Introducing Snapshots to Database Pointer Transactions *

Dag Nyström, Mikael Nolin, and Christer Norström
Mälardalen Real-Time Research Centre
Mälardalen University, Västerås, Sweden
{dag.nystrom, mikael.nolin, christer.norstrom}@mdh.se

Abstract

We present 2V-DBP-SNAP, an algorithm that allow hard real-time tasks in an embedded real-time control system to read a snapshot of a number of data elements in a real-time database. Furthermore, 2V-DBP-SNAP allows these data elements to be shared with soft real-time tasks, which access them using a database query language, and with other hard real-time tasks that use database pointers. 2V-DBP-SNAP allows temporal behavior and memory consumption to be accurately predicted. Introducing snapshot transactions is beneficial for embedded control-systems, such as for engine control in an automotive system, since a snapshot of the state of the environment can be collected, e.g., the state of all cylinders in the engine. 2V-DBP-SNAP is lightweight and predictable, both with respect to computational and memory overhead, and is therefore highly suited for resource constrained systems.

1. Introduction

As complexity in automotive control systems grows larger [6], the need for a structured way to handle and maintain data is needed [17]. One attractive solution is to integrate the data into a real-time database management system (RTDBMS) [2, 13, 15]. An RTDBMS can benefit the system development and the run-time system by providing, for example, tools to structure and manage data at design-time, mechanisms to allow complex queries to be issued at run-time, and allowing hard and soft real-time tasks to share common data. In [14], a concurrency control algorithm, denoted *2-Version Database Pointer Concurrency Control algorithm* (2V-DBP) that allow hard and soft tasks to access shared data, was introduced. The algorithm, which combines the concept of database pointers [16] and relational transaction management, 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. However, one drawback with 2V-DBP is that it doesn't allow hard transactions to access more than one data element per transaction. Consider, for example, a control-task that reads

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

a number of data elements from the database, and calculates a new actuator-value which is written back to the database. Such a task would, under 2V-DBP, have to consist of multiple transactions, each accessing (or writing) one data element. For a majority of control tasks in a vehicular system, using multiple transactions is not only sufficient, but preferable, since such an approach favors freshness of data. For a majority of control applications, data freshness is more important than achieving atomic transactions. However, a minor part of the control-tasks might need such atomic transactions, i.e., the ability to read (or write) a set of data in an atomic operation. One example of this might be in engine-control in which you need to have a consistent view, and control, of the state of all cylinders in the engine. To be able to handle this behavior, we extend the 2V-DBP algorithm to support *snapshots* [18] , we call this algorithm *2-version database pointer concurrency-control with snapshots* (2V-DBP-SNAP).

In this paper, we present and evaluate the 2V-DBP-SNAP concurrency control algorithm, which has the following benefits; (i) It allow hard tasks to access non-snapshot data in the database without being blocked by any database locks . (ii) It overcomes the widely recognized problem that soft transactions with low priority and long execution times are penalized due to the likeliness of data conflicts [9]. (iii) It allows hard tasks to access a set of data, using snapshot transactions, to be read and/or written atomically. Blocking for snapshot transactions are deterministic and bounded.

2V-DBP-SNAP is a concurrency-control algorithm that is both resource-efficient and predictable, and is well suited for complex embedded real-time systems, such as automotive control-systems.

The outline of the paper is as follows; in section 2 the system model is described. Here, the task model, as well as the transaction model is presented. The paper continues in section 3, where the database pointer concept and the 2V-DBP algorithm, which 2V-DBP-SNAP is based on, is described. In section 4, the proposed algorithm, 2V-DBP-SNAP, is presented and evaluated. Finally, in section 5 the paper is concluded.

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 [17]: (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 [17]. The tasks use database transactions to access and manipulate shared data stored in a database. Since tasks are preemptive, the RTDBMS needs some form of concurrency control to maintain consistency given multiple concurrent accesses.

2.1. Snapshots

We define snapshots and their operations as follows:

Definition 1 *A **snapshot** is defined as a consistent instant view of a set of data elements.*

Definition 2 *A **snapshot requirement** on a set of data elements is the requirement that the data elements must be read as a snapshot.*

Intuitively, definition 2 imply that any operation that access data within a snapshot must be able to retrieve a snapshot of the state of the data elements, even if the data elements currently are updated by some other operation.

