REAL-TIME DATABASE SYSTEMS
Architecture and Techniques
The Kluwer International Series in Engineering and Computer Science

REAL-TIME SYSTEMS

Consulting Editor: John A. Stankovic, University of Virginia

Other books in this series:


Contents

List of Figures ix
List of Tables xi
Acknowledgments xiii
Preface xv
Contributing Authors xix

I OVERVIEW, MISCONCEPTIONS AND ISSUES

   Tei-Wei Kuo and Kam-Yiu Lam

2. Misconceptions About Real-Time Databases 9
   John A. Stankovic, Sang H. Son and Jörgen Hansson

3. Applications and System Characteristics 17
   Doug Locke

II REAL-TIME CONCURRENCY CONTROL

4. Conservative and Optimistic Protocols 29
   Tei-Wei Kuo and Kam-Yiu Lam

5. Semantics-Based Concurrency Control 45
   Tei-Wei Kuo

6. Real-Time Index Concurrency Control 59
   Jayant R. Haritsa and S. Seshadri

III RUN-TIME SYSTEM MANAGEMENT

7. Buffer Management in Real-Time Active Database Systems 77
   Anindya Datta and Sarit Mukherjee
8. Disk Scheduling
Ben Kao and Reynold Cheng

9. System Failure and Recovery
Rajendran M. Sivasankaran, Krithi Ramamritham and John A. Stankovic

10. Overload Management in RTDBs
Jörgen Hansson and Sang H. Son

11. Secure Real-Time Transaction Processing
Jayant R. Haritsa and Binto George

IV ACTIVE ISSUES AND TRIGGERING

12. System Framework of ARTDBs
Jörgen Hansson and Sten F. Andler

13. Reactive Mechanisms
Jonas Mellin, Joakim Eriksson and Sten F. Andler

14. Updates and View Maintenance
Ben Kao, K. Y. Lam and Brad Adelberg

V DISTRIBUTED REAL-TIME DATABASE SYSTEMS

15. Distributed Concurrency Control
Özgür Ulusoy

16. Data Replication and Availability
Özgür Ulusoy

17. Real-Time Commit Processing
Jayant R. Haritsa, Krithi Ramamritham and Ramesh Gupta

18. Mobile Distributed Real-Time Database Systems
Kam-Yiu Lam and Tei-Wei Kuo
VI PROTOTYPES AND FUTURE DIRECTIONS

19. Prototypes: Programmed Stock Trading
   Brad Adelberg and Ben Kao
   261

20. Future Directions
    Tei-Wei Kuo and Kam-Yiu Lam
    279

Index
   287
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Value functions of different types of real-time transactions</td>
<td>4</td>
</tr>
<tr>
<td>4.1</td>
<td>A RWPCP Schedule</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>A 2VPCP Schedule</td>
<td>35</td>
</tr>
<tr>
<td>5.1</td>
<td>Similarity of database states</td>
<td>48</td>
</tr>
<tr>
<td>5.2</td>
<td>The temporal depth of the preemption stack</td>
<td>50</td>
</tr>
<tr>
<td>5.3</td>
<td>R-recency and W-recency: time span of similar reads and writes</td>
<td>51</td>
</tr>
<tr>
<td>5.4</td>
<td>A SSP schedule</td>
<td>52</td>
</tr>
<tr>
<td>7.1</td>
<td>A triggering graph</td>
<td>80</td>
</tr>
<tr>
<td>7.2</td>
<td>Buffer Model</td>
<td>82</td>
</tr>
<tr>
<td>8.1</td>
<td>I/O Request Model</td>
<td>99</td>
</tr>
<tr>
<td>8.2</td>
<td>Disk Head Scheduling Example</td>
<td>101</td>
</tr>
<tr>
<td>9.1</td>
<td>Taxonomy</td>
<td>115</td>
</tr>
<tr>
<td>10.1</td>
<td>Scheduling architecture showing the information flow and component interactions</td>
<td>129</td>
</tr>
<tr>
<td>10.2</td>
<td>EDF schedule after overload resolution</td>
<td>136</td>
</tr>
<tr>
<td>11.1</td>
<td>Bell-LaPadula access restrictions</td>
<td>143</td>
</tr>
<tr>
<td>11.2</td>
<td>Dual approach</td>
<td>149</td>
</tr>
<tr>
<td>11.3</td>
<td>MCC architecture</td>
<td>150</td>
</tr>
<tr>
<td>13.1</td>
<td>Event monitoring process</td>
<td>171</td>
</tr>
<tr>
<td>13.2</td>
<td>Directed acyclic graphs of $E_1, E_2$ in event monitoring</td>
<td>172</td>
</tr>
<tr>
<td>13.3</td>
<td>ECA rule extended with temporal attributes</td>
<td>176</td>
</tr>
<tr>
<td>13.4</td>
<td>Response time in a real-time system using ECA paradigm</td>
<td>180</td>
</tr>
<tr>
<td>14.1</td>
<td>A Real Time Database System</td>
<td>186</td>
</tr>
<tr>
<td>14.2</td>
<td>Maximum Age (MA) and Unapplied Update (UU)</td>
<td>189</td>
</tr>
<tr>
<td>14.3</td>
<td>Inter-arrival distribution of the stock price updates of G.E. on 94/1/3</td>
<td>191</td>
</tr>
<tr>
<td>14.4</td>
<td>An Example for ACS and RCS</td>
<td>194</td>
</tr>
<tr>
<td>18.1</td>
<td>System Architecture of a MDRTDBS</td>
<td>246</td>
</tr>
<tr>
<td>18.2</td>
<td>Transaction Architecture under the Transaction Shipping Approach</td>
<td>253</td>
</tr>
<tr>
<td>18.3</td>
<td>Data Pre-fetching Mechanism</td>
<td>256</td>
</tr>
<tr>
<td>19.1</td>
<td>Standard rule to maintain comp_prices for stock price changes</td>
<td>267</td>
</tr>
<tr>
<td>19.2</td>
<td>Stock composite indices example</td>
<td>268</td>
</tr>
<tr>
<td>19.3</td>
<td>Triggered transactions for composite example</td>
<td>269</td>
</tr>
</tbody>
</table>
19.4 Unique transaction approach to maintain \texttt{comp\_prices}
for stock price changes 270
19.5 Unique transaction approach (on \texttt{comp}) to maintain \texttt{comp\_prices}
for stock price changes 212
19.6 Standard rule to maintain \texttt{option\_prices} for stock
price changes 273
19.7 Process architecture 275
19.8 The flow of tasks within STRIP 275
List of Tables

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Aircraft Mission Computer Data Characteristics</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Spacecraft Onboard Computer Data Characteristics</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>Air Traffic Control Data Characteristics</td>
<td>22</td>
</tr>
<tr>
<td>3.4</td>
<td>Industrial Control Data Characteristics</td>
<td>24</td>
</tr>
<tr>
<td>3.5</td>
<td>Command &amp; Control Data Characteristics</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>The lock compatibility matrix of 2VPCP</td>
<td>34</td>
</tr>
<tr>
<td>6.1</td>
<td>Index Node Lock Compatibility Table</td>
<td>61</td>
</tr>
<tr>
<td>10.1</td>
<td>Characteristics of transaction workload</td>
<td>135</td>
</tr>
<tr>
<td>10.2</td>
<td>Overload resolution actions</td>
<td>135</td>
</tr>
<tr>
<td>13.1</td>
<td>Event history example</td>
<td>173</td>
</tr>
</tbody>
</table>
This page intentionally left blank.
Acknowledgments

Dedicated to my wife I-Ju Chen and my parents Cheng-I Kuo and Chin-Yun Hsu for their love and support.

Tei-Wei Kuo

We would like to thank Prof. Stankovic for his valuable comments on the organization and the contents of this book. We also want to express our gratitude to the Distributed Real-Time Systems (DRTS) Research Group of the University of Skövde and the Distributed Real-Time Multimedia Databases of the University of Virginia for providing us valuable information regarding their real-time database prototypes which are included in the chapter for future directions. Special thanks go to all of the contributing authors, particularly Prof. Krithi Ramamritham and Prof. Sten Andler, for their support and valuable advice during the development of the book. At last, we would like to thank Mr. Reynold Cheng for helping us in integrating and formatting all of the chapters.

Kam-Yiu Lam and Tei-Wei Kuo
This page intentionally left blank.
Preface

In recent years, a lot of research work has been devoted to the design of database systems for real-time applications. A real-time database system (RTDBS) is usually defined as a database system where transactions are associated with deadlines on their completion times. In addition, some of the data items in a real-time database are associated with temporal constraints on their validity. Example applications include systems for stock trading, navigation, telephone management, and computer integrated manufacturing. In order to commit a real-time transaction, the transaction has to be completed before its deadline, and all of its accessed data items must be valid up to its commit time. Otherwise, the usefulness of the results will be seriously affected. For many cases, any deadline violation of a hard real-time transaction may result in disasters.

There are two major technology components in building a RTDBS: real-time systems and database systems. Unfortunately, these two technologies might have conflicting goals. For example, a database system is often designed to maximize the system throughput, while real-time techniques usually aim at meeting deadlines and improving the predictability of the system performance. Frequently, when conventional database technologies are applied to a RTDBS, many real-time requirements of the systems might be seriously affected and cannot be guaranteed. In the past decade, a lot of research effort has been devoted to the study of RTDBS, and a large number of research papers were published in this area [9]. Important conferences and special issues of international journals were organized and devoted to real-time database systems, such as the 1996 and 1997 International Workshop on Real-time Database Systems [2, 3], the 1995, 1997, and 1999 International Workshop on Active and Real-Time Database Systems [4, 5, 6], Information Systems [7], Real-Time Systems Journal [11], ACM SIGMOD Records [14, 15], IEEE Transactions on Knowledge and Data Engineering [8], and Journal of Systems and Software [10]. A recent issue of the Real-Time Systems Journal was also devoted to RTDBS [12].

Early study in RTDBS was focused on the design of efficient transaction scheduling algorithms and concurrency control protocols [1]. The goals are to satisfy the time constraints of transactions and the temporal properties of data items [13]. Other design issues essential to the performance of RTDBS are buffer management, I/O scheduling, etc. Although a large number of research papers have been published, there is a lack of a comprehensive system reference book which covers the important technologies in this area. Furthermore, there still exists many misunderstandings about this area, e.g., the relationship with active database systems and the performance issues of a RTDBS. It is the purpose of this book to provide an in-depth review on the current technologies
and algorithms in the design of RTDBS. Its target readers are graduate students, researchers, and practitioners in real-time systems and database systems.

The book is divided into six parts, which covers all the important topics in the area. Part I discusses the important basic concepts of RTDBS and the characteristics of their potential applications. Part II covers one of the most important issues in RTDBS: real-time concurrency control, which is essential in maintaining database consistency and providing a consistent view to transactions. Part III is on other run-time system management issues, such as buffer management, recovery and system failure handling, disk scheduling, security, and overload management. An important characteristic of many real-time database applications is the active property, which is important for generating timely responses to critical events in the external environment. The active issues in RTDBS are covered in Part IV. Since many RTDBS, such as traffic management systems and tele-communication management systems, are often distributed in nature, Part V addresses important issues related to the processing of transactions in a distributed real-time database system. They are commitment, distributed concurrency control, and replicated data management. The last part of the book, Part VI, describes a RTDB prototype and the future directions for RTDB research.

Kam-Yiu Lam and Tei-Wei Kuo

References


This page intentionally left blank.
Contributing Authors

Brad Adelberg  
Department of Computer Science, Northwestern University, Evanston, Illinois, USA

Sten F. Andler  
Department of Computer Science, University of Skövde, Skövde, Sweden

Reynold Cheng  
Department of Computer Science and Information Systems, University of Hong Kong, Hong Kong

Anindya Datta  
Centre for Electronic Commerce, Georgia Institute of Technology, Atlanta, Georgia, USA

Joakim Eriksson  
Department of Computer Science, University of Skövde, Skövde, Sweden

Binto George  
Department of Computer Science, Western Illinois University, Macomb, Illinois, USA.

Ramesh Gupta  
Goldencom Technologies, Inc., Fremont, California, USA.

Jörgen Hansson  
Department of Computer and Information Science, Linköping University, Linköping, Sweden

Jayant R. Haritsa  
Database Systems Lab, Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, India.

Ben Kao  
Department of Computer Science and Information Systems, University of Hong Kong, Hong Kong

Tei-Wei Kuo  
Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan
Kam-Yiu Lam
Department of Computer Science, City University of Hong Kong, Hong Kong

Doug Locke
TimeSys Corporation, USA

Jonas Mellin
Department of Computer Science, University of Skövde, Skövde, Sweden

Sarit Mukherjee
Panasonic Labs, Princeton, New Jersey, USA

Krithi Ramamritham
Department of Computer Science, University of Massachusetts, Amherst, Massachusetts, USA.

S. Seshadri
Lucent Bell Labs, Murray Hill, New Jersey, USA.

Rajendran M. Sivasankaran
GTE Laboratories, Waltham Massachusetts, USA.

Sang H. Son
Department of Computer Science, University of Virginia, Charlottesville, Virginia, USA

John A. Stankovic
Department of Computer Science, University of Virginia, Charlottesville, Virginia, USA

Özgür Ulusoy
Department of Computer Engineering, Bilkent University, Ankara, Turkey
PART I
OVERVIEW, MISCONCEPTIONS, AND ISSUES
Chapter 1

REAL-TIME DATABASE SYSTEMS: AN OVERVIEW OF SYSTEM CHARACTERISTICS AND ISSUES

Tei-Wei Kuo
Department of Computer Science and Information Engineering
National Taiwan University
ktw@csie.ntu.edu.tw

Kam-Yiu Lam
Department of Computer Science
City University of Hong Kong, Kowloon, Hong Kong
cskylam@cityu.edu.hk

1. INTRODUCTION

In the past two decades, the research in real-time database systems (RTDBS) has received a lot of attention [7, 8, 9, 10]. It consists of two different important areas in computer science: real-time systems and database systems. Similar to conventional real-time systems, transactions in RTDBS are usually associated with time constraints, e.g., deadlines. On the other hand, RTDBS must maintain a database for useful information, support the manipulation of the database, and process transactions. Some example applications of RTDBS are integrated manufacturing systems, programmed stock trading systems, air traffic control systems, and network management systems. Typically, these application systems need predictable response time, and they often have to process various kinds of queried in a timely fashion.

Differently from traditional database systems, RTDBS must not only maintain database integrity but also meet the urgency of transaction executions. Different performance metrics are adopted. Real-time transactions are, in general, classified into three types: No hard real-time transaction should have its deadline missed, and its deadline must be guaranteed by the system. On the other hand, any deadline violations of soft real-time transactions may only
result in the performance degradation of the system. The major performance metrics for soft real-time transactions are the number or percentage of deadline violations or their average or worst-case response time. Firm real-time transactions are a special kind of soft real-time transactions, except that firm real-time transactions will be killed when their deadlines expire. The major performance metric is the number or percentage of deadline violations.

The completion of a real-time transaction might contribute a value to the system. The relationship between the value of a real-time transaction and its completion time can be considered as a value function of time, as shown in Figure 1.1. After a soft real-time transaction misses its deadline, its value might decrease with time. A firm real-time transaction loses its value after its deadline expires. When a hard real-time transaction misses its deadline, its value becomes (possibly infinitely) negative. It means that a catastrophe might occur.

![Figure 1.1 Value functions of different types of real-time transactions](image)

It is possible that a system consists of a single type or multiple types of real-time transactions, or even mixed with non-real-time transactions. Here non-real-time transactions are referred to the traditional database transactions. When a transaction is requested periodically, such as the polling of the stock index, the transaction is called a periodic transaction; otherwise, it is a sporadic transaction.
2. EXTERNAL AND TEMPORAL CONSISTENCY

RTDBS differ from conventional database systems in that they often must deal with an external environment that imposes new relations between the data objects in the database and the dynamic real-world objects they model. For example, in an intelligent transportation system, traffic information must be recorded in the database in a timely manner, and at the same time, car drivers and travelers may inquire the most recent trip tips or traffic conditions.

Due to the relationship with the external environment, several new consistency constraints, beside the internal consistency constraints in traditional databases, were introduced [1, 2, 6]: An external consistency constraint is concerned with the absolute recency of data. An example of this is that the information about the stock index in the database of a stock trading system must be kept sufficiently up-to-date. A temporal consistency constraint is concerned with the relative recency of data. An example of this is that the values of two data objects, such as the coordinate values \(x\) and \(y\) of a vehicle, read by a transaction must be sufficiently correlated in time. Note that temporal consistency is different from external consistency in that the former deals with the relative recency of data, while the latter deals with the absolute recency of data. While external consistency may sometimes imply temporal consistency, there is, in general, no implication of one from the other.

Since the values of data items in many real-time databases are often highly dynamic, due to the rapidly changing conditions in the external environment. The validity of data values usually degrades rapidly with time. Efficient update methods are very important to maintain the temporal validity of data values. Although the recency of data values can be modeled by their ages, there is, in general, no uniform way to derive or reason the ages of data values. It is mainly caused by the difficulty in defining the age of data values derived by other pieces of data. Whether the data age of an data object is up-to-date may depend on two or more timing constraints. For example, suppose the value of a data object \(x\) depends on a data object \(y\), the update transaction of \(x\) never misses its deadline, but the update transaction of \(y\) often misses its deadline. Note that we cannot say that the \(x\) value is up-to-date simply because the transaction updating \(x\) is always timely.

3. DESIGN ISSUES

Obviously, RTDBS must try to satisfy transaction deadlines. In the past decades, various scheduling algorithms have been proposed to schedule transactions in a real-time fashion. A lot of study on RTDBS adopts the well-known rate monotonic algorithm or the earliest deadline first algorithm. The rate monotonic algorithm (RM) [5] is an optimal fixed-priority assignment scheme
for periodic transactions, where RM assigns transactions priorities inversely proportional to their periods. Such algorithm is very suitable to RTDBS consisting of only periodic hard real-time transactions. The *earliest deadline first algorithm* (EDF) is an optimal dynamic-priority scheduling algorithm, where EDF assigns a higher priority to any transaction which has a closer deadline. EDF fits any workload of periodic or non-periodic transactions. It is used for the scheduling of hard, soft, and firm real-time transactions. Besides RM and EDF, many real-time transaction scheduling algorithms derive transaction priorities based on the transaction value or criticality [4]: Transaction value is often described as a function $f(t)$ of the transaction completion time $t$ which can be a step function or any reasonable one. For example, for a hard real-time transaction $\tau$, let its value function $f(t) = v$, if $t$ is no larger than its deadline, where $v$ is the value of $\tau$ to the system. If $\tau$ is ready at time $t$ then $f(t) = 0$, for any $t$ less than the ready time, which means that nothing is gained before time $t$. If $t$ is over its deadline, $f(t)$ is an infinitely negative number, which means that the system value will become infinitely negative, as shown in Figure 1.1.

One of the most important issues in RTDBS is real-time concurrency control. In the past decades, various real-time concurrency control protocols had been proposed to resolve data conflicts among transactions. The system must not only satisfy the ACID properties of real-time transactions but also meet their deadlines. Real-time concurrency control protocols can be classified into conservative and optimistic protocols. Conservative protocols often rely on locks to avoid any possible violation of transaction serializability. Optimistic protocols usually allow transactions to access the database in any way they like. At the validation time, the system decides whether the commitment of a transaction may violate serializability. If any serializability violation may occur, the transaction is aborted. Other important system issues are recovery, buffer management, and real-time disk scheduling. It is highly essential to eliminate or minimize any unpredictable factors in system design.

Memory-resident databases are often mentioned in the implementation of real-time databases, especially when a system must have a highly predictable behavior in data retrieval and storing, e.g., those with sensors in many typical industrial applications. The implementation of real-time databases on main memory and with real memory addressing, in deed, avoids the unpredictability of disk access time and significantly improves the system performance. Nevertheless, it increases the vulnerability of the system because many system failures may easily wipe out the entire database residing on the main memory. Extra overheads must be paid in taking care of system recovery and logging. Methods, such as RAM-disks or server-based mechanisms, can be used to alleviate such overheads in normal system operation, due to logging and frequent system checkpointing [3, 11]. We must also emphasize that techniques developed for disk-resident databases cannot be directly applied to memory-
resident databases because of different implementation concerns. For example, in
disk-resident databases, a shallow and large-fanout B-tree index structure is
considered good for system performance because of the disparity of CPU and
disk speeds. However, such a B-tree index structure may not be welcome to
memory-resident databases.

How to respond to the occurrences of critical events in the external envi-
ronment in a timely fashion is another important issue in RTDBS. Researchers
have proposed to adopt "active" features, such as the triggering of transactions,
in RTDBS. The scheduling of updates must be emphasized to maintain the
temporal validity of data objects in the system. Recently, researchers also
started exploring the design of distributed real-time database systems (DRT-
DBS) and mobile real-time database systems (MRTDBS). Supporting real-time
processing of transactions over a distributed environment or a mobile network
is not easy. Most of the existing work on DRTDBS and MRTDBS is mainly
focused on the systems with soft and firm real-time transactions. Three of the
most important issues are distributed concurrency control, data replication, and
transaction commitment.

4. SUMMARY

In this edited book, we divide all important design issues of RTDBS into six
parts. Part I is focused on the basic concepts of RTDBS and the clarification their
mis-concepts. Part II discussed one of the most important issues in the design
of RTDBS: concurrency control. Other issues related to the management of the
systems are included in Part III. Part IV addresses real-time active issues, such
as triggered transactions. Part V reports the research findings in distributed and
mobile real-time database systems. Part VI is for prototype and future work.

References

Ph. D. Thesis, Department of Computer Sciences, The University of Texas
at Austin, August, 1994.

Databases with Consistency Constraints,” in Proceedings of the 16th VLDB
Conference, Brisbane, August 1990.

Concurrency Control and Recovery,” in Proceedings of the Third IEEE In-
ternational High Assurance Systems Engineering Symposium, Washington,
DC, November 1998.


Chapter 2

MISCONCEPTIONS ABOUT REAL-TIME DATABASES

John A. Stankovic and Sang H. Son
Department of Computer Science
University of Virginia, Charlottesville, USA

Jörgen Hansson
Department of Computer and Information Science
Linköping University, Linköping, Sweden

1. INTRODUCTION

Very often the key differences between general purpose computing and real-time computing are misunderstood. Two papers have appeared in the literature that focus on these misconceptions: one of these papers dealt with real-time computing in general [9], and the other dealt with real-time databases [10]. The impact of the first paper [9] was significant in that it spurred a lot of research that specifically focused on real-time issues. We believe that there is now a significant body of scientific and technological results in real-time computing, in part, due to the careful definition of real-time computing and the articulation of the important differences between real-time computing and general purpose computing found in that paper.

Today, the level of data sophistication employed in many real-time systems is growing, ranging from sensor data and various derived data typically found in the more classical real-time systems, to large, even global database management systems that must support real-time characteristics (such as the air traffic control system). Unfortunately, many misconceptions have arisen with respect to the real-time aspects of databases. The state of confusion seems to be similar to what existed in the 1980s with respect to the difference between real-time computing and general purpose computing; except now it is with respect to databases. Consequently, in this chapter we will carefully define real-time databases and state and dispel the most common misconceptions of real-time
In the following discussion we are addressing two distinct audiences. The first is the mainstream database researchers and users who usually do not have any experience with real-time issues and often feel that it is acceptable to simply use a commercial database management system and make it go fast! The second audience is the real-time system community who, in the past, have dealt with real-time data from sensors and data derived from this data. Usually all this real-time data was in main memory, and when higher level (non-real-time) data was required it was probably placed in a totally separate database system, outside the real-time data purview.

2. DEFINITION

A real-time database system is a database system where timing constraints are associated with transactions and data have specific time intervals for which the data is valid [7, 2]. The transaction timing constraints can be deadlines to complete by, times by which they must start, periodic invocations, deadlines, etc. It is not necessary that every transaction has a timing constraint, only that some transactions do. In addition to transaction timing requirements, data has time semantics as well. Data such as sensor data, stock market prices, locations of moving objects, etc. all have semantics that indicate that the recorded values are only valid for a certain interval of time. A real-time database makes this validity interval explicit, i.e., it is part of the database schemas. Transaction correctness is then defined as meeting its timing constraints and using data that satisfies absolute and relative timing consistency. Absolute time consistency means that data used by a transaction is still temporally valid (i.e., within its time validity interval). Logically this means that the value of this data item reflects the true state of the world to an acceptable degree of accuracy. Relative time consistency means that when a transaction uses multiple data items with a relative consistency constraint between them that the times at which those items were updated (sensed) are within the specified time interval. Logically, it means that the states they represent are within an acceptable range of each other.

Note also that a system which simply has data with real-time requirements, such as sensor data, does not constitute a real-time database system. Since a real-time database system is a database system, it has queries, schemas, transactions, commit protocols, concurrency control support, storage management, etc.
3. MISCONCEPTIONS

To better understand what real-time databases are and how important they are, 9 common misconceptions are enumerated and discussed. The first three misconceptions are variations of the misconstrued theme that real-time relates to speed. However, it is instructive to separate this general theme into these three related aspects.

**Advances in hardware will take care of the real-time database requirements.**

Technology will exploit faster and parallel processors to improve system throughput, but this does not mean that the timing constraints will be automatically met. In fact, with the increased size and complexity of the database and hardware it will become more and more difficult to meet and to show that the timing constraints will be met. Hardware alone cannot ensure that transactions will be scheduled properly to meet their deadlines, nor ensure that the data being used is temporally valid. For example, if a transaction more quickly uses out of date data, it is still an incorrect transaction. Or, transactions that run faster may still block in such a pattern that they have to be aborted and restarted. This delay may be longer than the associated speed up and cause missed deadlines.

**Advances in database technology will take care of real-time database requirements.**

Sometimes database designers claim that better database buffering, faster commit protocols, and novel query processing techniques will speed up databases to permit them to be used in real-time systems. While these techniques will help, they will not ensure that deadlines are met nor support the requirement that transactions only use valid (in time) data. The advances in database technology that will be required include time cognizant protocols for concurrency control, committing, transaction scheduling, and logging and recovery. There now exists ample evidence that such protocols are considerably better at supporting real-time transaction and data correctness than standard database protocols which simply go fast [1, 5, 8, 12]. Similar evidence exists when comparing general purpose timesharing systems to real-time systems: time cognizant algorithms perform better than those that just go fast when deadlines and other time constraints are important!

**Real-time computing is equivalent to fast computing.**

The objective of fast computing is to minimize the average response time of a set of transactions. However, the objectives of real-time databases are to meet the timing constraints and the data timing validity of individual transactions...
and to keep the database time valid via proper update rates. Again, to do this it is necessary to have time cognizant protocols. A simple, contrived example can illustrate this fact. Consider two transactions, A and B. Transaction A starts at time 1 and uses data item X and has a deadline at time 20 and an execution time of 10. Assume that transaction A begins executing at time 1 and locks X at time 2. At time 3 transaction B is invoked and it uses X, has an execution time of 2 and a deadline at time 6. Standard database protocols would have transaction B block (or possibly not even begin executing). Transaction A would then complete and release the lock on X at time 11. However, this is too late for transaction B and B misses its deadline. A time cognizant set of protocols would preempt transaction A at time 3 (because transaction B’s deadline is earlier than transaction A) and when B attempts to obtain X, one time cognizant solution (there are others) would be to abort transaction A, let transaction B finish (at time 5) and then restart transaction A which would also complete by its deadline. In this case both transactions complete correctly and on-time.

There is no need for a real-time database because we can solve all the problems with current databases.

This is a tricky issue. For example, using a current database system, it is possible to define a field for every relation (object) that contains the validity interval of that data. Then the transaction itself can check these fields to ensure absolute and relative validity. Further, the system can be modified to run some form of earliest deadline scheduling by controlling the priority of each transaction. However, this means that every transaction must program this capability itself instead of having the system support the capability. In fact, by including real-time scheduling in the system, the designers are moving towards a real-time database system. The problem is that if the transactions have timing constraints and data validity timing constraints then it is more efficient to build this support into the system rather than trying to cajole, manipulate, force fit a current typical database system into this needed set of capabilities. Further, if you actually do this force fitting, then you now have a real-time database system (but it will not likely be as efficient as if you developed these capabilities from the ground up). After all, all algorithms are programmable on a Turing machine but few people would advocate using a Turing machine to build real systems.

A real-time database must reside totally in main memory.

The primary reason for placing data in main-memory is to avoid delays introduced by disks, e.g., unpredictable seek and rotational delays. In most systems, I/O requests are scheduled in order to minimize average response time, maximize throughput or maintain fairness. Typical disk scheduling al-
algorithms for this type of disk scheduling are First-Come-First-Served (FCFS), Shortest-Seek-Time-First (SSTF) and the elevator algorithm SCAN. Typically, a database transaction performs a sequence of database read operations, computations, and then writes the data back to the database. However, since the deadline and the importance of the transaction are not considered when disk requests are scheduled, the timeliness of the transaction is jeopardized. In the same way traditional CPU scheduling algorithms, attempting to minimize response time or maximize throughput, have been shown to be inappropriate for real-time systems, the use of non-time-cognizant disk scheduling algorithms are not appropriate for scheduling disk requests. It has been shown [3] that disk scheduling algorithms that combine a scan and deadline requirement works considerably better than conventional algorithms.

Using a conventional database system and placing the database in main memory is sufficient.

It is sometimes argued that placing a conventional database in main-memory is a viable approach in order to gain performance and thereby make them suitable for real-time systems. Although it is true that main-memory resident databases eliminate disk delays, conventional databases still have many additional sources of unpredictability (such as delays due to blocking on locks, transaction scheduling, stolen processing time to handle external interrupts, etc.) that prevent time constraints of transactions from being ensured. Again, increases in performance can not make up for the lack of time-cognizant protocols in conventional database systems.

A temporal database is a real-time database.

Although both temporal databases [11] and real-time databases support the notion of time, the aspects of time they support are not the same. While a temporal database is aiming to support some aspects of time associated with information (e.g., time-varying information such as stock quotes), a real-time database tries to satisfy timing constraints associated with operational aspects of the database.

Time has several dimensions, but in the context of databases, two dimensions are of particular interest: valid time which denotes the time a fact was true in reality, and transaction time during which the fact was present in the database as stored data [6]. These two dimensions are in general orthogonal, although there could be some application-dependent correlations of the two times. A temporal database identifies those dimensions of time associated with the information maintained in the database and provides support to the user/applications to utilize such timing information, while in real-time databases those time dimensions imply some timing constraints.
Consider the difference between a temporal database and a real-time database via the following example. The military rank of Beetle Bailey can be specified in a temporal database as private during January 1, 1998, through June 30, 2001, at which time he will be promoted. It only states the timing fact that is believed to be true, regardless of when that information was entered. In most real-time databases, such static timing facts are not of primary concern. In a real-time database, the valid time is specified according to the semantics of the counterpart (external object) in the real world. When a value is entered into the database, its valid time specifies that the value can be assumed to represent the actual value (absolute time consistency). If the value of a sensor data was inserted into the database at time $T$ and its valid time is $t$, then the value must be updated within $T + t$ and if not updated in time, the value becomes stale and useless, or even dangerous. Current temporal database research does not pursue operational timing constraints such as maintaining correlation to real-time events in the real world and meeting deadlines.

Since meeting the timing constraints are essential in certain safety-critical database applications, a real-time database needs to provide a range of transaction correctness criteria that relax ACID properties. However, such approach is, in general, not acceptable in temporal databases. Because of their different objectives, the policies and mechanisms used to resolve data and resource conflicts in real-time databases are different from those in temporal databases. Temporal databases, along with other conventional databases, attempt to be fair while maximizing resource utilization. In real-time databases, timely execution of transactions is more important, and hence fairness and resource utilization become a secondary consideration.

There is no way to make guarantees or achieve predictability in a real-time database system.

It is sometimes argued that the predictability can not be enforced in a real-time database partly due to the complexity of making accurate and not too pessimistic estimates of the transaction execution times. The complexity rises from the fact that database systems have a number of sources of unpredictability [7]: dependence of the transaction’s execution sequence on data values; data and resource conflicts; dynamic paging and I/O; and transaction aborts resulting in rollbacks and restarts. We have already mentioned two solutions to overcome the unpredictability due to disk delays, namely, placing the database in main memory or adopting time-cognizant protocols for scheduling disk requests and managing memory buffers.

Due to data conflicts, transactions may be rollbacked and restarted, which increases the total execution time of the transaction, and in the worst case, unbounded number of restarts will occur, causing not only the transaction to miss its deadline, but may also jeopardize the timeliness of other transactions.
requesting resources. In real-time systems the set of transactions are normally well-known and can therefore be pre-analyzed in order to give estimates of the resources required, both in terms of execution time and what data that is needed. By scheduling transactions based on this information, conflicts and the number of transaction restarts can then be minimized and bounded.

A real-time database is a specialized database.

It is true that each real-time database application may have different timing constraints. However, it does not imply that a specialized database system must be developed from scratch for each application. An analogy would be to state that each real-time application needs to develop its own specialized real-time operating system, since its resource requirements and scheduling policies are different from others. Although the specifics of timing requirements can vary among applications, each application need the database support for specifying and enforcing its requirements.

At the same time, conventional database systems cannot be used for real-time applications simply by adding a few functional improvements. Since supporting timing constraints deals with the lowest level database access mechanisms, the overall architecture of database systems needs to be changed to be time cognizant. The reason is similar in explanation as to why certain time-critical applications need real-time operating systems (require fixed priorities, have known worst case latencies, be able to pin pages in memory, etc.) instead of conventional operating systems. However, we don’t need to develop different real-time operating system for each application.

4. SUMMARY

The field of real-time databases is growing in importance every day. It is important to know and dispel the common misconceptions. Doing this can give rise to more focussed and accurate research results and facilitate progress. We hope that commercial vendors of databases don’t perpetuate these 9 misconceptions which is sometimes done for marketing purposes,

References


Chapter 3

APPLICATIONS AND
SYSTEM CHARACTERISTICS

Doug Locke
TimeSys Corporation, USA

1. INTRODUCTION

As we look at real-time databases, it is helpful to consider some applications for which real-time databases are appropriate. In this chapter, we consider four different application domains. For each application domain we will discuss the underlying application performance requirements from which real-time database performance requirements are derived. The four application domains to be considered are:

- Avionics and Space
- Air Traffic Control
- Industrial Control
- Command and Control

Although there is a long design tradition for each of these application domains, this discussion does not necessarily reflect these traditions, generally because they use ad-hoc solutions rather than making use of real-time database technology. Instead, we consider the fundamental requirements for each application domain and what effects these requirements would have had if these application domains had been implemented using a real-time database. Few of these applications use any commercial databases at all because commercial real-time database implementations are not yet generally available.

Each application domain deals with an external environment that it controls or modifies within a set of bounded time constraints. This means that the application must use various sensors and other inputs to access information that describes its environment. Each piece of information must then be stored and retrieved in a timely way with an appropriate level of consistency with respect to every other piece of environment information.
For example, in an air traffic control system, the data to be managed includes the location of every aircraft in every sector of airspace being controlled. This information must remain consistent with all other available information about each flight, such as flight plans and weather, as well as information about each individual controller and his or her assigned airspace.

2. APPLICATION DOMAIN DESCRIPTIONS

There are many application domains that involve the use of real-time data in addition to these four, but these four clearly illustrate the nature of real-time data management in the presence of time constraints. For each application domain, we first discuss the general requirements for the domain and then the kind of information that must be managed. In addition, we discuss the relationship of the well-known ACID (Atomicity, Consistency, Isolation and Durability) properties to these applications and to their time constraints.

In each application domain description, we present a table containing some sample database parameters. These parameters include:

- **Size** — The approximate number of entities in the database. Knowing the approximate number of entities gives us some idea of how much memory will be needed, as well as the likely complexity involved in managing the data.

- **RTDB read/write response** — The approximate range of response times within which the database data must be read or written. The database designer uses the range of response times to determine the kinds of controls that can be supported. Longer times make it possible to use more complex management for the ACID properties. Note that the maximum values may constitute either a hard or soft deadline, depending on the characteristics of the data in the application.

- **External consistency** — The approximate required maximum latency of the database data with respect to the external environment. This latency bound includes all delays and processing/communication times involved from the time the data was valid in the environment to the time it is read from the database by an application component. Note that this represents approximately half of an end-to-end latency bound. The database implementation should support it as a hard deadline if any of the system end-to-end deadlines are hard.

- **Temporal consistency** — The approximate maximum time between readings of environment data that, even though the values may differ somewhat, can be considered to be equivalently valid representations of the external environment. This value determines the importance of maintaining the atomicity and consistency during database management.
Durability — The approximate maximum length of time for which the system is designed to run without restarting. This value gives some idea of how the database must be stored; i.e., whether it is reasonable for a memory-resident implementation or whether the data must be maintained in, or staged to, some long-term storage device.

2.1 AVIONICS AND SPACE

There are many different kinds of avionics and space applications, so this discussion can describe only a few in any detail. In general, avionics and space applications can be characterized as managing a set of sensors and actuators managing information describing and controlling an external environment. For example, a flight control system has sensors that describe the pilot’s intentions such as the position of the control yoke or rudder pedals, the operation of the control switches, attitude and airspeed sensors, etc. On the other hand, if the application is an onboard spacecraft mission processor, it is likely to be integrating the sensors for navigation, attitude control, and mission support. Its actuators include thrusters, attitude control systems, sensor controls, environment controls, computer and processing controls, etc.

