Concurrency Control Protocols for Persistent Shared Virtual Memory Systems*
(Extended Abstract)

Igor Nekrestyanov, Boris Novikov, Ekaterina Pavlova, and Serge Pikalev
University of St.-Petersburg, Russia
E-mail: {kate,igor,sep}@meta.niimm.spb.su
borisnov@acm.org

Abstract

A family of optimistic concurrency control protocols for real-time persistent systems with critical performance requirements and very tight deadlines is proposed.

Under assumptions that almost all transactions are extremely short and read-only and the entire database resides in main memory (which are realistic for this type of applications), we propose distributed concurrency control protocols which exploit the virtual memory management for concurrency control. Virtual memory management is required anyway, so the additional overhead is extremely low, especially for most frequent read-only transactions.

Two variations of the proposed protocol are described and compared with alternative approaches.

1 Introduction

The concurrency control techniques for database systems, developed in late 70-ties, work fine for conventional databases, but are inadequate for new application areas. To meet requirements of some of these new applications, several advanced transaction models were proposed and evaluated [3, 4]. Most of these new models relax some or all of ACID properties of transactions to allow certain types of co-operation or enhance data sharing between different transactions. This is especially important for applications where human co-operation is essential, e.g. various types of collaborative design or production. That is, significant research effort was spent to improve functionality of transactions.

However, these advanced models are of little, if any, help for systems where performance issues, such as response time, are critical, e.g. real-time applications. Typically, these systems rarely require sophisticated transaction interactions, but have severe additional constraints imposed on concurrency control. The additional requirement is that transactions may have deadlines for commit time [5]. Significant amount of work was recently done in this area [5, 1, 9]. In particular, it is shown in [5] that optimistic concurrency control protocols can significantly outperform traditional locking-based pessimistic ones, due to better CPU utilisation.

*This work was partially supported by Russian Foundation for Basic Research under grant 95-01-00636, and UrbanSoft Ltd. under contract 36/97.

Proceedings of the First East-European Symposium on Advances in Databases and Information Systems (ADBIS'97). St.-Petersburg, September 2–5, 1997.
St.-Petersburg: University of St.-Petersburg, 1997.

In this paper we study concurrency control for the special type of real-time systems with the following assumptions:

- most of transactions are read-only;
- most of transactions are extremely short and have very tight deadlines (say, the transaction should commit in tens of milliseconds after arrival);
- data items accessed by transactions are very small, but may have complex structure and links, so any mismatch with programming language should be avoided.

For the latter reason, the persistent store of the system in question should be organized as a type of an object-oriented database system.

Further, for external reasons, such as reliability and performance, the system is distributed over several processing units. To meet the performance requirements (regardless of concurrency control) it is necessary to place the entire database into main memory. Even under this assumption, the concurrency control becomes a problem due to unacceptable processing overheads (e.g. locking of a single data element usually will require several remote procedure calls, each of which may consume few milliseconds - a significant percentage of time allowed for the whole transaction before deadline.

On the other hand, the probability of conflicts is extremely low. For this reason, a concurrency control algorithm with low overhead is highly desirable.

A family of optimistic concurrency control protocols which can meet these requirements is described in this paper.

We assume that the database is main-memory resident. To keep overhead minimal, we choose page-level granularity of access control. The objects are expected to be very small, hence object-level control would result in increased amount of processing necessary to keep track of accesses. This decision might lead to decrease of concurrency, but this is not for read-only transactions, which are our major concern.

The rest of the paper is organised as follows.

In section 2 a model of the environment is described and motivated. Section 3 outlines the simplest version of the proposed concurrency control scheme. More sophisticated protocol with multiversion concurrency control is discussed in section 4. A comparison of the proposed techniques with other approaches is presented in section 5, and conclusion summarises results of the paper.

2 The Model

Conceptually, we assume that the real-time system runs on a multiprocessor hardware platform with shared memory. In realty, the shared memory may be virtual, implemented over a network.

This type of hardware attracted significant attention of researches during few last years. The interest is especially stimulated by appearance of 64-bit addressing, which makes possible to explore single virtual address space for several computers for implementation of operating systems and persistent object stores (e.g. [6, 10, 11, 8]). Another implementation of distributed shared virtual memory is proposed in [2] and used for implementation of distributed object directory.

However, in this paper we are not particularly interested in the implementation architecture of distributed shared memory and we describe our concurrency control algorithms in terms of a specific model of the system.

The hardware configuration in our model consists of several processors which all can access directly shared persistent virtual memory. We assume, for the purposes of this paper, that this memory is completely reliable (so that we do not need to consider recovery after memory failures, which can be efficiently implemented in a way similar to described in [7]). However, the processor failures are possible in our model.

