Synchronization and Recovery in an Embedded Database System for Read-Only Transactions

Subhash Bhalla and Masaki Hasegawa

The University of Aizu, Aizu-Wakamatsu, Fukushima PO 965-8580, Japan
{bhalla, d8041201}@u-aizu.ac.jp

Abstract. Transactions within an embedded database management system face many restrictions. These can not afford unlimited delays or participate in multiple retry attempts for execution. The proposed embedded concurrency control (ECC) techniques provide support on three counts, namely - to enhance concurrency, to overcome problems due to heterogeneity, and to allocate priority to transactions that originate from critical host.

1 Introduction

An embedded system is most often dedicated to a single application or small set of tasks. The software to manage them is small and simple. The operating conditions of the system are typically more restrictive than those of general purpose computing environments. An embedded system must continue to function without interruption and without administrator intervention.

In this paper, we focus on the Asilomar report "gizmo"databases [3]. These databases reside in devices such as smart cards, toasters, or telephones. The key characteristics of such databases are the following,

- the database functionality is completely transparent to users,
- explicit database operations or database maintenance is not performed,
- the database may crash at any time. It must recover instantly,
- the device may undergo a hard reset at any time. It requires that the database must return to its initial state, and
- the semantic integrity of the database must be maintained at all times.

As embedded systems define a specific environment and set of tasks, requiring expertise during the initial system configuration process is unacceptable. Many research efforts focus their attention on the maintenance of the system. For example, Microsoft's Auto Admin project [6], and the "no-knobs" administration. These have been identified as an area of important future research by the Asilomar authors [3].

2 Motivation - Embedded Database Systems

There are a few tasks that are typically performed by database administrators (DBAs) in a conventional database system. These tasks must be automated in an embedded system.

2.1 Outline of Requirements

Embedded systems typically perform simple queries. The relevant criteria are ease of maintenance, robustness, and small footprint. Of these three requirements, robustness and ease of maintenance are the more important criteria. Users must trust the data stored in their devices and must not need to manually perform anything resembling system administration in order to get their unit to work properly.

Application Level Database System. In an embedded database, the normal maintenance tasks must be automated. These are not necessarily based on the initial system configuration prepared by a user. There are five tasks that are traditionally performed by DBAs, but must be performed automatically in embedded database systems. These tasks are log archival and reclamation, backup, data compaction / reorganization, automatic and rapid recovery, and re-initialization from scratch.

Log archival and backup are tightly coupled. Database backups are part of any large database installation, and log archival is analogous to incremental backup [13]. There are a few implications of backup and archiving data in an embedded system. Consumers do not back up their VCRs or refrigerators, yet they back up their personal computers or personal digital assistants. We assume that backups, in some form, are required for gizmo databases (imagine having to reprogram, manually, the television viewing access pattern learned by some set-top television systems today) [13]. Furthermore, we require that those backups are nearly instantaneous or completely transparent,as users should not be aware that their gizmos are being backed up and should not have to explicitly initiate such backups.

Data compaction or reorganization has traditionally required periodic dumping and restoration of database tables and the recreation of indices. In an embedded system, such reorganization must happen automatically.

Recovery issues are similar in embedded and traditional environments with a few exceptions. While a few seconds or even a minute recovery is acceptable for a large server installation, no one is willing to wait for their telephone or television to reboot. As with archival, recovery must be nearly instantaneous in an embedded product. Secondly, it is often the case that a system will be completely reinitialized, rather than simply rebooted. In this case, the embedded database must be restored to its initial state, freeing all its resources. This is not typically a requirement of large server systems.

System Level Architecture. In addition to the maintenance-free operation required of the embedded systems,there are a number of requirements that fall out of the constrained resources found in the systems using gizmo databases. These requirements are: small footprint, short code-path, programmatic interface for tight application coupling and to avoid the overhead (in both time and size) of interfaces such as SQL and ODBC, application configurability and flexibility, support for complete memory-resident operation (e.g., these systems must run on gizmos without file systems), and support for multi-threading.

A small footprint and short code-path are common requirements for embedded database (EDB). However, traditional interfaces such as ODBC and SQL add significant size overhead and frequently add multiple context/thread switches per operation. These also add several IPC calls.

The rest of the manuscript is organized as follows. The next section describes database transactions. Section 4 presents a model of the system. It also describes the data sharing problems and a solution based on embedded concurrency control. Section 5 considers a proof of correctness. Section 6 presents related research activities. Finally, section 7 presents summary and conclusions.