The data managed by these kinds of systems, as with most data in real-time systems, can be classified into two categories. The first is a class of data related to environmental measurements that exhibit momentum such as velocity, roll rate, thrust, and altitude. Because of momentum, this class of data allows current values to be estimated from previous values, providing a level of redundancy that makes the system relatively forgiving of temporal data inconsistencies. The second class consists of information that does not exhibit the property of momentum, such as the health status of various spacecraft or aircraft systems, indicating that the equipment is, or is not, in a particular mode, a device is turned on or turned off, is available or unavailable, or has been selected or deselected by an operator or external interface.

With respect to the ACID properties, data exhibiting momentum frequently does not need atomicity to achieve an acceptable level of consistency because the error due to different periodic measurements can frequently be ignored. For example, consider a sensor providing the latitude and longitude of an aircraft. If atomicity is omitted, it is possible that a latitude reading from one period and a longitude from a previous period might be used together to represent the current position; the distance travelled between periods bounds the resulting error. This means that it is not always necessary to maintain all of the ACID properties for this kind of data under some circumstances. However, for system control data, it is much more likely that atomicity must be provided to achieve a satisfactory level of consistency. This means that the real-time database...
technologies discussed in this book become much more important for such data categories.

As an illustration, Table 3.1 shows some sample characteristics of an aircraft mission computer. A mission computer is one that manages the data needed for the aircraft mission. It would typically include aircraft navigation, tracking of expendable materials, operator information, etc. The database size reflects the need to manage a large number of air or ground tracks, air navigation facilities, operator information, etc. The read/write response time supports the need to store and retrieve this data with sufficiently low latency that combined with computation and input/output delays, the end-to-end time constraints can be met. The external consistency describes an acceptable level of error for position and attitude information (based on nominal values for speed and roll/pitch rates), while the temporal consistency reflects an acceptable level of error for most sensor data (e.g., radar returns, image processing, etc.) The durability value is derived from the length of a typical mission.

Table 3.2 shows similar data for a spacecraft onboard control computer. In this case, the database size reflects such information as sensor data, stored environment information (e.g., star maps, radiation readings, ephemeris information). The read/write response time is similar to that for an aircraft mission computer for the same reasons. The external and temporal consistency reflects the fact that the environment generally changes more slowly than that of an avionics processor. Similar to the aircraft mission computer, the durability value reflects the length of a mission.

### 2.2 AIR TRAFFIC CONTROL

The principal function of an air traffic control system is to track all of the aircraft in a portion of the airspace over some geographic area. For example, in the United States there are 20 Air Route Traffic Control Centers (ARTCCs)
that control all of the national airspace, in addition to many terminal control centers that manage aircraft in the vicinity of cities with one or more airports. An ARTCC can contain as many as several hundred air traffic controllers, each of whom controls a specific sector of the center’s airspace. Each sector controller uses several highly capable computer displays that show all of the relevant traffic information (e.g., aircraft identification, position, flight plan data, airways, conflict data, and weather) for that sector.

As an aircraft departs an airport, its crew communicates with air traffic controllers at the airport’s terminal facility until it departs the terminal area, when the flight is “handed off” to an ARTCC. Following the hand-off, depending on the distance to be covered by the flight, it is handed off from one sector or ARTCC to the next as it passes through various sector and altitude boundaries. Prior to departure from the airport, if the aircraft is operating under instrument flights rules (IFR), it will have filed a flight plan that describes the destination, the route of the flight, the type of aircraft, and other important information regarding the flight. During its flight, the aircraft reports its altitude and position using a radar transponder, and operates in continuous radio contact with air traffic controllers on the ground. All of this information must be continuously tracked by the air traffic control system and appropriately displayed.

In addition, each controller has a unique identification (ID) associated with a set of stored preferences regarding the settings of the displays in use, the types of controls to be used, and other unique preferences that he or she will use as the controlled aircraft moves into, or out of, the sector. The system must keep track of the current controller ID at each controller position and must use the personal preference information to manage the information displayed to each controller. During a controller shift change, or in the event of a display or computer failure at a controller station, the controller will move quickly to an available display position that must then be immediately adjusted to the controller’s current sector and preferences.

<table>
<thead>
<tr>
<th>Database size</th>
<th>5,000 entities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/write response time</td>
<td>0.05 ms. minimum 1.00 ms. maximum</td>
</tr>
<tr>
<td>External consistency</td>
<td>0.20 sec.</td>
</tr>
<tr>
<td>Temporal consistency</td>
<td>1.00 sec.</td>
</tr>
<tr>
<td>Durability</td>
<td>25 years</td>
</tr>
</tbody>
</table>

*Table 3.2 Spacecraft Onboard Computer Data Characteristics*
Weather information is especially critical to flights in the sector in which the weather occurs. Particularly in the case of potentially dangerous weather, such as icing conditions or thunderstorms, it is critical that the relevant weather information be immediately displayed so that the controller can route his or her traffic around the weather appropriately.

Although this means that the computers involved must precisely keep track of all the data associated with these functions, maintaining the data consistently, accurately, and without loss, it is equally important that the data be available at the time it is needed. Thus, there are specific time constraints that must be met, generally expressed using a mean, maximum, and 90th percentile for each of the critical functions involved.

For example, Table 3.3 shows some database performance figures that, while not specifically representing any particular air traffic control system, might be representative of some existing systems. The database size reflects all the aircraft that might be represented in a given ARTCC, all the weather information, all the active and pending flight plans, and all the physical entities (e.g., airports, navigation aids, radar installations, etc.) The read/write response time reflects the need to modify and access a large number of highly detailed pieces of information as radar, display, controller actions, and weather information passes through the system. The external consistency corresponds to the latency of handling and displaying radar (i.e., transponder) information, while the temporal consistency reflects the aircraft speed as it is represented on the display. The durability is derived from the lifetime of a flight plan (not including such data as controller preference information).

| Database size (approximate upper bound) | 20,000 entities (including tracks, weather, etc.) |
| Read/write response time | 0.05 ms. minimum |
| | 5.00 ms. maximum |
| External consistency | 1.5 sec. |
| Temporal consistency | 3.00 sec. |
| Durability | 12 hours |

Table 3.3 Air Traffic Control Data Characteristics
2.3 INDUSTRIAL CONTROL

Industrial control generally involves the management of a set of sensors and actuators reporting on and controlling a set of manufacturing and production processes. The information represented in the industrial control system includes such highly diverse examples as material or part locations, flow rates, vessel pressures, conveyor belt speed and position, chemical reactions conditions, and robotic controls. All of these exhibit time-critical characteristics that must be continuously maintained. In addition, industrial controls frequently involve the flow of supporting information such as engineering drawings, specifications, process models, status indicators, measurement of production employee activities, and many other kinds of information.

The data involved in such controls is generally considered critical to the overall management of the system for a number of reasons. Many manufacturing processes have safety implications, both to employees and to the surrounding environment. Government regulations dictate the maintenance of specific record-keeping processes. The profitability of the manufacturing process is strongly affected by the ability of the system to minimize inventories and work-in-process.

This implies, for example, that critical information such as the location of all parts and assemblies, maximum safe pressures and temperatures, location and amounts of dangerous substances, and other information must be maintained in a consistent state. This implies that at least the atomicity and consistency components of the ACID properties will be very important to many industrial control applications. However, the durability of the measured information is usually not nearly as critical over the long-term, with the exception of regulatory information.

The consequence of this is illustrated in Table 3.4 with an example of some of the derived database characteristics that might be typical of some industrial controls. The database size reflects the number of parts/assemblies that might be present on a large manufacturing floor, while the read/write response time reflects the speed that might be needed to ensure that safety critical temperatures/pressures are correctly maintained. The external and temporal consistency values might be those resulting from measurements of the position and speed of a conveyor, while the durability value is derived from the total time it might take for a part to travel from a storage area through its departure in an assembly from the manufacturing floor. It does not include the durability required, for example, for an “as-built-list” describing a complex assembly that must be retained for the life of the assembly following shipment.
2.4 COMMAND & CONTROL

Command and control systems are generally involved in handling massive amounts of widely diverse information derived from an external environment that must be comprehended by command personnel so that the environment can be managed, bringing the external situation to a successful conclusion. For example, in a battlefield command & control situation, arriving information comes from message traffic, intelligence information, multiple deployed systems such as tanks and aircraft, and many other sources. Some information comes via on-line communications such as wireless networks, while other sources arrive from voice communication entered into the system by a communication operator. Regardless of its source, all of this information must be consolidated into a single, coherent picture, accounting for the precision and accuracy of its source, and displayed to the decision maker. Thus, the command and control system must manage this data in such a way that it remains current and continuously available to the decision maker within a bounded amount of time. Such systems are frequently described as “systems of systems”. This means that the command and control system provides the central control over a large set of other systems such as messaging systems, fire control systems, etc.

The need for consistency among the data maintained by a command & control system derives from the system requirement to support one or more decision makers using information that accurately reflects the current status of the environment being controlled. For example, if the command & control application is a ground control facility for a space shuttle, the external consistency requirement is that the information presented must adequately reflect the current state of the shuttle and its environment within a short, bounded amount time. The exact time bound is defined by the type of information, such as the shuttle’s position data, attitude sensor data, equipment status data, experiment results data, or any of the other myriad kinds of information involved. In addition,
the information displayed at any single console must remain consistent with
information displayed at each of the other related consoles, resulting in a need
for temporal consistency. The need for durability for most of the information
being generated or handled is limited to its useful lifetime (not considering data
recording, which need not be immediately accessible during a mission), and is
usually measured in seconds or minutes.

The database size shown in Table 3.5 reflects the approximate number of
entities being managed in a large command & control system, such as a bat-
tlefield management system. The read/write response time of its database is
derived from the need to access a significant amount of information during a
correlation or sensor fusion operation whose end-to-end response time is on
the order of a second. The requirements for external and temporal consistency,
as well as the durability requirement (which excludes recording requirements)
might represent a space shuttle control application.

3. SUMMARY

These four sample application domains, while certainly not representative
of all real-time applications potentially using databases, illustrate some of the
widely diverse requirements involved. It is interesting to note that the numerical
values for the application data characteristics shown for each domain are gen-
erally stated neither in the “official” application requirements documentation,
nor in the design documentation that reflects derived application requirements.

These figures are not described in system requirements or design because
it is not generally recognized that these characteristics drive the design of the
data-handling mechanisms to be used by the application. Instead, most actual
application designs either use ad-hoc data handling mechanisms, or attempt
to use normal commercial database products that provide either anomalous
consistency behavior, or inappropriate response characteristics due to excessive

<table>
<thead>
<tr>
<th>Database size</th>
<th>50,000 entities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/write response time</td>
<td>0.50 ms. minimum 5.00 ms. maximum</td>
</tr>
<tr>
<td>External consistency</td>
<td>0.05 sec.</td>
</tr>
<tr>
<td>Temporal consistency</td>
<td>0.10 sec.</td>
</tr>
<tr>
<td>Durability</td>
<td>1 hour</td>
</tr>
</tbody>
</table>

*Table 3.5 Command & Control Data Characteristics*
adherence to the ACID properties. One of the main purposes of this book is to demonstrate how application designers can use these derived data requirements to define appropriate characteristics of real-time databases that will support them.

**References**


PART II
REAL-TIME CONCURRENCY CONTROL
Chapter 4

CONSERVATIVE AND OPTIMISTIC PROTOCOLS

Tei-Wei Kuo  
Department of Computer Science and Information Engineering  
National Taiwan University  
ktw@csie.ntu.edu.tw

Kam-Yiu Lam  
Department of Computer Science  
City University of Hong Kong, Kowloon, Hong Kong  
cskylam@cityu.edu.hk

1. OVERVIEW

One of the most important topics in the research of real-time database systems (RTDBS) is concurrency control [1, 2, 3, 4, 5, 10, 24]. The main issue is how to meet the urgency of transaction executions and, at the same time, to maintain the database consistency. In traditional database systems, transactions often suffer from an unbounded number of priority inversion. As a result, it is highly difficult to meet transaction deadlines.

Real-time concurrency control algorithms, in general, can be classified as either conservative or optimistic. Conservative algorithms prevent any violation of database consistency from happening. Various algorithms have been proposed for concurrency control of hard, soft, and firm real-time database systems. They are often self-restrained in the coordination of data access, and data objects are usually locked before they are accessed by a transaction. Transaction aborting is, in general, avoided to minimize the unpredictability of transaction completion time. Some variations of conservative algorithms adopt multi-data-version strategies to increase the concurrency level of the system.

On the other hand, optimistic concurrency control algorithms often delay the resolution of data conflicts until a transaction has finished all of its operations. A validation test is then performed to ensure that the resulting schedule is
serializable. In case of data conflicts, they usually use transaction aborts to maintain database consistency. Various validation schemes, such as wait-50 and sacrifice were proposed [3] to make the conflict resolution being priority-cognitive. Most the proposed algorithms based on the optimistic approach are for soft and firm real-time database systems. It is because the abort cost of a real-time transaction, under optimistic concurrency control algorithms, is often hard to quantify. It is very difficult to ensure the schedulability of the transactions under the optimistic approach.

In this chapter, we will first define the real-time transaction model. We then summarize recent work for conservative algorithms in terms of lock-oriented real-time concurrency control. Optimistic real-time concurrency control will then be discussed. A summary of the previous work and some suggestions for the future work will be presented in the last section of this chapter.

2. TRANSACTION MODEL AND SCHEDULING

A real-time transaction is the template of its instances. A transaction instance is a sequence of read and write operations on data objects. An instance of a transaction is scheduled for every request of the transaction. Real-time transactions are usually associated with deadlines.

Transaction priorities are often assigned according to their timing constraints such as periods (rate monotonic priority assignment) or deadlines (earliest deadline first priority assignment). The rate monotonic priority assignment (RM) [18] assigns transactions priorities inversely proportional to their periods. That is, a smaller-period transaction is assigned a higher priority. If transactions are not allowed to change their periods, transaction priorities are statically assigned, regardless of which of their instances is referred. The earliest deadline first priority assignment (EDF) assigns transactions priorities according to the distance of their deadlines and the current time. The closer the deadline, the higher the transaction priority is. In other words, the priority order of the instances of any two transactions may be dynamically changing. There are also many other priority assignment schemes based on slack [19], transaction criticality, transaction value [8], etc.

Real-time concurrency control must meet the urgency of real-time transactions but also maintain the consistency of the database. Depending on the adopted concurrency control algorithms, transactions may be abortable or non-abortable. When a transaction is aborted, it must be rolled back to restore the database back to its original state. In order to reduce the rollback cost and avoid cascading aborting, the delayed write procedure is often adopted in many concurrency control algorithms. Under the delayed write procedure, for each data object updated by a transaction, the update is done in the local area of the
transaction, and the actual write of the data object is delayed until the commit
time of the transaction.

A database can be considered as a collection of data objects. When a
lock-oriented concurrency control algorithm is adopted, each data object is
associated with a lock. Before a transaction locks a data object, it must lock
the data object in a proper mode, such as share (or read) or exclusive (or write)
mode. For the purpose of this chapter, we will use the terms “transaction” and
“transaction instance” interchangeably when there is no ambiguity.

3. LOCK-ORIENTED CONCURREN CY CONTROL

Conservative concurrency control algorithms usually require transactions
to lock data objects before the data objects are used. Any violation of data
consistency is usually prevented from happening, and transaction abort is often
minimized. The two-phase locking scheme (2PL) is the most popular locking
scheme adopted in lock-oriented concurrency control algorithms to preserve the
serializability of transaction executions, where any transaction that adopts 2PL
cannot request any new lock on any data object after the transaction releases a
lock. A real-time transaction scheduled by a lock-oriented concurrency control
algorithm with 2PL must lock data objects in a 2PL fashion.

3.1 READ/WRITE PCP ALGORITHM

The Read/Write Priority Ceiling Protocol (RWPCP) [24], which is an ex-
tension of the well-known Priority Ceiling Protocol (PCP) [23], adopts 2PL in
preserving the serializability of transaction executions. RWPCP is proposed
for concurrency control of periodic hard real-time transactions. RWPCP has
shown the effectiveness of using read and write semantics in improving the
performance of PCP. While PCP only allows exclusive locks on data objects,
RWPCP introduces a write priority ceiling $WPL_i$ and an absolute priority ceil-
ing $APL_i$ for each data object in the system to emulate share and exclusive
locks, respectively.

The write priority ceiling $WPL_i$ of data object $O_i$ is equal to the highest
priority of transactions which may write $O_i$. The absolute priority ceiling
$APL_i$ of data object $O_i$ is equal to the highest priority of transactions which
may read or write $O_i$. When data object $O_i$ is read-locked, the read/write
priority ceiling $RWPL_i$ of $O_i$ is equal to $WPL_i$. When data object is
write-locked, the readwrite priority ceiling $RWPL_i$ of $O_i$ is equal to $APL_i$.
A transaction instance may lock a data object if its priority is higher than
the highest read/write priority ceiling $RWPL_i$ of the data objects locked by
other transaction instances. When a data object $O_i$ is write-locked, the setting
of $RWPL_i$ prevents any other transaction instance from write-locking $O_i$
because $RWPL_i$ is equal to $APL_i$. When a data object $O_i$ is read-locked, the
setting of $RWPL_i$ only allows a transaction instance with a sufficiently high priority to read-lock $O_i$ in order to constrain the number of priority inversions for any transaction instance that may write-lock $O_i$, because $RWPL_i$ is equal to $WPL_i$.

A transaction $\tau$ uses its assigned priority, unless it locks some data objects and blocks higher priority transactions. If a transaction blocks a higher priority transaction, it inherits the highest priority of the transactions blocked by $\tau$ (priority inheritance). When a transaction unlocks a data object, it resumes the priority which it had at the point of obtaining the lock on the data object. The priority inheritance is transitive. Note that the resetting of priority inheritance can be efficiently implemented by using a stack data structure.

RWPCP is designed for hard real-time transaction scheduling. It is required that the system consists of a fixed set of transactions and a fixed collection of data objects. In return, RWPCP guarantees deadlock-freeness and at most one priority inversion for every transaction. Transactions are given fixed priorities and scheduled in a priority-driven fashion, such as that by the rate monotonic priority assignment [18].

**Example 1** RWPCP Schedule:

![Figure 4.1 A RWPCP Schedule](image)
We shall illustrate RWPCP by an example: Suppose that there are three transaction $\tau_1$, $\tau_2$, and $\tau_3$ in a uniprocessor environment. Let the priorities of $\tau_1$, $\tau_2$, and $\tau_3$ be 1, 2, and 3, respectively, where 1 is the highest, and 3 is the lowest. Suppose that $\tau_1$ and $\tau_2$ may read and write data object $S_1$, respectively, and $\tau_2$, and $\tau_3$ may read and write data object $S_2$, respectively. According to the definitions of ceilings, the write priority ceiling $WPL_1$ and the absolute priority ceiling $APL_1$ of $S_1$ are 2 and 1, respectively. The write priority ceiling $WPL_2$ and the absolute priority ceiling $APL_2$ of $S_2$ are 3 and 2, respectively.

At time 0, $\tau_3$ starts execution. At time 2, $\tau_3$ write-locks $S_2$ successfully, and $RWPL_2 = APL_2 = 2$. At time 4, $\tau_2$ arrives and preempts $\tau_3$. The write-lock request of $\tau_2$ on $S_1$ is blocked at time 6 because the priority of $\tau_2$ is no higher than $RWPL_2 = APL_2 = 2$. As a result, $\tau_2$ is blocked by $\tau_3$. At time 9, $\tau_2$ successfully write-locks $S_1$ ($RW PL_1 = APL_1 = 1$). At time 11, $\tau_1$ arrives and preempts $\tau_2$. The read-lock request of $\tau_1$ on $S_1$ is blocked at time 13 because the priority of $\tau_1$ is no higher than $RW PL_1 = APL_1 = 1$. As a result, $\tau_1$ is blocked by $\tau_2$. $\tau_2$ resumes its execution at time 13 and read-locks $S_2$ successfully because no data object is locked by any other transaction ($RWPL_2 = WPL_2 = 3$). $\tau_2$ then unlocks $S_2$ and $S_1$ at times 18 and 20, respectively. At time 20, $\tau_1$ resumes its execution and read-locks $S_1$ successfully ($RWPL_1 = WPL_1 = 2$). $\tau_1$ unlocks $S_1$ at time 22 and commits at time 23. $\tau_2$ resumes its execution at time 23 and commits at time 25. $\tau_3$ then resumes its execution at time 25 and commits at time 27.

### 3.2 VARIANTS OF RWPCP-BASED ALGORITHMS

There are a number of lock-oriented concurrency control protocols derived from RWPCP or PCP, e.g., [7, 11, 15, 16, 17, 22, 26]. In particular, Lam and Hung [15] further sharpened the RWPCP by proposing the idea of dynamic adjustment of serializability order for hard real-time transactions, where Lin and Son [13] first proposed the idea of dynamic adjustment of serializability order for optimistic real-time concurrency control. With a delayed write procedure, a higher-priority transaction instance may preempt a lower-priority transaction instance by using the Thomas Write rules when a write-write conflict exists, where a delayed write procedure requires every transaction instance to only update data objects in its local space and to delay the updating of the database until the commitment of the transaction instance. Here the Thomas Write Rule (TWR) [25] ignores obsolete write operations instead of rejecting them to prevent obsolete information from entering a database. The read-write conflict between conflicting transaction instances is partially resolved by allowing a higher-priority transaction instance to read the database even though a lower-priority transaction instance has write-locked the data object. Note that the delayed write procedure requires every transaction instance to only update data...
objects in its local space, and the above preemption in read-write conflict lets the higher-priority transaction instance precede the lower-priority transaction instance in the serializability order. Similar ideas were proposed by Lam, et al. [16, 17] in favoring read-only transactions. Weak correctness criteria are proposed to allow read-only transactions to have different views on the transaction serializability order.

<table>
<thead>
<tr>
<th>req \ locked</th>
<th>Read-Lock</th>
<th>Write-Lock</th>
<th>Certify-Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-Lock</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Write-Lock</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Certify-Lock</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 4.1 The lock compatibility matrix of 2VPCP

Kuo, et al. [7] proposed a two-data-version-based concurrency control protocol Two-Version PCP (2VPCP) to reduce the blocking time of higher-priority transactions based on the idea of dynamic serializability adjustment [13, 14] without relying on local data updates for transactions [15, 17, 13]. Each data object has two versions: consistent version and working version, where a consistent version contains a data value updated by a committed transaction instance, and a working version contains a data value updated by an uncommitted transaction instance. There are three kinds of locks in the system: read, write, and certify. Before a transaction reads (or writes) a data object, it must first read-lock (or write-lock) the data object. A read operation on a data object always reads from the consistent version of the data object. A write operation on a data object always updates the working version of the data object. It is required that, before a transaction commits, the transactions must transform each of its write locks into a certify lock on the same data object. As soon as a transaction obtains a certify lock on a data object, it can copy its updated working version of the data object to the consistent version. There is no requirement on the order or timing of lock transformations. The transformation of a write-lock into a certify-lock is considered as requesting a new certify lock. If the request of a certify-lock by a transaction instance is not granted, the transaction is blocked by the system until the request is granted. When a transaction terminates, it must release all of its locks. The compatibility matrix of locks is shown in Table 4.1, which is as the same as the compatibility matrix for the well-known Two-Version Two-Phase Locking [20].

As defined in RWPCP, the write priority ceiling \( WPL \) of data object \( O \) is equal to the highest priority of transactions that may write \( O \). The absolute
priority ceiling $APL_i$ of data object $O_i$ is equal to the highest priority of transactions that may read or write $O_i$. Since 2VPCP adopts a two-data-version approach and introduces a new lock called certify lock, the setting of the read/write priority ceiling $RWPL_i$ of each data object $O_i$ is modified as follows: The read/write priority ceiling $RWPL_i$ of each data object $O_i$ is set dynamically. When a transaction read-locks or write-locks $O_i$, $RWPL_i$ is equal to $WPL_i$. When a transaction certify-locks $O_i$, $RWPL_i$ is equal to $APL_i$. Note that any read operation on a data object always reads from the consistent version of the data object, and any write operation on a data object always writes into the working version of the data object.

The 2VPCP protocol is, then, extended to a distributed environment to process read-only transactions at client-side systems locally and to support predictable and efficient failure recovery. The 2VPCP protocol also guarantees deadlock-freeness and at most one priority inversion for every transaction.

**EXAMPLE 2** 2VPCP Schedule:

![Figure 4.2 A 2VPCP Schedule](image)

We illustrate the 2VPCP protocol by the same PCP example. At time 0, $\tau_3$ starts execution. At time 2, $\tau_3$ write-locks $S_2$ successfully, and $RWPL_2 = WPL_2 = 3$. At time 4, $\tau_2$ arrives and preempts $\tau_3$. At time 6, $\tau_2$ write-locks $S_1$ successfully because the priority of $\tau_2$ is higher than $RWPL_2$ ($RWPL_1 =$
At time 8, τ₂ read-locks S₂ successfully because the priority of τ₂ is higher than RWPL₂ (RWPL₂ = WPL₂ = 3). Note that τ₃ is behind τ₂ in the serializability order although τ₃ write-locks S₂ before τ₂ read-locks S₂. At time 11, τ₁ arrives and preempts τ₂. At time 13, τ₁ read-locks S₁ successfully because the priority of τ₁ is higher than RWPL₁ and RWPL₂. RWPL₁ is equal to WPL₁ = WPL₁ = 2. Note that τ₂ is behind τ₁ in the serializability order although τ₂ write-locks S₁ before τ₁ read-locks S₁. τ₁ then unlocks S₁ and commits at time 15 and 16, respectively. Right before time 18, τ₂ certify-locks S₁ successfully and copies the working version of S₁ into the consistent version because the priority of τ₂ is higher than RW PL₂ (RWPL₂ = W PL₂ = 3). At time 18, τ₂ unlocks S₂. At time 20, τ₂ unlocks S₁. At time 22, τ₂ commits, and τ₃ resumes its execution. Right before time 25, τ₃ certify-locks S₂ successfully and copies the working version of S₂ into the consistent version. At time 27, τ₃ commits.

This example demonstrates that one of the goals in designing the 2VPCP protocol is that a higher-priority transaction instance can utilize the consistent version of a data object to avoid being blocked by a lower-priority transaction instance due to read/write conflicts. The serializability order of transaction instances is no longer determined by the order of their conflicting lock requests and can be adjusted according to the priorities of the transaction instances.

### 3.3 ABORTING VERSUS BLOCKING

Although RWPCP and its variants provide ways to bound and estimate the worst-case blocking time of a transaction, they are usually pretty conservative, and it is often unavoidable to avoid lengthy blocking time for a transaction in many systems. Transaction aborting is suggested by many researchers to solve problems due to lengthy blocking time. One of the well known example is the High Priority Two Phase Locking protocol (HP-2PL) [1], in which the lock-requesting transaction may abort the lock-holding transaction if the former has a priority higher than the later. The algorithm is named as HP-2PL because transactions must lock data objects in a 2PL fashion, and the system always prefers higher priority (HP) transaction when any lock conflict occurs. HP-2PL is usually explored for the scheduling of soft and firm real-time transactions, instead of hard real-time transactions, because of the potentially unbounded number of transaction aborting. Various variants based on the HP-2PL have been proposed, such as that with conditional priority inheritance [6]. Under HP-2PL with conditional priority inheritance, the lock-requesting transaction may be blocked by the lock-holding transaction if the remaining execution time of the later is smaller than a half of its original execution time. Priority inheritance is applied if any blocking exits. Otherwise, transaction abort will be used to resolve the data conflict.
Transaction aborting has been incorporated into RWPCP-based algorithms for hard, soft, and firm real-time database systems. A number of researchers, e.g., [9, 10, 11, 21, 22, 26] have proposed lock-oriented concurrency control algorithms which rely on one transaction aborting to preserve data consistency and bound the worst-case number of priority inversion for critical transactions. In particular, Liang, et al. [11, 9] proposed a framework in trading the aborting cost with the blocking cost of transactions. Different levels of aborting relationship among transactions are considered, and the impacts of the aborting relationship are evaluated when the relationship is built in an on-line or off-line fashion. Shu, et al. [21, 22] demonstrated the strengths of transaction aborting in reducing priority inversion and in the improvement of the schedulability of critical transactions. In addition to the priority ceilings defined for data objects [23, 24], an abort ceiling is defined for each transaction to allow a higher-priority transaction to abort a lower-priority transaction which blocks the higher-priority transaction, if the abort ceiling of the lower-priority transaction is smaller than the priority of the higher-priority transaction. A non-trivial mechanism is proposed to determine when and how to set the abort ceiling of each transaction. Takada, et al. [26] also proposed a concurrency control protocol which allows the switchings of the aborting status of transactions to better manage the blocking and aborting costs of transactions.

The main idea behind transaction aborting is that when a higher priority transaction is blocked by a lower priority transaction due to resource competition, the higher priority transaction aborts the lower priority transaction if the lower priority transaction is abortable, and the lower priority transaction may introduce excessive blocking time to any higher priority transaction. If not, the higher priority transaction is blocked by the lower priority transaction. Whether a transaction is abortable or may impose excessive blocking on any higher priority transaction can be determined by an on-line or off-line schedulability analysis.

The aborting cost of a transaction by a higher-priority transaction $\tau'$ is often modeled as the CPU time that has been consumed by $\tau$ when $\tau$ is aborted by $\tau'$. The worst-case (total) aborting cost per request/instance of transaction can be approximated as the sum of the aborting cost of $\tau$ by all higher-priority transactions in its execution period (or between its ready time and commit time or deadline). Different concurrency control algorithms can guarantee different amounts of aborting cost. For example, the aborting algorithms proposed in [11] guarantee that any request of a lower priority transaction can be aborted at most once by a higher priority transaction within a period of the higher priority transaction. Let $b_r, d_r, p_r, c_r,$ and $ab_r$ be the worst case blocking time (from lower priority transactions), deadline, period, worst case computation requirement, and worst-case aborting cost of a transaction $\tau$, respectively.
With the rate monotonic analysis (RMA) [12], the following schedulability formula can be derived:

**Theorem 1**[11]: A transaction $\tau_i$ will always meet its deadline for all process phases if there exists a pair $(k, m) \in SP_i$ such that

$$\sum_{j \in HPC_i} \left( c_j \left\lfloor \frac{m p_k}{p_j} \right\rfloor \right) + c_i + b_i + a b_i \leq m p_k,$$

where $HPC_i$ is the set of transactions with a priority higher than $\tau_i$, and $SP_i = \{ (k, m) | 1 \leq k \leq i, m = 1,2, \ldots , \left\lceil \frac{p_i}{p_k} \right\rceil \}$. Each pair $(k, m)$ represents a scheduling time point $mp_k$ to test the schedulability of transaction $\tau_i$. Let $MPK$ be defined as $\{ mp_k | (k, m) \in SP \}$. The maximum blocking time which a transaction $\tau_i$ can tolerate becomes

$$\max_{k \in MPK} \left[ t - \sum_{j \in HPC_i} (c_j \left\lfloor \frac{t}{p_j} \right\rfloor) - c_i - a b_i \right].$$

The formula shows that there is a tradeoff between the maximum tolerable blocking time and aborting cost. The higher the aborting cost, the lower the tolerable blocking time is. The aborting cost of a transaction can be zero if the transaction is not abortable, as required in many algorithms such as RWPCP. If a transaction is set as abortable because it may introduce excessive blocking time to some higher-priority transaction, then the transaction has a non-zero aborting cost and can only tolerate a smaller amount of blocking time which is caused by lower-priority transactions. In order to tune the schedulability of an entire transaction system, we must determine which transaction is abortable or non-abortable in order to guarantee the schedulability of hard real-time transactions without unnecessarily sacrificing the response time of soft real-time or non-real-time transactions. We refer interested readers to [9, 11] for details.

4. **OPTIMISTIC CONCURRENCY CONTROL**

Optimistic concurrency control often assumes that the execution of a transaction consists of three phases: read, validation, and write phases. During the read phase, a transaction reads from the database but update data objects at its local space. When a transaction prepares to commit, it enters its validation phase to check for the satisfaction of the system correctness such as serializability. If a transaction passes its validation phase, then it enters its write phase by reflecting its updates in the local space into the database.

There are two major approaches for the validation: backward and forward validation. In the backward validation, the validating transaction is verified against committed transactions. A validating transaction is allowed to commit if the resulting schedule is serializable. In the forward validation, a validating transaction validates against other executing transactions. Both backward and forward validation methods resolve data conflict by abort of transactions. In the previous research studies [3], it had been found that the forward validation
is more suitable for real-time database systems as it provides a better freedom in resolving access conflict.

Various validation algorithms such as wait-50 and sacrifice were proposed by Haritsa, et al. [3, 4]. In the wait-50 scheme, if the number of conflicting transactions with higher priorities is greater than 50% of the total number of conflicting transactions, the validating transaction will be blocked until it is smaller than 50%. Otherwise, all conflicting transactions will be restarted even though some of their priorities are higher than the priority of the validating transaction. An extension was made on the wait-50 scheme, by considering wait-X, where X is the threshold for the percentage of all conflicting transactions which have higher priorities than the validating transaction. When X is equal to 50%, wait-X is wait-50. Haritsa, et al. found that when X=50, the miss ratio of all transactions is minimized in many cases.

Traditionally, the serializability order of transactions scheduled by optimistic concurrency control is determined by the validation order of executing transactions. Son, et al. [13, 14] proposed an optimistic concurrency control protocol based on the idea of dynamic adjustment of serializability order to avoid unnecessary restart. In particular, a timestamp-interval adjustment scheme is proposed to dynamically reorder transaction serializability order if possible:

When a transaction $\tau$ starts execution, it is assigned a timestamp interval $[0, \infty)$, which is the entire range of time interval. Whenever any other transaction passes its validation phase, the timestamp interval of $\tau$ is adjusted to reflect the data dependencies or access precedence of transactions. When $\tau$ successfully passes its validation phase, a final timestamp in its timestamp interval is assigned to $\tau$. The final timestamps of transactions determine their serializability order. That is, a transaction with a smaller final timestamp is before another transaction with a larger final timestamp in the serializability order. $\tau$ will be aborted if its time interval become null during its read phase.

The adjustment and validation procedures are defined as follows [14]:

Let $TI(\tau)$ and $TS(\tau)$ denote the timestamp interval and final timestamp of transaction $\tau$, $RS(\tau)$ and $WS(\tau)$ denote the read and write sets of $\tau$. Suppose that $\tau_v$ and $\tau_a$ denote a validated transaction and a transaction under timestamp-interval-adjustment, respectively. When a transaction $\tau_v$ enters its validation phase, the following validation procedure is invoked.

Validate($\tau_v$) {
    Select an arbitrary $TS(\tau_v)$ from $TI(\tau_v)$;
    For any executing transaction $\tau_a$ in its read phase Do
        Adjust($\tau_a$);
    Update the read and write timestamps of data objects
        in $RS(\tau_v)$ and $WS(\tau_v)$, respectively;
}
The selection of final timestamp for a validating transaction can be arbitrary. The maintenance of timestamps is for efficient implementation of the algorithm. We refer interested readers to [14] for details.

The adjustment procedure is a process to record the serializability order according to the conflicting types of operations:

\[ \text{Adjust}(\tau_a) \{ \]
\[ \quad \text{For any data object } O_i \text{ in } RS(\tau_v) \text{ Do } \{ \]
\[ \quad \quad \text{If } O_i \in WS(\tau_a) \text{ then} \]
\[ \quad \quad \quad TI(\tau_a) = TI(\tau_a) \cap [TS(\tau_v), \infty] ; \]
\[ \quad \quad \text{If } TI(\tau_a) == [] \text{ then} \]
\[ \quad \quad \quad \text{Restart } \tau_a ; \]
\[ \quad \} \]
\[ \quad \text{For any data object } O_i \text{ in } WS(\tau_v) \text{ Do } \{ \]
\[ \quad \quad \text{If } O_i \in WS(\tau_a) \text{ then} \]
\[ \quad \quad \quad TI(\tau_a) = TI(\tau_a) \cap [0, TS(\tau_v) - 1] ; \]
\[ \quad \quad \text{If } O_i \in WS(\tau_a) \text{ then} \]
\[ \quad \quad \quad TI(\tau_a) = TI(\tau_a) \cap TS(\tau_v), \infty] ; \]
\[ \quad \text{If } TI(\tau_a) == [] \text{ then} \]
\[ \quad \quad \text{Restart } \tau_a ; \]
\[ \} \]

The first For loop is to make sure that \( \tau_a \) is after the validated transaction \( \tau_v \) in the serializability order by adjusting the timestamp interval of \( \tau_a \). Note that if the timestamp interval of an executing transaction \( \tau_a \) is shut out, then it means that \( \tau_a \) may introduce non-serializable execution with respect to the validated transaction \( \tau_v \). The first If statement in the second For loop is to ensure that \( \tau_a \) is before \( \tau_v \) because \( \tau_a \) has read from the database before \( \tau_v \) updates the database in its write phase. The second If statement in the second For loop is to ensure that \( \tau_a \) is after \( \tau_v \) because \( \tau_a \) must update the database after \( \tau_v \) does so. The following example can be used to understand how the adjustment procedure works:

**Example 3** Dynamic Adjustment of Serializability Order: Let \( r_1[x] \) and \( w_1[x] \) denote that data object \( x \) is updated by transaction \( \tau_1 \), and \( c_i \) denote the commitment of \( \tau_i \). Consider the following schedule:

\[ \pi = w_1[x], r_2[x], r_2[y], w_1[y], c_1, c_2 \]

The validation of transaction \( \tau_1 \) first assigns a timestamp to \( \tau_1 \), say \( TS(\tau_1) = 50 \). The resulted timestamp-interval adjustment will set the timestamp of \( \tau_2 \) to
be $[0, 49]$. It is to ensure that the final timestamp of $\tau_2$ will be before $\tau_1$ because $\tau_2$ reads from the database before $\tau_1$ really updates the database in the write phase. Thus, their final timestamps determine their serializability order. □

Recently, researchers started working on real-time concurrency control for systems consisting of mixed real-time transactions, e.g., those with soft real-time, hard real-time, and non-real-time transactions. In such real-time database systems, a single strategy for concurrency control will not be sufficient. Integrated methods are proposed for resolving data conflicts among different types of transactions. For example, in [27], a two-level concurrency control framework is proposed for real-time database systems consisting of soft and non-real-time transactions. Under the framework, a master concurrency controller is proposed to detect any possible inter-class data conflicts, which are data conflicts between different types of transactions. Any intra-class data conflicts amongst transactions in the same class are detected and resolved by individual schedulers. Different concurrency control protocols can be incorporated into the systems. Under the framework, data conflicts between soft real-time and non-real-time transactions are resolved by performing a serialization check on the time-stamps of transactions. In their experiments, the base protocols for non-real-time and soft real-time transactions are 2PL and OCC-Wait-50, respectively.