Each processor possesses a certain amount of private volatile memory. The address space is sufficiently large to include all persistent memory at any time (consequently, we can assume that memory addresses may be used as object references, which is not essential for this paper, however).

Each transaction runs within a process (which may probably migrate from one processor to another, but this is not essential for our purposes). Each process has it's own virtual address space of equal size which potentially can include the whole shared persistent memory (as well as private volatile memory of the processor).

The memory is segmented into pages, and each process can access memory via the page table which is maintained for each address space separately.

The relationship between process and address space is not necessarily one-to-one correspondence, we only need to assume that only one transaction can run in the address space at any time.

In addition to (local) page tables for each address space, we need a global page table (probably maintained as a page table of the transaction manager process, which may run on any of processors).

Both hardware and the operating system provide some support for virtual memory (this assumption is not too restrictive). In addition, we assume that transaction manager can provide call-back functions to virtual memory manager and can influence the page replacement policy through these call-back functions.

We assume that hardware can trap both attempt to access a page which is not present in the active page table, and attempt to write data to a page which is marked as read-only in the active page table.

Finally, we need some low-level protection of shared memory against concurrent updates of a single location, as well as some communication facilities connecting the processors (so that they can notify each other on certain events). In other words, our concurrency control should ensure global consistency provided that local updates are protected by hardware.

3 The Rough Algorithm

The main idea of our protocols is to use virtual memory management to trace access to persistent data and to detect the inter-transaction conflicts.

The algorithm described in this section uses minimal additional data structures and is very simple, so that the overhead is minimised. However, it may cause unnecessary transaction aborts and decreased performance due to extra restarts.

We assume that the transaction manager has access to page tables of all address spaces that are in use on the same processor. The transaction manager may either use separate process or may be invoked from regular application process (e.g. via exception handler), and in any case it can also access global page table.

The transaction manager maintains a transaction counter, which should be globally unique. This counter may be also used to define serialization order. It may or may not be coupled with logging, or may reside in a single location of the persistent shared memory. In the latter case this location may become a bottleneck of the system. However, we need some kind of transaction counter to ensure serializability.

The protocol requires some data structures which allow to determine when the particular page was last updated. There are several ways to maintain this information. For example, each page may contain a log record number provided by logging service (hence at no cost for transaction manager).

Alternatively, the transaction manager can maintain a "black list" of recently updated pages.

To present the protocol, we further describe actions performed by transaction manager on certain events.

3.1 Transaction Identifiers

The transaction manager registers the new transaction, creating a globally unique sequential transaction identifier.

There are two possibilities to create the identifier:

- The identifiers may be obtained from a centralised service (e.g. via counter in shared memory, as mentioned above).
- Each transaction manager maintains local transaction counter, and the global counter is used only for writing transactions.

  The transaction identifier should contain:

  1. the value of global counter (advanced when a writing transaction commits),
  2. transaction manager identifier, and
  3. local sequential number (relative to the global number).

The advantage of the latter implementation is that the identifiers for most frequent read-only transactions can be assigned locally on each processor without any synchronization with other transaction managers. However, this structure of identifier is essential for the protocol described in the following section.

3.2 Read Operation

When the transaction attempts to read some data in the shared persistent memory for the first time, an addressing exception occurs, and appropriate virtual memory manager is invoked.

The required page is located (using the global page table) and inserted into the page table of the requester for read access to this page. However, this is done only if the page was not updated after start of the requesting transaction, otherwise the requester is aborted.

Once the page was inserted into local address space, it remains there at least until the transaction terminates (a kind of a non-steal policy), so all subsequent accesses to the same page can be performed without any interruptions or overheads other than hardware resolution of virtual addresses.

3.3 Write Operation

When a transaction attempts to update the data in shared persistent memory, an exception occurs, because initially each page is available in (at most) read-only mode.

The exception handler then creates a private copy of the page to be accessed and appropriately updates the local page table. This copy may or may not be allocated on the persistent memory.

In addition, the transaction is registered as writing transaction and the manager makes housekeeping records to be able to handle the private (shadow) copy properly when the transaction terminates.

3.4 Abort

In case of abort of the read-only transaction the information about this transaction is discarded. When writing transaction aborts (e.g. for it's internal reasons), all updated pages (that is, private copies) are simply discarded and all information about this transaction may be erased.

3.5 Read-Only Commit

When read-only transaction attempts to commit, the transaction manager validates if this transaction had conflicts with other transactions (that is, the transaction should be validated).