3 Database Transactions

We consider, an environment based on transaction classification. The transactions at the server end are considered to be short and these can be easily restarted on account of few failures. The critical client's transactions on the other hand are considered instant execution requests of highest (real-time) priority. The server is assumed to have a high capacity and receives a few cases of critical client update requests. In many cases, the transaction processing system can execute a critical client (cc) update, with little or no overheads. In the study, conflicts among two critical client transactions are separately discussed at the end for sake of simplicity. We demonstrate the ease of processing a long read (backup) transaction using the proposed model.

In order to preserve serializability, the conventional systems depend on 2 phase locking (2PL) protocol [2]. Whereas the 2PL protocol enforces a two phase disciple, the criteria of serializability does not dictate the order in which a collection of conflicting transactions need to execute [2]. This option provides an opportunity to make a modified system that follows 2 PL protocol at the TM's level, but can be flexible at the data manager's (DM's) level. It can permit a interference free and 'non-blocked' execution for critical host (CH) transactions. This change necessitates maintaining 'lock table' in the form of site level graphs. Although this is the first effort (to the best of our knowledge) to use the technique for embedded databases, many graph based techniques have been studied earlier by [7], [11], [12].

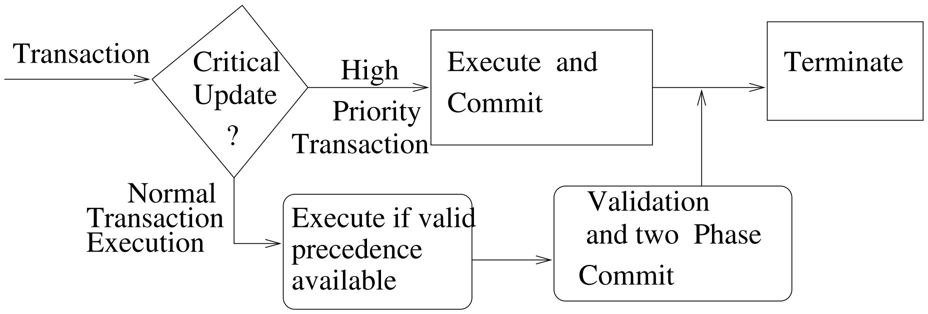
Transaction
Critical Update ?
High Priority Transaction
Execute and Commit
Terminate
Normal Transaction Execution
Execute if valid precedence available
Validation and two Phase Commit

Fig. 1. Execution of CH update transactions in isolation through embedded 2 phase locking based concurrency control

It is proposed to execute a critical host update (CHU) transaction in a special priority fashion. It may need to wait for another low-priority transaction, only if, that transaction has completed and local DM is participating in the second phase of a 2 phase commit.

The introduction of these possibilities integrates well with the existing transaction execution models. Earlier efforts at separating read-only transactions and update transactions exist [2]. The present study is an effort that proposes an implementation strategy for isolation of Serializable CHU transactions, for such an execution, that is free from interference by other transactions (Figure 1).

3.1 Transaction Execution

It is common for designers to extend the available approaches for concurrency control for use within the new system environments. However, we propose to study an analytical model and consider introduction of parallelism.

There have been some efforts at introducing parallelism within the concurrency control function. Earlier proposals attempt to eliminate interference between two classes of transactions. For example, processing Read-only transactions separately by using older versions of data, eliminate interference. Within the new classes, transactions are processed with no interference from each other's transactions. These can be considered to be executing in parallel. We propose to study the process of data allocation to executing transactions by using a stochastic process model. The model helps us in examining the parallel activity introduced by the use of classification of transactions. It also provides new insights that can lead to efficient processing of time-critical transactions. In the new environment, the time-critical transactions aim to execute with no interference from the ordinary transactions (Figure 1). In this light, the characteristics of the 2 Phase Locking based Concurrency Control scheme have been examined, within framework of a Real-Time (time-critical) database system.

4 The System Model

Based on the models of 2 phase locking and real-time computational environment with no slack time [9], a set of assumptions for executing transactions are organized. It is assumed that a 2 phase locking discipline is followed and the transaction execution is based on the criteria of serializability. Ideally, the CHU transactions should be able to do the following :

- a critical transaction may proceed without interference from other transactions.
- over ride conventional delays during execution
- integrate with existing modes of transaction executions. The two phases within the two phase locking (2PL) protocol must execute with no blocking;
- execute and commit, i.e., if phase 1 is completed, then phase 2 needs to complete.