Definition 3 *We refer to a set of data elements that have a snapshot requirement as **snapshot data**.*

Definition 4 *We define an operation as having a **snapshot property**, if the operation access snapshot data such that the snapshot requirement for the data is maintained.*

Intuitively, definition 4 imply that if an operation, e.g., a database transaction, access data elements within snapshot data (any combination of reads and writes) such that any other read operation still can get a snapshot of the data, the operation has a snapshot property. The easiest way to allow database transactions to have snapshot properties, would be to make them non-preemptive, i.e., a new transaction can never start before any currently executing transaction. Introducing concurrently executing transactions often makes it impossible for transactions to have snapshot properties, this includes well known traditional concurrency-control algorithms, such as 2-phase locking [7] and optimistic concurrency control [10]. One way of enabling transactions to have snapshot properties is to use multiple versions of all data elements in the database [8]. Achieving snapshot properties for all transactions and for all data in the database might require an unbounded number of versions. The overhead, both with respect to memory and CPU utilization, for applying such snapshots is often unacceptable. In 2V-DBP-SNAP, individual sets of data elements with snapshot requirements are identified, and only one extra version of each data element in the set is needed.

2.2. Application and task model

From a conceptual point of view, we divide the tasks in the system into three categories, namely, I/O-tasks, control-tasks, and management-tasks [17].

The I/O-tasks are hard real-time tasks, typically executed periodically, and 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. I/O-tasks touch very few, in most cases only one, data element in the database, and their transactions are always *precompiled*, (i.e., already formulated at compile-time). In some cases, consider the engine-control example from section 1, I/O-tasks might use snapshot data, i.e., all cylinders needs to be controlled atomically.

Task type	Soft Trans.	Hard Trans.	Snap. Trans.
Control tasks		(x)	(x)
I/O tasks		(x)	(x)
Management tasks	x		

Legend:
x - the transaction type is needed for the task type
(x) - the transaction type is needed for some tasks of the task type

Table 1. Transaction types for different task types.

Control tasks which also are hard real-time tasks, take a set of data values and derive new actuator values, thus 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 traditional RTDBMSs, since they focus on preserving transaction ordering rather than providing data freshness. Again, going back to the engine-control example, we show that control-tasks also might require snapshot capabilities.

Management tasks are soft real-time tasks, which executes soft database 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. We assume that all soft (i.e., management) tasks execute on lower priorities than all hard (i.e., I/O and control) tasks.

2.3. Transaction models

Tasks in the system interact with the RTDBMS through database transactions. Different types of tasks require different kinds of transactions. Therefore, transactions are divided into the following three classes:

1. Soft transactions, either precompiled or ad hoc (formulated and parsed at run-time) reside in soft real-time tasks. These transactions utilize a relational database query interface, e.g., SQL [5], for access. They provide a flexible and dynamic access to the data in the database to the system and are especially suited for management tasks, e.g., logging, diagnostics, and user interface (HMI) tasks, e.g., tasks controlling the instrument board.

2. Hard transactions, which are precompiled, reside in hard real-time tasks and have no snapshot properties. A hard transaction utilizes the database pointer interface, providing an efficient and predictable access to a single data element in the database. A majority of the transactions in a vehicle, i.e., vehicle control, would fall into this class.

3. Snapshot transactions, which are precompiled transactions with snapshot properties, also reside in hard real-time tasks. These transactions utilize the database pointer interface, but are allowed to access multiple data elements per transaction. Since snapshot transactions are more complex than hard transactions, and the need for snapshot data only apply to a limited number of tasks, one could expect few transactions of this class.