5. SUMMARY AND FUTURE WORK

Although a number of researchers have done excellent work on real-time concurrency control, it is still lack of a concluding remark on which real-time concurrency control protocol is more suitable to which kind of real systems. Instead of proposing yet another real-time concurrency control, it may be important, especially at this point, to investigate the performance of protocols under representative benchmarks. Also, many real-time concurrency control protocols assume the existence of a perfect "real-time" environment. It is highly important to study the impacts of various non-real-time system features on the performance of different real-time concurrency control protocols.

Although some researchers, e.g., [1, 3] have showed the superiority of the optimistic concurrency control protocol, we must emphasize that optimistic concurrency control, in general, does not fit the needs of hard real-time transaction scheduling because transactions may be repeatedly aborted, and/or the aborting cost of a transaction is often hard to quantify or bound. It is highly important to minimize the number of unnecessary aborts in order to minimize the wasting of system resources in re-executing transactions and to maintain the performance of optimistic concurrency control.
References


1. OVERVIEW

Real-time concurrency control has been an active research topic in the past decades. It has been recognized by a number of researchers that the notion of serializability is too strict a correctness criterion for concurrency control in accessing real-time data. In avionics software, for example, the precision of an answer to a query involving sensor data is often acceptable as long as the data is sufficiently timely, even though updates are sometimes performed in violation of the usual serializability criterion. Instead of read/write locks, a “cyclic executive” is also routinely used in these applications to enforce a set of timing constraints on data access which in turn guarantees data consistency, the usual database locking protocols being too inefficient for the purpose. Obviously, violation of serializability must be justified in the context of the semantics of the application domain.

Three increasingly restrictive criteria for correctness are commonly accepted and have been studied in detail. They are: final-state serializability, view serializability, and conflict serializability [17]. In recent years, other different correctness criteria and related concurrency control algorithms have been proposed for different purposes and application. The relaxation of correctness criteria not only helps the deployment of (real-time) database applications in various environments and application domains, such as distributed environments, but also provides the performance improvement which is otherwise impossible for many real-time applications. In other words, relaxed correctness criteria not only provide more functionality, e.g., queries of different precision levels under Epsilon-serializability [6, 19, 20], but also relax the constraints in transaction processing, such as similarity-based concurrency control [7, 8].
Kuo and Mok [7] explored the similarity of data to provide a semantic foundation for accessing data in a real-time fashion. Similarity is a binary relation on the domain of a data object. Every similarity relation is reflexive and symmetric, but not necessarily transitive. In a schedule, we say that two event instances are similar if they are of the same type (read/write) and access similar values of the same data object. The concept of similarity can be used to extend the usual correctness criteria for transaction scheduling [7] and to derive flexible and efficient concurrency control algorithms in uniprocessor, multiprocessors, and even mobile environments [8, 9, 12, 13]. The results justify the weaker notion of correctness that has been employed on an ad hoc basis in many real-time applications where the state information is "volatile" and the value of data depends on its timeliness. A distributed real-time data management interface was also built on an Intel multiprocessor machine [10]. Xiong et al. [22] further exploited temporal data similarity and evaluated the performance improvement of transactions when combinations of similarity and forced wait policies were considered. A force wait policy may force a transaction to delay further execution until a new version of sensor data becomes available.

Peng and Lin proposed the idea of compatibility matrix to allow transactions to acquire different degrees of consistency requirements [18]. Their work was motivated by avionic systems and automated factories that have a limited number of high-speed sensors with frequent user-initiated command processing. The rationale behind their work was that the consistency between the device readings and the current values used by transactions could be more important than the serializability of transactions. DiPippo and Wolfe [2] also proposed object-oriented techniques to support both logical and temporal consistency requirements. Compatibility functions of objects were proposed to allow the tradeoffs among these requirements.

Epsilon-serializability (ESR) [19, 20] formalizes the query behavior by deriving the formulae that express the inconsistency in the data values read by a query. Transactions are associated with limits of importing inconsistency and exporting inconsistency. Query transactions are allowed to view inconsistent data in a controlled fashion. Kamath and Ramamritham [6] then introduced the idea of hierarchical inconsistency bounds that allows inconsistency to be specified at different granularities such as transactions and objects. They provided mechanisms to control the inconsistency and reported the evaluation of the performance improvement due to ESR.

There are also many weak correctness criteria being proposed for non-real-time transaction systems [14, 15]: In particular, Garcia-Molina and Wiederhold in [5] discarded consistency considerations for read-only transactions, with the stipulation that, after read-only transactions have been removed, the resulting schedule should be serializable. Garcia-Molina and Salem [4] proposed the
concept of “SAGAS” so as to solve consistency problems brought on by long-lived transactions, where SAGAS are long-lived transactions that can be broken up into a collection of sub-transactions that can be interleaved in any way with other transactions. Thus, SAGA is not atomic but should be executed as a unit. It means that correct schedules can be nonserializable. Korth and Speegle [11] proposed a formal model which allows transactions to specify pre-conditions and post-conditions. These conditions can be specified in conjunctive normal form. They enforced serializability with respect to every conjunct in the conjunctive normal form by a criterion called predicatewise serializability. Their model also includes consideration of nested transactions and multiple versions. Update consistency [1] has been explored under broadcast environments such that clients can obtain data that is currently consistent “off the air” without contacting the server to obtain locks [21], where update consistency requires all update transactions being serializable. We refer interested readers to [14, 15] for more detailed summary of relaxed concurrency control for non-real-time transaction systems.

2. APPLICATION SEMANTICS - DATA SIMILARITY

Data objects in a real-time database may reflect entities in a real-world environment that changes continuously. The value of a data object cannot in general be updated continually to perfectly track the dynamics of its corresponding real-world entity. The time needed to perform an update alone necessarily introduces a time delay which means that the value of a data object cannot be instantaneously the same as the corresponding real-world entity. Fortunately, it is often unnecessary for data values to be perfectly up-to-date or precise to be useful. In avionic systems, the dynamics of a sensor or the environment may limit how fast sensor readings can change over a short time interval. For certain computations, engineers often consider the change in sensor readings over a few consecutive cycles to be insignificant in the execution of the avionic software. It is sometimes acceptable to use a reading that is not the most recent update in a transaction. Data values of a data object that are slightly different in age or in precision are also interchangeable as read data for transactions in many cases. This observation underlies the concept of similarity among data values.

As one might expect, there is no general criterion for determining whether two values are similar. It is imperative for us to justify and make explicit the implicit assumptions that are behind the currently ad hoc engineering practice. They can be the cause of costly errors. However, the proximity in data value or some mappings, such as rounding or “=”, may often be used in establishing similarity relations. Similarity can be formally defined a binary relation on the domain of a data object. Every similarity relation is reflexive and symmetric, but
not necessarily transitive. Different transactions can have different similarity relations on the same data object domain. *Two views of a transaction are similar* if and only if every read event in both views uses similar values with respect to the transaction. We say that *two values of a data object are similar* if all transactions which may read them consider them as similar. In a schedule, we say that *two event instances are similar* if they are of the same type and access similar values of the same data object. *Two database states are similar* if the corresponding values of every data object in the two states are similar.

![Figure 5.1 Similarity of database states](image)

As an example, consider the similarity of data read by two transactions in a railroad-crossing monitoring system. Suppose there are two data objects, *distance* and *velocity*, which provide information about the nearest approaching train. In the above figure, $S$, a point, is a database state in the database state space of the system. Let $\tau_1$ be a transaction that displays the *distance* and *velocity* of the approaching train on the monitoring system. The other transaction $\tau_2$ controls the crossing gate which depends only on the *distance* of the train. With different precision requirements, $\tau_1$ and $\tau_2$ consider values falling inside, respectively, $\tau_1$’s box and $\tau_2$’s box to be similar to their counterparts in the state $S$. Note that because $\tau_2$ does not read *velocity*, all values in the domain of *velocity* are similar to one another and therefore similar to that at $S$. Note that according to the terminologies defined in the previous paragraph, two values of a data object are similar if and only if all transactions that may read them consider them to be similar. We say that all database states in the overlapping of $\tau_1$’s box and $\tau_2$’s box are similar to $S$.

A minimal restriction on the similarity relation that makes it interesting for concurrency control is the requirement that it is preserved by every transaction, i.e., if a transaction $\tau$ maps database state $s$ to state $t$ and state $s'$ to $t'$, then
$t$ and $t'$ are similar if $s$ and $s'$ are similar. We say that a similarity relation is regular if it is preserved by all transactions. We are interested in regular similarity relations only.

### 3. SIMILARITY-BASED CONCURRENCY CONTROL

The notion of strong similarity was introduced in [7] which has the property that swapping any number of similar events in a schedule will always preserve similarity in the output. The definition of regular similarity only requires a similarity relation to be preserved by every transaction, so that the input value of a transaction can be swapped with another in a schedule if the two values are related by a regular similarity relation. Unless a similarity relation is also transitive, in which case it is an equivalence relation, it is in general incorrect to swap events an arbitrary number of times for many transactions in a schedule. The notion of strong similarity is motivated by the observation that the state information of many real-time systems is “volatile”, i.e., they are designed in such a way that system state is determined completely by the history of the recent past, e.g., the velocity and acceleration of a vehicle are computed from the last several values of the vehicle’s position from the position sensor. Unless events in a schedule may be swapped in such a way that a transaction reads a value that is derived from the composition of a long chain of transactions that extends way into the past, a suitable similarity relation may be chosen such that output similarity is preserved by limiting the “distance” between inputs that may be read by a transaction before and after swapping similar events in a schedule. For the purpose of this section, it suffices to note that if two events in a schedule are strongly similar (i.e., they are either both writes or both reads, and the two data values involved are strongly similar), then they can always be swapped in a schedule without violating data consistency requirements.

Specifically, we assume that the application semantics allows us to derive a similarity bound for each data object such that two write events on the data object must be strongly similar if their time-stamps differ by an amount no greater than the similarity bound, i.e., all instances of write events on the same object that occur in any interval shorter than the similarity bound can be swapped in the (untimed) schedule without violating consistency requirements.

The basic strategy of the Similarity Stack Protocol (SSP) [8] can be summarized as follows: Transactions are normally scheduled according to their priorities which can be dynamic (e.g., earliest-deadline-first algorithm) or static (e.g., as determined by the rate monotonic assignment algorithm), with the provision that transaction execution follows the stack discipline, i.e., if transaction $\tau$ starts after transaction $\tau'$ then $\tau'$ cannot resume until after $\tau$ finishes. However, no transaction is allowed to start execution if it conflicts with another transaction that has already started but not committed such that the conflicting read/write
events may not be arbitrarily swapped under the similarity relation. Note that conflicting transactions cannot block one another as long as their event conflicts can be resolved by appealing to the similarity bound. To prevent deadlocks and unlimited blocking, however, the SSP protocol does not allow a transaction to start before all higher-priority transactions have completed.

Conceptually, we shall assume the existence of a stack for each processor. We call it the preemption stack of the processor. When a transaction instance is scheduled on a processor, it is pushed onto the top of the preemption stack. At any time, the transaction instance at the top of a preemption stack is the one being executed on the processor. A transaction instance preempts another transaction instance by being pushed on top of the latter on the preemption stack. When a transaction instance commits, it is popped off from the top of its preemption stack.

To increase concurrency, we adopt the atomic data set concept of [16] and partition the transactions into disjoint subsets which we call interactive sets such that no two transactions from different interactive sets may conflict. However, we do not require transactions in an interactive set to run exclusively on one processor. The key idea of the SSP protocol is to use the preemption stacks to restrict the maximum time interval spanning two conflicting transaction instances that may overlap in their execution. Since the preemption stack contains all the transactions that have started execution but not committed, enforcing a bound on the temporal depth of the preemption stacks achieves the desired effect. For each preemption stack, a different integer $accu_{ij}$ for every interactive set is adopted as a measure of the temporal depth of the $i_{th}$ stack for the $j_{th}$ interactive set, where the $i_{th}$ stack is for the $i_{th}$ processor.

For example, suppose that $\pi$ is a schedule of three transactions $\tau_1$, $\tau_2$, and $\tau_3$ on one processor as shown in the above figure. Let $\tau_1$ and $\tau_2$ be in the first interactive set and $\tau_3$ be in the second interactive set. That is, $\tau_1 (/\tau_2)$ and $\tau_3$ are independent in data access. Suppose $\tau_1$, $\tau_2$, and $\tau_3$ have computation requirements 1, 2, and 3, respectively. As shown in the above figure, $\tau_3$ which starts execution after $\tau_2$ committed is running at time $t$. Since $\tau_1$ is preempted by $\tau_2$ and has not committed, there is one scheduled but uncommitted transaction instance in the first interactive set on the first processor. Therefore, the temporal...
We now derive bounds called R-recency & W-recency bounds to limit the temporal depth $accu_{ij}$ of the $i_{th}$ stack for the $j_{th}$ interactive set: Suppose $sb_i$ is a similarity bound for data object $O_i$ so that any two writes on $O_i$ within an interval shorter than $sb_i$ are interchangeable and strongly similar. We define a W-recency bound $w_i$ which is the upper bound to be enforced by the SSP protocol on the temporal distance between two write events on $O_i$ which belong to overlapping transaction instances. Obviously, the choice of $w_i$ is constrained by $w_i \leq sb_i$. We also define a R-recency bound $\delta_i$ which is the upper bound to be enforced by the SSP protocol on the temporal distance between a read and a write event on $O_i$ which belong to overlapping and conflicting transaction instances. Suppose the read and write events in question are $r_i$ and $w_i$, and $\tau_i$ occurs before $w_i$. If we want to swap $r_i$ and $w_i$ without violating the correctness criterion, the write event, call it $w'_i$ that $r_i$ reads from must be sufficiently close to $w_i$ so that these two write events are strongly similar. Specifically, if there is an update transaction on $O_i$ which is scheduled at least once every $p_i$ time units, then the two write events are strongly similar if the distance between the two write events $w_i$ and $w'_i$ is at most $sb_i$ and $sb_i \geq \delta_i + 2p_i$. (Here, we assume that in the absence of additional application information, the update transaction may actually perform the write operation on $O_i$ anywhere inside a period of $p_i$ time units, and hence a read event may read from a write event almost $2p_i$ time units away.) Since we assume that the maximum distance between a read event $r_i$ and the write event $w'_i$ it reads from is $2p_i$, we have $sb_i \geq \delta_i + 2p_i$. Hence, $\delta_i$ is constrained by $\delta_i \leq sb_i - 2p_i$.

The limitation of the temporal depth bound $accu_{ij}$ can be done simply by blocking any transaction whose execution may cause the sum $(accu_{ij} + accu_{ik})$ to exceed $w_j$ or $\delta_j$ for some $i$ and $k$. Only intuitive way to maintain the above
condition is to keep each $a_{i,j}$ to not exceed $\beta = \min\left(\frac{\delta_i}{T}, \frac{\omega_j}{T}\right)$ so that the stacks cannot grow unexpectedly in the future. We call $\beta$ as the recency bound. We refer interested readers to [S, 9] for details and extension.

**EXAMPLE 4 A SSP Schedule**

We illustrate the working of SSP by an example [8]. Suppose there are four transactions $\tau_1$, $\tau_2$, $\tau_3$, and $\tau_4$ in a two-processor environment. Let $\tau_1$, $\tau_2$, $\tau_3$, and $\tau_4$ have computation requirements 1, 2, 1, and 2, respectively, and let them have the same period 5. Suppose $\tau_1$ conflicts with $\tau_2$, $\tau_2$ conflicts with $\tau_3$, and $\tau_3$ conflicts with $\tau_4$. According to the Similarity Stack Policy (SSP), these four transactions are in the same interactive set. For a typical similarity bound, we assume that the application permits data updates to be missed by one cycle, as is the case in many avionic software systems. As shown in the above figure, this can be taken to mean that data values shall remain valid for three periods. Thus, we can reasonably assume that a similarity bound of this interactive set is 15. The recency bound $\beta$ of their interactive set is then 2.5 from the formulas for W-recency and R-recency. As a result, we allow $\tau_1$ ($/\tau_3$) and $\tau_2$ run concurrently on different processors, as they are scheduled by RM (or even EDF).

The SSP protocol was shown being deadlock-free and guarantees at most one priority inversion for every real-time transaction. There are at most two context switches counted for each transaction instance. The correctness of transactions scheduled by SSP is validated by the notion of the similarity:

**Definition:** [7] View Similar:

Two schedules are view-similar if and only if

1. They are over the same set of transaction instances.
2. They transform (strongly) similar initial database states into similar database states under any interpretation.

3. Every transaction instance has similar views in both schedules for any initial state and under any interpretation.

A schedule is view $\Delta$-serializable iff it is view-similar to a serial schedule. It can be shown that all SSP schedules are view $\Delta$-serializable [8]. In other words, all SSP schedules are similar to serial schedules. Note that if a schedule is view-equivalent to another schedule, then it is view-similar to that schedule, but the converse may not hold. The view-similarity relation between schedules is reflexive and symmetric but not necessarily transitive.

The similarity-based concurrency control has been extended in several ways: (1) The concept of similarity was further explored to provide a sufficient condition for achieving data synchronization for free [9]. In other words, real-time transactions satisfying this condition can be scheduled correctly by any process scheduling discipline designed for the independent processes model, e.g., the rate monotonic scheduling algorithm (RM), earliest deadline scheduling algorithm (EDF), where no locking of data is assumed. The correctness of our approach can be justified by $\Delta$-serializability. (2) A distributed real-time, data-access interface, Real-Time Object Management Interface (RTOMI), was built on an Intel multiprocessor computer [10]. It was designed to facilitate the implementation of real-time, data-intensive applications and provide a testbed for the methodologies of process scheduling and data distribution based on the similarity concept. (3) The definition of similarity was combined with the idea of transaction skipping to provide a theoretical foundation for reducing the workload of a real-time transaction system [3]. Guidelines were also proposed to adjust the execution frequencies of a static set of transactions, and their correctness was proved. The strengths of the work were then verified by simulation experiments on an air traffic control example. We refer interested readers to [3, 9, 10] for details.

4. EPSILON-BASED CONCURRENCY CONTROL

Epsilon-serializability (ESR) is a generalization of classic serializability (SR) by allowing some limited amount of inconsistency in transaction processing. Similar to data similarity, ESR enhances concurrency since some non-SR execution schedules are permitted. For example, some epsilon-transactions (ET’s) that just perform queries may execute in spite of ongoing concurrent updates to the database. Thus the query ET’s may view uncommitted, i.e., possibly inconsistent, data. Concretely, an update transaction may export some

---

1The contents of this section is a summary of [6, 20]. We refer interested readers to [6, 20] for details.
inconsistency when it updates a data item while query ET’s are in progress. Conversely, a query ET may import some inconsistency when it reads a data item while uncommitted updates on that data item exist. The correctness notion in ESR is based on bounding the amount of imported and exported inconsistency for each ET.

A database is defined as a set of data items. Each data item may carry a value. A database state is the set of all data values. A database state space is the set of all possible database states. A distance function \( \text{distance}(u,v) \) is defined over every pair of states \( u \) and \( v \) where the distance function is the absolute value of the difference between two states (on real numbers). For example, the distance (of an account item) between $40 and $70 is $30. Thus, if the current account balance is $40, and $30 is credited, the distance between the new state and the old state is $30.

ESR defines correctness for both consistent states and inconsistent states. In case of consistent states, ESR reduces to classic serializability. In addition, ESR associates an amount of inconsistency with each inconsistency state, defined by its distance from a consistent state. Informally, inconsistency in a data item \( x \) with respect to a query \( q \) is defined as the difference between the current value of \( x \) and the value of \( x \) if no updates on \( x \) were allowed to execute concurrently with \( q \). A query \( q \) imports inconsistency when it views, i.e., reads, an inconsistent data item. Conversely, an update transaction exports inconsistency when it updates, i.e., writes to, a data item while query ET’s that read the data item are in progress.

An application designer specifies the limit for each ET and the transaction processing (TP) system ensures that these limit are not exceeded during the execution of the ET. For example, a bank may wish to know how many millions of dollars there are in the checking accounts. If this query was executed directly on the checking accounts during the banking hours, serious interference would arise because of updates. Most of the updates are irrelevant since typical updates refer to small amounts, compared to the query output units, which is in millions of dollars. Hence, we must be able to execute the query during banking hours. Specifically, under ESR, if we specify an import-limit for the query ET, for example, of $100,000, for this query, the result also would be guaranteed to be within $100,000 of a consistent value (produced by a serial execution of the same transactions). For example, if the ET returns the value $357,215,000 then at least one if the serial transaction executions would have yielded a serializably query result in the $325,215,000 ± $100,000 interval.

Since a query does not update data, it does not affect the permanent state of the database. Furthermore, Ramamritham, et al. [20] have assumed that updates do not import inconsistency, i.e., they operate on consistent database states. Thus, assuming that each update ET maintains database consistency, updates do not affect the consistency of the database. The only effect of the
updates is on the inconsistency of the data read by queries. ESR is mainly based on eventual consistency, in which data consistency is guaranteed by serializability, and it formalizes the query behavior by deriving the formulae that express the inconsistency in the data values read by a query. A schedule is epsilon-serializable only if it is serializable after all of its queries are removed. To limit importing inconsistency and/or exporting inconsistency of transactions, epsilon-serializability relies on the maximum allowable "changes" per transaction, especially in forward divergence control which prevents inconsistency violation from occurring at all. Kamath and Ramamritham [6] later introduced the idea of hierarchical inconsistency bounds that allows inconsistency to be specified at different granularities such as transactions and objects. They provided mechanisms to control the inconsistency and reported the evaluation of the performance improvement due to ESR.

5. SUMMARY AND FUTURE WORK

Real-time concurrency control has been an active research topic in the past decades. The notion of serializability is often too strict a correctness criterion for many real-time application systems. The relaxation of correctness criteria not only helps the deployment of real-time database applications in various environments and application domains but also provides the performance improvement which is otherwise impossible for many real-time applications. Tailoring a real-time database system to cater to the needs of a wide variety of users and achieve a better utilization of resources is very important in many real-time applications. An interesting extension to the past work is to characterize the domain-specific semantics of data in application systems, such as multimedia, real-time knowledge base, advanced communication and control systems, and to let engineers investigate the structure of a real-time database for efficient semantics-based data accesses.

References


This page intentionally left blank.
Chapter 6

REAL-TIME INDEX CONCURRENCY CONTROL

Jayant R. Haritsa
Database Systems Lab, Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore 560012, INDIA.
haritsa@dsl.serc.iisc.ernet.in

S. Seshadri
Lucent Bell Labs
600 Mountain Avenue, Murray Hill, NJ 07974, USA.
seshadri@research.bell-labs.com

1. INTRODUCTION

Database systems implement indexes to efficiently locate data items on disk and thereby significantly improve query response times. While indexes are an integral component of traditional DBMS, they assume even greater significance in the real-time environment due to the stringent time constraints placed on transaction completion. Therefore, it can be reasonably expected that well-designed RTDBS will make extensive use of indexes to enhance their ability to meet transaction deadlines.

A practical example of the above scenario is the following: The rapid spread of cellular communication technology has resulted in database systems having to support environments with a large of number of highly mobile users [12]. Due to the users mobility, the spatio-temporal characteristics of the answers, and the need to conserve battery power, these mobile interactions with the database system have to be processed in real-time and indexes will be essential for quickly processing frequent operations such as location updates (“I am here”) and location queries (“Where is X”).

For RTDBS supporting high transaction processing rates, the contention among transactions concurrently using the index may itself form a performance bottleneck. In particular, note that while contention for the physical resources...
can always be reduced by purchasing more and/or faster hardware, there exists no equally simple mechanism to reduce index contention since the base data, and hence indexes, cannot be “manufactured”. Similarly, even in environments where the contention on base data is largely absent due to transactions accessing mostly disjoint sets of data, index contention may continue to pose performance problems since a common access structure is used to access this base data. Therefore, in this sense, index contention is a more “fundamental” determinant of RTDBS performance as compared to other shared system resources.

From the above discussion, it is clear that a pre-requisite for RTDBS to realize the benefits of indexes is incorporation of index concurrency control (ICC) protocols that are tuned to the objectives of the real-time environment. However, while a rich body of ICC protocols is available for traditional DBMS, and RTDBS design has been investigated for over a decade, research into real-time ICC protocols is not only very sparse but also only of recent vintage. This lacuna is rather surprising since, as mentioned earlier, it appears highly reasonable to expect that RTDBS will make extensive use of indexes to quickly access the base data and thereby help more transactions meet their deadlines. In the remainder of this chapter, we describe our efforts, which represent the first work in the area, to address this lacuna.

Our study is carried out in the context of real-time applications with “firm-deadlines” [8] – for such applications, completing a transaction after its has expired is of no utility and may even be harmful. Therefore, transactions that miss their deadlines are considered to be worthless and are immediately “killed” – that is, aborted and permanently discarded from the RTDBS without being executed to completion. Accordingly, the performance metric is KillPercent, the steady-state percentage of killed transactions.¹

2. B-TREE INDEXES AND PROTOCOLS

While a large variety of index structures have been proposed in the literature, commercial database systems typically use B-tree indexing [3] as the preferred access method. In particular, they implement the B⁺ variant in which all data key values are stored at the leaf nodes of the index; hereafter, our usage of the term B-tree refers to this variant.

The transactional operations associated with B-trees are search, insert, delete and append of key values. Search, insert and delete are the traditional index operations. The append operation is a special case of the insert operation wherein the inserted key is larger than the maximum key value currently in the index. In this case, the new key will always be inserted in the rightmost leaf node of the B-tree. The distinction between appends and generic inserts

¹Or, equivalently, the percentage of missed deadlines.
is important since when appends are frequent\footnote{For example, when tuples are inserted into a relation in index key order.} the rightmost leaf node of the index effectively becomes a “hot-spot”. Finally, note that search operations arise out of transaction reads while the insert, delete and append operations arise out of transaction updates.

The basic maintenance operations on a B-tree are split and merge of index nodes. In practical systems, splits are initiated when a node overflows while merges are initiated when a node becomes empty. An index node is considered to be safe for an insert if it is not full and safe for a delete if it has more than one entry (index nodes are always safe for searches since they do not modify the index structure). A split or a merge of a leaf node propagates up the tree to the closest (with respect to the leaf) safe node in the path from the root to this leaf. If all nodes from the root to the leaf are unsafe, the tree increases or decreases in height. The set of nodes that are modified in an insert or delete operation is called the scope of the update.

B-tree ICC protocols maintain index consistency in the face of concurrent transaction accesses and updates. This is achieved through the use of locks on index nodes. Index node locks are typically implemented in commercial DBMS using latches, which are “fast locks”\cite{17}. An important aspect of latches is that deadlocks involving latches are not detected and ICC protocols have to therefore ensure that latch deadlocks can never occur.

The index lock modes discussed in this chapter and their compatibility relationships are given in Table 6.1. In this table, IS, IX, SIX and X are the standard “intention share”, “intention exclusive”, “share and intention exclusive” and “exclusive” locks, respectively\cite{5}.

<table>
<thead>
<tr>
<th>mode</th>
<th>IS</th>
<th>IX</th>
<th>SIX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>IX</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>SIX</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>X</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 6.1 Index Node Lock Compatibility Table

Some B-tree ICC protocols use a technique called lock-coupling in their descent from the root to the leaf. An operation is said to lock-couple when it requests a lock on an index node while already holding a lock on the node’s parent. If the new node is found to be safe, all locks held on any ancestor are released. A variant of this technique is optimistic lock-coupling, wherein,
regardless of safety, the lock at each level of the tree is released as soon as the appropriate child has been locked.

There are three well-known classes of B-tree ICC protocols: Bayer-Schkolnick, Top-Down and B-link. Each class has several flavors and we discuss only a representative set here.

2.1 BAYER-SCHKOLNICK PROTOCOLS

We consider three protocols in the Bayer-Schkolnick class [2] called B-X, B-SIX and B-OPT, respectively. In all these protocols, readers descend from the root to the leaf using optimistic lock-coupling with IS locks. Their update protocols, however, differ: In B-X, updaters lock-couple from the root to the leaf using X locks. In B-SIX, updaters lock-couple using SIX locks in their descent to the leaf. On reaching the leaf, the SIX locks in the updaters scope are converted to X locks. In B-OPT, updaters make an optimistic lock-coupling descent to the leaf using IX locks. After the descent, updaters obtain a X lock at the leaf level and complete the update if the leaf is safe. Otherwise, the update operation is restarted, this time using SIX locks.

2.2 TOP-DOWN PROTOCOLS

In the Top-Down class of protocols (e.g. [20, 15]), readers use the same locking strategy as that of the Bayer-Schkolnick protocols. Updaters, however, perform preparatory splits and merges during their index descent: If an inserter encounters a full node it performs a preparatory node split while a deleter merges nodes that have only a single entry. This means that unlike updaters in the Bayer-Schkolnick protocols who essentially update the entire scope at one time, the scope update in Top-Down protocols is split into several smaller, atomic operations. In particular, a lock on a node can be released once its child is either found to be safe or made safe by the preparatory split or merge mentioned above.

We consider three protocols in the Top-Down class called TD-X, TD-SIX and TD-OPT, respectively: In TD-X, updaters lock-couple from the root to the leaf using X locks. In TD-SIX, updaters lock-couple using SIX locks. These locks are converted to X-locks if a split or merge is made. In TD-OPT, updaters optimistically lock-couple using IX locks in their descent to the leaf and then get an X lock on the leaf. If the leaf is unsafe, the update operation is restarted from the index root, using SIX locks for the descent.

2.3 B-LINK PROTOCOLS

A B-link tree [16, 22, 15] is a modification of the B-tree that uses links to chain together all nodes at each level of the B-tree. Specifically, each node in a B-link tree additionally contains a high key (the highest key of the subtree
rooted at this node) and a link to the right sibling, which are used during a node split or a node merge. A node is split in two phases: a half-split, followed by the insertion of an index entry into the appropriate parent. Operations arriving at a newly split node with a search key greater than the high key use the right link to get to the appropriate node. Such a sideways traversal is called a link-chase. Merges are also done in two steps [15], via a half-merge followed by the appropriate entry deletion at the next higher level.

In B-link ICC protocols, readers and updaters do not lock-couple during their tree descent. Instead, readers descend the tree using IS locks, releasing each lock before getting a lock on the next node. Updaters also behave like readers until they reach the appropriate leaf node. On reaching the leaf, updaters release their IS lock and then try to get an X lock on the same leaf. After the X lock is granted, they may either find that the leaf is the correct one to update or they have to perform link-chases to get to the correct leaf. Updaters use X locks while performing all further link chases, releasing the X lock on a node before asking for the next. If a node split or merge is necessary, updaters perform a half-split or half-merge. They then release the X lock on the leaf and propagate the updates, using X locks, to the higher levels of the tree, holding at most one X lock at a time.

As discussed above, B-link protocols limit each sub-operation to nodes at a single level. This is in contrast to Top-Down protocols which break down scope updating into sub-operations that involve nodes at two adjacent levels of the tree. B-link protocols also differ from Top-Down protocols in that they do their updates in a bottom-up manner. We consider only one B-link protocol here, which exactly implements the above description. This protocol is referred to as the LY protocol in [23], and was found to have the best performance of all the above-mentioned protocols with respect to the conventional DBMS metric of transaction throughput.

3. REAL-TIME INDEX CONCURRENCY CONTROL

In the previous section, we described the functioning of the various classical ICC protocols. We now move on to considering the major issues that arise when incorporating these protocols in an RTDBS environment. In particular, we discuss how real-time priorities are incorporated, how transaction serializability is ensured and, finally, how the undos of the index actions of aborted transactions are implemented.

3.1 PRIORITY INCORPORATION

Since satisfaction of transaction timing constraints is the primary goal in RTDBS, transactions are usually assigned priorities to reflect the system resources each transaction’s urgency relative to other concurrently executing
transactions – the Earliest Deadline policy [14], for example, assigns priorities in the order of transaction deadlines.

The ICC protocols described in the previous section do not take transaction priorities into account. This may result in high priority transactions being blocked by low priority transactions, a phenomenon known as priority inversion in the real-time literature [24]. Priority inversion can cause the affected high-priority transactions to miss their deadlines and is clearly undesirable. We therefore need to design priority-cognizant schemes for ICC protocols in order to adapt them to the real-time environment.

Latches (i.e., locks on index nodes) are typically held only for a short duration, in contrast to data locks which are usually retained until EOT. Therefore, it may be questioned as to whether adding priority-cognizance to latches is really necessary. The reason this can make a difference is the following: The time for which computation is performed on a node is usually much shorter than the time it takes to retrieve a node from disk. Given this observation, if two transactions conflict on a parent node but require access to different children, then, in the absence of priority-cognizance, a higher priority transaction may have to wait for a lower priority transaction while this lower priority transaction retrieves its desired child from the disk. This delay can prove to be significant, especially in resource-constrained environments, as confirmed in our experiments.

3.1.1 Priority Preemption and Priority Inheritance. Priority can be incorporated into the ICC protocols in the following manner: When a transaction requests a lock on an index node that is held by higher priority transactions in a conflicting lock mode, the requesting transaction waits for the node to be released (the wait queue for an index node is maintained in priority order). On the other hand, if the index node is currently held by only lower priority transactions in a conflicting lock mode, the lower priority transactions are pre-empted and the requesting transaction is awarded the lock. The lower priority transactions then re-perform their current index operation (not the entire transaction).

As an alternative to the above “priority pre-emption”-based approach, we could also consider techniques based on “priority inheritance” [24]. Here, low priority transactions that block a high priority transaction inherit the priority of the high priority transaction. The expectation is that the blocking time of the high priority transaction will be reduced since the low priority transactions will now execute faster and therefore release their index nodes earlier.

The positive feature of the inheritance-based approach is that, unlike pre-emption, it does not run the risk of operation restart and therefore does not waste resources. However, a serious limitation is that due to the nature of index traversal, it is possible that a high-priority transaction may be blocked at multiple nodes during its index processing, the cumulative effect of which
Real-Time Index Concurrency Control

may result in a missed deadline. We, therefore, utilize the priority preemption approach in our study.

3.1.2 Physical Updates. Independent of whether priority-pre-emption or priority inheritance is used to address the priority inversion discussed above, we recommend a different procedure for the special case where a low priority transaction is in the midst of physically making updates to a node that is currently locked by it. In this situation, it appears best to allow the transaction to just complete these updates and release the lock on the updated node in the normal manner, without taking any special action. The reasons are the following:

- A transaction starts making a physical update only after all the required nodes are in memory. Therefore, no disk accesses are required during the updation process, resulting in swift completion. This means that the priority inversion period is very small, having only a negligible performance effect.
- If priority preemption is employed, the physical updates that have already been made have to be undone, which may require commensurate or more effort as compared to just allowing the update to complete.
- On the other hand, if priority inheritance is used, the protocol code will become significantly more complex and therefore slower to execute, negating the real-time objectives.

3.2 TRANSACTION SERIALIZABILITY

The KillPercent performance metric applies to entire transactions, not to individual index actions. We therefore need to consider transactions which consist of multiple index actions and ensure that transaction serializability is maintained.

In conventional DBMS, a popular mechanism for providing transaction data concurrency control (DCC) is the well-known ARIES Next-Key-Locking protocol [18]. We now describe as to how this protocol could be adapted to the real-time domain. In the following discussion, we use the term “key” to refer to key values in the index and the term “next key” (with respect to a key value $k$) to denote the smallest key value in the index that is $\geq k$.

Our variant of the Next-Key-Locking protocol works as follows: A transaction that needs to perform an index operation with respect to a specific key (or key range) first descends the tree to the corresponding leaf. It then obtains from the database concurrency control manager, the appropriate lock(s) on the associated key(s). For point searches an S lock is requested on the search key value (or the associated next key, if the search key is not present). A range
search operation has to acquire S locks on each key that it returns. In addition, it also acquires an S lock on the next key with respect to the last (largest) key in the accessed range. Inserts acquire an X (exclusive) lock on the next key with respect to the inserted key, acquire an X lock on the key to be inserted, insert the key, and then release the lock on the next key. Deletes, on the other hand, acquire an X lock on the next key with respect to the deleted key, acquire an X lock on the key being deleted, delete the key, and then release the lock on the deleted key.

