systems*, 17, September 1992.

[2] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*, chapter 13.10.1 Immediate Ceiling Priority Inheritance. Addison-Wesley, second edition, 1996. ISBN 0-201-40365-X.

[3] S. Cannan and G. Otten. *SQL - The Standard Handbook*. MacGraw-Hill International, 1993.

[4] J. Huang, J. Stankovic, K. Ramamritham, and D. Towsley. Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 35–46. Morgan Kaufmann, September 1991.

[5] T.-W. Kuo, Y.-T. Kao, and L. Shu. A Two-Version Approach for Real-Time Concurrency Control and Recovery. In *Proceedings of the Third IEEE International High Assurance Systems Engieering Symposium*. IEEE Computer Society, November 1998.

[6] T.-W. Kuo and A. K. Mok. SSP: a Semantics-Based Protocol for Real-Time Data Access. In *Proceedings of 14th IEEE Real-Time Systems Symposium*, pages 76–86. IEEE Computer Society, December 1993.

[7] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[8] D. Nyström, A. Tešanović, C. Norström, and J. Hansson. Database Pointers: a Predictable Way of Manipulating Hot Data in Hard Real-Time Systems. In *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications*, pages 623–634, February 2003.

[9] D. Nyström, A. Tešanović, C. Norström, J. Hansson, and N.-E. Bånkestad. Data Management Issues in Vehicle Control Systems: a Case Study. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pages 249–256. IEEE Computer Society, June 2002.

[10] Polyhedra Plc. http://www.polyhedra.com.

[11] K. Ramamritham. Real-Time Databases. *International Journal of distributed and Parallel Databases*, 1(2):199–226, 1993.

[12] R. Rastogi, S. Seshadri, P. Bohannon, D. W. Leinbaugh, A. Silberschatz, and S. Sudarshan. Logical and physical versioning in main memory databases. In *The VLDB Journal*, pages 86–95, 1997.

[13] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Embedded Databases for Embedded Real-Time Systems: A Component-Based Approach. Technical Report MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-43/2002-1-SE, Dept. of Computer Engineering, Mälardalen University, January 2002.

[14] H. Thane, A. Pettersson, and D. Sundmark. The Asterix realtime kernel. In E. Tovar and C. Norström, editors, *Proceedings of the Work-in-progress and Industrial Session of the 13th Euromicro Conference on Real-Time Systems,Delft Netherlands*. http://citeseer.nj.nec.com/thane01asterix.html, June 2001.