The task types and the transaction types that they use are summarized in table 1.

```
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

3. Database pointers with versioning

Before presenting our proposed concurrency control algorithm, we will recapitulate the database pointer concept presented in [16], and the concurrency control algorithm, denoted 2V-DBP [14], which our algorithm is based upon.

3.1. Database pointers

Database pointers allow individual data elements in a RTDBMS to be accessed in an efficient and predictable manner. They are intended as a complement to the relational data model, without limiting the expressibility of the relational model.

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:

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 and its type. The `version`, `update`, and `trans` entries are used by 2V-DBP, and will be discussed in section 3.2
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*`.

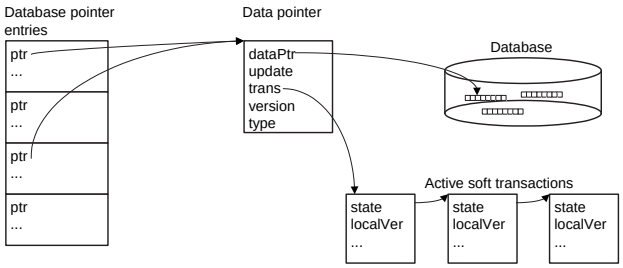


Figure 2. The data structures used by database pointers under 2V-DBP.

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.

3.2. The 2-version database pointer concurrency control algorithm (2V-DBP)

The 2V-DBP algorithm allows hard database transactions to execute without being blocked by soft database transactions [14]. 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 exist in the database in a similar way as in the two-version priority ceiling protocol [11], and the two-version two phase locking [3].

To maintain the versions of data elements, the following data structures are used (see figure 2):

- A second version of the data element (`version`) and the (`update`) flag which 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.
- A list of all active soft transactions, where each entry consists of, among other information, the current state (`state`), see section 3.3, of the transaction, and local working copies of the data elements used by each transaction (`localVer`).
- A pointer (`trans`) to any soft transaction holding a write lock on the data element is added to the data pointer. This pointer is necessary, in order to ensure that soft transactions can commit atomically, and thus will reveal no intermediate data to other transactions during its execution.

3.3. Soft transactions

The soft transactions utilize the relational part of the RTDBMS, and use an extended form of *two-phase locking high priority* 2PL-HP [1]. Soft transactions pass through the following four steps during its execution:

The Begin of Transaction step (BOT) in which the transaction becomes active. For dynamically formulated queries (i.e. ad hoc queries), the query is parsed.

The lock-obtaining step in which the transaction obtains all locks necessary to complete. Obtaining a data is performed in the following sub-steps:

1. Obtaining of the appropriate database lock, i.e., a write or read-lock. Arbitration of any lock-conflicts are solved according to 2PL-HP, i.e., in favor of the higher prioritized transaction.
2. Copying of the locked data to a local working copy (`localVer` in figure 2). If this data is acquired using a write-lock, the `update` flag of any data pointer to the locked data is also set to `clean`. Both these activities, e.g., copying to a local version, and setting the flag are performed in one non-preemptive operation.

After the data has been obtained, the local working copy might be modified.

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 data conflicts. However, when the transaction enters the committing step it cannot be aborted any longer, thus higher prioritized transactions in conflict will have to wait for the transaction to release the conflicting lock. Before writing a data element back to the database, the `update` flag is checked. If the data element is `dirty` (see section 3.4), it has been updated since it was read and the transaction will not overwrite the data element. This is to ensure that the soft transaction is serialized before any hard transactions that have updated the data element, see section 3.5.

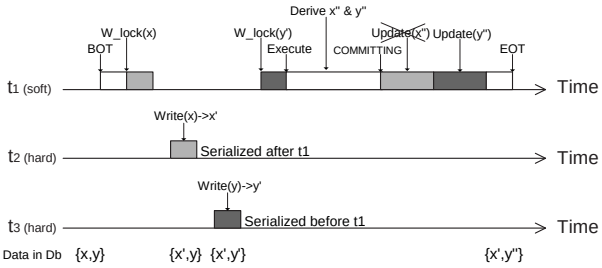
The End Of Transaction (EOT) step in which the transaction releases all locks, and the transaction is completed. When the EOT step has been executed, all changes to the database made by the transaction are made visible to other transactions. In 2V-DBP, performing this step is done by simply changing the state of the transaction to `NO_TRANS`.