In traditional Next-Key-Locking, all the above-mentioned locks (unless otherwise indicated) are acquired as needed and released only at the end of the transaction (i.e., strict 2PL). In the RTDBS domain, however, a real-time version of 2PL should be used – in our study, we use 2PL High Priority (2PL-HP) [1] which incorporates a priority pre-emption mechanism similar to that described above for the ICC protocols. An important difference, however, is that transactions restarted due to key-value-lock preemptions have to commence the entire transaction once again, not just the current index operation.

Earlier studies of DCC protocols in firm RTDBS (e.g. [8]) have shown optimistic algorithms to usually perform better than locking protocols, especially in environments with significant data contention. In light of this, our choice of 2PL-HP may seem surprising – the rationale is that open problems remain with respect to integrating optimistic DCC schemes with index management [7, 19], making them currently infeasible for incorporation in complete database systems.

3.3 UNDO TRANSACTIONS

Real-time transactions may be aborted due to priority resolution of data conflicts (the 2PL-HP protocol mentioned above), or due to missing their deadlines and therefore being killed. This results in transaction aborts being much more common in RTDBS than in conventional DBMS. (In a conventional DBMS, aborts are usually resorted to only to resolve data deadlocks, and deadlocks occur very infrequently in practice [26].)

For aborted transactions, it is necessary to undo any effects they may have had on the index structure; since aborts are common, as mentioned above, the processing of these undos may have a significant performance impact. Therefore, the undos of the index actions of aborted transactions must be considered in the design of real-time ICC protocols.

We hereafter use the term undo transaction to refer to transactions that require undoing of their index actions. The undo transaction removes all the changes that the original transaction had made on the index structure and also releases all its data locks. Note that is imperative to complete the undo transactions quickly since regular index operations cannot proceed with an
inconsistent index. To ensure this, undo transactions are treated as “golden” transactions, that is, they are assigned higher priority than all other transactions executing in the system. Among the undo transactions, the relative priority ordering is the same as that which was existing between them during their earlier (pre-abort) processing.

4. PERFORMANCE RESULTS

Using a detailed simulation model of a firm-deadline RTDBS, we evaluated the performance of the various real-time ICC protocols over a wide range of index contention environments. The full description of our experiments and results are available in [11] – here, we restrict our attention to three representative workloads: **Low Index Contention (LIC)** – 80% searches, 10% inserts and 10% deletes; **Moderate Index Contention (MIC)** – 100% inserts; and **High Index Contention (HIC)** – 25% searches and 75% appends.

Apart from the primary performance metric of *KillPercent*, two additional performance metrics related to fairness were also evaluated: *SizeFairness* and *TypeFairness*. The *SizeFairness* factor captures the extent to which bias is exhibited towards transactions based on their sizes (number of data items accessed). The *TypeFairness* factor captures the extent to which bias is exhibited towards transactions based on their type. Two types of transactions were considered: *read-only* and *update* – read-only transactions are composed exclusively of search index actions while update transactions include at least one update (insert, delete or append) index action.

Our experimental results showed that index conflict has a considerable impact on the real-time performance, *highlighting the need for designing sophisticated real-time ICC protocols*. Further, two factors characteristic of the (firm) real-time domain: Addition of priority and discarding of late transactions, significantly affected the performance of the ICC protocols, as compared to their behavior in conventional DBMS [23]. This outcome justifies the need to re-evaluate ICC protocol behavior in the real-time environment.

More specifically, the following observations were made:

- Adding preemption to index latches resulted in tangible performance benefits, in accordance with our expectations (Section 3.).

- No appreciable difference was observed between the performance of the corresponding protocols from the Bayer-Schkolnick and Top-Down classes (i.e., between B-X and TD-X, B-SIX and TD-SIX, B-OPT and TD-OPT). This is because the number of exclusive locks held at one time on the scope of an update is hardly different for the two classes for trees with few levels and large fan-outs, which is typically how B-trees are organized. We will therefore simply use X, SIX and OPT to denote these protocols in the sequel.
All the protocols were noticeably unfair with regard to one or both fairness metrics. For example, B-link and OPT progressively favor smaller-sized transactions with increasing loading levels, while SIX shows extreme bias favoring read-only transactions.

The X protocol performed very poorly with respect to the other protocols, due to the root of the B-tree becoming a severe bottleneck.

Surprisingly, B-link missed many more deadlines under heavy loads as compared to lock-coupling protocols. This is in contrast to conventional DBMS where B-link always exhibited the best throughput performance [23]. The reason for the altered performance in the real-time environment is the following: Algorithms such as SIX apply a form of load control by giving preferential treatment to read-only transactions. B-link, on the other hand, tends to saturate the disk due to largely treating transactions uniformly independent of type and therefore misses significantly more deadlines. That is, the very reason for B-link’s good performance in conventional DBMS – full resource utilization – turns out to be a liability here.

An additional factor is that in conventional DBMS, the slower updaters of SIX clogged the system, resulting in much higher contention levels and poor performance – in the firm real-time environment, however, that does not happen because transactions are discarded as soon as their deadlines expire. Therefore, this type of clogging is inherently prevented.

The OPT protocol performed almost as well as B-link even under HIC conditions. This is also in contrast to conventional DBMS, where OPT performed poorly in this scenario – a large number of restarts were caused by high contention, resulting in the root becoming a bottleneck due to the large number of second-pass updaters. In the real-time environment, however, prioritization of transactions causes a marked decrease in the number of index operation restarts, as explained below.

In the no priority case, several first pass transactions see the same unsafe leaf node before the first transaction which saw it as unsafe has completed its second pass and made the leaf safe (by splitting or merging). This is the source of the large number of restarts. In the prioritized environment, however, the highest priority transaction overtakes other transactions at the nodes it traverses during its descent and therefore completes its second pass very quickly. This results in only relatively few other transactions seeing the leaf node while it is unsafe. In essence, the time period for which a node is unsafe is much smaller in the prioritized environment as compared to the non-real-time environment.
4.1 EFFECT OF UNDOS

We now move on to considering the performance impact of modeling the undos of the index actions of aborted transactions. It is tempting to surmise that there is really no need to explicitly evaluate this feature based on the following argument, which was advanced in [4]: The number of undos can be expected to be directly related to the transaction kill percentage, that is, a higher kill percentage leads to more undos. Given this, implementing undos in the model would only result in further increasing the difference in performance of the protocols seen in the above experiments. This is because the poorly performing algorithms would have more clean-up work to do than the better protocols and therefore miss even more transaction deadlines. In summary, there is a direct positive feedback between the undo overhead and the kill percentage. Due to this relationship, incorporating undo actions will correspondingly increase, but not qualitatively alter, the observed performance differences between the various protocols.

While the above argument may appear plausible at first sight, our study showed that the impact of undos is not so simply characterized and that unexpected effects can show up in certain environments.

Firstly, under LIC and MIC environments, undos had only a marginal adverse impact on performance. Further, this marginal impact was not limited to light loads – it also occurred for high kill percentage values where we would expect the undo overheads to be considerable due to the large number of aborted transactions.

Even more interestingly, the protocols often performed better with undos than in the absence of undos, especially in the lower range of the loading spectrum! While the improvement is typically small, for SIX a significant improvement was observed. These results, which suggest that the presence of overheads helps to improve performance may appear at first glance to be illogical. A careful investigation showed, however, that there are subtle effects that come into play in the undo model, which do indeed cause this apparently strange behavior. We explain these reasons below.

1. The primary reason for little adverse performance impact in the LIC and MIC environments is that the disk is the bottleneck in these experiments. Undo operations, however, typically only need the CPU since the index pages they require will usually be already resident in the buffer pool as they have been accessed recently. In essence, undo operations primarily impose a CPU overhead, not a disk overhead. This means that only when the CPU itself is a bottleneck, as in the HIC environment, are there the large differences that we might intuitively expect to see.

2. The number of index node splits and merges reduces significantly in the presence of undos. This is because they compensate for the actions of
aborted transactions. This is especially significant in light of the fact that index node splits and merges are expensive operations since they necessitate additional disk I/O.

3. The reason for SIX performing so much better with undos at low arrival rates in the LIC environment is that the average latch wait time for the normal operations is greatly reduced by the presence of undo operations. This is explained as follows: In SIX, readers usually descend the tree very quickly since they use IS locks which are compatible with the SIX locks of updaters. Updaters, on the other hand, have to wait for each other since SIX locks are mutually incompatible. Due to the lock-coupling nature of SIX, this may result in a convoy phenomenon, wherein there is a chain of updaters from the leaf to the root, each holding a latch that the next one wants. Since index trees are usually of small height, such convoys can occur very easily. Moreover, the convoy may persist for quite a while since the updater on the leaf node typically has to wait for the leaf node to be brought from the disk before it can release its latch on the parent node (and that too, only if the leaf is safe).

The presence of undo transactions eliminates the convoy bottlenecks described above. This is because, by virtue of their “golden” priority, they preempt during undo processing all transactions holding an incompatible latch in their tree descent. In particular, they preempt updaters holding leaf level latches and waiting for their disk requests to be completed. This eliminates the convoy bottleneck and allows other updaters to gain possession of the higher level latches and quickly descend to the leaf nodes, especially since latching times are very small as compared to disk access times.

The benefit of convoy elimination is felt only under light and moderate loads but not under heavy loads. This is because, under heavy loads, the disk is almost fully utilized and becomes the primary bottleneck. In this situation, removing the latching bottleneck proves ineffective and is actually harmful since it increases the amount of wasted work due to the large number of preemptions.

4.2 RANGE QUERIES

For range queries, we observed that the results are qualitatively similar to those seen with point queries. Quantitatively, however, the performance differences between the protocols generally increased. The reason for this increase is that range searches significantly increase the level of data contention in the system since they hold locks on several key values simultaneously.
5. THE GUARD-LINK PROTOCOL

One of the outcomes of the results presented above is that it appeared that the performance of B-link could be improved by adding a load-control component. Accordingly, we designed a new protocol called GUARD-link, which augments the real-time variant of the B-link protocol with a feedback-based admission control mechanism called GUARD (Gatekeeping Using Adaptive earliest Deadline). The details of the algorithm are in [11].

On evaluation, we found that GUARD-link, by virtue of its admission control policy which successfully limits transaction admissions to a sustainable number, significantly reduced the kill percentage of B-link in the overload region and thereby provided the best performance over the entire loading range. This clearly demonstrates the need for admission control in real-time index management. Interestingly, such need for load control has also been identified in other RTDBS modules (e.g. [9, 21]).

Apart from its good MissPercent performance, GUARD-link also provided close to ideal fairness, not discriminating either based on transaction size or on transaction type. Therefore, we suggest that RTDBS designers may find the GUARD-link protocol to be a good choice for real-time index concurrency control.

6. RELATED WORK

The material discussed in this chapter is based on our earlier papers [4, 10, 11]. Apart from this, the only other work that we are aware of on real-time ICC appeared very recently in [13]. While our focus was on improving the real-time performance of classical ICC protocols by suitable use of priority and admission-control mechanisms for both point and range queries, the emphasis in [13] is on “relaxing” traditional index structures in a manner that improves timeliness for batch queries, where several keys are provided at a time. In particular, a B-tree that is re-balanced only periodically, but not continuously, is considered. Specialized techniques to increase the buffer hit ratio across transactions and to directly access the index from internal nodes are evaluated and shown to provide improved real-time performance.

7. SUMMARY AND FUTURE WORK

In this chapter, we have attempted to characterize the state-of-the-art with regard to real-time index processing. This topic is of importance since it appears highly reasonable to expect that RTDBS will make extensive use of indexes to quickly access the base data and thereby help more transactions meet their deadlines. We studied the real-time performance of the Bayer-Schkolnick, Top-Down and B-link classes of ICC protocols under a range
of workloads and operating conditions. We also proposed and evaluated a new ICC protocol called GUARD-link, which augments the classical B-link protocol with an admission control mechanism. The GUARD-link protocol provided the best real-time performance in terms of both the number of missed deadlines and the fairness with respect to the distribution of these missed deadlines across transaction classes. Finally, we identified some unexpected performance outcomes of executing the undos of index actions of aborted transactions.

We now outline a variety of issues that appear appropriate for further research: First, we have not considered in our study a high-performance ICC protocol called ARIES/IM that was proposed in [17]. In [23], several reasons were advanced as to why ARIES/IM might be expected to exhibit performance behavior similar to that of the B-link protocol in the conventional database domain. While we expect a similar outcome to also arise in the real-time environment, confirmation of this expectation requires actual implementation and evaluation of the ARIES/IM protocol.

Second, as mentioned in Section 3., an alternative to the priority-preemption approach that we considered here is priority inheritance. It would be interesting to evaluate the performance behavior of this scheme. Similarly, instead of treating undo transactions as “golden” transactions, a priority inheritance approach could be used there too to ensure that these transactions interfere only minimally with ongoing processing.

Third, we assumed here that all transactions have the same “criticality” or “value” [25], and therefore the performance goal was to minimize the number of killed transactions. An important open area is the characterization of ICC protocol behavior for workloads where transactions have varying values, and the goal of the RTDBS is to maximize the total value of the completed transactions. These results could also be extended to “soft-deadline” applications, where there is a diminishing residual value for completing transactions even after their deadlines have expired.

Finally, another interesting issue is to extend our analysis to multi-dimensional index structures such as, for example, R-trees [6]. We expect that these structures would be utilized in real-time applications such as mobile databases that include spatial data in their repositories.

Acknowledgements S. Seshadri worked on this project when he was with the Dept. of Computer Science and Engineering, Indian Institute of Technology, Mumbai, India. Brajesh Goyal, S. R. Narayanan and V. Srinivasan made significant contributions at various stages to this project. Jayant Haritsa’s work was supported in part by research grants from the Dept. of Science and Technology and the Dept. of Bio-technology, Govt. of India.
References


PART III
RUN-TIME SYSTEM MANAGEMENT
This page intentionally left blank.
Chapter 7

BUFFER MANAGEMENT IN REAL-TIME ACTIVE DATABASE SYSTEMS

Anindya Datta
Georgia Institute of Technology
Atlanta, GA 30332-0520
anindya.datta@mgt.gatech.edu

Sarit Mukherjee
Panasonic Labs
Princeton, NJ

1. INTRODUCTION

Real-time, active database systems (RTADBSs) have attracted a substantial amount of interest in recent times. Such systems amalgamate the properties of real-time systems and active systems. Specifically, RTADBSs are characterized by triggered transactions that arrive with deadlines. The most widespread envisioned use of RTADB is in control systems, i.e., systems that monitor and control external environments. Examples of such systems are process control systems such as nuclear reactor control systems, automated stock trading systems, telecommunications network management systems, etc. A common theme in all these scenarios is that automatic control actions must be taken in response to semantically incorrect operation of the system. Such reactive behavior is usually operationalized by Event-Condition-Action (ECA) rules [1] with time constraints. An example may be the following rule: ON event report, IF temperature in reactor is greater than 1000° C, THEN reduce pressure by 10 atm. WITHIN 10 s. This rule illustrates the active and real-time nature of RTADBSs. A nice feature of RTADBSs is the potential for the use of "canned transactions" [2]. Usually, control systems are "closed loop" systems where two primary classes of transactions dominate: (a) state reporting transactions, where distributed sensors report the values of predetermined data items (e.g., process sensors reporting the state of environment variables such as tempera-
ture, pressure, reaction kinetics in reactors); and (b) control action transactions, which are fired in response to unacceptable system behavior detected by analyzing the reported system state. Because both these transaction classes consist of transactions whose instances arrive repetitively, it is possible to determine the read/write sets of these transactions a priori. We will exploit the canned transaction phenomenon in this chapter. We also feel it is important to state the context of this work. This chapter reports part of our ongoing research that explores transaction execution in RTADBSs. We are currently looking at various aspects of transaction management, including CPU scheduling, overload management, concurrency control, buffer management etc. For our study on scheduling and overload management, refer to [3]. In this document we report PAPER, a buffer management algorithm developed for active, realtime transactions. Our preliminary results indicate PAPER outperforms other buffer management algorithms under a wide range of system loading and resource availability conditions.

2. MOTIVATION AND RELATED WORK

Data buffering is an important issue in database systems. Due to the fact that databases are usually too large to reside completely in memory, data is mostly disk resident. The principle of intelligent data buffering consists of retaining a strategic part of the database in main memory such that transactions do not have to go to disk often to fetch data, thereby reducing response times. In general data buffering is based on the principle of transaction reference behaviors [4].

Buffer management refers to a collective set of strategies that dictate what data is held in memory and for how long. The two basic components of any buffer management policy are buffer allocation and buffer replacement [5]. Buffer allocation strategies attempt to allocate buffer frames to transactions, while buffer replacement policies attempt to identify victim frames, i.e., candidate frames for replacement. In non-real-time systems the goal of buffer management is to reduce transaction response times. In real-time systems a good buffer management strategy should reduce number of transactions missing their deadlines.

In spite of extensive buffer management studies in traditional, non-real-time database frameworks [5,6,7], not much is reported in RTDBSs contexts. In all, we were able to identify three papers [8, 9, 10] that report "priority" cognizant buffer management algorithms. Of these, the work reported in [10] is the only one that considers a real-time context. The other two consider prioritized buffer management in the context of a "DBMS with priorities." Three priority cognizant buffer management algorithms are postulated: Priority-LRU (PERU) and priority DBMIN in [8] and Priority Hints (PH) in [9]. Priority DBMIN primarily applies to query processing and not transaction processing and is
thus not directly relevant to our work in this chapter. Both PERU and PH apply

to prioritized transaction processing and are therefore considered relevant. As
mentioned above, these algorithms were not developed in the context of a
RTDBSs. Moreover, the above algorithms consider "static priorities", i.e.,
transaction priorities (which translate into frame priorities) are constant. In
RTDBSs, however, as is well known, transaction priorities change during the
course of their lifetimes (e.g., in the Earliest Deadline (ED) [11] algorithm,
priorities increase as transactions get closer to their deadlines). Due to the non-
real-time nature of PH and PERU, as well as the static nature of the associated
priorities, these are not directly applicable to RTDBS scenarios. Nevertheless
these papers and algorithms provide important insights into the behavior of
prioritized buffer management algorithms and a case could definitely be made
for adapting these algorithms to RTDBSs and studying their performance.

Another very interesting aspect regarding the above three papers is that
they are somewhat contradictory in their conclusions: whereas Huang and
Stankovic [10] report no substantial performance enhancements with priority
cognizant buffering policies, Carey and coworkers [9, 8] conclude that incor-
porating priorities is quite beneficial. A part of this contradiction could be
attributed to the fact that former report looks at a RTDBSs while the latter ones
do not. However, all three papers are fundamentally concerned with prioritized
buffer management, and it is interesting to note that they reached such different
conclusions. Perhaps it was the fact that authors in [10] assume no a priori
knowledge of transaction access patterns while the other algorithms do. Such
an assumption is quite justified in real-time scenarios (the canned transaction
assumption) as pointed out in several papers in the literature [12, 13, 2]. In sum-
mary, it appears that the issue of buffer management in RTDBSs is quite open
- moreover, there appears to be room for improvement especially by assuming
some prior knowledge about transactions. As Ramamritham [2] succinctly puts
it, "... the jury is still out on this issue and further work is needed." In this
chapter, we show that it is possible to design better performing real-time buffer
management schemes.

Apart from the above reasons, we would like to reiterate that this work is
being done in the context of our broader research goal of studying transaction
processing in RTADBSs. Also, the algorithm introduced in this chapter, PA-
PER, exploits the special characteristics of both real-time and active databases.

3. EXECUTION MODEL FOR ACTIVE REAL-TIME
TRANSACTIONS

In this section we explain, briefly, a model of execution of real-time triggered
transactions, which forms a significant part of the workload in a RTADBS. Note
that we make the canned transaction assumption, i.e., the read and write sets of
incoming transactions are known. The system model we consider in this chapter may be regarded as comprising of finite sets $D$ of data items, $C$ of pre-defined triggering conditions and $T$ of pre-defined triggered transactions. Triggering conditions are formulae consisting of a term or conjunction of terms. Each term is of the form $d_i l_i$, where $d_i$ is a data item and $l_i$ is a constant. An example of a triggering condition is: temperature $\geq 1000$. If a data item $d_j \in D$ appears in triggering condition $c_i \in C$, then $d_j$ is said to participate in $c_i$. Transactions in $T$ are triggered by the satisfaction of triggering conditions in $C$.

Certain transactions upon execution may update data items which may result in the satisfaction of triggering conditions, resulting in the triggering of more transactions. Thus the system may be regarded as consisting of data items, triggering conditions and transactions that interact with each other in a pre-defined manner.

To facilitate the representation of the above model, we use a structure that we have termed a triggering graph (TG). TGs capture the relationships between the sets $D$, $C$ and $T$ explained above. A formal definition is provided below.

**DEFINITION 1** A triggering graph (TG) is a 6-tuple $(D, C, T, E, W_p, W_t)$ representing a (weighted) tri-partite graph with vertex set $D \cup C \cup T$ and edge set $E \cup ((D \times C) \cup (C \times T) \cup (T \times D))$. $W_p$, the satisfaction function, denotes a mapping from $(D \times C)$ to the set of probability values. $W_t$, the time interval function, denotes a mapping from $(C \times T)$ to the set of non-negative real numbers. A marked TG is a TG in which a nonempty set of vertices $V \subseteq D$ is identified as being “marked”.

---

$^1$ may be another data item as well, but that case is a trivial extension of the case reported.
An example TG is shown in Figure 7.1. Figure 7.1 may be interpreted according to the explanation of our model given above. Square nodes denote data items, circular nodes model triggering conditions, and oval nodes model triggered transactions. Edges in the TG represent relationships between the node sets. The edge set $E_{DC} \subseteq (D \times C)$ models the participates relationship explained above. The weights on these edges denote the probability that the update of a data item results in the satisfaction of the corresponding consistency constraint. These probability values are not arbitrary. Rather, they are computed, while the system is operational, by noting the number of times a data item is updated and the number of times corresponding trigger conditions (i.e., the conditions in which that data item participates) are satisfied. Thus, these probability values are dynamically updated as the system runs for longer and longer durations. For example the edge $(d_1, c_1)$ models the fact that the data item $d_1$, participates in the triggering condition $c_1$. Moreover, the weight $p_{11}$ on this edge denotes that an update of $d_1$ will result in the satisfaction of $c_1$ with a probability $p_{11}$. Note that all circular nodes with indegree greater than one are complex constraints, i.e., are the conjunction of multiple terms. The edge set $E_{CT} \subseteq (C \times T)$ models the triggering relationship. The weights on these edges denote the time constraint imposed on the triggered transaction. For example the edge $(c_1, t_1)$ denotes the fact that upon satisfaction of $c_1$ transaction $t_1$ would be triggered. The weight $t_{c_{11}}$ on this edge denotes that $t_1$ will be triggered with a time constraint of $t_{c_{11}}$. Finally edge set $E_{TD} \subseteq (T \cup D)$ models the read and write sets of transactions. Edges shown as dashed lines denote a read access of the appropriate data item, while solid lines denote write access. For example the dashed edge $(t_1, d_1)$ models the fact that transaction $t_1$ reads data item $d_1$, while the solid edge $(t_1, d_2)$ indicates that $t_1$ writes $d_2$. In Section 6.2 we will see how this graph is used in our algorithm.

Note also that the execution model supports both parallel detached mode with causal dependencies and sequential detached mode with causal dependencies.

4. BUFFER MANAGEMENT

4.1 BUFFER MODEL

Before describing our algorithm, we briefly present our buffer model. Under our model, each transaction is allocated a piece of memory as its working space. We regard each of these memory spaces as the private buffers of active transactions. We work under the assumption that there is enough memory space for private buffers for all concurrent transactions in the system. This notion of private working space provides two benefits: (a) it helps recovery using the after image journaling approach [10]; and (b) it captures intra-transaction locality of transaction reference behavior [4]. Between the pool of private buffers and the disks on which the database is resident, lies the global buffer. This buffer
is shared by all concurrent transactions. The global buffer space captures two different locality of reference behaviors: inter-transaction locality and restart-transaction locality [4]. Figure 7.2 illustrates the data buffering architecture assumed.

![Buffer Model](image)

Figure 7.2 Buffer Model

### 4.2 BUFFER MANAGEMENT ALGORITHM

Our buffer management algorithm is titled PAPER - Prefetching Anticipatorily and Priority based Replacement. We first state the basic philosophy behind PAPER and then describe different aspects in greater detail. The basic philosophy behind PAPER is simple.

1. The buffer replacement policy of PAPER is priority based - it assigns a priority to global buffer frames. The priority of a given page in memory changes depending on its anticipated reference pattern and the priority of the transactions that are to access that page.

2. Given that transaction data access and triggering patterns are predefined (recall the triggering graph), it is possible to anticipate, with some associated uncertainty, future transaction arrival patterns, given the current state of the system. PAPER uses this knowledge to prefetch data items that it anticipates will be required by near future arrivals.

Two distinct components of PAPER emerge from the above description: (a) the priority based page replacement component; and (b) the anticipatory prefetching component. Accordingly, we think of two versions of the algorithm: (i) PAPER− which just implements the prioritized replacement part, i.e., component (a) above, and (ii) PAPER+ which implements both components. Through this dichotomy we study the effects of priority based replacement alone in the reduced version of the algorithm, PAPER−, after which we investigate the effects of prefetching in RTADBS through the full version of the algorithm, PAPER+. Another way of looking at the dichotomy is to note that PAPER− is a real-time algorithm by virtue of the fact that it does not exploit transaction triggering information and thereby does not prefetch. PAPER+ on the other hand is the real-time, active version, exploiting both components. As
will be shown later, PAPER performs better than other algorithms. Thus, in this paper we not only contribute to the area of RTADBS, we contribute to buffer management in RTDBSs as well.

5. BUFFER MANAGEMENT IN REAL-TIME DATABASE SYSTEMS

Before actually presenting the algorithms, we discuss some of the issues involved in RTDBSs and the reasons why existing policies are deemed inappropriate in such systems.

Achieving a low page fault rate is a common goal of buffer management algorithms. PAPER does not depart from this common philosophy, as will be discussed below. However, certain characteristics of real time data base systems need to be examined.

Transaction deadline: Transactions in a RTDBS typically arrive with time constraints. As a result, the primary performance goal of the RTDBS is the achievement of a low miss ratio, rather than the traditional "low average response time" criterion [14]. This important difference has guided the exploration of new scheduling and concurrency protocols in RTDBSs. Our intuition suggests that for the same reason, buffer management needs to be re-examined as well.

Deferred disk writes: Conventional DBMSs defer disk writes, i.e., committed pages are not written to disk immediately, but either wait until when their carrying buffer is victimized through a page fault, or are written en masse by an asynchronous write engine [15, 4]. Both methods, however, present some undesirable side effects when applied to temporally constrained transaction processing. In the "write-on-demand" scenario, the actual cost of the disk-write operation is effectively transferred to the demanding transaction. This occurs regardless of the transaction's time constraint, which is clearly undesirable. If, on the other hand, an asynchronous write engine is used, it effectively allocates itself exclusive use of disk resources periodically. Meanwhile, active transactions are effectively suspended, which is clearly undesirable when these transactions have time constraints.

Clean and dirty frames: In most algorithms, no distinction is made during victimization between clean and dirty frames. Consider two pages, \( p_1 \) and \( p_2 \) which are deemed to be equally available as victims, but \( p_1 \) is dirty while \( p_2 \) is clean. A traditional buffer management policy would choose as its victim whichever of \( p_1 \) or \( p_2 \) came first in its internal structure. In other words, both pages would have the same likelihood of being reused. We believe that this should not be the case in time constrained applications. Rather, in real-time scenarios, the clean page \( p_2 \) should be victimized first, since this would potentially save a costly disk write. From the above discussion, the following
two principles should appear as desirable for buffer management policies in RTADBS:

1. A buffer management policy for a RTADBS should victimize pages that will only be used by transactions that are further away from their deadlines, and

2. Given any set of pages that are equally likely candidates for replacement, the clean frames should be victimized first.

Below, we present a policy that follows these two principles.

6. THE ALGORITHMS

We first present PAPER−, the pure real-time version of our algorithm and subsequently augment it to PAPER+ for RTADBS.

6.1 PAPER−

The buffer allocation and replacement policy of PAPER− is based on the discussion in Section 5. We distinguish between pages in memory that are likely to be used by a currently executing transaction ("reuse" pages), and those that are not ("free" pages). Upon page fault, PAPER− will victimize a free page if any is available, and a reuse page otherwise. To determine which pages are likely to be used, an arriving transaction \( t_i \) is asked to provide information about its working set. This information is then transformed into a set \( U_{t_i} \) of \((p_j, n_{ij})\) pairs, where \( p_j \) is page that \( t_i \) intends to use and \( n_{ij} \) is the number of times that \( t_i \) will need to touch \( p_j \). During its life, \( t_i \) will update this usage set \( U_{t_i} \) as it touches each page \( p_j \) in its working set, i.e., for each access of page \( p_j \), its counter \( n_{ij} \) would be decremented by 1. Once the counter reaches 0, it is known that the transaction does not require \( p_j \) any longer. Finally, upon termination, \( t_i \) will update any remaining \((p_j, n_{ij})\) pair where \( n_{ij} \) is not zero. Thus, a page \( p_j \) in memory will be in the reuse pool if there exists at least one currently executing transaction \( t_i \) with a pair \((p_j, n_{ij})\) \( \in U_{t_i} \) where \( n_{ij} > 0 \).

Within the reuse pool, frames are differentiated as follows. From the discussion in Section 5., it is clear that victims should not be chosen from pages that are likely to be used by transactions that are close to their deadlines. We say that any buffer page that is likely to be used by a transaction is "owned" by that transaction. Since there might be more than one transaction that would claim such "ownership" over any given page, we assign each page the deadline of its principal owner, i.e., the owner with the smallest deadline. The reuse pool is thus sorted in order of earliest deadline, such that pages that have the latest deadline principal owner are victimized first.

Also, since a transaction may claim principal ownership of more than one page, multiple pages may have the same deadline. We distinguish between
such pages using their dirty bit. Following from our discussion in Section 5., PAPER\textsuperscript{−} victimizes clean frames before dirty ones. The complete key upon which the reuse pool is sorted thus becomes the pair \((d_p, b_p)\), where \(d_p\) is the deadline assigned to page \(p\) and \(b_p\) is the clean bit of \(p\), where \(b_p\) is null if the page is dirty and 1 otherwise.

The free pool, on the other hand is not sorted, but is divided into two subpools: a clean pool and a dirty pool. Each pool follows the simple first-come, first-served (FCFS) policy for replacement. Also, as was the case for the reuse pool, we first select victims from the clean pool while it is not empty. To summarize, we present below the sequence of steps to be followed in PAPER\textsuperscript{−} when a victim frame is needed.

6.1.1 Implementation of PAPER\textsuperscript{−}. As explained in the previous section, PAPER\textsuperscript{−} relies on two sets of pages, namely the reuse and free pools, forming two buffer pools. The free pool in turn is composed of two sets of pages, a clean pool and a dirty pool. The frames in the reuse pool are sorted in order of the composite key \((\text{deadline}, \text{clean-bit})\), such that pages in frames with a higher key are replaced first. The free pool, on the other hand, is simply divided into two sets, one for dirty frames, and one for clean ones. The \text{deadline} field of each frame is thus neglected in the free pool.\textsuperscript{2} We now proceed to the algorithmic presentation of PAPER\textsuperscript{−}. Throughout this discussion, we utilize the terms \text{Page()} and \text{Frame()} to represent two functions internal to the DBMS. The \text{Page()} function, serviced by the data manager, returns the page on which a particular data item is located, while the \text{Frame()} function returns information from the system page table, namely the memory frame in which a particular page is located, or NULL if the page is not in memory.

Upon entry of a transaction to the system, the \text{System_entry} procedure below is executed. This procedure considers each element of the arriving transaction’s working set. For each element, it updates the appropriate usage counter. If, in the process, the transaction becomes the principal owner of the corresponding frame, one of two things occurs. If it is not in the reuse pool, the page is brought in; otherwise, it is relocated depending on its new deadline.

On each page reference thereafter, the \text{Touch-page} procedure below is followed. The entry corresponding to the page being touched is found within the usage set \(U\) of the transaction and its usage count is decremented by 1. If this usage count reaches 0, then we know that the page will not be used again by the touching transaction. If the touching transaction was the principal owner of the page, then if the page is in memory it is relocated into the free pool if it has no other owner, and in the reuse pool otherwise.

\textsuperscript{2}In the free pool, the deadline is in fact set to infinity. This is done to simplify the prefetching algorithm, as will be discussed in Section 6.2
**System_Entry**

Let the operation set \( OS_t \) of a transaction \( t_i \) be represented as a set of pairs \{\( (d_1, o_1), (d_2, o_2), ..., (d_n, o_n) \)\}, such that \( (d_i, o_i) \) indicates that \( t_i \) will perform \( o_i \) - a read or write - upon data item \( d_i \). Note that a write access constitutes two page touches while a read access is a single page touch.

On entry of transaction \( t_i \) into the system at current time \( A_{ti} \) with deadline \( D_{ti} \) and operation set \( OS_{ti} \), perform the following:

For all pairs \( (d_j, o_j) \in OS_{ti} \) do

1. if \( \text{Page}(d_j) \) does not appear in \( U_{ti} \) then
   1. \( U_{ti} \leftarrow \text{Page}(d_j, 0) \cup U_{ti} \)
   2. if \( (t_i \) is principal owner of \( \text{Page}(d_j) \) \&\& \( \text{Frame}(\text{Page}(d_j)) \neq \text{NULL} \) then
      \( \text{Reuse}_\text{Pool} \leftarrow \text{Frame}(\text{Page}(d_j)), D_{ti} \cup \text{Reuse}_\text{Pool} \)

Select \( (p, n) \in U_{ti} \) where \( p_j = \text{Page}(d_j) \)

if \( o_j = \text{WRITE} \) then \( n \leftarrow n + 2 \)
else \( n \leftarrow n + 1 \)

**Touch_Page**

Let \( t_i \) be an active transaction that is touching a page through an operation \( (d_j, o_j) \)

Select \( (p_j, n_{ij}) \in U_{ti} \) where \( p_j = \text{Page}(d_j) \)

\( n_{ij} \leftarrow n_{ij} - 1 \)

if \( n_{ij} = 0 \) then

1. if \( (t_i \) was the principal owner of \( p_j \) \&\& \( p_j \) is in memory \) then
   1. if \( (t_i \) still has a principal owner of \( t_{PO} \) \) then
      \( \text{Reuse}_\text{Pool} \leftarrow \text{Frame}(p_j), D_{tPO} \cup \text{Reuse}_\text{Pool} \)
   else Add \( \text{Frame}(p_j) \) in the appropriate \textit{free pool}
Buffer Management in Real-Time Active Database Systems

When a transaction terminates after committing, having performed all its operations, it will own no pages. If, however, there occurs an abnormal termination or restart (e.g., through deadline expiry or through the validation phase of an optimistic CC algorithm), then PAPER\(^-\) actively empties the usage set \(U\) of the terminating transaction along with any effects this transaction might have upon buffer frames. As can be seen from the algorithm shown below, this operation takes place as follows. For each element of the transaction’s \(U\) set, if the terminating transaction was the primary owner of the page and the page has not yet been swapped out, then if there exists a new principal owner for the page, the corresponding frame is relocated in the reuse pool, otherwise, it is simply moved to the free pool. This is shown in the Abnormal-Termination procedure.

**Abnormal-Termination**

Let \(t_i\) be an active transaction that is either aborting or restarting.

For each \((p_j, n_{ij}) \in U_t\) do

if (\(t_i\) is the principal owner of \(p_j\) A \(p_j\) is in memory) then

if (\(p_j\) still has a principal owner of \(t_{PO}\)) then

Reuse_Pool ← \((Frame(p_j), D_{t_{PO}}) \cup Reuse-Pool)\)

else Add Frame\((p_j)\) in the appropriate free pool

To complete our description of the buffer allocation and replacement of PAPER\(^-\), we present the replacement policy. Upon a page fault, PAPER\(^-\) first attempts to select a candidate victim from the clean subpool in the free pool. If none are available, then the dirty subpool is used. If the free pool does not contain any available candidate victims, then the reuse pool is inspected, and the frame with the highest key is selected and replaced.\(^3\)

This concludes our discussion of PAPER\(^-\). As can be easily seen from the above discussion, PAPER\(^-\) is a pure real-time algorithm, relying solely on deadline information of transactions.

Below we present PAPER\(^+\), which includes, in addition to all features of PAPER\(^-\), the anticipatory prefetching component of the algorithm introduced at the beginning of Section 4.2.