In the following section, a scheme to execute transactions as per a precedence order is described.

4.1 Definitions : Embedded Database System

Embedded database system (EDS) consists of a set of data items (say set 'D'). The EDS is assumed to be based on a server that are occasionally accessed by critical hosts. The site supports a transaction manager (TM) and a data manager (DM). The TM supervises the execution of the transactions. The DMs manage individual databases. Each critical host supports a TM, that interacts with an EDB server. That performs other TM functions of interaction with other DMs. The network is assumed to detect failures, as and when these occur. When a site fails, it simply stops running and other sites detect this fact.

4.2 The Transaction Model

We define a transaction as a set of atomic operations on data items. The system contains a mixture of instant priority real-time transactions (CHU, or CH reads) and ordinary transactions. We assume that the ordinary transactions can be aborted, in case of a data conflict with the real-time transactions.

The use of real-time database systems is growing in many application areas such as, industrial process control systems, and other time-critical applications. Many approaches for implementation of Real-Time systems are being studied [10]. In the real-time system environment, a critical transaction (computational task) is characterized by its computation time and a completion deadline. These systems are characterized by stringent deadlines, and high reliability requirements.

4.3 Embedded Concurrency Control for Critical Data Operations

It is proposed to execute a CHT in isolation. It permits the CHT to proceed by locking data items. This step reduces the value of 'n' as the domain of locked items is confined to CHTs only. The norm for processing other transactions is based on an additional validation check, as per the criteria of serializability. For this purpose, each transaction is validated before commit.

The validation test for other transactions(OTs) uses the following criteria :

1. (normal) No data item, read by the transaction (OT), has been updated by a transaction after the read, that is -
   Read-set (OT) ∩ Write-set (more recently committed transactions) ; and
2. (additional) No data item read by the transaction (OT), is in the locked item list of executing CHTs, that is -
   Read-set (OT) ∩ Locked-items (CHTs) .

The transactions that fail to meet the first criteria are aborted, and restarted. The transactions that fail to meet the second criteria can be made to delay commit, so as to let the executing CHT complete its execution. The algorithm for performing, the validation check is given below (Figure 2.). The possibility of repeated rollback of an ordinary transaction can be eliminated (It can be submitted as a low priority CHT). It can be observed that, such an execution introduces a non-interference environment for a CHT.

The above test is strictly conflict based and OTs need not perform any locking and can be made completely dependent upon a validation test [4].

```
Procedure Validate (OT);
Valid := true;
For each X ∈ Read-set (OT) do
        if X ∈ write-set (Transactions committed after read by OT)
        then Valid := false, exit loop; end ;
        if X ∈ (Locked items list of CHTs )
        then wait for release and Valid := false, exit loop;
end;
If valid
then for each X ∈ write-set (T) do
< Allot a commit sequence number to T >
< Commit write-set (T) to database >

else restart (T);
end.
```

Fig. 2. Validation procedure for Embedded Concurrency Control (ECC)

Correctness Criteria. The critical host update transactions execute as per the criteria of serializability by virtue of the 2PL protocol [2]. As the CHTs completely ignore the presence of OTs, these transactions are executed as per the notion of optimistic concurrency control, with an enhanced validation check. The validation check ensures that an OT is Serializable with respect to,

1. the previously committed transactions, and
2. the executing critical host transactions.

Performance Considerations. A drawback associated with adoption of validation based approaches is the possibility of repeated rollbacks. However the proposed scheme can prevent these rollbacks by resubmitting a rejected trans-

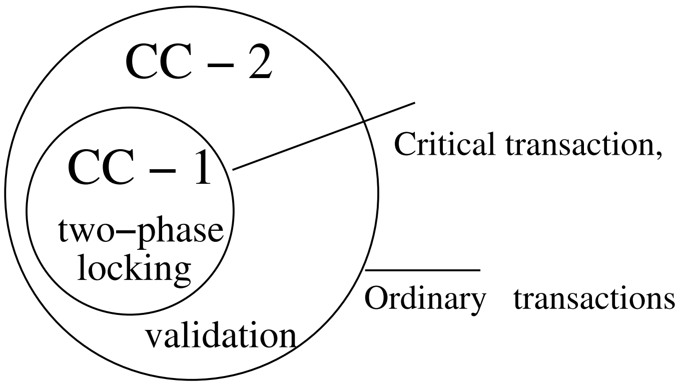

Fig. 3. Performance gain for Critical Transactions

action as a low priority CHT. This will make the OT execute as per the 2PL protocol and prevent the repeated rollbacks.