The complexity of the validation depends on the data structure used to track updates and on the amount of updates performed by writing transactions. More detailed description of possible alternatives and trade-offs will be included in the full version of the paper.

3.6 Update Commit

When writing transaction attempts to commit, the transaction manager performs validation procedure. The transaction is aborted (and planned for restart), if another writing transaction was committed after the start of the transaction in question. This is a very rough decision, actually two writing transactions cannot run in parallel, but under assumption that writing transactions are extremely infrequent this decision may be acceptable. To make the decision, it is sufficient to compare the identifier of the last committed writing transaction and the transaction in question.

If the transaction may be committed, then its updates are reflected in the global page table and all other transaction managers are notified that the transaction is being committed.

Then the updated pages are removed from all local address spaces in which they were present. If a read-only transaction will not access such page again, it may commit, otherwise it will be aborted on attempt to re-read the updated page.

3.7 Crash Recovery

Under assumption that the persistent memory is reliable (that is, the recovery is implemented at another layer), few actions should be performed when processor restarts.

All page tables for this processor should be cleared, and new transactions can be executed provided that the transaction identifiers can be consistently obtained from the persistent memory.

Some additional clean-up activity may be necessary if private copies of updated pages for failed transactions are kept in persistent memory.

3.8 Correctness

The correctness of the algorithm described above is obvious due to the fact that all writing transactions are executed serially. Every commited read-only transaction can be serialized before the first writing transaction committed after the start of read-only transaction in question.

4 Version-Based Approach

In this section we describe an enhancement of the protocol which avoids multiple aborts of read-only transactions when writing transaction is committed.

In this protocol, we use multiple versions of pages and allow read-only transactions to access previous states of updated pages.

To implement this feature the transaction manager should maintain additional data structures to keep track of existing versions for each (recently) updated page, and this data should be consistent for all transaction managers.

Each version of a page has associated version number, which is equal to global part of the transaction identifier (which is increased when writing transaction appears).

During the initialisation of a transaction an interval of admissible version numbers for this transaction is calculated.

4.1 Read Operation

When a transaction attempts to read data from a page of shared persistent memory, the transaction manager looks for the page address in the global page table. If the page is found there, this page has only one version, which is made accessible to the requester.

Othervise the transaction manager looks for appropriate version of a page in the list of versioned pages and selects a version which is suitable for the requesting transaction (the selection is based on the transaction identifier and the version number associated with each version of a page).

4.2 Write Operation

If transaction attempts to write to the latest (or the only) version of a page it can proceed the same way as in the Rough algorithm, otherwise the transaction is aborted.

4.3 Abort

The transaction aborts are processed the same way as in the rough protocol.

4.4 Read-Only Commit

The read-only transaction can commit regardless of any other circumstances, because proper versions of each page it accessed were provided at the first access.

The only responsibility of the transaction manager is to perform clean-up, if necessary.

4.5 Update Commit

The validation of the writing transaction includes check of all updated pages. If for some pages which were updated by the transaction in question more recent versions exis the transaction should be aborted, otherwise it commits.

Some special care is required to update version control tables, but we omit the details here.

4.6 Crash Recovery

As for rough protocol, nothing except clean-up is required.

4.7 Obsolete Versions

The detection and removal of the obsolete versions in a distributed environment is not a trivial task. Two albernatives to solve it are possible: a kind of “garbage collection” process or a set of appropriate counters.

The latter solution seems preferable because it avoids occasional increase of workload due to activation of a garbage collector. A table of counters can be maintained with only hardware protection feature assumed in our model (single location may be protected from accses of other processors). No conflicts are possible because each processor updates only it’s own counters.

4.8 Correctness

The correctness of this protocol is even more obvious because every commited read-only transaction work with same state of database which was at the time of its start.

4.9 Variations

There are few design issues not mentioned in the description of both algorithms.

During normal processing, new pages are included into page tables of address spaces used by transactions, but they are not removed until the transaction terminates. Different strategies are possible for cleaning of local address spaces:

- the address space may be cleaned on termination of each transaction;
- pages are removed (on transaction commits) only when the size of page table exceeds certain threshold[1]

Note that the address space is cleaned unconditionally if transaction was aborted for some reason.

The first of alternatives results in increased processing overhead, while the second may increase the number of restarts (because probably transaction inherits some pages which are not necessary for it).

Another issue is removal of obsolete versions. To detect the fact that the version is obsolete, some sort of garbage collector is required. Again, we omit details here.

5 Comparison

5.1 Rough vs Version

The only advantage of the rough scheme is the extremely low overhead for successfully committed read-only transactions. This advantage, together with the simplicity of the algorithm, may be very significant under our assumptions, especially when writing transactions are extremely rare.