**6.2 ANTICIPATORY PREFETCHING: AUGMENTATION OF PAPER\(^-\) TO PAPER\(^+\)**

It has been recognized in the literature [16], that prefetching based strategies (as opposed to demand paging) appear to be appropriate for some real-time

\(^3\)Recall that the sorting key for the reuse pool is a composite key of the form \((deadline, clean-bit)\), as discussed in Section 6.1.
Page Replacement

if \((Clean\_Pool \neq 0)\) then
    \(F \leftarrow \text{Element from } Clean\_Pool\)
    \(Clean\_Pool \leftarrow Clean\_Pool - F\)

else if \((Dirty\_Pool \neq 0)\) then
    \(F \leftarrow \text{Element from } Dirty\_Pool\)
    \(Dirty\_Pool \leftarrow Dirty\_Pool - F\)

else \(F \leftarrow \text{Element from } Reuse\_Pool\)
    \(Reuse\_Pool \leftarrow Reuse\_Pool - F\)

Replace \(F\)

systems. Also by analyzing the triggering graph for a particular RTADB, it is possible to anticipate, with a certain degree of uncertainty, what transactions may be triggered. For example, if a transaction \(t_i\) arrives in the system, using the write set of \(t_i\), we can mark the appropriate \(D\) nodes in the TG (recall the discussion in Section 3.), and extract the subgraph induced by the nodes reachable from the marked nodes. Clearly, the \(C\) nodes in this subgraph are the triggering conditions that may be satisfied by the updates of the marked \(D\) nodes. Moreover, the \(T\) nodes in this subgraph are the transactions that would be triggered were the conditions actually satisfied. Subsequently, following the appropriate \((T \times D)\) edges one can re-mark appropriate \(D\) nodes that would be touched by the triggered transactions and start the entire process again. In this way, an \(n\)-step lookahead could be performed, and a set of data items (and consequently a set of data pages) may be identified. In addition, by simple probabilistic analysis one can easily compute the probabilities that particular transactions would be triggered and consequently particular data items needed. This could be very simply done using the weights of the \((D \times C)\) arcs, i.e., the \(p_{ij}\) values. Thus, given the arrival of a transaction into the system, it is possible to predict, with associated probabilities, which data pages may be needed subsequently. The prefetching stage of PAPER\(^+\) does precisely that. Below we outline algorithmically, the steps in the prefetch component of PAPER\(^+\):

**Anticipatory Prefetching.** On the arrival of transaction \(t_i\) into the system at current time \(A_{ti}\) with deadline \(D_{ti}\) and write set \(WS_{ti}\) perform the following:

1. Mark appropriate \(D\) nodes, such that each marked node

2. Based on this marking perform a 1-step lookahead as described before. This lookahead would yield two related pieces of information:
A set of pairs \{((t_1, pr_1), (t_2, pr_2), \ldots, (t_n, pr_n))\}. If the pair \((t_i, pr_i)\)
appears in this set it means that transaction \(t_i\) may be triggered with probability \(pr_i\).

A set of data items \\{\(d_1, d_2, \ldots, d_j\)\} that the identified transactions (in the above step) will need.

3. Based on step 2, compute prefetch priorities for the data items needed. The process for this computation is shown below. The prefetch priorities indicate the criticality or the algorithm's "degree of confidence" that the data item will actually end up being needed.

4. Based on the priorities computed in step 3, the prefetch requests are introduced in prioritized prefetch pools and prefetch pages that are in the free pool are moved to the anticipated pool. These structures are described below.

Clearly, prefetch page requests compete with demand page requests. In PAPER+, all demand page requests have higher priority than prefetch requests. In other words a prefetch request gets served only when the appropriate disk is free, i.e., not serving a demand page request. The next logical question therefore is: are the disks free often enough in order for prefetching to have an impact. We ran extensive experiments under a wide range of operating conditions to verify the appropriateness of prefetching. It turns out that even under extreme load conditions disk utilization was never more than 80%. In other words, there were significant opportunities for prefetching to work. This motivated us to incorporate prefetching in PAPER+.

At this point we turn our attention to three important aspects of prefetching in PAPER+: (a) prefetch priority computation; (b) prefetch pools; and (c) anticipatory frame holding. These are described below.

**Prefetch priority computation.** The philosophy behind prefetch priorities is the following: the prefetching algorithm outlined above is going to output a set of data items (pages) that will be required by transactions which may arrive. Moreover, the algorithm will also output probabilities that particular transactions will be triggered. Also, it can be easily assumed that we will be unable to satisfy every prefetch request as all demand page requests are processed with higher priority than each prefetch request. Therefore, it makes sense to prioritize the prefetch requests in order to make it more likely that "more urgent" requests will be served before "less urgent" ones. We illustrate the process of prefetch priority computation through an example:

**Example.** Let transaction \(t_i\) enter the system at time \(A_i\) with deadline \(D_i\). Assume that the 1-step lookahead algorithm outlined above outputs the information that \(t_i\) will trigger \(t_j\) and \(t_k\) with probabilities \(p_j\) and \(p_k\) and time
constraints \( c_j \) and \( c_k \), respectively. Additionally assume that it is known that \( t_j \) and \( t_k \) would be triggered at times \( T_{rj} \) and \( T_{rk} \), respectively. In that case the deadlines of \( t_j \) and \( t_k \) would be \( T_{rj} + c_j \) and \( T_{rk} + c_k \) respectively. Also, it is easily intuitively understood that prefetch priorities, i.e., the urgency with which the data items belonging to a transaction's working set would be needed is (a) directly proportional to the triggering probability of that transaction; and (b) inversely proportional to the deadline of the transaction. Based on this logic, the prefetch priority of the data items (pages) corresponding to \( t_j \)'s working set is \( p_j/(T_{rj} + c_j) \). Similarly, the prefetch priority of the data items (pages) corresponding to \( t_k \)'s working set is \( p_k/(T_{rk} + c_k) \). The only problem with the scenario described above is that the triggering times of \( t_j \) and \( t_k \) are not possible to predict, i.e., \( T_{rj} \) and \( T_{rk} \) are unknown. Therefore we estimate the trigger times by making the assumption that the most likely trigger time would be at the midpoint of a transactions allowed lifetime, i.e., halfway between its arrival time and its deadline. For \( t_j \) and \( t_k \), that time is the midpoint of the allowable lifetime of \( t_i \). In other words, we make the assumption that \( T_{rj} = T_{rk}(D_{ti} + A_{ti})/2 \). Then the prefetch priority of each data item in \( t_j \)'s working set is \( p_j/(D_{ti} + A_{ti})/2+c_j) \). Similarly, the prefetch priority of each data item in \( t_k \)'s working set is \( p_k/(D_{ti} + A_{ti})/2+c_k) \). In PAPER+, we do one more "tweak" to the method outlined above to get our priorities. Basically, we prioritize by taking the inverse of the above expressions. One can easily see that the inverses give the virtual deadlines associated with each page request and can be used to prioritize the request. The virtual deadline for transactions \( t_j \) and \( t_k \) of the above example will be \( (D_{ti} + A_{ti})/2+c_j)/p_j \) and \( (D_{ti} + A_{ti})/2+c_k)/p_k \) respectively.

Prefetch pools. In PAPER+, we do not send the prefetch requests directly to disk. Rather, they are sent to a number of prioritized pools maintained inside the system. There is one pool for each disk. Whenever the demand pool for a disk is empty, the highest priority prefetch request is served from its associated prefetch pool. The primary motivation for holding prefetch requests in specialized pools is that we want to have more control on the prefetch requests than on the demand page requests. There are two reasons for this:

Depending on the level of activeness of the underlying ADBMS, there might be a lot more prefetch requests generated per arriving transaction than demand page requests. If the prefetch requests were allowed to proceed directly to the disk queues, these queues would grow indefinitely. However, by holding these requests in controllable pools, we can control the rate of growth by eliminating requests that are outdated. The procedure to do this is outlined later in this chapter. Because of the large number of prefetch requests generated, it is likely there would be a large amount of redundancy, e.g., the same page may be requested several times. Thus, it is necessary to be able to exert some amount
Buffer Management in Real-Time Active Database Systems

of control in order to reduce these redundancies. The proposed solution can be summarized as follows:

1. All demand requests would have higher priority than all prefetch requests. If a disk queue is empty, the highest priority request from its corresponding prefetch pool would be serviced. Before sending a prefetch request to disk, PAPER+ checks if the page already exists in buffer. If this search succeeds, the prefetch request is temporarily ignored and the next one checked.

2. It is quite possible that different prefetch requests for the same page would be generated with identical or different priorities (as more than one transaction may need the same page). In PAPER+, we make no attempt to eliminate this redundancy, thereby reducing overhead. This does not cause problems, as the issue discussed in the previous item takes care of this. For example, assume that two requests exist for page p with virtual deadlines \(v_i\) and \(v_j\) where \(v_i < v_j\). Now further assume that the request with virtual deadline \(v_i\) is served. If the request with virtual deadline \(v_j\) ever reaches the head of the pool, our algorithm will check if p is in memory. If it is, the request will not be served. However, if it is not, i.e., it has been swapped out since the time it was brought in by the earlier request, the request with virtual deadline \(v_j\) will get served and the page will be swapped in again.

3. Also, it is possible that pages which are released by their active owners are potential prefetch candidates. In PAPER−, these pages are selected for replacement only when none of the frames that are either owned by executing transactions or tagged for prefetching is available. In other words, pages that might be used by anticipated transactions are only victimized when there are no frames that will not be used by existing or anticipated transactions. If this were not so, then such “anticipated” pages might find themselves swapped out by a faulting transaction, only to be swapped back in at a little later through either the faulting of another active transaction or the servicing of a prefetch request.

A prefetch request \(R\) is represented in PAPER+ by a pair \((p_R, VD_R)\), consisting of the page, \(p_R\), to be brought into memory and the "virtual deadline", assigned to this requested page through the computation presented previously. To each triggering transaction \(t_i\), we associate a set \(P_{t_i}\) of all such requests that were caused by \(t_i\). This set is generated upon transaction arrival, updated as requests are serviced, and destroyed upon transaction termination. Concurrently, each disk \(\delta\) has an associated \textit{prefetch-pool}_{\delta} where prefetch requests particular to that disk are placed. Each such pool is sorted so that requests having the earliest \(VD_R\) will be serviced first.
Anticipatory frame holding. In PAPER+, frames are released from the reuse pool when no existing transaction intends to use the page any further. At this stage, the frame is made available for replacement, be it for the servicing of page faults for existing transactions or the servicing of prefetch requests in anticipation of potential transactions. Yet, some of these released frames are likely to contain pages that are candidates for prefetching. We thus introduce a third frame pool into the scheme, in addition to the reuse and free pools. This anticipated pool holds all frames that contain pages that either have been prefetched or have been requested for a prefetch and are not currently owned by an active transaction. This buffer pool is inserted between the reuse pool and the free pool, i.e., the frames located there are less likely to be used than frames in the reuse pool, but less likely candidates for replacement than frames in the free pool. Within this new anticipated pool, the frames are sorted in order of decreasing virtual deadline, such that the frame with the largest virtual deadline is selected for replacement first. Since there might, as is the case in the reuse pool, be more than one frame with the same virtual deadline, frame cleanliness is also taken into consideration in the sorting. The complete sorting key for the anticipated pool thus consists of the pair \((VDR, \text{clean-bit})\). Note that the frame data structure used here is thus the same as for the other two pools, consisting of a (virtual) deadline, a clean-bit and the associated frameID.

To know which pages are candidates for prefetching, PAPER+ keeps a simple Prefetch-count array of prefetch request counters. Each time a transaction requests a prefetch on a page, the corresponding entry in the Prefetch-count array is incremented by one. Upon termination the transaction decreases the appropriate prefetch counter for each entry in the list. The basic structure having been described, it is easy to see how it efficiently manages the prefetching process. Below we illustrate how the basic prefetch management operations are performed with this structure.

Servicing prefetch requests. Whenever a disk \(\delta\) becomes idle, the prefetch request with the lowest deadline in \(\text{prefetch-pool}_\delta\) is serviced, as is shown in the following algorithm:

The above procedure first identifies an available clean victim frame in which to insert the page to be prefetched. This frame is, of course, taken only from either the Clean Pool of the free pool or the Anticipated pool itself. If there exists no such frame, then the procedure simply exits. Otherwise, \(\text{prefetch-pool}_\delta\) is searched for a request for a page that is not currently in memory. Once such a request is found, if its virtual deadline is earlier than that of the page in the victimized frame,\(^4\) then the request is actually serviced, the frame is then

\(^4\)As explained in Section 5., frames located in the free pool are given an infinite deadline. This ensures that they will always be used as victims for prefetching.
Service_Prefetch_Request
When a disk $\delta$ becomes idle, the following algorithm is executed:
if ($\text{Clean}\_\text{Pool}=0$) then
  if there is no clean frame in the $\text{Anticipated}\_\text{Pool}$ then
    exit
  else $F \leftarrow$ clean frame from the $\text{Anticipated}\_\text{Pool}$
else $F \leftarrow$ frame from the $\text{Clean}\_\text{Pool}$
Find the first prefetch request $(p_R, VD_R) \in \text{prefetch}\_\text{pool}_\delta$ where $p_R$ is not in memory
if ($VD_R < F.\text{deadline}$) then
  Fetch $p_R$ into $F$ from disk $\delta$
  $\text{Anticipated}\_\text{Pool} \leftarrow (F, VD_R) \cup \text{Anticipated}\_\text{Pool}$
  Remove $(p_R, VD_R)$ from $\text{prefetch}\_\text{pool}_\delta$

inserted into the $\text{anticipatory pool}$, and the prefetch request is removed from the $\text{prefetch}\_\text{pool}_\delta$.

Prefetch awareness in buffer pool management. The basic buffer management policy introduced in Section 5. needs to be augmented with the introduction of the different steps involved in ensuring that the buffer allocation and replacement policies are cognizant of the anticipatory prefetching nature of PAPER+. The algorithms are adjusted as follows:

- On entry of a transaction to the system, the steps outlined in the Anticipatory Prefetching algorithm earlier in this section are performed after completion of the System_Entry algorithm from Section 5.

- On each page reference, the Touch_Page procedure presented in Section 5. is enhanced to ensure that pages that are released from the reuse pool are not sent directly to the free pool if they are prefetch candidates. Instead, they are inserted in the $\text{anticipatory pool}$.

- Also, the Page_Replacement procedure is augmented to take the $\text{anticipated pool}$ into account. Namely, frames in the $\text{anticipated pool}$ are victimized only when no frames are available in the free pool, but before any frame in the reuse pool.

- Finally, when a transaction terminates, whether through completion or abnormal termination, PAPER+ removes from the system all prefetch requests that the transaction emitted on arrival. Upon termination, a transaction will have triggered all its children (which, upon arrival, will have moved the necessary frames into the reuse pool) and not triggered
some potential children (which will thus not need the pages tagged for prefetch by the parent transaction).

Having presented the details of our proposed algorithm along with the reasons for implementing such a policy, we now turn to a short discussion of the results of the extensive performance evaluation we conducted. For a detailed description of this performance evaluation, refer to [17].

7. SUMMARY AND FUTURE WORK

In order to evaluate the performance of the protocols described so far, we conducted an extensive simulation of the execution of these algorithms on an RTADB. For a detailed description of the simulation model, consult[17]. We compared the performance of PAPER to three different buffer management protocols: the well-known LRU [5], and two other leading priority-based buffer management protocols – Priority LRU and Priority-Hints [8, 9]. The primary performance metrics used were Transaction Miss Ratio (TMR) and Page Fault Rate (PFR). The results showed that PAPER consistently outperformed the other algorithms across a variety of workload conditions.

In this chapter, we presented PAPER, a buffer management algorithm designed to study the needs of RTADBSs. The algorithm consists of a priority-based buffer management policy coupled with a prefetching mechanism. The buffer management policy relies on the system priority of the active transactions to determine proper allocation and replacement of pages. The second component of PAPER, the prefetching mechanism, utilizes the canned-transaction properties associated with RTADBs to try to reduce the number of page faults of triggered transactions. The proposed algorithm was simulated, and the results from this simulation clearly show that PAPER outperforms other DBMS buffer management policies: less transactions miss their deadlines. However, this superiority, although always present is not always significant. It appears that prefetching, in particular, cannot provide significant performance improvements except in some cases.

Acknowledgements The work of first author (A. Datta) was supported in part by a small grant awarded by the Office of Vice president for Research and the University of Arizona Foundation, University of Arizona.

References


Chapter 8

DISK SCHEDULING

Ben Kao
Reynold Cheng
University of Hong Kong
kao, ckcheng@csis.hku.hk

1. INTRODUCTION

Many real-time applications handle large amounts of data and require the support of a real-time database system. Examples include telephone switching, radar tracking, media servers, and computer-aided manufacturing. Many of these database systems are disk-resident because the amount of data they store is too large (and is too expensive) to be stored in nonvolatile main memory. For instance, a radar surveillance system for tracking flying objects stores the images of known aircraft models in an RTDB, to be matched against those of an intruder in real-time [1]. As another example, an intelligent network stores in its services databases voluminous data (e.g., customer service records, account records, network traffic management data, etc.) to keep track of various activities that occur in the network [2].

In these applications, disk accesses often dominate the execution time of a real-time transaction. An effective disk scheduling algorithm is thus very crucial for the system to attain a high performance. Comparing with CPU scheduling, disk scheduling is even more difficult. The main reason is that disk seek time, which accounts for a very significant fraction of disk access latency, depends on the disk head movement. Hence, the servicing order of I/O requests and their service times exhibit an intricate dependency. This chapter focuses on the design of the I/O subsystem in an RTDBS, particularly, disk scheduling.

There are a number of studies on RTDB I/O subsystem design. Here, let us first draw a roadmap of the issues and problems addressed by these previous works. Further discussion summarizing their discoveries will be presented in the various sections of this chapter.

Before we talk about scheduling I/O requests, we must understand the objective of the scheduler and the characteristics of the requests we schedule.
In a real-time system, the most important objective is to satisfy the real-time constraints (e.g., deadlines) of the transactions that issue the I/O requests. In [3, 1] Abbott and Garcia-Molina observe that read and write requests are usually associated with different timing constraints because they are issued by uncommitted and committed transactions respectively. Based on this observation, they propose a model (k-buffer) for handling reads and writes. We discuss this model in Section 2.

With an understanding of the timing property of I/O requests, the next step is to assign priorities to the requests based on their timing constraints. In Section 3, we discuss a number of such priority assignment policies [3, 1,4,5]. Several studies have shown that the simple earliest-deadline-first (EDF) scheduler does not work well on disk. This is because EDF does not optimize disk seek, and as a result, the service rate of the disk is significantly lowered. In Section 4, we give an account on various disk scheduling algorithms [6, 1,5, 7]. In general, these algorithms improve the performance of the system by striking a balance between minimizing disk seek and meeting I/O request deadlines.

Besides employing a smart disk scheduler, another method to improve the I/O performance is to use multiple disks [8,9]. Section 5 discusses the various issues of applying multiple disk systems in an RTDBS. These issues include data replication, data placement and scheduling.

2. READS AND WRITES

A typical transaction consists of a number of read and write operations. In most database systems, transaction reads and writes do not go directly to the disk, instead, they are intercepted by the buffer manager. Given a read operation, the buffer manager checks if the data requested is cached in its buffer pool. If so, the cached data is returned to the transaction; otherwise, a disk-read request is issued. In the latter case, the transaction is blocked while the data is being read from the disk. A write operation, however, always updates a copy of the data item in memory buffer. The buffer manager will eventually write the updated buffers back to disk, for example, when the buffer manager is running out of buffers or when the transaction commits.

It is noted in [3, 1] that if the buffer pool is large enough to hold all the updates of unfinished transactions, then disk-read requests are issued by uncommitted transactions while write requests are issued by committed transactions. This observation implies that while read requests carry the timing constraints of their issuing transactions, write requests do not have explicit timing constraints. Instead, writes are done to avoid buffer overflow. It is thus interesting to see how reads and writes should be scheduled when their urgencies are of different nature.
In [1], the authors propose a \textit{k-buffer} model for the disk controller to schedule real-time I/O requests. In the model (Figure 8.1), the disk controller separates read and write requests. Read requests are associated with the deadlines of their issuing transactions. They are queued at the read queue according to their deadlines. Write requests, on the other hand, are spooled at the \textit{k-buffer}. The purpose of the \textit{k-buffer} is to delay the service of write requests. As we have pointed out, write requests do not have explicit deadlines. They are held at the \textit{k-buffer} so that they would not compete with the more urgent read requests. In the next two sections, we discuss how the disk scheduler would decide the servicing order of the I/O requests.

3. DEADLINE-BASED SCHEDULING

The simplest strategy to scheduling requests is earliest-deadline-first (EDF). Under this strategy, every I/O request is assigned a deadline and the disk scheduler would always service the request with the earliest deadline first. As we have discussed, read requests are issued by uncommitted transactions. They could thus inherit the deadlines of the transactions which issue them. Write requests, on the other hand, do not have explicit timing constraints. They should be serviced only when there are no read requests, or when the \textit{k-buffer} is almost full.

In [1], two heuristics are proposed for deciding when a write is serviced. The first heuristic simply counts the number of free slots in the \textit{k-buffer}. If the number goes below a space threshold, an arbitrary write request in the \textit{k-buffer} is serviced. This heuristic ensures that a minimum number of slots in the \textit{k-buffer} are maintained at all time so that there are always free slots available to receive write requests from the buffer manager. The second heuristic assigns an \textit{artificial deadline} $D_w$ to write requests using the formula:

$$D_w = t + \left(1/\lambda_w\right) \times \text{(number of free slots in k-buffer + 1)}$$
where $\lambda_w$ is the average arrival rate of write requests and $t$ is the current time. Essentially, $D_w$ is the expected time at which the $k$-buffer gets filled up. In other words, it is the deadline before which a write request must have to be completed to avoid overflowing the buffer.

Other variations of the deadline assignment policy have been proposed to improve performance [5, 4]. For example, Chen et al. [5] suggest that a read request should not simply inherit the ultimate deadline of the issuing transaction. This is because the transaction's deadline does not faithfully reflect how urgent a request is. This is especially true for those requests that are the first few steps of a transaction. A better strategy is to evenly distribute the slack of a transaction to its read/write operations. This policy, called EVEN, is achieved by assigning to each operation a step deadline. The step deadline of the $i$-th read/write operation of a transaction is given by step-deadline:

$$\text{step-deadline}(i) = a + \frac{i}{n}(L - a)$$

where $n$ is the total number of steps of the issuing transaction, $L$ and $a$ are the transaction's deadline and arrival time, respectively. Under this policy, an earlier read request gets a smaller step deadline, and hence a higher service priority. The motivation of EVEN is that if two transactions have the same deadline, but one has more steps than the other, then the early requests of the transaction with more steps will have a higher priority. By expediting the early steps, the longer transaction has a better chance of saving up enough amount of slack for the execution of the later steps.

Besides EVEN, other heuristics of assigning deadlines are studied in [5]. One method is to assign loose slack to early steps and tight slack to later steps (Early-Loose-Late-Tight or ELLT). Another scheme is to assign tight slack to early steps and loose slack to later steps (Early-Tight-Late-Loose or ETLL). Through experimental studies, it is shown that using step deadlines can significantly reduce the percentage of missed deadlines of transactions. In some high load cases, savings of up to 53% of deadlines are reported. The study also finds that ELLT performs slightly better than EVEN and ETLL. This is because ELLT gives an almost finished transaction a higher priority. This allows locks and other resources being held by the transaction to be released earlier, benefiting the execution of other transactions.

4. DISK HEAD SCHEDULING ALGORITHMS

In the last section, we discuss how deadlines are assigned to I/O requests so that EDF scheduling can be applied on disk. A number of studies [3, 6, 9, 1,
10, 5], however, show that the disk performance can be significantly improved if disk seek overhead is also taken into account. In this section, we discuss a number of disk head scheduling algorithms that integrate EDF with seek-optimizing methods such as Shortest Seek Time First (SSTF) and SCAN[11]. To illustrate the different algorithms, let us consider the following example, as shown in Figure 8.2.

Suppose we have four requests A, B, C and D in the I/O queue with their deadlines ($dl$) in the following order:

$$dl(A) < dl(B) < dl(C) < dl(D).$$

The position of the data needed by each request is shown in Figure 8.2. If EDF scheduling is employed, the service order would be:

**EDF:** A, B, C, D.

We note that in this case, the head sweeps the disk back and forth four times, or 32 tracks. Considering that the requests can be satisfied in only 11 track movement (in the order of D, B, C, A), apparently EDF is not a very smart way of scheduling the disk head if response time or throughput is a concern.

Algorithms for shortening disk head movement have been devised [11]. These include popular algorithms like SSTF and the elevator algorithm (SCAN). SSTF, as its name implies, always services the request with the shortest seek distance from the current head position. SCAN, on the other hand, moves the head from one end of the disk to the other and then back, servicing whatever requests are on its way, and changing direction whenever there are no more requests ahead in its direction. Referring to the example in Figure 8.2, SSTF and SCAN produce the following servicing schedule:

**SSTF/SCAN:** D, B, C, A.
We note that the disk head moves over 11 tracks, or three times fewer than in the case of EDF.

The problem with SSTF and SCAN, as applied to real-time systems, is that the deadlines of requests are not considered. In our example, request A, which has the earliest deadline, is serviced last. There is thus a trade-off between maximizing throughput and meeting system’s timing constraints. Methods that combine the properties of EDF and SCAN are very desirable. In what follows, we describe two middle-ground I/O scheduling algorithms: one applies the idea of SCAN to EDF, and another adds the flavor of EDF to SCAN.

When EDF is used, the disk head may pass through tracks on which there are other low priority requests. The Elevator principle says “do pick them up because the disk head is already there!” In [1], Abbott presents the FD-SCAN algorithm based on this principle. FD-SCAN stands for Feasible Deadline SCAN. Any request whose deadline is determined to be impossible to meet is discarded. Simply stated, FD-SCAN follows EDF in always “targetting” the disk head towards the track with the request that has the earliest deadline, but also services whatever requests are on its way. Consider the earlier example, the servicing order under FD-SCAN would be:

\[
\text{FD-SCAN: } C, A, D, B. \]

We note that in this example, the disk head moves a similar distance as SCAN does but the request A (which has the earliest deadline) is served sooner.

In Abbott’s study, FD-SCAN is tested against other disk scheduling algorithms including FCFS, SCAN, SSTF, and EDF. Simulation results show that FD-SCAN performs best among the algorithms tested in terms of the ability to meet deadlines. This property is most prominent when the load of the I/O subsystem is high. Also, the performance advantage of FD-SCAN is persistent through a wide range of system parameter settings.

In [6], the problem of long seek time exhibited in EDF is addressed. It is argued that the use of fine-grain priority gives EDF a FCFS-like average seek time (with possibly even worse response time). Their idea (which we will call the Highest Priority Group First (HPGF)) is to blur the boundaries of priority. Disk requests are grouped into a small number of priority levels even though the transactions issuing the I/O requests may have distinct deadlines. Once these groups are formed, the disk is scheduled to service the highest priority group first. In case there is more than one request in the highest priority group, SCAN is used for the intra-group scheduling. Referring to our example, if requests A and B are in a high priority group, and requests C and D are in a low priority group, the service order under HPGF would be:

\[
\text{HPGF: } B, A, C, D. \]
We note that in the example, the disk movement is much less than what EDF would require, while the requests with earlier deadlines are served before those with later deadlines.

Through a series of experimental studies [6], it is found that HPGF performs better than SCAN in meeting deadlines. This benefit is achieved at a cost of a prolonged average response time. However, the study shows that the response time degradation mainly affects low priority requests. High priority requests, on the other hand, experience response times which are very close to what SCAN provides.

In [5], another method of integrating EDF and SSTF is proposed. The method is called Shortest Seek and Earliest Deadline by Value (SSEDV). SSEDV is described as a window algorithm because it only considers a small number (a window) of requests that are at the head of the disk queue when the scheduler is picking one to serve. By limiting the number of requests considered, the cost of scheduling is reduced.

For each request \( r_i \) in the window, SSEDV assigns the following value to it:

\[
value_i = \alpha d_i + (1 - \alpha)\delta_i
\]

where \( d_i \) is the distance from the current head position to the track \( r_i \) seeks, and \( \delta_i \) is the remaining lifetime of \( r_i \). The request with the smallest value is served first. The variable \( \alpha \) is a parameter that controls whether SSEDV should behave more like EDF or SSTF. For example, if \( \alpha \) is close to one, then value is dominated by the term \( d_i \). In this case, SSEDV is degenerated to SSTF [10].

The motivation of SSEDV is that if a request with a far-in-the-future deadline is very close to the current disk head position, the scheduler might as well service the request first, since the cost incurred is minimal. In [5], it is shown that the disk performs very well in meeting deadlines when \( \alpha \) is set in the range of 0.7 to 0.8. In fact, their experiments show a performance gain of up to 38% over other algorithms such as SSTF, SCAN, FD-SCAN and HPGF.

The last disk head scheduling policy we describe in this section is Urgent Group and Shortest Seek Time First (UG-SSTF) [7]. In this policy, I/O requests are partitioned into three groups: an Urgent Group (UG), a Miss Group (MG), and a Deferrable Group (DG). An (urgent) request belongs to UG if it must be serviced immediately or else it may miss its deadline\(^3\). If a request has already missed its deadline, it is classified as an MG request. All other requests are assigned to DG.

---

\(^2\)The remaining lifetime of \( r_i \) is defined as the amount of time between the current time and \( r_i \)'s deadline.\(^3\)Specifically, at the instant of scheduling, a request is urgent if its deadline is earlier than current time + maximum disk seek time + full rotational latency + the transfer time of the data.
Under UG-SSTF, the scheduler always picks a request in UG (if there is one) that is closest to the head position to service first. The rationale is that among all the urgent requests, it is better to serve the one with the shortest seek time than the one with the earliest deadline. When there are no urgent requests, priority is given to requests in MG rather than those in DG. SSTF is again employed in scheduling the requests in MG. The reason for giving MG a higher priority is to reduce the mean tardy time of requests. As pointed out in [7], most studies have concentrated on reducing deadline miss ratio, but not so much on providing a low mean tardy time. By giving a higher priority to MG, the requests which have missed their deadlines can get reasonable response times. Finally, if both UG and MG are empty, EDF is used to schedule DG requests.

The study in [7] addresses the trade-off between meeting request deadlines and reducing the mean tardy time. To save as many deadlines as possible, tardy requests will have to be put on hold. This implies that the mean tardy time would be high in order to achieve a low miss ratio. UG-SSTF provides a mechanism to control the trade-off and to balance the system requirements of reducing both tardiness and miss rate. Through simulation experiments, it is claimed that UG-SSTF can reduce the miss ratio of FD-SCAN and SSEDV by 18-24%. The mean tardy time of UG-SSTF is also much smaller than that of FD-SCAN.

5. MULTIPLE-DISK SYSTEMS

For many real-time applications, a single-disk system is hardly adequate to meet the performance requirements of the applications. The most serious problem of a single-disk storage system is that all the I/O requests of conflicting or non-conflicting transactions contend for one disk [9]. This severely lowers the performance of an RTDBS, especially under an overload situation.

Multiple-disk systems [12, 13] are often used to increase the system’s storage capacity and throughput, and to improve the system’s reliability (through data replication). When a multiple-disk system is incorporated into an RTDBS, many interesting questions arise. First, how shall the data be placed strategically on the different disks so that contention is minimized? Second, if data is replicated on multiple disks, which disk is picked to service a read request? Third, shall the multiple disks share the same request queue? Or, shall the disks hold their own queues? In this section, we briefly discuss the issues of applying a multiple-disk I/O system in an RTDBS [14, 8].

In [9], Cheng et al. study three data placement schemes of RAID for an RTDBS. The simplest way is to organize the disks as independent units and each file is stored on one disk only. An obvious drawback of this scheme is that if there are several transactions requesting the same file, they will compete for
the same disk. Another placement method is to stripe a file across the disks. With disk striping, a file is divided into a number of small blocks. Successive blocks are stored on successive disks in a round robin fashion. Disk striping improves the performance of an RTDBS because if several transactions are requesting different blocks of the same file, the requests can be satisfied by different disks simultaneously. This significantly reduces disk contention.

A problem with the simple disk striping scheme is that an I/O request can be satisfied by only one disk. To further reduce disk contention, a data placement method called disk-striping-replication is proposed [9]. In this scheme, each block of a file is replicated in several disks. The scheduler thus has the flexibility of choosing a disk to service a read request, presumably the one that can satisfy the timing constraint of the request.

In [9], the authors describe a method of picking a disk to service a read request. In the study, it is assumed that each disk keeps its own queue. Also, all read requests are associated with deadlines but writes are not. Each disk queue orders its read requests according to the requests’ deadlines, earliest first. Writes are only serviced when there are no reads.

Given a read request $r$ and a disk $D$ which holds the data requested by $r$, the scheduler estimates whether inserting $r$ into $D$’s disk queue will cause any requests (including $r$) to miss their deadlines. In particular, all requests whose services are pushed back by $r$ will be re-evaluated to see if they would still make their deadlines. Request $r$ is assigned to $D$ only if no missed deadline is caused; Otherwise, another disk which holds the data is tried. If no disk is picked, the read request is aborted. In that case, all other requests of the same transaction will also be aborted. The essence of this prior test algorithm is that the feasibilty of an arriving request for meeting its deadline is estimated before it is executed. By discarding infeasible requests before they are executed, the disks are more effectively utilized.

Through a series of experimental studies [9], it is found that a multiple-disk system using a disk-striping-replication scheme and the prior test algorithm gives a significant improvement in I/O performance over a single-disk system.

6. SUMMARY AND FUTURE WORK

In a disk-resident real-time database system, disk I/O represents a major portion of a transaction’s execution time. Traditional disk scheduling policies aim at minimizing the average response time of disk requests or maximizing the throughput of the disk. In a real-time system, these scheduling policies are not adequate since the timing constraints of the transactions which issue the disk requests are not considered. In this chapter, we discussed various issues concerning the design of an I/O system for real-time database systems. We presented several techniques for assigning priorities to I/O requests. We
looked at several disk head scheduling algorithms which reduce the disk seek overhead and meet the deadlines of requests. The use of multiple-disk systems in an RTDBS was also mentioned.

Finally, we note that relatively little work has been done on multiple-disk systems [8,9]. Many interesting questions are left unanswered:

- To apply the idea of *k-buffer* to a multiple-disk system, we can simply provide each disk with a *k-buffer*. Since write requests are not evenly distributed among the disks, the *k-buffers* of some disks may be under-utilized. An alternative strategy is to provide a single *k-buffer* pool for the system. All the write requests are buffered in this pool. Can the utilization of *k-buffer* be improved by the latter strategy?

- In the *disk-striping-replication* scheme, each block of a file is replicated in several disks. The function of replication is to alleviate the contention of a disk as mentioned in the previous section. However, if the number of replicas for each item is not limited, a lot of write requests will be generated. This can hinder the processing of read requests. In addition, excessive replication can waste a lot of disk space. So we should be careful in controlling the number of replicas for each item. We notice that hot items (i.e., frequently accessed items) generate a lot of read requests. If these items are not replicated, serious contention problem can be created. We should therefore create more replicas for them in order to relieve contention. Other ‘colder’ items require fewer replicas. An interesting question would be: how should one choose the number of replicas for a data item?

- In [9], it is assumed that each disk keeps its own queue. Will the performance be improved if a single queue is shared by multiple disks? Park et.al [8] studied the performance of 2 queueing schemes for a mirrored-disk system, which consists of only 2 disks. Their experimental results showed that sharing a single queue among the disks yields a better I/O performance than assigning a separate queue to each disk. How would their result be generalized to a system with an arbitrary number of disks?

As the price of disks continue to drop and more systems demand better disk performance, we believe that a multiple-disk system will become an essential component of an RTDBS. The above questions are therefore worth considering.

References


This page intentionally left blank.
1. INTRODUCTION

Current databases need logging and recovery to maintain the correctness properties of transactions and consistency of the database under failures. Also, logging and recovery have tremendous performance implications. While there has been a lot of work that deals with logging and recovery algorithms for traditional disk resident databases ([1,2,3,4,5]) and Main Memory Databases (MMDB) ([6, 7]), researchers have not systematically explored the issue of logging and recovery in Real-Time Databases (RTDB). In fact, there is a need for designing new algorithms for logging and recovery in RTDBs because the sequential nature of logging and the lack of time and priority cognizance during recovery are not in tune with the priority oriented and preemptive nature of activities in RTDBs.
This chapter motivates the need to do logging and recovery differently in Real-Time Databases (RTDBs), and presents SPLIT, a partitioned logging and recovery algorithm for RTDBs, and ARUN\(^1\), a suite of logging and recovery algorithms for RTDBs. In order to improve performance, ARUN makes use of the Solid State Disk (SSD) technology that is referred to as Non-Volatile High Speed Store (NVHSS) in this chapter. The primary motivation for this work comes from the fact that conventional logging and recovery are not suitable for priority oriented RTDBs as illustrated in section 2. The other motivating factor is that the characteristics of NVHSS technology, which is capable of reducing unpredictability and improving i/o performance in RTDBs, must be methodically taken into account during the entire design of RTDBs.

The logging and recovery algorithms are based on dividing data into a set of equivalence classes derived from a taxonomy of data and transaction attributes. For example, data can be broadly classified into *Hot* and *Cold* depending on the type of the data (like *critical, temporal* etc.), and type of the transactions (like *high priority, low priority* etc.), that accesses the data. The logging and recovery algorithms assume that the classes of data are disjoint and that the recovery of one class can be done without having to recover the other classes, and if one class is recovered, then the transactions accessing that class can proceed without having to wait for the system to recover the other classes. For instance, in TONICS ([8]), a network management system consists of different kind of data such as the performance, traffic, configuration, fault and security management data. The fault management data, which could potentially get updated frequently, is very critical to the operation that recovering it immediately after a crash can improve the performance of the system.

2. PRINCIPLES UNDERLYING RTDB LOGGING AND RECOVERY

Conventional databases have a sequential log, use the Steal buffer management policy (allowing dirty data from data buffers to migrate to disk), and No Force commit policy (allowing a transaction to commit before the disk resident data is updated). Sequential logging and No Force policies are motivated by the slow disk access times (due to random seeks), and Steal is motivated by the potentially inadequate buffer space. In short, conventional logging and recovery are optimized for databases where both the data and log are disk resident.