Although, in this approach the OTs face an enhanced level of validation, there are three positive aspects that are associated with our proposal. Firstly, the OTs avoid the problem of repeated rollbacks. Also, such a mixed mode of execution enhances the overall level of concurrency, because unlike locking based approaches the validation check is based on testing conflicts using the read-set of the committing transaction. Finally, the main item under focus concerning the is performance is the execution of CHT. Although the overall level of concurrency is expected to be improved, the performance of CHTs is enhanced as these execute with no interference with the other transactions within the system (Figure 3). A performance evaluation study has been presented in [5].

4.4 Incremental Corrections to Global Read Contents

We propose an algorithm based on asynchronous computing ([4]). This algorithm has two stages. In the first stage, the log of the transactions is recorded during global-read. In the next stage, the backup copy is corrected by using the log of the transactions in off-line mode. The entity conditions and the normal transactions work the same as the original incremental global-read algorithm [1]. In this proposal, there are no rejected transactions.

In the proposed algorithm, a global read transaction is started. It locks a small part of the database as it proceeds. The database items that are read by the global read are colored as black. Other transactions that update data in the database are colored based on the items accessed by them. Transactions that read black data items are colored as black. Other transactions are colored as white or grey (mixed read-set). These transaction update database. The database items updated by the colored transactions are colored as grey. At the time of commit, if an entity's color is black and is updated by a black or gray transaction, then its contents are noted by using the log of the transactions. Later, the copied version of database (inconsistent version read by global read copy) is corrected by using the log (in off-line mode) (Figure 4.). This proposal creates a complete backup of the database which is consistent with the time, when the global-read is completed. This proposal does not need more storage, as a small log can be maintained in the main memory. Also, if a separate system recovery log is prepared for recovery by the system. By combining the backup log with system recovery log, no additional storage is needed.

The Algorithm - Transformation of Database States. As shown in Figure 4, phase 1, generates a modified log during the execution of a global-read transaction. This log (called the **color log**) contains a color marking for each update transaction. On completion of the global-read, the data read by the global-read contains an inconsistent version of the database. In phase 2, modifications are applied to make the data consistent, as shown in figure 4. An algorithm to generate the color log and for later generation of a consistent database version is described in this section. The various aspects of the proposed scheme are discussed below.

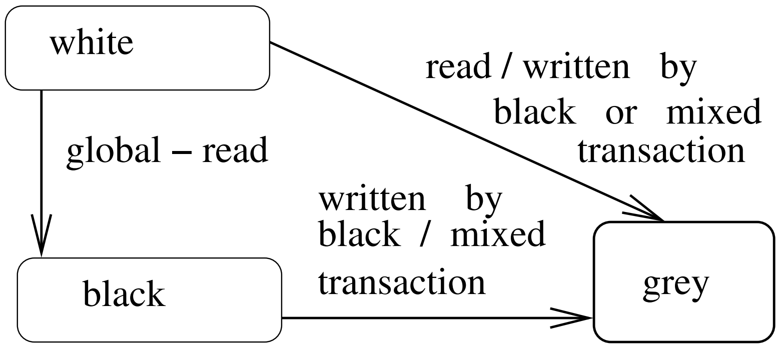
white
read / written by black or mixed transaction
global – read
written by black / mixed transaction
black
grey

Fig. 4. State transition diagram for data entities

Global-Read Processing. The global-read transaction (T_{gr}) divides the database entities into black and white colors. At the beginning, all database entities are white. Gradually, entities read by a T_{gr} are colored black. In addition, all entities that are written by black or mixed transactions are colored as gray. White data entitity that are read by black or mixed transaction are also colored as gray. Thus all data entities are colored as white, or black or gray. Normal update transactions are colored white or black depending on the color of data entities being updated by them. A transaction is termed as mixed (gray color), based on the following conditions,

– a transaction that updates a mixture of black and white data entities;
– a transaction that reads a data entity that is colored as gray; and
– a transaction that writes on a data entity that is colored gray.

The color of a transaction can be determined at the time of its commit by examining the color of data items in its, read-set and write-set. Normal read-only transactions are not colored and proceed normally subject to the constraints of the two-phase locking protocol.