3.4. Hard transactions

All hard transactions use database pointers. Hard transactions can access the same data elements as soft transactions, however hard transactions are never blocked by database locks. However, hard transactions take database locks in consideration

	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.
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.
t_1	$W\_lock(y')$		t_1 obtains write lock in y, thus obtaining a local copy of y'.
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 $x- > x''$, since it was updated by a hard transaction since it was read.
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.

(a) Execution scenario under 2V-DBP



(b) Time line of execution scenario

Figure 3. An execution-trace of three transactions

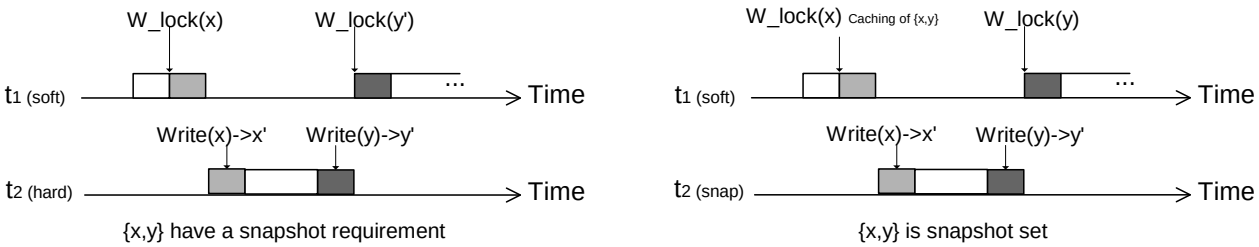
and access the database differently if the data element is locked.

Hard transactions that write to the database will, if the data is write locked by a soft transaction, set the `update` flag to `dirty` to indicate that the data element is not to be overwritten by the soft transaction. The execution-time of a hard transaction is short and close to constant, since it only contains a handful of sequential instructions. Furthermore, hard transactions are in its entirety executed non-preemptive. Experiences from implementing 2V-DBP show that the duration of these non-preemptive sections are significantly shorter than the interrupt latency of typical real-time operating systems. From a schedulability analysis perspective, the operating system itself introduces a greater blocking, and the hard transaction do therefore not influence the maximum blocking factor in the system.

3.5. 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 real-time data [12], since freshness of data often is more important than maintaining the traditional serialization order.

In 2V-DBP, for each soft transaction, a feasible serialization of all transactions can be found. However, different soft transactions can have different perceptions of the actual serialization order. Thus, 2V-DBP introduces a relaxed serialization order that favors data freshness. As an example of this relaxation, consider the execution trace depicted in figure 3(a). Figure 3(b) illustrates this trace on a time line for each transaction when using 2V-DBP. 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.



(a) Faulty trace under 2V-DBP

(b) Correct trace under 2V-DBP-SNAP

Figure 4. Execution-traces for transactions using snapshot data

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

In its current form, 2V-DBP is not sufficient to support the notion of snapshot data just by allowing hard transactions to read or update multiple data elements. Consider the example depicted in figure 4(a) in which the hard transaction t_2, preempting the soft transaction t_1, is allowed to update multiple data elements. In this example, t_1 reads x and y', hence t_1 only sees partial results of the updates performed by t_2. The two correct scenarios in this example would be that t_1 either reads x and y (t_1 is serialized before t_2), or that it reads x' and y' (t_1 is serialized after t_2). In this example, the snapshot requirement of $\{x, y\}$ is violated.

From the above example we see that introducing hard transactions with snapshot properties need a more elaborate handling, therefore we introduce the 2V-DBP-SNAP algorithm, which is an extension to 2V-DBP, that introduces a third transaction-class, namely the snapshot transaction. Snapshot transactions are allowed to read and/or update several database pointers during its execution, and are guaranteed to have a snapshot property, that is; (i) all the data read during its execution represent a snapshot of the database, and (ii) no intermediate results from the transaction will be visible in the database prior to its completion.

The main idea of 2V-DBP-SNAP is to cluster data that have snapshot requirements, i.e., are used by snapshot transactions, into *snapshot sets*. A snapshot set is defined as a set of data elements that are used by one or more snapshot transactions. In figure 5, three snapshot sets (ab, x, and y) can be seen. Note that snapshot transactions a and b have a common data element, thus they share the same snapshot set. Snapshot transactions are only allowed to access data element within its snapshot set. When a soft transaction locks a data element within a snapshot set, it must cache the entire snapshot set to its local working copy before it may continue to execute.