This section systematically shows why conventional logging and recovery algorithms will not work well in RTDBs where transactions have timing constraints such as deadlines and some of the data, referred to as temporal data,

\(^1\)ARUN - Algorithms for Recovery Using Non-Volatile High Speed Store.
have validity intervals associated with them after which they become invalid. It would suffice to say that the RTDBs that are considered in this chapter are rich in terms of data and transaction characteristics. In this chapter, data class is a set of data items that have certain properties and these properties can be derived from data types (like critical, temporal etc.) and/or transaction types (like critical, high priority etc.) of transactions that access the data items. Also, temporal data is not explicitly addressed in this chapter.

Following are some of the desirable properties of logging and recovery algorithms for RTDBs.

- Reduced log traffic
- Speedy logging and recovery
- Transaction priority oriented logging and recovery
- Data class oriented logging and recovery

The first two properties imply that logging and recovery should be fast which is a desirable property for conventional databases too. But, real-time is not equivalent to fast ([9]). The last two properties are key to the design of logging and recovery algorithms for RTDBs where the transaction priorities and data properties are taken into account.

To improve logging and recovery performance in RTDBs this chapter combines the following techniques:

- Partitioned and parallel logging and recovery.
- Ephemeral Logging.
- Using NVHSS as a fast persistent backup store.
- Data and transaction class based logging.

Each of the above is discussed in detail.

2.1 PARTITIONED AND PARALLEL LOGGING AND RECOVERY

The following example motivates the need for partitioning the log in RTDBs. In RTDBs, usually, cpu scheduling, i/o scheduling and concurrency control conflict resolution are based on transaction priorities ([10, 11, 12, 13]). But, conventional (non-partitioned) logging and recovery techniques are sequential activities that do not work well in such a priority oriented and preemptive RTDB setting. The following example illustrates that it is difficult to attain the key (last two) desirable real-time properties for RTDB logging and recovery algorithms using conventional sequential log. Let $T_1$ be a high priority transaction updating data items $H_1$ and $H_2$. Let $T_2$ be a low priority transaction updating data items $C_1$, $C_2$, $C_3$ and $C_4$. Let the sequence of updates be $W_1(H_1)$, $W_2(C_1)$, $W_2(C_2)$,
$W_3(C_3)$, $W_5(C_4)$, and $W_1(H_2)$. This situation can arise in a multiprocessor with combined logs where $T_1$ and $T_2$ are executing on different processors\(^2\).

If transaction $T_1$ wants to commit, then it has to flush the log until $W_1(H_2)$ which means it has to flush the log records of a low priority transaction.

In this chapter, this phenomenon of high priority requests doing work for low priority requests is referred to as Priority Diversion\(^3\). Priority diversion will increase the response times of high priority transactions thus making the conventional sequential logs unsuitable for RTDBs. Similarly, recovery of important data items cannot be done separately, but has to be integrated with the reading of log records of unimportant data items (and recovery of unimportant data items). Let $H_1$ and $H_2$ be hot (frequently accessed) data items and $C_1$, $C_2$, $C_3$ and $C_4$ be cold (infrequently accessed) data items. During recovery, in order to recover hot data items $H_1$ and $H_2$ the log records of cold items $C_1$, $C_2$, $C_3$ and $C_4$ have to be read (and possibly recovered). This example shows that a sequential log, and recovery using that log will pose problems in a priority based, preemptive RTDB setting.

Partitioned and parallel logging and recovery in itself is not a new idea. Parallel logging in conventional databases was explored in [14]. In [15], data placement, logging and recovery issues in real-time active databases are discussed which includes the idea of partitioned logs in a real-time context. In [16], log partitioning and related performance evaluation for real-time MMDB are presented. In all this work, there are no details of partitioned logging and recovery algorithms together with details of required data structures.

### 2.2 EPHEMERAL LOGGING

While the last subsection motivated the need to partition the log in RTDBs, this section presents ephemeral logging technique which helps to partition the log, and reduce the amount of log to process at recovery. We claim that ephemeral logging is suitable for RTDBs and substantiate this claim by presenting novel RTDB logging and recovery algorithms that take advantage of ephemeral logging.

An extreme case of partitioning would be to write separate logs for each transaction which will facilitate logging based on transactions’ priorities for RTDBs. Using example 1, separate logs would imply that transactions $T_1$ and $T_2$ will have separate logs.

Now, log of $T_1$ can be flushed to the stable storage without having to flush log of $T_2$ thus avoiding priority diversion. Also, with transaction based logs, the log of a particular transaction could be discarded at the end (commit or abort) of that

---

\( ^2\)This can also happen in a single processor case if $T_1$ is waiting for a disk I/O or there is priority inversion.

\( ^3\)to indicate that high priority requests are diverted from what they are supposed to be doing.
transaction, if the data pages accessed by that transaction are forced at commit. Transaction based logs that can be discarded at the end of the transaction are referred to as **ephemeral transaction logs** (as in [17]). Transaction based logs that cannot be discarded are referred to as **non-ephemeral transaction logs**.

The advantages of ephemeral logging are

- Transaction priority based logging and recovery is possible making it suitable for RTDBs.
- checkpointing becomes unnecessary, since data items are forced at commit.
- After a system crash, potentially, any data item can be undone on demand since there is not much log to process.

The main disadvantages of ephemeral logging are

- The lack of permanent system-wide log that can be used as an audit trail.
- There is a need for a high speed stable storage to force data items and logs because ephemeral logging does not work well with disk-resident log or data ([18]).

### 2.3 USING NVHSS

NVHSS technology is imminent. The presence of database accelerators and log accelerators using SSD only confirm the hypothesis that SSD will become an integral part of future databases ([19]). However, we strongly believe that database designers should take Non-Volatile High Speed Store (NVHSS) such as SSD into account in their initial designs and not just use them in a brute force and adhoc manner, as it is used now, to improve performance.

The following reasons make NVHSS a suitable technology for RTDBs:

- Log write time becomes low, if the logs are written to NVHSS.
- Force time becomes low, if the data items are forced to NVHSS.
- Unlike disks, NVHSS access time is a constant. This improves the predictability of the system.
- Ephemeral logging, that is suitable for RTDBs, is facilitated because all the data pages can be forced to NVHSS and the logs can be written to NVHSS.

The fact that adding NVHSS to a database will improve the performance is obvious ([20]). But, NVHSS could be used as a general second level buffer, a data class specific second level buffer, a logging device or some meaningful combinations of the above. For instance in [21], NVHSS is used just a fast stable write buffer. Also, whereas in [17] logging and recovery algorithms for databases with non-volatile main memory are presented certain subtle, practical issues like the possibility of corruption of main memory by

---

1 A page (4K) write that takes 20 ms (5 ms) in a random (sequential) disk setting takes 100mus in NVHSS.
system failures are not considered. In [17], the study considers an integrated safe where the approach is to write page based before images, which can be memory consuming. Also, [17] does not deal with priority based, preemptive RTDBs. Basically, these studies don’t consider the requirements of RTDBs to design the logging and recovery algorithms in a systematic way.

2.4 DATA AND TRANSACTION CLASS BASED LOGGING

In this section, we argue that there are advantages to maintaining separate logs, but not one per transaction, but one per transaction class. Transaction class oriented logging is a coarser granularity of logging than individual transaction oriented logging, i.e., each transaction class has a sequential log associated with it. This can work well if the transactions from different classes do not access the same data pages/objects at the same time. Otherwise, with transaction class based logging the log records of a data item can be in different logs and all these log records (and hence the logs) might have to be processed to recover the data item.

On the other hand, if data classes are disjoint (a reasonable assumption), then each data class can have its own log. For instance, in TONICS, the fault management data is independent of the traffic management data and it is desirable that the fault management data is recovered first. Also, the disjoint logs of related data can work well with the notion of eventual consistency ([22]) where even the related data can be inconsistent for some time and then they are made consistent at some later point in time. This dataclass oriented logging will facilitate data class oriented recovery which is a desirable feature for logging and recovery in RTDBs. This will facilitate partitioned and parallel logging and recovery of different classes of data items. The logging and recovery can be prioritized depending on the data class. Classes of data which are immediately needed by transactions can be recovered first and the remaining classes of data can be recovered in the background. While data class oriented logging is suitable for systems with or without NVHSS, transaction based logs work well only if NVHSS is present. Of course, NVHSS will reduce the overheads of logging and commitment.

Having introduced the ingredients of partitioned logging, ephemeral logging, NVHSS, and transaction/data classes, we now discuss the attributes of data in detail to pave the way for the development of a suite of algorithms that mix the ingredients depending on the data attributes of interest.
3. DATA, TRANSACTION AND LOG TYPES IN RTDBS: A TAXONOMY

Figure 9.1 gives a taxonomy of data attributes along seven dimensions. The possible values that an attribute can have are given along with the attribute in brackets. The data type attribute is the type of the data which could be critical, temporal, hot or cold. Data types themselves are not completely orthogonal. For instance, there could be a cold data item that is critical.

The other data attributes are straightforward. The transaction type attribute is the type of the transaction that accesses the data. The Home Media is the media where the data resides. The steal and force attributes stand for the steal and force policies that are used to manage the data. Lock granularity stands for the granularity of locking on the data. The log type attribute stands for the type of the log that is used to manage the data. The log type has its own attributes. The home media attribute indicates where the log resides and the temporality attribute indicates if the log is ephemeral or permanent.

The data type and the type of the accessing transaction are application dependent attributes. All other attributes are policy decisions that the RTDB makes to process the data. These attributes, i.e., the home media, steal policy, force policy, locking granularity and log type are closely related attributes and are not totally orthogonal. For instance, a data item can not have ephemeral log and not be forced at commit time.

---

5The data type of this item would be critical.
Data class, as defined in section 2., is a set of data items that have certain properties and these properties can be derived from data types and/or transaction types of transactions that access the data items. Log class is derived from the temporality of the log, home media of the log, home media of the data, lock granularity, and Steal and Force policies.

Let $L_1, L_2, ..., L_n$ be the classes of log. Let $C_1, C_2, ..., C_m$ be the classes of data. Now, a general logging scheme for the different classes of data can be expressed as a set of 2-tuples \{(c_1, l_1), (c_2, l_2), ..., (c_q, l_q)\} where $c_i$ is a set of data classes \{C_i, C_j, ..., C_k\} such that if $\exists C_k \in c_i$ then $C_k \not\in c_j, \forall i, j, i \neq j, 1 \leq i, j \leq q$ and $l_i$ is a log class $L_k$ where $1 \leq k \leq n$. Also, if $l_i = l_j, i \neq j$, then the physical logs of $c_i$ and $c_j$ are separate. Informally,

- A set of data classes $(c_i)$ is associated with one log class $(L_k)$.
- One data class cannot belong to two different sets $c_i$ and $c_j$, i.e., a data class cannot be logged in two different ways.
- If two different sets of data classes, say $c_i$ and $c_j$ are associated with the same class of log ($l_i = l_j$), say $L_k$, then the physical logs in which the log records for $c_i$ and $c_j$ are written are different. Also, the physical log to which the data classes that belong to a particular set of data classes $c_i$ is written is the same.

For the sake of simplicity and without loss of generality, the following specific log and data classes are considered in the rest of this chapter. The first class, referred to as $EL$, is ephemeral logging where the data and log are NVHSS resident, with Force policy and object level locking. There are two subclasses of $EL$, namely, $EL_s$ and $EL_n$ that refer to $EL$ with Steal and $EL$ with No Steal, respectively. The second class, referred to as $PL$, is permanent logging where the data and log are disk resident with No Force/Steal policies and object level locking. Adding NVHSS to the second class can give rise to other subclasses of logging and recovery algorithms where either the data is NVHSS resident or the log is NVHSS resident or both are NVHSS resident.

A system can have combinations of the $EL$ and $PL$ algorithms where some classes of data have $EL$ and some $PL$. Also, two primary equivalence classes of data with respect to logging and recovery are proposed. The first class consists of critical, temporal, hot and cold items that are accessed by critical or high priority transactions. The second class consists of cold data items accessed by low priority transactions. It is possible to have different sets of equivalence classes in practice. This proposed classification, apart from being a pedagogical aid, is also a reasonable way to classify data in RTDBs. The claim that the classification is reasonable will be strengthened, if the first class of data items comprises a small percentage of the total data items in the database, which is a reasonable assumption and a necessary one for the ARUN suite of algorithms to perform well with limited amount of NVHSS. For instance, the access skew in TPC-C benchmarks is such that 84% of the access go to about
20% of the tuples and in terms of page accesses 75% of the page accesses go to 25% of the pages ([23]). Also, given the application characteristics, data types and transaction types, coming up with these equivalence classes in itself is a non-trivial topic of research. For ease of exposition, the first data equivalence class is referred to as Hot and the second equivalence class is referred to as Cold.

Traditional logging scheme can be expressed as \{\{(Hot, Cold), PL\}\}, i.e., permanent, combined disk based log, with Steal No Force policies and object level locks for disk resident data items for any data type and transaction type.

The next three sections present the RTDB logging and recovery algorithms that have been developed with the principles discussed in section 2, as the guidelines. The algorithms are SPLIT, a logging and recovery algorithm for RTDBs without NVHSS, and General ARUN (g-ARUN), Real-Time ARUN (rt-ARUN) and Hybrid Real-Time ARUN (hrt-ARUN) that belong to ARUN, a suite of logging and recovery algorithms for RTDBs with NVHSS.

4. SPLIT ALGORITHM

Logging in SPLIT can be expressed as \{\{(Hot, PL), (Cold, PL)\}\}. Essentially, SPLIT uses separate permanent sequential logs for Hot and Cold items. In SPLIT, there are separate, permanent sequential logs for Hot and Cold data items. The log for Hot items is physically realized by having two logs, namely, HotUndoLog and HotRedoLog. The reasons behind this split are:

- The amount of log generated by Hot items will be usually higher than the amount of log generated for Cold items. Hence, splitting the Hot log further will enhance partitioned and parallel logging.
- To show how a log can be split into undo and redo parts could prove useful. For instance, if page level locks are used with No Steal policy for Hot items, then HotUndoLogs need not be flushed but used only for transaction aborts.

The logging technique for PL class log assumes page level redos and operation level undos with compensation logs. This is similar to ARIES, a logging and recovery algorithm used in commercial databases ([4]).

4.1 DATA STRUCTURES

In this subsection, details of the data structures that are used in the SPLIT algorithm are presented. It should be noted that some of the data structures that are presented here are also used in the ARUN suite of algorithms. Detailed pseudo-code of all the algorithms are presented in [18]. Following are some of the important attributes of a permanent, sequential log:

- currLSN - The Log Sequence Number (LSN) of the log record that is currently being used.
- stableLSN - The LSN of the log record until which the log has been stabilized.
The important fields of a log record are as follows:

- **Type** - Type of the log record. It can be compensation, update, or commit related such as commit, abort or prepare.
- **t** - Identifier of the transaction that wrote the log record.
- **ob** - Identifier of the object to which the updates of this log record were applied.
- **updt** - The operation responsible for the log record.
- **Undo-Details** - Details of the undo operation to be performed. Present only in update type log records.
- **Redo-Details** - Details of the redo operation to be performed. Present only in update type log records.

The **XTable** entry consists of the following fields:

- **t** - Identifier of the transaction that is represented by the entry.
- **State** - State of the transaction that could be active, prepare or commit.

Some of the important fields in the transaction data structure are:

- **hotUndoLastLSN** - The LSN of the latest log record written by the transaction in the HotUndoLog.
- **hotRedoLastLSN** - The LSN of the latest log record written by the transaction in the HotRedoLog.
- **coldLastLSN** - The LSN of the latest log record written by the transaction in the ColdLog.
- **hotUndoNextLSN** - The LSN of the next record to be processed during rollback in the HotUndoLog.
- **coldUndoNextLSN** - The LSN of the next record to be processed during rollback in the ColdLog.

The commit log record for SPLIT consists of the following important fields:

- **hotRedoPrevLSN** - The LSN of the previous log record written by the transaction in the HotRedoLog.
- **hotUndoLastLSN** - The LSN of the last log record written by the transaction in the HotUndoLog.
- **coldLastLSN** - The LSN of the last log record written by the transaction in the ColdLog.

The **4.2 NORMAL PROCESSING**

Below is a description of how different events are processed in SPLIT. When a transaction begins execution, the transaction is added to the **XTable**. When a transaction updates an object, depending on whether the object is **Hot** or **Cold**, the corresponding undo and redo log records are added to **HotUndoLog** and **HotRedoLog** or **ColdLog**, respectively. On commit of a transaction, a commit record is written to the **HotRedoLog** and the Write Ahead Log (WAL) protocol is followed where the log records are forced to stable **HotRedoLog** and **ColdLog**. If there are **Hot** objects whose uncommitted changes could migrate to disk then the **HotUndoLog** is also flushed. In case of an abort of a transaction, the data updated by the transaction is undone, compensation log records (like in ARIES) are written, an abort record is written and finally the log is flushed. The compensation log Record (CLR) will be written both in the **HotRedoLog** and **HotUndoLog**. The redo part will be written in the
4.3 CRASH RECOVERY

In SPLIT, crash recovery is done in two phases where each phase has a forward and backward pass. In the first phase, the data items belonging to Hot are recovered by processing the redo and undo logs (HotRedoLog, HotUndoLog). In the second phase, ColdLog is processed to recover data items belonging to Cold. The forward pass of the first phase consists of analysis and redo of HotRedoLog. The backward pass consists of undoing the Hot data items using the HotUndoLog. The second phase consists of two passes (like ARIES), a forward pass of ColdLog where all updates are redone and a backward pass where the losers are undone.

Two transaction lists that are maintained during recovery, namely, Winners, Losers. The Winners consists of all transactions that are committed and have to be redone, and the Losers consists of transactions that have to be undone, i.e., active transactions that did not have a chance to commit (or abort) at the time of crash. Before the first phase of recovery starts, Winners = Losers = φ. During phase 1, if a transaction begin record is encountered, an entry is made in the XTable, and the transaction is added to the Losers. If a redo record of an update on an object is encountered that redo is done. If a commit or abort record of a transaction is encountered then the transaction is deleted from the Losers. This Losers list is used in the backward passes of both the phases.

SPLIT algorithm can be easily extended to accommodate more data classes. For instance, say, if critical data has to be a separate class, then the data structures will have to be augmented with critical LSN and Critical Log. The order of recovery will be to first process the CriticalLog, then process the HotLog and finally process the ColdLog. The only constraint is that the commit record should be written to the log that will be recovered first because the Losers list is necessary for the backward pass (undo recovery). Also, SPLIT algorithm will perform well only if there are multiple log disks because the advantage of sequential disk access might be lost if there are multiple logs and a single disk.

5. GENERAL ARUN (G-ARUN)

The previous section discussed SPLIT, a logging and recovery algorithm for disk resident databases that permits parallel and partitioned logging and recovery. This section presents g-ARUN, an algorithm that belongs ARUN suite of algorithms with ephemeral logging that is facilitated by the presence of NVHSS. In all the ARUN algorithms that are presented, the XTable is also
made NVHSS resident, thus making it easier to find out during crash recovery which transactions were active before the crash. The XTable entry can be used as the commit record, if necessary.

5.1 NORMAL PROCESSING

As seen in section 3., logging in g-ARUN can be expressed as $\{[\{\text{Hot}, \text{Cold}\}, \text{EL.}\}\}$. In g-ARUN, there is one combined ephemeral transaction undo log ($\text{undoLog}$) per transaction for both Cold and Hot items. When a data item is flushed, the undo log of all transactions that accessed this data item will be flushed to NVHSS.

All the functions in the ARUN suite take the log combination type as a parameter. For g-ARUN the log combination type is combined and for rt-ARUN and hrt-ARUN the log combination type is split. In g-ARUN, like in SPLIT, when a transaction begins execution, the transaction is added to the XTable. When a transaction updates an object, since the log combination type is combined, an undo log record is written to the transaction’s ephemeral undo log, i.e., undoLog. At commit, the objects updated by the transaction are atomically flushed to NVHSS. In case of an abort of a transaction, the data updated by the transaction is undone using the undoLog.

5.2 CRASH RECOVERY

In ARUN, crash recovery is done by loading the XTable from NVHSS and undoing all transactions that were active at the time of crash. The active transactions are undone using the transaction specific UndoLog.

The problems with g-ARUN are

- While recovery can be made cognizant of transaction priority, i.e., undoing of transactions can be according to transactions’ priorities, it is very difficult to make the recovery cognizant of data priority. For instance, to recover the Hot items first, one has to process all the log records of a transaction.
- Hot items could be stolen to disk. This implies that at the time of recovery Hot items might have to be read from disk to be undone.
- Ephemeral logging means that there is no system-wide log to keep track of what happened.
- Forcing all the data items to NVHSS implies a large NVHSS. Also, there has to be a flusher daemon process to flush the pages from NVHSS to Disk. If this flusher is not aware of Hot and Cold classes then during recovery the disk might have to be accessed to recover Hot data items that the flusher might have flushed to make space in the NVHSS.

These considerations motivate rt-ARUN.
6. REAL-TIME ARUN (RT-ARUN)

Logging in rt-ARUN can be expressed as $\{(\text{Hot}, EL), (\text{Cold}, EL)\}$. There are two ephemeral transaction based undo logs for each transaction, namely, $hotUndoLog$ and $undoLog$ for $Hot$ and $Cold$ items, respectively. Basically, the undoLog that was used in g-ARUN is used for $Cold$ data items and there is a separate $hotUndoLog$ for $Hot$ data items. When a data item is flushed, depending on its type, the corresponding undo log of all the transactions that accessed this data item will be flushed to NVHSS.

The log combination type for rt-ARUN is $split$. In rt-ARUN, like in SPLIT, when a transaction begins execution, the transaction is added to the $XTable$. When a transaction updates an object, since the log combination type is split, depending on the class of the object, an undo log record is written either to $undoLog$ or $hotUndoLog$. In case of a commit, the objects updated by the transaction are atomically flushed to NVHSS. In case of an abort of a transaction, the data updated by the transaction is undone using the $undoLog$ and $hotUndoLog$.

Algorithm rt-ARUN, like g-ARUN, suffers from the problem of not having a system-wide log to keep track of what happened to the system and the problem of requiring a large NVHSS to force all the data items.

7. HYBRID-REAL-TIME ARUN (HRT-ARUN)

Logging in hrt-ARUN can be expressed as $\{(\text{Hot}, EL), (\text{Cold}, PL)\}$. In this case, the NVHSS need not be as large as in the case of g-ARUN or rt-ARUN. If the $Hot$ items form a reasonably small subset of the database, then the amount of NVHSS to force the data items need not be very large. Also, the permanent, sequential $ColdLog$ can act as a system-wide log where the important details of the high priority transactions can also be logged. This will be an approximate log for $Hot$ data items in the sense that not all committed details about them will be available, because this $ColdLog$ will not be flushed when $Hot$ data items are committed or when high priority transactions commit.

7.1 NORMAL PROCESSING

The log combination type for hrt-ARUN is $split$. In hrt-ARUN, like in SPLIT, when a transaction begins execution, the transaction is added to the $XTable$. When a transaction updates an object, the $Hot$ items are processed as in rt-ARUN and the $Cold$ items are processed as in SPLIT. In case of a commit, the $ColdLog$ is flushed as in SPLIT and the $Hot$ objects are atomically flushed to NVHSS as in rt-ARUN. In case of an abort (Hybrid-ProcessAbort), the transaction based $hotUndoLog$ is used to undo the $Hot$ items and the system-wide $ColdLog$ is used to undo $Cold$ items.
7.2 CRASHRECOVERY

Crash recovery in hrt-ARUN is done in two phases. The first phase consists of recovering the Hot items using the transaction based hotUndoLogs of all active transactions. The second phase, that is similar to the second phase of SPLIT, consists of two passes, a forward and a backward pass, where the ColdLog is used to recover the Cold items.

8. SUMMARY AND FUTURE WORK

In many real-time applications, a considerable amount of data is kept in main memory. However, even though main memory costs are dropping and sizes are increasing, the sizes of the databases are also increasing very rapidly. As in [24], we believe that the assumption that the database is only partially main memory resident must underlie the design of a practical database system.

Motivated by this assumption, this chapter showed the need to design novel logging and recovery algorithms for disk resident RTDBs by showing that conventional logging and recovery algorithms are not suitable in a priority oriented RTDB setting. We presented a taxonomy of data characteristics and proposed two data classes that are derived from data types and transaction types. Based on a set of driving principles, we methodically developed a suite of algorithms targeted at real-time databases.

SPLIT, a logging and recovery algorithm for disk-resident RTDBs without NVHSS, was also presented. The main idea behind SPLIT is to partition the log based on data classes thus enhancing parallel and partitioned logging and recovery. Also, we showed the need to design RTDBs with NVHSS which is capable of reducing unpredictability and improving the i/o performance in RTDBs. This was shown by exploring how NVHSS impacts logging and by presenting ARUN, a suite of novel logging and recovery algorithms that uses NVHSS to facilitate ephemeral logging where there are separate logs for each transaction that can be logged and recovered in priority order and discarded when the transactions complete. Three specific algorithms were presented, namely, g-ARUN, rt-ARUN and hrt-ARUN. While g-ARUN and rt-ARUN use pure ephemeral logging, hrt-ARUN uses both ephemeral logging and permanent, sequential logging. Details of correctness and performance analysis of the above algorithms are presented in [18].

There are a number of research problems to be investigated:

• Performance Evaluation: Currently, the proposed algorithms are being implemented on RADEX (Real-Time Active Database Experimental System) for the purpose of experimental evaluation.

• Dynamic Classes: The data classification is static and hence the algorithms based on this classification is also static. What happens if a data
item migrates from one class to another is an issue to be considered for future work.

References


Chapter 10

OVERLOAD MANAGEMENT IN RTDBS

Jorgen Hansson  
*Department of Computer and Information Science*  
Linköping University, Linköping, Sweden

Sang H. Son  
*Department of Computer Science*  
University of Virginia, Charlottesville, USA

1.  INTRODUCTION

A transient overload denotes a system state that lasts for a finite duration where the resource demand exceeds the available system resources. As a consequence the system fails to fully enforce its consistency requirements (temporal, logical, external). To minimize the effects of a transient overload, resource management should ensure controlled and graceful performance degradation. While several scheduling policies, e.g., Earliest Deadline First and Least Slack, produce optimal results under non-overload conditions, they generally have poor performance in overload situations.

Real-time database systems often operate in situations of imminent overload, and hence, the system should have, in addition to an overload detection mechanism, a mechanism and a policy defining how a pending transient overload should be resolved. Resource management in real-time database systems can be divided into several activities: *admission control* determining which transactions should be granted system resources; *scheduling* determining the execution order of admitted transactions; *overload management* determining how to resolve transient overloads. Hence, scheduling focuses on *when* to execute transactions, and admission control and overload management focus on selecting *which* transactions should be allowed to execute.

Possible overload resolution policies include rejecting and terminating transactions during overloads, transaction migration, and transaction replacement. These techniques change the resource demand, requested by transactions, to a level that can be handled by the system. Additional approaches to resolving
overloads are to add more resources to the system, or change the correctness criteria of the system.

Transaction rejection implies that transactions causing an overload are rejected upon their arrival. In contrast, transaction termination implies that overloads are resolved by aborting transactions eligible for execution, i.e., transactions that have been admitted to the system. The resource demand can be decreased by executing transactions partially, e.g., imprecise computation [5] where transactions are decomposed into one mandatory and one optional task, where only mandatory parts are executed during transient overloads, or replace transactions with alternative transactions that have smaller resource requirements but still produce satisfactory results, although with less quality than the original ones. An overload can also be resolved by increasing available resources, e.g., in distributed database systems migration of transactions to a node with surplus capacity can resolve the local overload.

In this chapter, we introduce a scheduling architecture, and an overload resolution algorithm that resolves transient overloads by rejecting non-critical transactions and replacing critical ones with contingency transactions. The algorithm, denoted OR-ULD (Overload Resolution - Utility Loss Density), has the following characteristics:

- it operates in transient overload conditions, i.e., only invoked in case a transient overload is detected;
- it resolves transient overload by (i) scrutinizing current resource reservations made by admitted transactions, and, if necessary, (ii) revoke previous reservations;
- it handles multi-class transaction workloads, where classes are discriminated by their criticality;
- it enables the use of multiple overload resolution strategies, e.g., transient overloads can be resolved by controllably drop non-critical transactions and selectively replace critical transactions with contingency transactions;
- it separates overload management from scheduling and can therefore be combined with various scheduling policies, given that the scheduler is able to detect transient overloads and can indicate in which interval the overload is occurring.

2. TERMINOLOGY AND ASSUMPTIONS

In this work, we consider workloads consisting of preemptable transactions that have hard critical or firm deadlines; hard critical transactions are sporadic and firm transactions are aperiodic. Each transaction $\tau_i$ is pre-declared and
pre-analyzed with known worst-case execution time $w_i$. This information is made available to the scheduler and the admission controller as a transaction arrives in the system. In addition, each critical transaction $\tau_i$ has a contingency transaction $\bar{\tau}_i$, which is an autonomous executable entity. Their deadlines and processing requirements are relative to their original transactions, and can be invoked for execution thereby replacing the original transaction. Further, a critical transaction is said to successfully complete if, and only if, the corresponding original transaction $\tau_i$ or contingency transaction $\bar{\tau}_i$ completes before the corresponding deadline $d_i$ or $\bar{d}_i$.

The following temporal attributes related to the temporal scope constituted by a transaction $\tau_i$ are known (the corresponding temporal attributes of a contingency transaction $\bar{\tau}_i$ are indicated by a bar):

- **ready time** $r_i$ — the earliest time at which a transaction may start its execution (transactions are ready upon arrival);
- **deadline** $d_i$ — the time at which the transaction execution should be complete;
- **deadline tolerance** $\delta_i$ — represents tardiness that can be allowed, i.e., when transaction execution must be complete;
- **worst-case execution time** $w_i$ — the upper bound on execution time of the transaction, independent of the current state of the database and the system;
- **remaining execution time** $\zeta_i$ — upper bound on remaining execution time of the transaction;
- **value function** $v_i(t)$ — representing the importance and criticality of the transaction as a function of its completion time $t$;
- **blocking time** $b_i$ — the upper bound on blocking time that this transaction may experience due to resources held by lower priority transactions;
- **worst-case critical section** $w_{crit_i}$ — the upper bound on execution time of the longest critical section of the transaction; and
- **abort time** $o_i$ — the upper bound on time needed to abort the transaction.

Generally, the contingency transaction $\bar{\tau}_i$ has significantly smaller resource and processing requirements in comparison to the original transaction $\tau_i$. As the contingency transaction $\bar{\tau}_i$ is a substitute action, producing a result of less quality, it is of interest to minimize the number of replacements. It is therefore usually preferred to execute the original transaction $\tau_i$ if its timeliness is not jeopardized (as opposed to execute the contingency transaction $\bar{\tau}_i$). In
our work, the desire of completing the original transaction as opposed to the contingency transaction is represented by their value functions, describing the utility contributed to the system once the transaction completes.

A value function, denoted $v_i(t)$, or $\bar{v}_i(t)$ respectively, describes the relative importance of the transaction $\tau_i$ and $\tau_i$ in relation to other transactions. In our work, transactions have value functions that are piecewise constant. This is a simplification of a more general value function where the utility contributed to the system may change more freely over time. We use the following mathematical model to express the value function $v_i(t_1)$ of a transaction $\tau_i$, yielding the utility contributed to the system when the transaction terminates at time $t_i$:

$$v_i(t_i) = \begin{cases} v'_i(t), & r_i \leq t_i \leq d_i \\ v''_i(t), & d_i < t_i \leq d_i + \delta_i \\ -p_i, & t_i > d_i + \delta_i \quad 0 \leq p_i \leq \infty \end{cases}$$

Throughout our discussion, we also assume that transactions are scheduled by the Earliest Deadline First algorithm, and that transactions are preemptable. Moreover, we assume a dedicated processing element for performing resource management services (i.e., admission control, scheduling, and overload handling). The database is main-memory resident, and hence, the blocking of transactions due to disk delays are avoided. The concurrency control scheme adopted is OCCL-SVW (optimistic concurrency control using locking and serial validation write) [4]. Our choice was based on previous performance studies that showed that OCCL-SVW performs better than two-phase locking. However, our proposed framework and overload resolution strategy are independent of OCCL-SVW and can be combined with other concurrency control schemes.

3. OVERLOAD RESOLUTION ALGORITHM

3.1 SCHEDULER ARCHITECTURE

The architecture supporting the scheduling activities consists of an admission controller, a transaction scheduler, an overload resolver, a dispatcher, and transaction queues. The **admission controller** tests for schedulability of new transactions upon their arrival. The **transaction scheduler** performs scheduling and concurrency control of admitted transactions by outlining an initial and feasible schedule. The scheduling policy may be any optimal dynamic scheduling policy (in the sense that if there is a feasible schedule, it will find one). In our work the scheduling policy is the well-known Earliest Deadline First (EDF), blocking is handled by Stack Resource Policy (SRP) [1] and, hence, admissions are based on the schedulability with EDF using SRP. Our proposed **overload resolver** is invoked when a transaction, given its original resource requirements,
cannot be admitted to the system. The resolver computes the amount of processing time that needs to be released in order to resolve the transient overload, and initiates a negotiation of the requirements of the admitted transactions and the new transaction. The \textit{dispatcher} performs dispatching according to the outlined schedule. The \textit{ready queue} contains the set of admitted but not yet completed transactions.

\section{3.2 COMPONENT INTERACTIONS}

Figure 10.1 shows data flow (solid lines) and control flow (dashed lines) among the components. New transactions arriving to the system are placed in an arrival queue. New transactions are then tested for schedulability by the admission controller. In the case of transient overloads, the admission controller acts as a transaction filter that decides whether to admit or deny new transactions. Thus, admission of a new transaction is granted only if it is schedulable with already admitted transactions considering the execution times. Hence, the admission controller guarantees that the transaction scheduler does not become overloaded, i.e., it is able to schedule admitted transactions, although the overall system may be facing a transient overload. If a transaction passes the
schedulability test, it is admitted to the system and sent to the scheduler. If a transaction fails the test, the admission controller invokes the overload resolver which analyzes earlier resource reservations in comparison to the needs of the newly arrived transaction. That is, the overload resolver determines whether or not it is worthwhile to accept a new transaction, given the potential loss of utility due to de-allocation of reserved resources. Resources can be released either by dropping a transaction, or by replacing the original transaction with a contingency transaction. We call these actions overload resolution actions (ORAs). The overload resolver develops an overload resolution plan (ORP), which consists of a carefully selected set of ORAs.

If it is decided that the new transaction should be admitted, then the overload resolver informs the scheduler, which executes the ORP. If the transaction should be rejected, then the overload resolver notifies the admission controller to reject it.

Information about admitted transactions is stored in the admitted transaction table. The workload information in this table is accessed by the admission controller, the scheduler, and the overload resolver (updates are made by the admission controller and the scheduler only).

3.3 ADMISSION CONTROL ALGORITHM

The primary task of the admission controller is to perform a schedulability test of new transactions upon their arrival, determining whether or not a new transaction should be admitted for execution. In our case, using EDF and SRP, the admission controller is pessimistic in the sense that it performs admission based on the worst-case execution times and does not take any chances. Processing time is initially reserved once transactions are admitted. Hence, transactions that are admitted are given a prognosis that they will get their desired amount of resources. Resource reservations are scrutinized only in the case of a transient overload. In the worst-case overload scenario, given the load hypothesis, this implies that all non-critical transactions must be terminated, and transactions must be replaced with their contingency transactions.

Let $\tau_A$ denote the ordered set of admitted transactions as follows: $\tau_A = \{\tau / 1 \leq i \leq n\}$ where $d_{i+1} \leq d_i$ for $1 < i \leq n$. Moreover, let $\tau_n$ represent a new transaction that is tested for admission. Hence, the schedulability test is performed on the unified set, denoted $\tau$, of admitted transactions and $\tau_n$, i.e., $\tau = \tau_A \cup \{\tau_n\}$.

A critical section is nontrivial if it involves a resource that can cause blocking of other transactions. With SRP, the blocking time $b_i$ is given by the maximum

---

1It should be noted that admitted transactions are not guaranteed to complete on time, since transactions may be replaced/dropped during overloads or aborted due to data conflicts not related to the scheduler.
worst-case execution time of the longest nontrivial critical section of any transaction $T_i$ such that $d_i < d_j$ ($i \neq k$). This maximum includes the worst-case execution time of all the critical sections of other transactions that might subject $T_i$ to priority inversion while it is in the critical section.

Assuming that the system is at a time $t_0$ before which no transactions are requested, the following condition should be satisfied in order to guarantee the schedulability of a set of periodic and aperiodic transactions using SRP with EDF [1] (deadline $d_i$ is relative to $t_0$):

**Theorem 1 (Baker, 1991)** A set of $n$ (periodic and aperiodic) transactions is schedulable by EDF scheduling with SRP semaphore locking if

$$\forall k = 1, \ldots, n \left[ \left( \sum_{i=1}^{k} \frac{w_i}{d_i} \right) + \frac{b_k}{d_k} \leq 1.0 \right]$$

*Proof.* See [1, p. 83].

Assume that the system has executed for some time and that the currently executing transaction at time $t_0$ is not in a critical section. The following condition should be satisfied in order to guarantee the schedulability of a set of periodic and aperiodic transactions using SRP with EDF (the proof is given in [2]).

