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.

Proceedings of the 12th 16th Euromicro Conference on Real-Time Systems (ECRTS'04)
1068-3070/04 $20.00 © 2004 IEEE

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.

Proceedings of the 12th 16th Euromicro Conference on Real-Time Systems (ECRTS'04)
1068-3070/04 $20.00 © 2004 IEEE

```
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:

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.

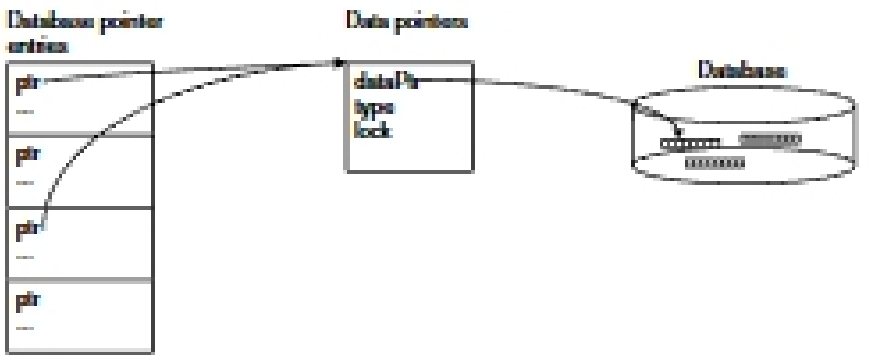

Figure 2. The data structures for database pointers.

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,

Proceedings of the 12th 16th Euromicro Conference on Real-Time Systems (ECRTS'04)
1068-3070/04 $20.00 © 2004 IEEE