5 Proof of Correctness

While the earlier proposals avoid inconsistency by not allowing certain transactions to commit, our proposals permit a normal execution activity during the execution of the global-read transaction. All such updates that could have been missed partially or fully, are rewritten on the database copy, during phase 2. These updates by gray or black transactions, can generate inconsistency. Concurrent updates by white transactions are read by the global-read transaction, during global-read.

6 Related Work

Occasionally, leading researchers in the database community convene to identify future directions in database research. The most recent of discussion is the

1998 Asilomar report. It identifies the embedded database as one of the important research areas in database research [3]. Also, market analysts identify the embedded database market as a high-growth area in the commercial sector as well [8].

The Asilomar report identifies a new class of database applications, which they term "gizmo" databases, small databases embedded in tiny mobile appliances, e.g., smart-cards, telephones, personal digital assistants. Such databases must be self-managing, secure and reliable. Thus, the idea is that gizmo databases require plug and play data management with no database administrator (DBA), no human settable parameters, and the ability to adapt to changing conditions. More specifically, the Asilomar authors claim that the goal is self-tuning, including defining the physical DB design, the logical DB design, and automatic reports and utilities [3]. To date, few researchers have accepted this challenge, and there few research studies on the subject [13].

7 Summary and Conclusions

In a embedded database system, transactions need to backup data and perform updates. A possibility is demonstrated by considering the critical client, as a host issuing update (or read) transactions. This class of transactions can be executed as an instant priority real-time transaction with no slack time available. By adopting transaction classification, many changes can be accommodated within the conventional locking at a low cost that enable the database updates by critical clients.

References

1. P. Amann, Sushil Jajodia, and Padmaja Mavuluri, "On-The-Fly Reading of Entire Databases", *IEEE Transactions of Knowledge and Data Engineering*, Vol. 7, No. 5, October 1995, pp. 834-838.
2. Bernstein P.A., V.Hadzilacos and N.Goodman, *Concurrency control and recovery in database systems*, Addison-Wesley, 1987.
3. Bernstein, P., Brodie, M., Ceri, S., DeWitt, D., Franklin, M., Garcia-Molina,H., Gray, J., Held, J., Hellerstein, J., Jagadish, H., Lesk, M., Maier, D., Naughton, J., Pirahesh, H., Stonebraker, M., Ullman, J., "The Asilomar Reporton Database Research", *SIGMOD Record*, Vol. 27, No. 4,pp. 74-80, 1998.
4. Bhalla S., "Improving Parallelism in Asynchronous Reading of an Entire Database", *Proceedings of 7th High Performance Computing (HiPC 2000) conference*, Banglore, December 2000, published by LNCS vol. 1970.
5. Bhalla S., "Asynchronous Transaction Processing for Updates With no Wait-for State", *Proceedings of 9th High Performance Computing (HiPC 2002) conference*, Banglore, December 2002, published by LNCS vol. 2552.
6. Chaudhuri, S., Narasayya, V., "An Efficient, Cost-Driver Index Selection Tool for Microsoft SQL Server," *Proceedings of the 23rd VLDB Conference*, Athens, Greece, 1997.
7. Eich, M.H., and S.H.Garard, "The performance of flow graph locking", *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 477-483, April 1990.

8. Hostetler, M., "Cover Is Off A New Type of Database," *Embedded DB News*, http://www.theadvisors.com/embeddeddbnews.htm, 5/6/98.
9. Korth H.F., E. Levy, and A. Silberschatz, "Compensating Transactions: a New Recovery Paradigm," *Proceedings of 16th International Conference on Very Large Databases (VLDB)*, Brisbane, Australia, 1990, pp. 95-106.
10. Ramamritham K., "Real-Time Databases", *Distributed and Parallel Databases*, Kluwer Academic Publishers, Boston, USA, Vol. 1, No. 1, 1993.
11. Reddy P. Krishna, and S. Bhalla, "A Nonblocking Transaction Data Flow Graph Based Protocol For Replicated Databases", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No. 5, October 1995.
12. Reddy P. Krishna, and S. Bhalla, "Asynchronous Operations in Distributed Concurrency Control", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 15, No. 3, May 2003.
13. Seltzer, M., and M. Olson, "Challenges in Embedded Database System Administration", May 2005, http://www.sleepycat.com/docs/ref/refs/embedded.html