**Corollary 1**

A set of $n$ (periodic and aperiodic) transactions is schedulable by EDF scheduling with SRP semaphore locking if

$$\forall k = 1, \ldots, n \left[ \left( \sum_{i=1}^{k} \frac{\xi_i}{d_i} \right) + \frac{b_k}{d_k} \leq 1.0 \right]$$

*Proof.*

4. **OVERLOAD MANAGEMENT ALGORITHM**

We now define the OR-ULD algorithm. First, temporal intervals necessary for the OR-ULD algorithm are introduced, and then we give a detailed description of the algorithm with an example that illustrates how the algorithm works.

4.1 **OVERLOAD INTERVALS**

If we are about to resolve an impending transient overload by releasing resources, and given EDF’s overload behavior, it is important to have an understanding of the time interval over which the transient overload is extending.
The reason is that to resolve an overload it is not the case that just any transaction can be dropped or replaced (e.g., consider the domino effect caused by EDF).

De-allocation of processing time is done in an interval referred to as the critical overload interval (COI). Formally, we define COI as follows:

**Definition 2 (Critical Overload Interval)** The critical overload interval \([t_0, t_1]\) denotes the interval starting at time \(t_0\) when the overload is detected, and ends at time \(t_1\) when the first transaction is missing its deadline, i.e., \(COI = [t_0, t_1]\) where \(t_1 = \min \{\tau_i\}\) and where \(\tau\) is the set of tardy transactions in \(\tau\), i.e., \(\tau = \{\tau_i|\tau_i \in \tau \land \tau_i > d_i}\)\).

Note, resolving the overload here using the minimum required time is guaranteed to resolve the transient overload. However, releasing the same time outside the critical overload interval will not resolve the transient overload. Let us formally define critical overload interval. Again, consider the situation when the set of transactions can be feasibly scheduled with EDF, but where is not schedulable.

### 4.2 Overload Resolver

The basic idea behind the overload resolver is to generate an overload resolution plan (ORP) that resolves the impending transient overload by de-allocating time from previously admitted transactions. Thus, in the case of an overload, the resource reservations are scrutinized, and they may be reduced by substituting transactions having contingency transactions, or de-allocated by dropping transactions. Hence, transactions that are admitted are only given a prognosis that they will get their desired amount of resources, i.e., a conditional guarantee. Dropping and replacing transactions result in a utility loss relative to the initial expectations (processing time is wasted if the transactions have already started to execute). Hence, it is of interest to minimize the utility loss. Note that a new transaction will eventually provide (when complete) some utility to the system, and this must be contrasted with the utility loss caused by de-allocating sufficient resources in order to admit the new transaction. More precisely, the resolver

(i) determines COI and computes the amount of processing time (\(\Delta s\)) that needs to be de-allocated in COI in order to resolve the overload;

(ii) generates one or more nearly optimal overload resolution plans (ORPs), where an ORP consists of a set of overload resolution actions (ORAs) that de-allocate resources among admitted transactions; and

(iii) decides whether it is advantageous to carry out one of the ORPs considering the relative utility loss or gain of executing the ORP and accepting
the new transaction in comparison to simply rejecting it or substituting it.

>From the set of valid ORAs, i.e., ORAs with finite utility loss (and release resources in the critical overload interval), we build an ORP by selecting a subset of ORAs that minimizes the total utility loss while deallocating the required amount of time to admit the new transaction. ORAs imposing an infinite penalty by dropping critical transactions will not be generated since these are, by definition, not considered valid.

**Computing the Time Saved by an ORA.** The amount of saved time, denoted \( \xi^d_i \) in a critical overload interval as a result of performing a specific ORA (d=drop or r=replace) on an arbitrary transaction \( \tau_i \), is computed as follows. When dropping the transaction \( \tau_i \), the amount of time de-allocated is \( \xi^d_i = \zeta_i - o_i + \psi_i \) where \( \zeta_i \) is the remaining execution time, \( o_i \) expressed the time for aborting the transaction, and \( \psi_i \) is related to blocking time, defined as \( \psi_i = \max(0, w_{crit_i} - \max\{b_j|j \neq i\}) \).

Since we are using SRP, we know that a high-priority transaction \( \tau_i \) may be blocked only once by a low-priority transaction. Thus, \( b_i \) is the worst-case execution time of all critical sections among low-priority transactions for transaction \( \tau_i \). If transaction \( \tau_i \) has the longest critical section, then \( w_{crit_i} \) time units have been reserved for blocking, but if it is decided to drop or replace transaction \( \tau_i \), then transaction \( \tau_i \) will not enter its critical section and, hence, will not block any other transaction, implying that time may be saved. The amount of time saved is then the difference in execution time between the longest critical section of \( T_i(\tau_i) \) and the longest critical section \( b_j \) \((j \neq i)\).

The amount of time de-allocated when an original transaction \( \tau_i \) is replaced by its contingency transaction \( \tilde{\tau}_i \), having the same deadline (i.e., \( d_i = \tilde{d}_i \)) can be computed in a similar way. Here, the execution time of the longest critical section of the contingency transaction \( \tilde{T}_i \) has to be considered, i.e., we get \( \xi^r_i = \zeta_i - o_i - \tilde{w}_i + \psi_i \) where \( \psi_i = \max\left(0, w_{crit_i} - \max\{\tilde{b}_i, b_j|i \neq j\}\right) \).

**Computing the Utility Loss Caused by an ORA.** The amount of utility loss, denoted \( \gamma^d_i \), as a result of a specific ORA (d or r) on an arbitrary transaction \( \tau_i \) is computed as follows. When dropping the transaction, the utility loss is given by the loss of benefit of meeting the deadline and the penalty of missing the deadline \( d_i \) (in the case where a transaction has a firm deadline, the utility loss thus equals the loss of benefit only), i.e. \( \gamma^d_i = v_i(t_i) - v_i(t'_i) \) where \( t_i \) \((t_i \leq d_i)\) is the original termination time, and where \( t'_i \) \((t'_i > d_i)\) is the new effective termination time.
By replacing a transaction $\tau_i$ with its contingency transaction $\tau_i$, the amount of utility loss is the difference between the benefit contributed by $\tau_i$ and the benefit contributed by $\tau_i$, i.e. $\gamma_i^r = v_i(t_i) - v_i(t_i)$.

**Algorithm for Generating the Overload Resolution Plan.** OR-ULD attempts to select the best ORAs for saving the required amount of time in the critical overload interval. Selection of ORAs is made by relating the utility loss caused to the amount of resources that are de-allocated by an ORA. OR-ULD computes the utility loss density ($\gamma_i^r / \xi_i^r$) for each ORA, and orders the set of actions by their utility loss density. The algorithm consists of the following steps:

(i) Generate the set of possible ORAs in COI and compute their utility loss density.

(ii) Sort ORAs by utility loss density.

(iii) Iterate through the ORAs in the order of decreasing utility loss density, and add them to the ORP until the total amount of saved time by the new ORP (overload resolution plan) is greater or equal to the amount of time required.

In order to provide a better understanding of how the algorithm for selecting ORAs works, and what computations are performed, we present an example.

**EXAMPLE 1 Overload Resolution by OR-ULD**

Consider a set of admitted transactions $\tau_A = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$, where $\tau_1$ and $\tau_2$ have critical deadlines and contingency transactions ($\tau_1, \tau_2$), and $\tau_1, \tau_4$ and $\tau_5$ have firm deadlines (the characteristics of the transactions are table 10.1). A new transaction $\tau_n$ arrives to the system at time $t=30$ causing a transient overload. If $\tau_n$ is admitted and scheduled, the consequence is that $\tau_2$ and $\tau_5$ will miss their deadlines.

In order to admit transaction $\tau_n$, $w_n$ time units must be available. Since, $\tau_2$ has the smallest slack of transactions having a later deadline than $\tau_n$, $\tau_2$ is the transaction that will be affected the most. Before $\tau_n$ arrived to the system, $\tau_2$ had a slack of $s_2 = 20$ time units and, hence, if $\tau_n$ arrived to the system, $\tau_2$ its deadline by $\Delta s = 30$ time units ($s'_2 = w_n - 20$). Therefore, if $\tau_n$ were admitted, then 30 time units must be released in the critical overload interval.

OR-ULD goes through the list of possible and valid ORAs, sorted by utility loss density in ascending order (see figure 10.2). The algorithm adds ORAs to the selected set $A_{ORP}$ until enough time has been released. Hence, we get $A_{ORP} = \{\alpha_i^r\}$. The cost of $A_{ORP}$ is 200, which should be compared to the utility contributed by $\tau_n$ and the penalty of rejecting it. Although the cost of $A_{ORP}$ is higher than the utility contributed by $\tau_n$, admitting $\tau_n$ is a
### Table 10.1 Characteristics of transaction workload $\tau$

<table>
<thead>
<tr>
<th>$\tau_i$</th>
<th>$r_i$</th>
<th>$w_i$</th>
<th>$d_i$</th>
<th>$v_i(t_i \leq d_i)$</th>
<th>$v_i(t_i &gt; d_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>0</td>
<td>80</td>
<td>430</td>
<td>350</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>$\tilde{\tau}_1$</td>
<td>0</td>
<td>30</td>
<td>430</td>
<td>200</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>30</td>
<td>80</td>
<td>280</td>
<td>350</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>$\tilde{\tau}_2$</td>
<td>30</td>
<td>40</td>
<td>280</td>
<td>100</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>$\tau_3$</td>
<td>50</td>
<td>100</td>
<td>260</td>
<td>200</td>
<td>0</td>
</tr>
<tr>
<td>$\tau_4$</td>
<td>100</td>
<td>50</td>
<td>250</td>
<td>400</td>
<td>0</td>
</tr>
<tr>
<td>$\tau_5$</td>
<td>110</td>
<td>50</td>
<td>350</td>
<td>300</td>
<td>0</td>
</tr>
<tr>
<td>$\tau_\eta$</td>
<td>120</td>
<td>50</td>
<td>230</td>
<td>180</td>
<td>$-\infty$</td>
</tr>
<tr>
<td>$\tilde{\tau}_\eta$</td>
<td>120</td>
<td>30</td>
<td>230</td>
<td>100</td>
<td>$-\infty$</td>
</tr>
</tbody>
</table>

Table 10.2 Overload resolution actions

<table>
<thead>
<tr>
<th>$\alpha_i^x$</th>
<th>$\gamma_i^x$</th>
<th>$\xi_i^x$</th>
<th>$\gamma_i^x / \xi_i^x$</th>
<th>valid</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha_1^x$</td>
<td>200</td>
<td>45</td>
<td>4.44</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>$\alpha_2^x$</td>
<td>250</td>
<td>45</td>
<td>5.55</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>$\alpha_3^x$</td>
<td>300</td>
<td>50</td>
<td>6.00</td>
<td>no</td>
<td>(remaining time not in COI)</td>
</tr>
<tr>
<td>$\alpha_4^x$</td>
<td>400</td>
<td>50</td>
<td>8.00</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>$\alpha_5^x$</td>
<td>150</td>
<td>15</td>
<td>10.00</td>
<td>no</td>
<td>(remaining time not in COI)</td>
</tr>
<tr>
<td>$\alpha_6^x$</td>
<td>$\infty$</td>
<td>35</td>
<td>$\infty$</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>$\alpha_7^x$</td>
<td>$\infty$</td>
<td>45</td>
<td>$\infty$</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>
better choice since the alternative overload resolution plan, suggesting that \( \tau_n \) should be rejected, imposes infinite utility loss and, hence, is not valid. If we consider admitting the contingency transaction \( \tau_n \), we need to release ten time units \((w_n - 20)\). As it turns out, in this example, the algorithm will produce an overload resolution plan equal to the former one and, hence, the cost efficiency of admitting \( \tau_n \) is worse. Considering this, the overload resolver will decide to resolve the overload by dropping \( \tau_3 \) and admitting \( \tau_n \). The new EDF schedule is shown in table 10.2.

By selecting ORAs in reverse order of utility loss density, we ensure that de-allocated time is cost efficient with respect to other ORAs. Given a workload where transactions are similar in length and utility, transactions close to completion will not be selected due to the limited time they would save if dropped. Critical transactions can only be replaced (never be dropped), and the amount of time saved by replacing a transaction is often more limited. Once the remaining execution time of a critical transaction is smaller than the worst-case execution time of its corresponding contingency transaction, replacing the critical transaction is in general not justified.

As we have seen in the example, when an arriving transaction has a contingency transaction there is an opportunity of directly accepting the contingency transaction instead. Admitting the contingency transaction requires less, if any, processing time to be de-allocated than the original transaction, but it also contributes less utility. The amount of time that needs to be released in COI in order to admit \( \tau_i \) \((\tau_i) \) is \( \Delta s = w_n - S(\Delta s = \hat{w}_n - S) \), where \( S \) is defined as \( S = \min_{\tau_i \in \tau_i} \{ s_i \} \) where \( \tau_i \in \tau_i \wedge d_i > d_n \), and \( s_i \) is the slack of \( \tau_i \).

We generate two ORPs and compute the relative cost of implementing them (note that (ii) and (iii) are exclusive): (i) admitting original transaction \( \tau_n \);
(ii) admitting contingency transaction $\tau_i$ (critical transactions only); and (iii) rejecting transaction $\tau_j$ (non-critical transactions only). The most cost-effective approach will then be selected, i.e., the one with the best overall utility gain or loss. The overall change in utility, denoted $\Phi$ can be computed as follows, and the ORP with the lowest overall utility loss (or highest utility gain) will then be selected (note, OR-ULD denotes a function that returns the total utility loss of releasing $w_n - S$ time in COI).

\[
\begin{align*}
\text{admit } \tau_\eta: & \quad \Phi_1 = v_\eta(r_\eta \leq t_\eta \leq d_\eta) - \text{OR-ULD}(w_\eta - S) \\
\text{admit } \tau_\eta: & \quad \Phi_2 = \Phi_\eta(r_\eta \leq t_\eta \leq d_\eta) - \text{OR-ULD}(w_\eta - S) \\
\text{reject } \tau_\eta: & \quad \Phi_3 = -v_\eta(r_\eta \leq t_\eta \leq d_\eta) + v_\eta(t'_\eta) \text{ where } t'_\eta > d_\eta
\end{align*}
\]

It should be noted that in our approach transactions may be downgraded, or even dropped, by the overload resolver. Once this happens, a transaction will not be upgraded again. We have considered the cost of frequent invocations of a transaction upgrader as significantly higher than the pay-off. Of course, this may depend on the characteristics of the application considered.

5. PERFORMANCE ANALYSIS

The OR-ULD algorithm has been implemented and an extensive simulation-based performance analysis has been conducted. Due to space limitation, we omit the simulation results and instead we present our conclusions based on the performance analysis (for details on the simulation results, see [2, 3]). In our experiments we used a workload consisting of sporadic critical transactions also having contingency transactions, and aperiodic non-critical transactions with firm deadlines. As comparison we used an admission based EDF. The results show that the OR-ULD algorithm (i) gracefully degrades performance during overloads by increasing the number of contingency transactions (replacing the original transactions) and gradually dropping non-critical transactions, (ii) ensures the timeliness of critical transactions (within a certain operational envelope), and (iii) produces near-optimal results\(^2\). A system utilizing both admission control and overload resolution by resource re-allocation, as defined by the architecture outlined in this article, offers a substantial increase in completion ratio during transient overloads. In particular, the enforcement of critical time constraints is guaranteed in a wider operational envelope.

\(^2\text{Results are not optimal since not necessarily all non-critical transactions are being dropped and all original transactions with contingency transactions are being replaced during heavy overloads.}\)
6. SUMMARY AND FUTURE WORK

We have in this chapter described the characteristics of transient overloads and proposed an approach and an algorithm, denoted OR-ULD, which dynamically resolves transient overloads in dynamic real-time database systems. Specifically, transient overloads are resolved by selecting a set of actions that release the necessary amount of time among admitted transactions in order to enable the admission of a new transaction. Performance studies show that OR-ULD gracefully degrades performance during transient overloads without jeopardizing the timeliness of critical transactions when the system is working within its operational envelope. A system utilizing overload resolution by resource re-allocation, as outlined, gives a substantial improvement in enforcing critical time constraints during transient overloads, and outperforms a system utilizing admission control only.

We believe the approach is unique and has several advantages. First, the overload resolution strategy takes advantage of the expressive power of value functions. Particularly, the natural and explicit representation of criticality using negative utility enables OR-ULD to enforce time constraints of critical transactions. Second, the approach separates overload management from scheduling and admission control. The overload management algorithm can be used with other admission control and scheduling policies that have a mechanism for detecting transient overloads and are able to indicate the critical interval in which a transient overload is occurring. Third, the approach enables resolution of transient overloads by using multiple strategies, i.e., dropping transactions, replacing transactions with contingency transactions, etc. For the first time it is possible to balance these decisions against each other and determine the most beneficial way of resolving a transient overload in order to admit a new critical transaction.

Our work has primarily focused on scheduling and overload resolving workloads consisting of hard critical and firm transactions, i.e., transactions with no deadline tolerance, hence, the two types of overload resolution actions we have used are dropping and replacing a transaction. However, there is a third way of resolving overloads and release resources, namely, postponing the completion of soft deadline transactions outside the critical overload, and thereby take advantage of the acceptable deadline tolerance given by soft time constraints. We intend to address this in future work.

References


This page intentionally left blank.
Chapter 11

SECURE REAL-TIME TRANSACTION PROCESSING

Jayant R. Haritsa
Database Systems Lab, Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore 560012, INDIA.
haritsa@dsl.serc.iisc.ernet.in

Binto George
Dept. of Computer Science
Western Illinois University, Macomb, IL 61455, USA.
b-george@wiu.edu

1. INTRODUCTION

Many RTDBS applications arise in electronic financial services, safety-critical installations and military systems where enforcing security is crucial to the success of the enterprise. For example, consider the environment of an electronic (open bid) auction on the World-Wide-Web with online auctioneers and bidders. Typically, the auction database contains “secret” information such as bidders personal details including private keys, credit-worthiness and past bidding patterns; the purchase price and ownership history of the items that are being auctioned; the list of “in-house bidders” – these are bidders planted by the auction house to provoke other bidders by artificially hiking the maximum bid; etc. The database also contains “public” information such as bidder public keys and authentication certificates; the starting bid price, the minimum bid increment and the time for delivery for each item; the sequence and state of bids for items currently under auction; etc. It is expected that the secret information is known only to the auctioneers whereas the public information is available to both bidders and auctioneers.

In the above environment, the auction service provider faces a problem of three dimensional complexity: (1) There is a considerable amount of data to be consulted, processed and updated, and while doing so the database
consistency should be maintained; (2) There are time constraints associated with various operations – for example, a bid is valid only if registered in the database within a pre-specified time period after submission (in the Flash Auction at http://www.firstauktion.com, bids that arrive more than five minutes after the previous bid is registered are invalidated); (3) During every stage of the bidding process, data security must be ensured — unauthorized access to the secret information by bidders may help them gain unfair and financially lucrative advantages over other competitors.

We investigate here the design of information systems that can simultaneously and effectively meet the above three challenges, that is, on the design of Secure Real-Time Database Systems (SRTDBS). In particular, we focus on the development and evaluation of high-performance secure real-time concurrency control (CC) protocols.

A number of requirements have been identified in the literature for a DBMS to be considered secure [6]. We restrict our attention here to secrecy, that is, the prevention of unauthorized knowledge of secret data, which is an especially important requirement for RTDBS due to the sensitive nature of their application domains. More precisely, we wish to enforce multilevel secrecy wherein, given a set of transaction security classes, the data associated with a particular class should not be revealed to transactions of lower security classes. In the sequel, we use the term security synonymously with multilevel secrecy.

Our study is carried out in the context of real-time applications with “firm-deadlines” [18] – for such applications, completing a transaction after its has expired is of no utility and may even be harmful. Therefore, transactions that miss their deadlines are considered to be worthless and are immediately “killed” – that is, aborted and permanently discarded from the RTDBS without being executed to completion. Accordingly, the performance metric is KillPercent, the steady-state percentage of killed transactions.\footnote{Or, equivalently, the percentage of missed deadlines.}

\section{SECURITY MECHANISMS}

Most secure DBMS attempt to achieve the secrecy objective by incorporating access control mechanisms based on the well-known Bell–LaPadula model [22]. This model is specified in terms of subjects and objects. An object is a data item, whereas a subject is a process that requests access to an object. Each object in the system has a classification level (e.g., Secret, Classified, Public, etc.) based on the security requirement. Similarly, each subject has a corresponding clearance level based on the degree to which it is trusted by the system.

The Bell–LaPadula model imposes two restrictions on all data accesses:
1. A subject is allowed **read** access to an object only if the former’s clearance is *higher than* or *identical to* the latter’s classification.

2. A subject is allowed **write** access to an object only if the former’s clearance is *identical to* or *lower than* the latter’s classification.

![Figure 11.1 Bell-LaPadula access restrictions](image)

The Bell-LaPadula conditions effectively enforce a “read below, write above” constraint on transaction data accesses (an example is shown in Figure 11.1), and thereby prevent direct unauthorized access to secure data. They are not sufficient, however, to protect from “covert channels”. A covert channel is an indirect means by which a high security transaction can transfer information to a low security transaction [21]. For example, if a low security transaction requests access to an exclusive resource, it will be delayed if the resource is already held by a high security transaction, otherwise it will be granted the resource immediately. The presence or absence of the delay can be used as a “signaling” or encoding mechanism by a high security transaction passing secret information to the low security transaction. Note that, from the system perspective, nothing “obviously illegal” has been done in this process by the conspiring transactions.

Covert channels that use the DBMS’s **physical resources** such as the CPU or the disk as the medium for passing on information can be tackled by introducing “noise” in the form of dummy transactions that make use of these resources (similar to the “pump” scheme proposed in [19]). However, this approach is impractical for covert channels that use **data** as the medium (for example, presence or absence of a lock on a pre-determined data item). This is because, unlike physical resources which are typically few in number, the number of data items is usually enormous, especially in a DBMS. In fact, in heavily loaded systems, noise at the physical resources may be generated “for free”, but this
will probably never be the case for data since it is trivial to insert an additional
data item that is of relevance only to the conspiring transactions. Therefore,
explicitly making data access covert-channel-free is more vital than doing the
same for resource access.

Covert channels based on data can be prevented by ensuring that low security
transactions do not “see” high security transactions – this notion is formalized in
[16] as non-interference, that is, low security transactions should not be able to
distinguish between the presence or absence of high security transactions. This
can be implemented, for example, by providing higher priority to low security
transactions whenever a conflict occurs between a low security transaction
and a high security transaction. From a system perspective, it translates to
implementing a concurrency control (CC) mechanism that supports the non-
interference feature. We address this issue here.

3. INTEGRATING SECURITY AND CONCURRENCY
CONTROL

A variety of challenging problems, ranging from priority assignment to
ensuring performance fairness, arise when we attempt to integrate security into
the RTDBS framework – these issues are discussed in this section.

3.1 PRIORITY ASSIGNMENT

An SRTDBS has to simultaneously satisfy two requirements, namely, pro-
vide security and minimize the number of killed transactions. Unfortunately,
the mechanisms for achieving the individual goals often work at cross-pur-
poses [17]. In an RTDBS, high priority is usually given to transactions with
earlier deadlines in order to help their timely completion. On the other hand,
in secure DBMS, low security transactions are given high priority in order
to avoid covert channels (as described earlier). Now consider the situation
wherein a high security process submits a transaction with a tight deadline in
an SRTDBS. In this case, priority assignment becomes difficult since assigning
a high priority may cause a security violation whereas assigning a low priority
may result in a missed deadline.

One solution, predicated on the notion that security is of paramount im-
portance, is to assign transaction priorities based primarily on clearance levels
and only secondarily on deadlines. In this scheme, priorities are assigned as
a vector $P = (\text{LEVEL}, \text{INTRA})$, where \text{LEVEL} is the transaction clearance
level and \text{INTRA} is the value assigned by the priority mechanism used within
the level. Clearance levels are numbered from zero upwards, with zero corre-
ponding to the lowest security level. Further, priority comparisons are made
in lexicographic order with lower priority values implying higher priority.
With the above scheme, transactions at a lower clearance have higher priority than all transactions at a higher clearance, a necessary condition for non-interference. For the intra-level priority mechanism, any priority assignment that results in good real-time performance can be used. For example, the classical Earliest Deadline assignment [23], wherein transactions with earlier deadlines have higher priority than transactions with later deadlines. For this choice, the priority vector would be $P = (\text{LEVEL}, \text{DEADLINE})$ – this priority assignment is used in most of our experimental evaluations.

### 3.2 SUPPORTING NON-INTERFERENCE

In conjunction with the above priority assignment policy, it would seem at first glance that, in principle, any real-time CC protocol could be used in an SRTDBS and that the actual choice of protocol would be based only on the relative performance of these protocols. However, not all the previously proposed real-time CC algorithms are amenable to supporting security requirements. For example, consider the 2PL Wait Promote algorithm proposed in [1]: This protocol, which is based on 2PL, incorporates a priority inheritance mechanism [26] wherein, whenever a requester blocks behind a lower-priority lock holder, the lock holder’s priority is promoted to that of the requester. In other words, the lock holder inherits the priority of the lock requestor. The basic idea here is to reduce the blocking time of high priority transactions by increasing the priority of conflicting low priority lock holders (these low priority transactions now execute faster and therefore release their locks earlier).

The Wait Promote approach is not suitable for SRTDBS. This is because it permits the blocking of high priority transactions by low priority transactions which violates the requirement of non-interference between the transactions of different security levels.

To generalize the above observation, a real-time CC protocol that permits, to even a limited extent, high priority transactions to be adversely affected by low priority transactions, a phenomenon known as priority inversion in the real-time literature [26], cannot be used in a secure RTDBS. Apart from Wait Promote, other examples of real-time CC algorithms that fall into this category include 2PL-CR [1], 2PL-OS/BI [3] and WAIT-50 [18].

### 3.3 RESOURCE ALLOCATION

Apart from the noise approach mentioned earlier, covert channels can also be eliminated by static resource allocation policies, wherein each transaction security class has a set of pre-assigned resources. This approach is taken, for example, in replicated secure architectures wherein a multi-level secure DBMS is built by “stacking” single-level database systems. As pointed out in [5], the
drawback of such strategies, however, is that they may result in an inefficient and inflexible use of resources.

Yet another possibility is to use "snapshot" protocols (e.g. [4]), wherein two versions of data items are provided – for data of a given security level, higher security transactions access the older version whereas transactions of the same level access the current version. A problem with this approach, however, is that higher security transactions, especially long-lived ones, have to base their results on arbitrarily stale data [5]. In fact, a performance study in [8] shows instances where the highest security level transactions read data items that have been subsequently overwritten by fifteen other transactions.

The above approaches are especially problematic in the real-time context since they may result in: (a) A large increase in the number of killed transactions due to poor resource utilization; or (b) Incorrect results since real-time applications typically respond to current situations and therefore should not utilize stale data. Due to these reasons, the challenge in the RTDBS domain is to design “dynamic” and “one-copy” CC policies that are demonstrably secure.

3.4 FAIRNESS

A major problem arising out of the preferential treatment of low security transactions is that of “fairness” – a disproportionately large fraction of the high security transactions miss their deadlines and are killed. Note that this is an especially problematic issue because it is the “VIPs”, that is, the high security transactions, that are being discriminated against in favor of the “common-folk”, that is, the low security transactions.

Unfortunately, designing dynamic mechanisms for achieving fairness that are completely free of covert channels appears to be fundamentally impossible [20]. This is because the fairness-inducing mechanism can itself become the medium of information compromise. The issue then is whether it is possible to design fair systems while still guaranteeing that the information leakage bandwidth (from covert channels) is within acceptable levels.

4. SECURE CC PROTOCOLS

In this section, we describe the CC protocols that are evaluated in our study. These include two well-known real-time protocols: 2PL-HP [1], which is based on locking, and OPT-WAIT [18], which is based on optimistic concurrency control. Both these protocols are completely free from priority inversion and can therefore be used to resolve conflicts in an SRTDBS.

Apart from the above real-time algorithms, we also consider S2PL [27], a recently proposed secure locking-based protocol that does not include any real-time-specific features. We include it here, however, for the following reasons: First, it serves as a baseline against which to compare the real-time
CC algorithms. Second, we use it in one of the “dual-CC” protocols described in Section 5. Finally, it has been used in some of the previous research work on SRTDBS (these are discussed in Section 8).

In the remainder of this section, we describe the 2PL-HP, OPT-WAIT and S2PL protocols.

4.1 2PL HIGH PRIORITY

The 2PL High Priority (2PL-HP) scheme [1] modifies the classical strict two-phase locking protocol (2PL) [10] by incorporating a priority conflict resolution scheme which ensures that high priority transactions are not delayed by low priority transactions. In 2PL-HP, when a transaction requests a lock on a data item that is held by one or more higher priority transactions in a conflicting lock mode, the requesting transaction waits for the item to be released (the wait queue for a data item is managed in priority order). On the other hand, if the data item is held by only lower priority transactions in a conflicting lock mode, the lower priority transactions are restarted and the requesting transaction is granted the desired lock. Note that 2PL-HP is inherently deadlock-free if priorities are assigned uniquely (as is usually the case in RTDBS).

4.2 OPT-WAIT

The OPT-WAIT algorithm [18] modifies the classical forward (or broadcast) optimistic CC protocol (OPT) [25] by incorporating a priority wait mechanism. Here, a transaction that reaches validation and finds higher priority transactions in its conflict set is “put on the shelf”, that is, it is made to wait and not allowed to commit immediately. This gives the higher priority transactions a chance to make their deadlines first. While a transaction is waiting on the shelf, it is possible that it may be restarted due to the commit of one of the conflicting higher priority transactions. If at any time during its shelf period, the waiting transaction finds no higher priority transactions remaining in its conflict set, it is committed, restarting in the process the lower priority transactions (if any) in its conflict set.

4.3 S2PL

A secure locking-based protocol called Secure 2PL (S2PL) was recently proposed in [27]. The basic principle behind Secure 2PL is to try to simulate the execution of conventional 2PL without blocking the actions of low security transactions by high security clearance transactions. This is accomplished by

\footnote{A new reader joins a group of lock-holding readers only if its priority is higher than that of all the waiting writers.}
providing a new lock type called virtual lock, which is used by low security transactions that develop conflicts with high security transactions. The actions corresponding to setting of virtual locks are implemented on private versions of the data item (similar to optimistic CC). When the conflicting high security transaction commits and releases the data item, the virtual lock of the low security transaction is upgraded to a real lock and the operation is performed on the original data item. To complete this scheme, an additional lock type called dependent virtual lock is required apart from maintaining, for each executing transaction $T_i$, lists of the active transactions that precede or follow $T_i$ in the serialization order. The complete details are given in [27].

In our implementation of S2PL, we have had to make some modifications since the algorithm (as described in [27]) does not eliminate interference under all circumstances – the details of the security loopholes and our fixes for these loopholes are given in [14].

5. THE DUAL-CC APPROACH TO SECURE CONCURRENCY CONTROL

In this section, we move on to discussing our new dual-CC approach to secure real-time concurrency control. Our design is based on the observation that in the secure environment there are two categories of conflicts: inter-level and intra-level. Inter-level conflicts are data conflicts between transactions belonging to different security clearance levels whereas intra-level conflicts are data conflicts between transactions of the same level. The important point to note here is that only inter-level conflicts can result in security violations, not intra-level conflicts. This opens up the possibility of using different CC strategies to resolve the different types of conflicts. In particular, we can think of constructing mechanisms such that inter-level conflicts are resolved in a secure manner while intra-level conflicts are resolved in a timely manner. For example, S2PL could be used for inter-level conflicts while OPT-WAIT could be used to resolve intra-level conflicts.\(^3\)

The advantage of the dual-CC approach, pictorially shown in Figure 11.2, is that the RTDBS can maximize the real-time performance, by appropriate choice of intra-level CC protocol, without sacrificing security. This is in marked contrast to the tradeoff approach suggested in [8] which requires the application to compromise on security in order to achieve enhanced real-time performance (a more detailed assessment of the tradeoff approach is presented in Section 8.). Another advantage of the dual-CC approach is that the separation of security and timeliness concerns makes it possible to use even unsecure real-time CC

\(^3\)A similar, but unrelated, dual-protocol strategy has been used earlier for the purpose of task scheduling in the Secure Alpha project [17].
algorithms (e.g., Wait-Promote, WAIT-50) for resolving intra-level conflicts!

The dual-CC approach therefore empowers the use, even in the secure RTDBS domain, of the rich set of real-time CC algorithms developed during the last decade.

5.1 ENSURING SERIALIZABILITY

At first glance, it may appear that concurrently using multiple CC mechanisms could result in violation of the transaction serializability requirement. This could happen, for example, if the serial orders enforced by the individual mechanisms were to be different. A detailed study of a generalized version of this problem is presented in [31], wherein the transaction workload consists of a mix of transaction classes and the objective is to allow each transaction class to utilize its preferred CC mechanism. They propose a database system architecture wherein intra-class conflicts are handled by the class’s preferred CC manager while inter-class conflicts are handled by a new software module called the Master Concurrency Controller (MCC) that interfaces between the transaction manager and the multiple CC managers. The MCC itself implements a complete concurrency control mechanism and ensures a single global serialization order in the entire database system by using a Global Ordering Scheme [31].

For our study, we adapt the above architecture to the secure real-time environment. In our implementation, each security level has a Local Concurrency Controller (LCC) that resolves intra-level data conflicts. The LCCs can use any high performance real-time CC protocol without risking security violations. The Master Concurrency Control mechanism, however, is chosen to be a secure protocol and is responsible for handling all inter-level data conflicts. Figure 11.3 shows an example of this implementation for a two security level
system. The proof of correctness and other details of our adaptation of the MCC architecture are given in [14].

![MCC architecture](image)

**Figure 11.3** MCC architecture

### 5.2 DUAL-CC PROTOCOLS

We have developed and evaluated three protocols, based on the dual-CC approach: WAIT-HP, HP-WAIT and S2PL-WAIT. These protocols are described below:

**WAIT-HP:** Inter-level data conflicts are resolved by OPT-WAIT and intra-level conflicts are resolved by 2PL-HP.

**HP-WAIT** This protocol is the “mirror” of WAIT-HP – inter-level data conflicts are resolved by 2PL-HP while intra-level conflicts are resolved by OPT-WAIT.

**S2PL-WAIT:** Inter-level data conflicts are resolved by S2PL and intra-level conflicts are resolved by OPT-WAIT. The scheme operates as follows: After the intra-level conflicts are resolved by OPT-WAIT, the transaction enters the “virtual commit” state, and subsequently the inter-level data conflicts are resolved by S2PL. After all conflicts are resolved, a transaction “really” commits. Recall that S2PL is *not* a restart-oriented algorithm and allows a high security transaction to continue regardless of the virtual commit of a low security conflicting transaction.
6. PERFORMANCE RESULTS

Using a detailed simulation model of a firm-deadline SRTDBS, we evaluated the real-time performance of the various secure CC protocols described in the previous sections. Further, to help isolate and understand the performance cost that occurs due to having to eliminate covert channels, we also simulated the performance achievable for each of these protocols if only “direct” unauthorized access was prevented (by using the Bell–LaPadula conditions) – for this scenario, a priority assignment of (P = DEADLINE) was used.

Our experiments, which covered a variety of transaction workloads and system configurations, demonstrated the following:

1. Under “normal” loads (KillPercent values in the range of 0 to 20 percent), the overall performance of the secure system is worse than that of the direct system, whereas under heavy loads (Killpercent greater than 20 percent), it is the other way around. Within the secure system, the performance of high-security transactions is significantly worse than that of the low-security transactions. Among the secure (individual) CC protocols, OPT-WAIT performed the best in minimizing the kill percentages on both an overall basis and on a per-clearance-level basis. These results show that OPT-WAIT, which provided excellent performance in traditional (unsecure) real-time concurrency control [18], continues to perform well even in the secure real-time domain.

2. The dual-CC combinations of HP-WAIT and WAIT-HP generally perform worse than pure OPT-WAIT. However, the dual combination of S2PL-WAIT performs better than OPT-WAIT, especially at lower kill percentage levels. This is because S2PL is a non-restart-oriented algorithm unlike both OPT-WAIT and 2PL-HP, and therefore ensures reduction of the harm done to high-security transactions.

The above results highlight the power and flexibility that is provided by the dual-CC approach. In fact, it may be possible to develop dual protocols that perform even better than S2PL-WAIT by appropriately choosing the constituent protocols.

3. At light loads when virtually all transactions make their deadlines, all the CC protocols are (trivially) fair. As the loading increases, however, the protocols become increasingly unfair since they selectively miss the deadlines of Secret transactions to accommodate the Public transactions. With regard to the relative fairness of the protocols, OPT-WAIT is the best among the individual CC protocols, whereas S2PL-WAIT beats OPT-WAIT on this metric also.
7. FAIRNESS ACROSS SECURITY LEVELS

As discussed above, our experimental results showed that even with the S2PL-WAIT protocol, a marked lack of fairness exists with respect to the real-time performance of the high security classes, especially in environments with significant resource contention. In this section, we discuss the fairness issue in more detail.

Policies for ensuring fairness can be categorized into static and dynamic categories. Static mechanisms such as resource reservation can provide fairness while still maintaining full security. However, they may considerably degrade the real-time performance due to resource wastage arising out of their inability to readjust themselves to varying workload conditions.