The disadvantages of the rough protocol are:

- Any commit of writing transaction implies enormous number of restarts. If the address spaces are not cleared on transaction commit, unnecessary restarts may occur.
- A validation procedure is required during commit of read-only transaction.

[1] The latter option cannot be used directly for the version-based algorithm due to neccesity of cleaning up references to old versions. In this case adress spaces of all transaction, that potentially used old versions, should be cleaned unconditionally.

The major advantage of versioned scheme is that read-only transactions can always commit, hence most important and time-critical (under our assumptions) transactions are never restarted.

It is hardly possible to obtain realistic analytical performance estimations for the system in question, therefore, simulation and measurements on the prototype are necessary to find out when one of the schemes is better than another.

5.2 Other Approaches

Any concurrency control algorithm is either pessimistic, based on locking, or optimistic. The former algorithms prevent conflicts forcing transactions to wait for requested resources, while the latter execute transactions concurrently and then check if the actual schedule is serializable.

The optimistic algorithms can be further divided into ”forward validation” and ”backward validation” types. The former algorithms abort all running transactions which are in conflict with the transaction to be committed, while the latter abort the transaction under validation if it is not serializable with already committed.

Both algorithms described in this paper are optimistic backward validation.

The common disadvantages of all optimistic algorithms are:

Restarts The number of restarts will not be significant due to our assumptions, especially for versioned scheme, where the most important read-only transactions are not restarted at all. For Rough algorithm, the number of restarts will be greater than for standard optimistic algorithm due extra restarts of writing transactions.

Validation Normally requires significant computational resources and is a major reason of poor performance. In versioned scheme the validation is not present for read-only transactions, but is hard for writing transactions.

Thus, we achieve major objection of our algorithm - high performance for most critical read-only transactions.

Huge amount of auxiliary data Both protocols exploit page tables which are used to support virtual addresses and should be maintained anyway.

For these reasons the proposed algorithms should outperform standard optimistic algorithm taking into account the specific requirements and assumptions.

The pessimistic concurrency control cannot provide acceptable performance due to locking overhead and can result in increased number of missing deadlines due frequent waits.

6 Conclusion and Future Work

In this paper we propose an optimistic concurrency control scheme for real-time persistent stores based on shared virtual memory. The major advantage of this scheme is extremely low processing overhead, especially for most critical and most frequent types of transactions.

The actual performance gains should be further investigated.

Further performance improvements may be expected when this concurrency control technique is combined with other functionality, e.g. access control or recovery. We plan to study these topics in our future work.

References

[1] S. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring. DeeDS: towards a distributed and active real-time database systems. *ACM Sigmod Record*, 15(1):38–40, Mar. 1996.

[2] K. Barker, R. Peters, and P. Graham. Distributed Shared Memory for Interoperability of Heterogeneous Information Systems—*Position Statement*. In *OOPSLA Workshop on Interoperable Objects—Experiences and Issues*, Oct. 1995.

[3] A. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan-Kaufmann, February 1992.

[4] T. Harder and K. Rothermel. Concurrency control issues in nested transactions. *VLDB*, 2(1):39–74, Jan. 1993.

[5] L. Juhnyoung and H. S. Sang. *Performance of Concurrency Control Mechanisms in Centralized Database Systems*. Prentice-Hall, 1996.

[6] E. Koldinger, J. Chase, and S. Eggers. Architectural Support for Single Address Space Operating Systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, Oct. 1992.

[7] E. Levy and A. Silberschatz. Incremental Recovery in Main Memory Database Systems. Technical Report TR-92-01, Dept. of Computer Science, University of Texas at Austin, January 1992.

[8] A. Lindstrom, A. Dearle, R. di Bona, A. Norris, J. Rosenberg, and F. Vaughan. Persistence in the grasshopper kernel. In *Proceedings of the 18th Australasian Computer Science Conference*, pages 329–338, Feb. 1995.

[9] P.Krzyżagórski and T.Morzy. Optimistic concurrency control algorithm with dynamic serialization adjustment for firm deadline real-time database systems. *ADBIS"95*, 1:21–28, jun 1995.

[10] R. Schmidt, J. Chase, and H. Levy. Using Shared Memory for Read-Mostly RPC Services. In *Proceedings of the 29th Hawaii International Conference on System Sciences*, Jan. 1996.

[11] M. Shapiro and P. Ferreira. Larchant-RDOSS: a Distributed Shared Persistent Memory and its Garbage Collector. In *Proceedings of the 9th International Workshop on Distributed Algorithms*, Sept. 1995.