The result of using 2V-DBP-SNAP on the faulty example in figure 4(a) is depicted in figure 4(b). In this case, the soft transaction t_1 caches both x, and y, when locking x, and then use the cached value, when locking y. In the example we see that t_1 no longer sees a partial result of t_2 and the snapshot requirement of $\{x, y\}$ is now fulfilled.

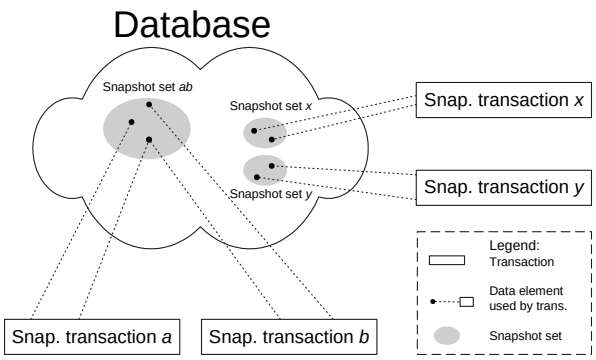


Figure 5. Snapshot sets in 2V-DBP-SNAP

4.1. Snapshot sets

Partitioning data into snapshot sets is a straightforward procedure, which can be easily automated. The algorithm works as follows:

Definition 5 *Let ss_x denote the set of data elements accessed by snapshot transaction x.*

The algorithm passes through the following two steps:

1. Let **set** SS be the set of all data elements sets, thus $SS = \{ss_1, ss_2, ..., ss_n\}$.
2. While there exist pairs of data elements sets (ss_x, ss_y) such that $ss_x \in SS, ss_y \in SS$ and $ss_x \cap ss_y \neq \varnothing$: Remove ss_x and ss_y from SS and add a new data elements sets $ss_{xy} = ss_x \cup ss_y$ to SS.

SS now contains all snapshot sets (which are all disjoint sets of data elements) of the database.

Both steps of the algorithm are straightforward since all snapshot transactions are precompiled, and thus all data elements possibly accessed are known at compile-time.

Partitioning data into snapshot sets is typically performed off-line, using a tool. The algorithm can, however, be applied during run-time. This would allow the system to create new snapshot transactions during run-time. However, we will not elaborate further on this in this paper.

4.2. The 2V-DBP-SNAP data structures

The 2V-DBP data structures from figure 2 must be extended to incorporate the snapshot sets. This is done by adding a new data structure, the *snapshot pointer*, which have a similar functionality as the data pointer described in section 3.1. In difference from the data pointer, which maintain a single data element, a snapshot pointer handle sets of data, i.e., all data within one snapshot set.

The snapshot pointer contains the following entries, see figure 6:

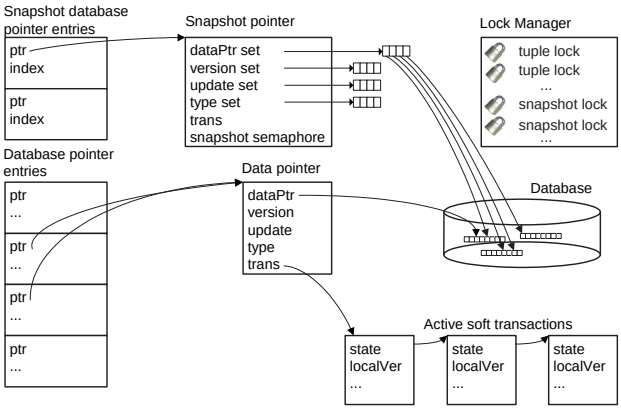


Figure 6. The extended data structures used by 2V-DBP-SNAP.

- **The** `dataPtr set`**,** `version set`**,** `update set` **and** `type set` **entries.** These entries are equivalent to the `dataPtr`, `version`, `update` and `type` entries in the data pointer, but are extended to sets with one element per data element in the set.

- **The** `trans` **entry.** This entry is similar to the `trans` entry in the data pointer, and contain a pointer to any soft transaction currently holding a *snapshot lock* on the snapshot set. Snapshot locks are ordinary database locks, maintained by the RTDBMS lock manager [15], that are used by soft transactions when accessing a snapshot. The usage of snapshot locks is further explained in section 4.5.

- **The** `snapshot semaphore` **entry.** This entry is used to enforce mutual exclusion among snapshot transactions accessing the same snapshot set, The usage of snapshot semaphores is further discussed in section 4.3.

4.3. Introducing snapshot transactions

A snapshot transaction operate in a similar fashion as hard transactions, with the difference that a snapshot transaction can read and manipulate multiple data elements. Snapshot transactions, as hard transactions, use the database pointer interface to access data elements. In addition to this a `begin of transaction` and an `end of transaction` are used to indicate what part of the application code is within the transaction. In figure 7, the pseudo code for a control task with a snapshot transaction is depicted. In the initialization part of the task (lines 2 - 4), the database pointers are bound to its respective data elements. The actual transaction (line 6-11) is then executed periodically.

The work-flow of a snapshot transaction is as follows:

1. **Begin of transaction.** In this step, the transaction is started (line 6 in figure 7). It consists of two steps, namely:

```
1 TASK engineControl(void) {
2   int s1,s2,a1,a2;
3   DBPointer *sens1,*sens2, act1, act2;
4   // Task initialization...
5   while(1){
6     snapBeginofTrans();
7     s1=read(sens1);
8     s2=read(sens2);
      //control-code to derive a1 & a2 from s1 & s2
9     write(act1,a1);
10    write(act2,a2);
11    snapEndofTrans();
12    waitForNextPeriod();
    }
  }
```

Figure 7. Example of a snapshot transaction.

- The snapshot semaphore is obtained, in order to ensure mutual exclusion among all snapshot transaction accessing the same snapshot set.
- Identify if a soft transaction is currently holding a write-lock on the snapshot set, see section 4.5. If this is the case, the `update`-flag must be set to `dirty` for all data elements manipulated by the transaction. This is similar as for hard transactions executing under 2V-DBP.

2. **Execute the transaction.** That is, read and write to any data elements within the snapshot set (lines 7-10 in figure 7). Calculations to derive the transaction results can also be executed.

3. **End of transaction.** In this step, the transaction is ended (line 11 in figure 7). In this step, the snapshot semaphore is released.

The snapshot semaphore, which is crucial to ensure transaction isolation among snapshot transactions, might introduce blocking on other, higher prioritized snapshot transactions executing on the same snapshot set. This blocking is, however, deterministic if a real-time semaphore algorithm, such as the immediate inheritance protocol is used [4], and if the worst case execution time is bounded for each snapshot transaction. It is noteworthy that this blocking affects neither snapshot transactions accessing other snapshot sets, hard transactions nor soft transactions.

4.4. Hard transactions under 2V-DBP-SNAP

Hard transactions under 2V-DBP-SNAP can only modify data elements that are not in any snapshot set. However, it is possible for a hard transaction to read a data element from a snapshot set. Not allowing hard transactions to update data in a snapshot set is not a limitation, but a consequence of the concept of snapshots. If data elements within a snapshot set would be individually updated by different transactions, the snapshot requirement would be violated. However, if a behavior where individual elements of snapshots are updated is wanted, snapshot transactions updating these data elements can be used.

Transaction type	**Soft**	**Hard**	**Snapshot**
Soft	L		
Hard	V	M	
Snapshot	S & V	D	M or D

Legend:
L - Database locks
V - Versioning
M - Mutual exclusion
D - Disjoint data
S - Snapshot set

Table 2. Serialization policies between transaction types.

4.5. Extending soft transactions

Soft transactions executing under 2V-DBP-SNAP differ from soft transactions executing under 2V-DBP in the following two ways:

1. In difference to snapshot transactions, soft transactions cannot read individual data elements from the snapshot at arbitrary points in time. If a soft transactions need to access a data element in a snapshot set, the entire set must be cached to the local working copy. This is a non-preemptive operation. If the transaction, later in its execution, requests a different data element within the same snapshot set, the transaction uses the cached copy. Even though the soft transaction reads the complete set it might choose to only update a subset of the elements in the set.

2. Prior to reading the snapshot set, the corresponding snapshot lock must be obtained. Since the snapshot set is a shared resource it needs to be protected, the same way as ordinary data tuples. Just as for any lock in 2V-DBP, snapshot locks are managed by the lock manager in the RTDBMS, they can be either read-, or write-locked, and lock conflicts are resolved using the 2PL-HP policy.

4.6. Serialization in 2V-DBP-SNAP

In 2V-DBP-SNAP, a set of different serialization techniques are used. Table 2 show the different policies used between different transaction types. We refer to [14] for a discussion about the serialization among soft and hard transactions, and instead focus on the serialization policies used for snapshot transactions. Table 2 show three possible serialization cases, namely serialization among:

1. **Two concurrent snapshot transactions.** In this case, there are two possible sub-cases; (i) If the transactions access different snapshot sets, no conflicts can occur, and (ii) if the transactions access the same snapshot set, the snapshot semaphore will ensure mutual exclusion (and thus no conflicts). The transactions are serialized in the order they obtain the snapshot semaphore.

2. **A snapshot transaction concurrent with a hard transaction.** In this case, there are three possible sub-cases; (i) A hard transaction is not allowed to write to data elements in a snapshot set, therefore no conflicts can occur. (ii) If a hard

transaction reads a data element after the snapshot transaction has updated it, the hard transaction is serialized after the snapshot transaction. (iii) If a hard transaction reads a data element before the snapshot transaction has updated it, the hard transaction is serialized before the snapshot transaction.

3. **A snapshot transaction concurrent with a soft transaction.** Since soft transactions read the entire snapshot set in one non-preemptive operation , it can be viewed as one data element (from a soft transactions perspective). Figure 4(b) illustrates this. If a snapshot set is manipulated by a snapshot transaction after being cached by a soft transaction, the versioning algorithm will ensure that, just as for soft transactions in 2V-DBP, only the clean data elements (determined by the `dirty` flag) are updated. Thus, a snapshot transaction is serialized before a soft transaction iff it is executed before the soft transaction caches the snapshot set, otherwise the snapshot transaction is serialized after the soft transaction.

4.7. Evaluation of 2V-DBP-SNAP

The aim of 2V-DBP-SNAP is to allow hard control-tasks to achieve a consistent view and control of the state of the environment. This aim must be fulfilled without the introduction of unpredictable, or significant blocking. In fact, 2V-DBP-SNAP introduces two forms of blocking for snapshot transactions.

First, the snapshot semaphore, introduce blocking among concurrently executing snapshot transactions. This approach can be compared to using shared variables which are protected by a semaphore. One benefit of managing this in database transactions instead of in the application code, is that minimal snapshot sets automatically can be constructed off-line, thus increasing the concurrency in the system. Furthermore, deadlocks caused by erroneous semaphore usage can be avoided since only one semaphore is used per transaction.

Second, soft transactions introduce blocking on snapshot transactions, since soft transactions needs to read (or update) a set of data elements in a non-preemptive operation. If, during the creation of the snapshot sets, the snapshot transactions overlaps to a large extent, the snapshot sets might get, in fact, arbitrarily large. Note, however, that this blocking is deterministic, and the maximum blocking factor can be derived and analyzed off-line. From our experience with automotive control-systems [17], only few data elements in a system have a need for snapshot semantics, and thus the snapshot sets are not likely to be large. An implementation of 2V-DBP-SNAP under the Asterix real-time operating system (rtos) [19] executing on an 133Mhz Intel 486 processor show that snapshots with up to 939 data elements can be used before reaching blocking times that exceeds the interrupt latency of the rtos, see figure 8. From the figure we can see that the rtos latency roughly is approximately 250μ sec. The snapshot caching in the soft transaction reaches this time at approximately 940 data elements. The evaluation show that, in the normal case, soft transaction that cache a snapshot will not affect the maximum blocking of the system, since snapshot sizes will not reach this size.

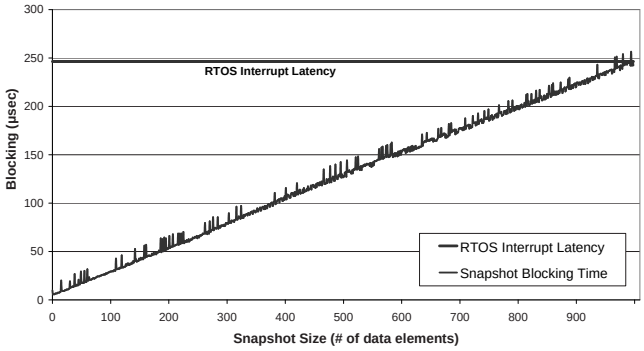


Figure 8. Blocking introduced by soft transactions caching a snapshot set

Another limitation in 2V-DBP-SNAP is snapshots used in interrupts. Introducing blocking for interrupt handlers is not preferable, however, in its current form, 2V-DBP-SNAP, may introduce blocking for interrupt handlers that executes snapshot transactions. One possible solution to this is to manage this data in the application, by letting the interrupt handler write the data to a buffer, which is then handled by a task executing a snapshot transaction.

However, these limitations do not negatively influence the system, or the users of the system. Instead 2V-DBP-SNAP is designed with automotive system's requirements in mind, and its minimal overhead is to a high degree suitable for use in such systems.

5. Conclusions

We have presented a concurrency control algorithm which is based on a previous algorithm, denoted 2V-DBP [14], which allows co-existence of soft real-time, relational database transactions (soft transactions), hard real-time database pointer transactions (hard transactions) [16].

The presented algorithm, called 2-version database pointer concurrency-control with snapshots (2V-DBP-SNAP), extends 2V-DBP by introducing snapshot support. This allows hard real-time tasks to get a consistent instant view of a set of data elements in the real-time database. Furthermore, this data can be shared with soft database transactions without violating the snapshot requirements on the data, i.e., the requirement to be able to get a snapshot of the data elements. To be able to allow transactions to have snapshot support, the concept of snapshot sets are introduced.

2V-DBP-SNAP is designed to be lightweight with respect to overhead and blocking of transactions. An implementation of 2V-DBP-SNAP show that this also is the case. The algorithm is intended for hard real-time control application, e.g., automotive control systems, in which hard control-tasks need a consistent view (or a consistent control) of multiple data elements.

Future work include introducing wait-free or lock-free algorithms to the snapshot sets in order to allow concurrently executing snapshot transactions to access the same snapshot set.

References

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

[2] S. F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring. DeeDS Towards a Distributed and Active Real-Time Database System. *ACM SIGMOD Record*, 25(1):38–40, 1996.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Publishing Company , 1987.

[4] 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.

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

[6] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano - a Revolution in On-Board Communications. Technical report, Volvo Technology Report, 1998.

[7] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *The communications of the ACM*, 19(11):624–633, November 1976.

[8] T. Gustafsson. Maintaining data consistency in embedded databases for vehicular systems. Linköping Studies in Science and Technology Thesis No. 1138. Linköping University. ISBN 91-85297-02-X, 2004.

[9] 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.

[10] H. T. Kung and J. T. Robinsson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[11] 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.

[12] 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.

[13] J. Lindstrom, T. Niklander, P. Porkka, and K. Raatikainen. A Distributed Real-Time Main-Memory Database for Telecommunication. In *Proceedings of the Workshop on Databases in Telecommunications*, pages 158–173. Springer, September 1999.

[14] D. Nyström, M. Nolin, A. Tešanović, C. Norström, and J. Hansson. Pessimistic Concurrency Control and Versioning to Support Database Pointers in Real-Time Databases. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 261–270. IEEE Computer Society, June 2004.

[15] D. Nyström, A. Tešanović, M. Nolin, C. Norström, and J. Hansson. COMET: A Component-Based Real-Time Database for Automotive Systems. In *Proceedings of the Workshop on Software Engineering for Automotive Systems*, pages 1–8. The IEE, June 2004.

[16] 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.

[17] 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.

[18] H. Sundell and P. Tsigas. Simple Wait-Free Snapshots for Real-Time Systems with Sporadic Tasks. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications*. Springer-Verlag, August 2004.

[19] 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.