Dynamic mechanisms, on the other hand, have the ability to adapt to workload variations. However, as observed in [20], providing fairness in a workload adaptive manner without incurring covert channels appears to be fundamentally impossible since the dynamic fairness-inducing mechanism can itself become the medium of information compromise. For example, consider the following situation: A conspiring Secret process submits a workload such that the Secret class performance degrades. The fairness mechanism subsequently tries to improve the Secret class performance by allocating more resources for the Secret class. A collaborating Public transaction could now feel the reduction in the availability of system resources and thereby sense the presence of the Secret process in the system. Therefore, this mechanism could be exploited for covert signaling.

In summary, for full-secure applications, unfairness can only be mitigated to an extent by a judicious choice of CC protocol, but not completely eliminated. Therefore, it is interesting to consider the more tractable problem of whether it is possible to dynamically provide fairness while guaranteeing an upper bound on the bandwidth of the covert-channel introduced by the fairness mechanism.

We investigated the above issue in [14] and proposed a simple feedback-based transaction admission control mechanism for achieving fairness. The bandwidth of the covert channel introduced by this mechanism was bounded by appropriately setting the feedback period. In particular, the bandwidth bound was set to one bit per second, an acceptable level as per the so-called “Orange Book” [7], which defines US military security standards. Experimental evaluation of this system indicates that it provides close to ideal fairness at little cost to the overall real-time performance.

8. RELATED WORK

The material discussed in this chapter is based on our earlier papers [11, 12, 14]. Apart from this, the only other work that we are aware of on SRTDBS
concurrency control is a series of papers by Son et al [9, 24, 28, 29, 30]. In this section, we briefly summarize these papers and contrast them with our study.

A concurrency control protocol called Adaptive 2PL (A2PL) that attempts to balance the dual requirements of security and timeliness was presented in [9, 28, 29]. In their scheme, transactions dynamically choose between an unsecure version of 2PL-HP and the secure S2PL described in Section 4. The goal of the A2PL protocol is to tradeoff security for real-time performance with the tradeoff depending on the state of the system and the application’s requirements. In contrast, in our work, we have assumed that full security is a fundamental requirement and that it is not permissible to improve the real-time performance at the cost of security.

There are also a few design and implementation difficulties associated with the A2PL protocol:

1. In A2PL, the decision to use 2PL-HP or S2PL is a function of the system state and the application requirements. The priority assignments in these protocols are independent and may therefore work at cross purposes. For example, consider the case where a tight-deadline Secret transaction requests a data item held by a slack-deadline Public transaction. Assume that the A2PL protocol chooses the real-time option to resolve this conflict and therefore grants the lock to the requester aborting the Public holder. Assume that another slack-deadline Public transaction now makes a conflicting request on the same data item. If the A2PL protocol decides to follow the secure option to resolve this conflict, the Secret holder will have to be restarted. This means that a transaction that was earlier assisted based on deadline considerations now gets hindered on security considerations, negating the priority assignment objective.

2. Since the A2PL protocol takes only data conflicts into account, it cannot control the conflicts at other resources. Therefore, here again, a transaction that was assisted to make its deadline at the data may be hindered at the physical resources.

3. For applications where full security is a fundamental requirement, the A2PL protocol reduces to plain S2PL, which does not have any real-time features.

In [24], a concurrency control protocol that ensures both security and timeliness is proposed. For this scheme, however, the RTDBS is required to maintain two copies of each data item. Further, transactions are required to obtain all their

---

4The tradeoff approach, and alternative schemes to implement the tradeoff, have also been considered in the Secure Alpha project [17], which investigated the interactions between security and timeliness in the context of a distributed real-time operating system.
data locks before starting execution (i.e., conservative locking). These requirements limit the applicability of the protocol. In our work, we have considered more general database environments where all data items are single-copy and transactions acquire data locks dynamically.

Another feature of their work is that it is primarily addressed towards “soft-deadline” applications, that is, real-time applications in which there is value to completing tasks even after their deadlines have expired. In contrast, we have concentrated on firm-deadline applications. The type of deadline has a significant impact on both the performance evaluation model and on the interpretation of the results, as observed earlier for (unsecure) real-time transaction concurrency control [1, 18].

9. SUMMARY AND FUTURE WORK

In this chapter, we have attempted to characterize the state-of-the-art with regard to secure real-time transaction processing. We first identified that, in order to satisfy the requirement of non-interference, only those real-time concurrency control protocols that are free from priority inversion can be used in a secure RTDBS. This requirement ruled out several previously proposed real-time CC protocols, including algorithms such as 2PL Wait Promote and WAIT-50. Then, we studied the relative performance of the secure versions of the 2PL-HP and OPT-WAIT real-time concurrency control algorithms as well as S2PL, a non-real-time secure algorithm. OPT-WAIT performed best in these experiments across all metrics.

We also proposed a novel dual-CC approach to secure concurrency control wherein different concurrency control algorithms are used to resolve inter-level conflicts and intra-level conflicts. The dual combination of S2PL-WAIT performed better than OPT-WAIT in our experiments. Another advantage of the dual-CC approach is that the separation of security and timeliness concerns makes it possible to use even unsecure real-time CC algorithms for resolving intra-level conflicts.

Finally, we showed that secure RTDBS are inherently biased against high security transactions. This problem can be addressed by implementing a simple feedback-based transaction admission control mechanism, whose information leakage bandwidth can be bounded by appropriately setting the feedback period.

We now outline a variety of issues that appear appropriate for further research: First, we have only considered environments wherein the security levels are fully-ordered – in general, however, security hierarchies may be partially ordered, typically in the form of lattices. It is therefore imperative to extend the CC protocols discussed here to this more general framework. Further, additional security requirements such as data integrity and availability should also be considered.
Second, the performance characteristics of snapshot-based concurrency control algorithms and the performance tradeoffs involved in replicated architectures need to be assessed.

Third, we considered only one class of covert channels in our study, namely, timing channels, wherein information transfer is by modulation of response times. A complementary class of covert channels is known as storage channels, wherein information transfer is through modulation of the success or failure of access of storage locations [2]. Solutions for plugging security loopholes based on storage channels need to be devised.

Finally, high-performance secure protocols for other real-time transaction processing components, including buffer management and distributed commit, need to be designed. A first step in this direction was taken in our recent study of real-time secure buffering [13, 15].

Acknowledgements Binto George worked on this project when he was a graduate student at the Database Systems Lab, Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, India. Jayant Haritsa’s work was supported in part by research grants from the Dept. of Science and Technology and the Dept. of Bio-technology, Govt. of India.

References


PART IV
ACTIVE ISSUES AND TRIGGERING
This page intentionally left blank.
Chapter 12

SYSTEM FRAMEWORK OF ARTDBS

Jörgen Hansson  
Department of Computer and Information Science  
Linköping University, Linköping, Sweden

Sten F. Andler  
Department of Computer Science  
University of Skövde, Skövde, Sweden

1. INTRODUCTION

This chapter addresses issues and problems associated with incorporating active functionality and real-time constraints into a single database system. As of now, there are some research database platforms that provide support for both active and real-time capabilities, e.g., DeeDS [3], BeeHive [20], REACH [7], and STRIP [1]. This is in contrast to the envisioned rather large number of applications that can benefit from the use of such a system [16, 12]. In this chapter, which borrows heavily from [11], we provide a general introduction to active real-time database systems and elaborate on the advantages of the technology. Moreover, we discuss identified problems, potential solutions, and current research projects within active real-time database systems.

When studying complex real-time applications, several common characteristics and requirements of the real-time system appear. First, massive amounts of data are often gathered from the environment and the data must be stored by the real-time system. Second, in some applications the amounts of data needed to produce an output, given a specific input, are significant, i.e., the size of the database used for making a decision is significant. Geographical information systems (sophisticated flight maps) in airplanes are an example of this. Third, such systems are responding to physical events, i.e., they incorporate reactive behavior. Fourth, complex real-time systems act in dynamic environments where not everything is known a priori, which implies that dynamicity and adaptability are desirable, and often necessary, properties of real-time systems. Fifth, the system must be able to handle complex workloads, since tasks dif-
fer in strictness, tightness, arrival patterns, etc. Hence, we conclude that a complex real-time system needs support for efficient storage and manipulation of data (i.e., database functionality), support for specifying reactive behavior, and it must be able to adapt to its environment in order to cope with dynamic situations.

As mentioned earlier, real-time systems must respond to external events in the physical environment. The events are signaled to the real-time system which, based on the event type, determines what the next step should be. Even though real-time systems generally exhibit reactive behavior, no general methods or techniques for uniform specification of reactive behavior have been developed in the real-time research community. While active databases use the concept of Event-Condition-Action (ECA) rules (on event $E$, if condition $C$ is true, then trigger action $A$ for execution) as a building block for specifying reactive behavior, active database systems do not explicitly consider time constraints of transactions, hence, even though the ECA-rules can be used for specifying timeouts, active database systems do not guarantee the timeliness of the triggered action(s).

In HiPAC [9] it was proposed that a time constraint could be attached to the action part of a rule. This implies a change in semantics as pointed out by Ramamritham [17], were it is identified that traditional ECA rules cannot express the following semantics:

```
ON event E
IF condition C
DO <COMPLETE> action A <WITHIN t seconds>
```

The semantics of the action in the rule above are to complete action $A$ within $t$ seconds. In general, the time constraint can be relative to (i) the time of event occurrence or (ii) the time of event detection. The former refers to the actual time $T$ when the event was generated, whereas the latter refers to the time when the event was detected by the system, which can be a time $T + d$, where $d$ is the event detection time relative to event occurrence.

Evaluating ECA rules normally include several phases. In the event detection phase, events that have occurred are signalled to the active database system. In the triggering phase, corresponding rules to detected events are triggered. In the rule scheduling phase, it is determined in which order rules should be processed and when condition evaluation and action execution should occur. In the condition evaluation phase, the conditions of triggered rules are evaluated. In the action execution phase actions are executed. These phases are not necessarily performed sequentially. They depend on the modes used. Event-condition coupling expresses when condition evaluation should be performed relative to the rule triggering event. The condition-action coupling expresses
when a triggered action should be executed with respect to condition evaluation [15].

Three basic and distinct coupling modes are immediate, deferred, and detached coupling. Immediate coupling implies that condition evaluation (action execution) is done immediately after event detection (condition evaluation), all within the same transaction. If deferred coupling, is used, then condition evaluation (action execution) is not immediate but is performed within the same transaction, normally at the end of that transaction. Detached coupling refers to when condition evaluation (action execution) is performed in a separate transaction from the one in which the event was detected (or the condition evaluated).

2. TECHNOLOGY ISSUES

In real-time database systems, timeliness and predictability are of paramount importance, and are achieved by eliminating the sources of unpredictability. Ramamritham [17] has identified several sources of unpredictability: dependence of the transaction’s execution sequence on data values; data and resource conflicts; dynamic paging and I/O; and transaction aborts resulting in rollbacks and restarts. Distributed systems have the additional complexity of communication delays and site failures. When incorporating reactive behavior in a real-time database system by using techniques and methods developed within the active database community, several additional incompatibilities and sources of unpredictability can be identified. Similar incompatibilities will be discovered when incorporating timeliness and predictability into an active database using techniques developed within the real-time community. In this section we identify some of these incompatibilities and discuss how they can be addressed in complex real-time applications.

2.1 EVENT DETECTION

In real-time applications, the system must respond to an external stimulus within an upper time bound, where the response usually entails performing a computation and reporting the result. Mapping this to the ECA model, actions are triggered by event occurrences. However, whether the deadline is relative to the time of event occurrence or event detection, and the importance of the triggered transactions, has a dramatic impact on the requirements of the real-time system. If the deadline is relative to the event occurrence, time-cognizant mechanisms for event handling and transaction triggering must explicitly be catered for in the system, as opposed to when deadlines are relative to event detection, where no upper bound can be obtained for the time between event occurrence and detection. However, real-time systems are inherently reactive and have to respond to external events within an upper time bound, implying that
transaction deadlines normally are relative to the time of occurrence. Hence, to guarantee that triggered transaction deadlines are met, each such transaction should be triggered before the latest possible start time of the transaction, which means with enough time to perform scheduling operations and to meet the deadline of the triggered transaction.

ECA rules provide a good model for specifying reactive behavior and enforcing constraints, and for triggering actions upon event occurrences. The original ECA model neither provides mechanisms for specifying time constraints, nor for guaranteeing that time constraints are enforced. Timeliness in this case is no longer a matter only of transaction scheduling, since the time constraints of the actions are determined by the time constraints of the events. Depending on the characteristics of the event, the deadline may be relative to the event occurrence, which is typical for external events occurring in the physical environment, or it may be relative to the time when the system detected the event, in which it is more likely that the event is internal. Hence, in order to obtain a notion of guarantee or schedulability, not only must the set of triggered transactions be considered, but methods and algorithms that are performed between event detection and transaction triggering must be time-cognizant. Hence, methods for event detection, rule selection and triggering and condition evaluation should be predictable.

If traditional event detection is adopted, i.e., events are processed in the same order as they occurred, then uncontrolled behavior can result with the timeliness being jeopardized as an effect of event showers and transient overloads. In [6] Berndtsson and Hansson suggested a scheme for prioritized event detection and rule handling, where events are detected and rules evaluated with respect to the time of event occurrence and the criticality of the actions that may be triggered upon each event. Such an event detection scheme is appropriate for real-time systems, where deadline criticality varies, or where the tightness between time of event occurrence and the latest start time of the action determines the priority. The scheme suggests that events are handled in a strictly prioritized manner, where the priority is determined by the degree of criticality of the most critical action which may be triggered upon that event. Moreover, the tightness of the time constraint of the triggered deadline could also be reflected in the priority assigned. It is suggested that the rule set is analyzed statically in order to determine the criticality of events, which are then parameterized with this information. It is suggested that event showers can be handled by using filtering mechanisms which are sensitive to critical events and thereby can filter out these and present them to the system first.

One implication with the approach is that in systems where it is likely that an event will be involved in rules where the actions vary in criticality and where the event is likely to be part of at least one rule with a critical action, there is risk
that the system will be loaded with critical events. The solution is to logically distinguish between events that generate critical and non-critical events.

2.2 RULE EVALUATION

Upon event detection, the appropriate rules should be selected, that is, those rules that should be triggered as a response to the event occurrence. Within active object-oriented database systems, techniques can be broadly categorized into the centralized approach, indexing of rules by classes, and rules associated with specific events. With the centralized approach, all the rules must be notified to determine which rules are subject to evaluation. By indexing the rules by classes of events efficiency is increased [6]. Neither method can guarantee that no unnecessary rule triggering is performed. It is therefore best that rules should be associated with specific events, and then notify the rules that are specifically interested in that event [6].

In the HiPAC project [9] it was suggested that the cost of evaluating rules should be embedded in the execution cost of the transaction, which results in problems when cascaded firing of rules occurs. Two ways of solving this problem were suggested: (i) restrict rule behavior, or (ii) limit the coupling modes. The first approach implies that rule behavior can be restricted to only allow non-cascade firing of rules. Thus, rules cannot trigger other rules as part of their action part. The second approach implies that coupling modes can be limited to only detached, thereby disallowing immediate and deferred.

Rule conditions in active databases are implemented as either boolean expressions, methods, or database queries. Sophisticated implementations of active databases can apply techniques proposed for query optimization in order to speed up the condition evaluation [10].

2.3 COUPLING MODES

Actions are carried out during the execution phase. As transactions are executed, new events may be generated which may cause cascaded rule firing, which will impose additional workload on the system. Unrestricted cascading of rule firings may cause system overload. Hence, cascading must either be bounded with respect to execution time, restricted in occurrence or prevented entirely. The core issue is when rule execution should be performed with respect to the triggering transactions. In addition, some of the coupling modes are not appropriate for real-time purposes since they may introduce unbounded execution times. When studying the set of coupling modes for event-condition, and condition-action, the following combinations of modes are possible in active database systems: (1) immediate-immediate, (2) immediate-

---

1Detached coupling may be either dependent or independent, but will not affect the list for analysis.
deferred, (3) immediate-detached, (4) deferred-deferred, (5) deferred-detached, (6) detached-immediate and (7) detached-detached. Predictability can be enforced by restricting the set of coupling modes to those not affecting the execution of the triggering transactions.

2.3.1 Event-Condition Coupling. The coupling modes used for event-condition coupling may be a source of unpredictability. The real-time scheduler has, based on the worst-case execution time of the triggering transaction, guaranteed that the deadline of the triggering transaction will be met. Given that the triggering transaction generates new events, and that immediate or deferred coupling is used, this implies that the event detection and condition evaluation cost will be charged to the triggering transaction due to the necessary blocking of the currently executing transaction. Hence, in order not to override the allocated worst-case-execution time, the time consumed on event detection and condition evaluation must be controlled, i.e., the number of events and condition evaluation time must be predictable. If this is not done, the coupling combinations 1-5 cannot be adopted in real-time environments, implying that the rules are reduced to detached event-condition coupling.

For detached event-condition coupling, condition evaluation is performed in a separate transaction, which is scheduled in the same way as any other transaction. Hence, the detached coupling mode does not jeopardize the timeliness of the guaranteed transactions nor predictability of the system. However, an additional workload is imposed on the system that must be handled by the real-time scheduler.

2.3.2 Condition-Action Coupling. Given that condition evaluation passes the test, i.e., an action should be executed, immediate and deferred actions are then executed as sub-transactions on behalf of the triggering transaction. Actually, similar reasoning as for the immediate and deferred event-condition coupling can be applied. Naturally, immediate and deferred condition-action coupling may to an even greater extent jeopardize the timeliness of the already guaranteed triggering transaction. Hence, the execution time of the transaction is prolonged in proportion to the number of rules triggered and executed in immediate and deferred mode, which may cause blocking delays of other transactions that arrived before the rules were triggered. If the number of cascaded sub-transactions and their worst-case-execution time cannot be bounded, immediate and deferred condition-action coupling have no applicability in real-time systems.

As detached condition-action coupling is performed in separate transactions, provided that event detection and condition evaluation is predictable, the temporal behavior of the triggering transaction is not affected, i.e., neither predictability nor timeliness is jeopardized. Referring to the list of coupling
modes, this means that the coupling combinations 3, 5 and 6 can be adopted in a real-time environment.

Three distinct types of dependent detached actions have been identified by Branding et al [7]. Parallel detached coupling with causal dependency implies that the triggered transaction must not commit until the triggering transaction has performed a commit. The triggered transaction may start execution before this though. Sequential detached coupling with causal dependency implies that the triggered transaction should only start executing after the triggering transaction commits. Temporal behavior of the triggering transactions is not affected. Exclusive detached coupled actions with causal dependencies are mainly used for defining contingency actions, that is actions that should commit if and only if the triggering transaction aborts. The execution of the contingency may very well be executed in parallel.

3. SUMMARY AND FUTURE WORK

There are several applications that require both active and real-time capabilities. Without support for both of these in underlying database systems, the designers of such applications are faced with several problems when considering their implementation platform.

Currently, there are no commercial, off-the-shelf, active real-time database platforms available. Hence, the designer has to either use an active database platform, use an real-time database platform, or build an active real-time database platform. The two first alternatives make trade-offs between either active capabilities or real-time capabilities. Even if an active database is chosen, the current support for active capabilities in commercial systems is limited. The last alternative is most likely to satisfy the application’s requirements. However, it will also imply that a large amount of time will be spent on building a new type of database system. Typically, this is something that cannot be accomplished within the time frame of a short term project.

When building an active real-time database system, some incompatibilities can be identified by studying existing methods within each discipline. These incompatibilities are also sources of unpredictability, and often these jeopardize the timeliness and predictability directly. When combining active and real-time database technology, methods and algorithms executed during signaling, triggering, rule scheduling (i.e., how rule conflict sets are processed and the time of condition evaluation and action execution), evaluation, and execution phases must be time-cognizant and enforce predictability. We have in this chapter elaborated on some of the issues that need to be resolved in order to make the marriage between active database systems and real-time database systems a successful one. Timeliness and predictability can be enforced by restricting the set of couplings modes to those not affecting the execution of the
triggering transactions. Unrestricted cascaded rule firing may cause overloads, implying that cascaded rule firing must be bounded. In addition, more work must be done in the area of predictable event detection and monitoring methods, rule triggering methods and dynamic action priority assignment algorithms to turn active real-time database systems into practical systems.

References


This page intentionally left blank.
Chapter 13

REACTIVE MECHANISMS

Jonas Mellin
Joakim Eriksson
Sten F. Andler

Department of Computer Science
University of Skövde, Skövde, Sweden

1. INTRODUCTION

In this chapter, we discuss a more implementation-oriented view of ECA rules, event monitoring, rule management, and rule distribution. We describe how events and rules, as described in Chapter 12, are specified, processed and used in applications. We also discuss how rules can be used in distributed databases.

1.1 OVERVIEW OF EVENT MONITORING

Event monitoring and event composition is addressed in active databases, where the HiPAC project [10] introduced seminal work. Time constraint monitoring is another stream of research, in which a unified approach was presented by Liu et al. [15]. To use event composition in a real-time database system, it must have predictable resource requirements and scale well with respect to the complexity of event specifications.

![Figure 13.1 Event monitoring process [17]](image)

Figure 13.1 Event monitoring process [17]
The process of event detection is to receive events from sources (e.g., software probes) and notifying event subscribers (e.g., application threads or database rules) when some specified event occurs (depicted in Fig. 13.1). Each component (source, monitor, subscriber) may reside in a separate address space. Briefly, composite event detection can be viewed as the process of detecting higher-level events consisting of other lower-level events. Each event occurrence is represented by an event instance carrying parameters denoting its type, in which scope it occurred, and when it occurred. An example of composite event detection is monitoring of foreseeable behaviors, such as a sequence of transaction start followed by transaction termination (either commit or abort). When the associated transaction termination occurs, the event monitor signals this behavior to the subscriber(s). In this case, a subscriber could be a deadlock detector.

Composite event detection [9, 16], performed by event monitors, is defined as the process of looking for an expected behavior in an event history that is a strictly ordered, recorded event stream. Such expected and foreseeable behaviors are described in terms of composite events that are specified using operators (such as conjunction, disjunction, and sequence [9, 12]) combining composite and primitive events. Primitive events [9], generated at event sources, are instantaneous and atomic predefined elementary occurrences in the system (such as start of transactions).

A common optimization of event composition is to keep the intermediate state of event composition, based on an event history, by each event monitor. That is, composite event detection may be viewed as processing an event history, storing an intermediate state representing the behaviors that may occur in the future (incompletely composed events), and signalling the outcomes (events) when they occur [5].

---

![Figure 13.2](image.png)

**Figure 13.2** Directed acyclic graphs of $E_1; E_2$ in event monitoring [17]
It is common to store the parameters of composite events in directed acyclic
graphs [17], because many event specification languages, e.g. Snoop [9] and
SAMOS [12], are based on operator grammars. One can argue that Snoop
is directly related to operator grammars. The data structure of a detected
composite event specified by $E_1; E_2$ (a sequence) is depicted in Fig. 13.2(a).
In Snoop, the internal representation of this specification has a similar structure,
depicted in Fig. 13.2(b). Each incoming edge to the “;” operator is associated
with a queue. Each element in these queues points to an event instance, e.g., $e_1^n$
and $e_2^n$ respectively. The superscript denotes the ordinal of the event instances
and the subscript denotes the type. When the operator detects the sequence, it
generates a composite event instance consisting of the constituents of $E_1$ and
$E_2$, e.g., $e_1^n; e_2^n$.

For example, given the event history in Tab. 13.1, the table shows the content
of the queues after each time step using the chronicle event consumption mode
that matches constituents based on the ordinals [8]. After time step 2, the
queue of edge E1 contains two elements. After time step 3, $e_1^1$ is removed from
this queue and composed with $e_2^1$ to form $e_1^1; e_2^1$, which is sent on the outgoing
edge(s). Note that at time $t_3$, the state of the queues is shown after execution
of the sequence operator but before any further event composition.

<table>
<thead>
<tr>
<th>Time</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Events detected</td>
<td>$e_1^1$</td>
<td>$e_2^1$</td>
<td>$e_2^2$</td>
</tr>
<tr>
<td>Edge Queue E1</td>
<td>$e_1^1$</td>
<td>$e_1^2$</td>
<td>$e_1^2$</td>
</tr>
<tr>
<td>Edge Queue E2</td>
<td>Always empty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Edge Queue E1; E2</td>
<td>$e_1^1; e_2^1$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Table 13.1 Event history example [17]*

### 1.2 OVERVIEW OF RULE MANAGEMENT

Rule handling can be divided into managing of rules and execution of rules.
Included in rule management are specification and storage of rules. Execution
of rules involves retrieval of triggered rules, condition evaluation and action
execution. Since an active real-time database system is assumed, timeliness
is of paramount importance. Therefore, all functions must be performed in a
predictable and efficient way.

As described in Chapter 12, the most used paradigm for specifying rules in
an active database is the ECA rule. In order to allow timely execution, rules
must be extended with temporal attributes. Using these attributes, temporal
behavior and constraints may be expressed.
Rules are triggered when a specified event occurs. When the event notification arrives at the rule manager, all rules triggered by this event must be retrieved from the rule base and be prepared for execution. The time to retrieve rules brings potentially unpredictable overhead to the system, and it is therefore critical to keep this time to a minimum. Methods for storing and retrieving rules must be carefully considered to allow predictable and efficient retrieval.

When all triggered rules have been retrieved, all their conditions must be evaluated to determine whether the action should be executed. Traditionally, in active databases, the condition is expressed as a query to the database, but can be any executable code or expression returning a Boolean value. In either case, predictability is the key issue. When the condition of all triggered rules have been evaluated, the corresponding actions of rules with conditions that evaluated to true must be executed, either in the same transaction or in a separate transaction.

There is always a compromise between flexibility and efficiency; the more static a system is, the more decisions can be made off-line, and thereby run-time overhead can be kept low. If a rule base can be made static, that is, no rules are added, deleted or altered at run-time, the rules can be analyzed, optimized and compiled into the rule manager, thereby improving efficiency.

1.3 OVERVIEW OF RULE DISTRIBUTION

In a system where the database is distributed over several nodes, reactive mechanisms can also be distributed along with the database. The issues to consider are whether events detected on one node are allowed to trigger rules on another node, if composite events can consist of events detected at different nodes, at which node a rule should be triggered, whether a spawned action should be executed on another node than the node on which the rule was triggered, and so on. Often, there is no definite answer to any of these questions, but a specific application demands a specific design of the mechanisms.

In a traditional database, description of data is separated from the program and stored with the data. This facilitates both data sharing and support of data manipulation. In an active database, rules for acting on the data in the database are also separated from the application (as well as from the representation of the data) and stored in the database. As data in the database of a distributed system is often replicated in several nodes, rules could be scheduled to run in any node that contains a copy of the data. This implies that rules in a replicated database can function as an automatic distribution mechanism as outlined in [1], which sketches an example of an application using distributed rules. In that example, the rule database is fully replicated, in that the system has a complete copy of the database and its rules on every node. This means that most of the issues mentioned above can be disregarded.
2. SPECIFICATION

This section describes how events and rules are specified in DeeDS [2] and suggests how specification of rule distribution may be given.

2.1 SPECIFICATION OF EVENTS

The specification of events in DeeDS consists of three parts, (i) an event expression, (ii) the subscribers, and (iii) real-time constraints. An event specification is often part of some other specification, such as ECA rules.

In contrast to active non-real-time databases, the event specifications in DeeDS must allow real-time constraints. For example, event priority is such a specification which can support end-to-end guarantees. If event specifications are given as a part of another specification, then some constraints can be derived from the surrounding specification. For example, event priorities can be derived from the temporal attributes of rules.

The event specification \( tr\text{Start}; \langle policy = \text{chronicle}, \text{match tid} \rangle tr\text{Term} \) is an example implying that we want to monitor transaction start followed by the corresponding transaction termination (commit or abort) in a chronicle context. The events must match on the transaction identifier (\( tid \)). Note that it is not feasible to add the condition for matching parameter as an event parameter condition, e.g. \( (tr\text{Start}; \langle policy = \text{chronicle}\rangle tr\text{Term})[tid(tr\text{Start}) == tid(tr\text{Term})] \), because such a composite may never be generated by the sequence operator. For example, if \( e^1_1 \) and \( e^1_2 \) in Tab. 13.1 are generated by different transaction identifiers, then \( e^1_1 \) would be incorrectly removed from the edge queue (while no event would be signalled). Thus, if an event \( e^m_2 \), \( m > 1 \) from the same transaction as the now removed \( e^1_1 \) is generated at a later stage, then there is no event signalled (which is an incorrect behavior, as the two events should have matched).

2.1.1 Real-time constraints. To limit the resource requirements of event monitoring [16], it is necessary to add temporal constraints that prune the intermediate state (in particular, the internal queues). That is, without temporal constraints the internal queues may grow indefinitely and, hence, there is no upper bound on the event composition processing. Transactions are used in active databases to prune the intermediate state. In real-time active databases, this is insufficient, because it is not normally feasible to associate events occurring outside the system with a transaction.

The temporal specification depends on the monitored objects [16]. If uniquely identified objects are monitored, then it is possible to derive the temporal constraints from the real world. For example, if we know that a vehicle passing one sensor ought to pass another sensor within a given time, then it is possible to use this information to prune the queues. If we do not
constant tank.period = 15s
event read-tanklevel-period periodic tank.period
rule read-tanklevel
on read-tanklevel-period
if true
do deadline tocc + tank.period criticality firm execution.time 600ms
begin
dbstore("tanklevel", tank.read_level());
end

Figure 13.3 ECA rule extended with temporal attributes

have uniquely identified objects, e.g. in control application where continuously varying sensors are sampled, then the temporal constraints may be derived from the control application. In this environment, events expire when they are of no use to the control algorithm.

For example, \(\text{sens}_1; \langle \text{policy} = \text{chronicle}, \text{match} \text{vid} \rangle \text{sens}_2 \text{within} 10s\) implies that a vehicle with vehicle identifier \text{vid} is expected to pass sensor \text{sens}_2 within 10 seconds after it has passed sensor \text{sens}_1. This specification can be translated into a set of event expressions [16] that generates either a success event if the expected behavior occurred or a failure event if it fails. Moreover, these expressions can be used to prune the internal queues.

### 2.2 SPECIFICATION OF RULES

As mentioned above, traditional ECA rules do not include the possibility of specifying temporal attributes for rules. Examples of such attributes are worst-case execution time, earliest release time, deadline and criticality. A few proposals have been made on how to incorporate some simple temporal attributes into the rule specification [7, 18]. In the DeeDS project [2], a full-fledged proposal of ECA rules extended with temporal attributes is presented [11].

As a simple example, consider the rule in Figure 13.3. This specified rule periodically reads the water level in a tank, and stores the value in the database. The reading and storage is calculated to take at most 600 ms. Since the criticality of the action is firm, it must be completed before the next period starts or else the operation is aborted. The deadline is expressed relative to the time of occurrence of the triggering event (\text{tocc}). The rule is triggered by a previously defined event \text{read-tanklevel-period}, which is a periodic event raised every 15 seconds. The condition is specified to be always true since this
rule should always be executed. The action thus has as temporal attributes a deadline, a criticality and an execution time. These attributes are sent to the scheduler on action spawning. The action part consists of a single database operation, which stores the reading of the tank level into the database.

3. PROCESSING

This section describes how rules (including events) are processed, i.e., the necessary processes, time requirements and distribution concerns.

3.1 EVENT MONITORING PROCESSING

In detail, event processing consists of five different categories of functionality: (i) event subscription that includes lookup and signaling, (ii) operator composition algorithms that describe the algorithm for the operators in the event specification language, (iii) mechanisms that are used to realize the operator algorithms, (iv) event parameter condition evaluation, (v) memory management that is desirable to tailor for event composition [17], and (vi) timeout service.

3.1.1 Event subscription. When an event occurs, it is sent to an event monitor. At the monitor, a lookup is performed. To each found subscriber, which may be an event operator or an external thread, the event occurrence is signalled as an event instance. At each operator, the event instance is processed. If the operator generates a composite event, then it is signalled to the subscribers of that operator. This signalling can be optimized by connecting it directly to other operators. This approach does not need to perform a lookup. However, this optimization require both operators to be able to guarantee synchronized access to shared data.

In addition, event priorities must be managed. To do this, one way is to signal an event instance once for each event priority it is used for. Without this approach, the order of occurrence cannot be maintained. It is possible that a system can get swamped with events, but given safe pointers [17] the associated overhead can be reduced.

Finally, although event composition is processed depth-first, delivery to external subscribers (not event operators) must be delayed until all processing associated with an event has been completed. The reason is twofold: (i) we need to guarantee that no event is delivered ahead of any critical event in the same event processing round, to ensure that the subscribers may guarantee timeliness of critical events, and (ii) if a mode change is associated with an event, then this must be delivered as a sentinel (last event). In the latter case, if a mode change event were delivered before other events that it is a constituent of, then there would be no guarantee that subscribers are in an appropriate mode for receiving these events.
3.1.2 Operator composition algorithms. Concerning operator algorithms, the semantics of the operators are similar among the proposed event specification languages. Most of them have binary operators such as conjunction, disjunction, and sequence. In terms of interval operators (operators that relate to interval defined by enabling and disabling events) there are absence, event enabler [8], and history operators. The history operators are troublesome, because they can aggregate an arbitrary number of constituents [16].

The semantics of operators are determined by the event consumption policy. In [9] four contexts are defined: recent, chronicle, continuous, and cumulative. Out of these, recent, chronicle, and non-overlapping continuous are useful for real-time computations. The reason is that they do not generate more than one event instance for each occurrence signalled to an operator. The recent context matches only the most recent occurrences of constituents, which is suitable for control applications. The chronicle context matches constituents based on the ordinal (or time). The continuous context generates potential trends where the non-overlapping variant generates the largest interval between two constituents. The cumulative context adds all incoming event instances to one composite, which is delivered to the recipients.

3.1.3 Mechanisms for operator algorithms. The operator algorithms can be realized by various mechanisms. Among the investigated mechanisms are: event graphs [9], (colored) Petri-nets [12], finite state automata [13], temporal calculi for communication systems [3], and real-time logic monitoring [15]. There is currently no clear evidence of which is the best mechanism to use, but Petri-nets and real-time logic monitoring require more computational steps to perform event composition. However, this is only one part of event monitoring.

3.1.4 Event parameter condition evaluation. Condition evaluation is desirable to check if events have expired. Without condition evaluation, expired events may be signalled to rules and processing capacity may be wasted. It is important to keep condition evaluation as cheap as possible, e.g. by only allowing equivalence checking of parameters, because this may have a severe impact on performance.

3.1.5 Memory management. One important part is memory management. If safe pointers are combined with self-referencing memory [17], then event composition has been shown to take at least 20% less overhead compared to approaches without this solution.

3.1.6 Timeout service. A timeout service is required for real-time monitoring. The reason is that we need relative temporal events that occur at the
end of an interval after a specified event to prune the internal queues. Without a timeout service, it is not possible to guarantee an upper bound on the event composition process, because the internal queues may grow indefinitely.

3.2 RULE PROCESSING

3.2.1 Storage of rules. To keep rule retrieval overhead to a minimum, it must be considered how to store rules effectively for timely retrieval. On event notification, all rules triggered by the event must be retrieved. Of course, a linear search through the entire rulebase is highly inefficient as the size of the rulebase increases. Suggestions have been made on storing rules by classes, but this will also retrieve rules that are not eligible to execute. By indexing the rule base by the triggering event [4], retrieval overhead is kept to an absolute minimum; only those rules triggered by a specific event are being considered for further evaluation.

As mentioned before in Section 1.2, it is beneficial to make the rulebase static, no rules can be added, deleted or altered at runtime. Although this might seem very restrictive, it should be understood that a real-time system, once taken into operation, is rarely changed at runtime. However, to achieve some degree of flexibility, rules can be grouped into sets that are activated in certain states of the system, or modes of operation [11]. In a specific mode, only a subset of all defined rules are sensitive to events. When a mode change occurs, currently enabled rules are disabled, and another set of rules is enabled. Of course, such rule sets can overlap, where a rule can be enabled in multiple modes.

3.2.2 Execution of rules. Rule execution involves several steps (see Figure 13.4). In order to obtain timely rule execution, each step must be predictable and also sufficiently efficient by avoiding overhead.

The total response time can be divided into three parts: event detection time, rule execution time and action execution time. The former two can be further subdivided. For event detection, there might be some time from when the event occurs to when it is actually detected. The corresponding delay is usually even longer for composite event detection.

When an event monitor delivers a notification of the event to the rule manager, the rule manager needs to retrieve all rules triggered by this event and thereafter evaluate all rule conditions, if present. For those rules whose condition evaluates to true, the actions are prepared for execution, and information about these actions are sent to the scheduler for execution.

3.2.3 Evaluate conditions. The condition part of a rule defines a valid system state under which the action is to execute. In its simplest form, it is a logical expression using local or global variables and constants. Conditions can
also be specified as a query to the database, where a non-empty reply denotes success. The latter conditions suffer from the same timeliness unpredictability as any database operation. For maximum expressibility, any method call returning a Boolean value can be used as condition. In such case, timeliness is completely dependent on the programmer of these methods.

3.2.4 Executing actions. Since timeliness is the key issue in real-time systems, it is required to avoid unpredictable behavior. One such behavior is unpredictable extension of transaction execution time. If an action is to be executed in the same transaction in which the event was raised (immediate or deferred coupling mode) the total execution time of the transaction is increased. This may mean that a reschedule is needed, and that formerly guaranteed transactions might miss their deadline. Therefore, it has been argued that all actions should be executed in separately schedulable transactions.

When it has been determined that an action is to be executed, a new transaction is started, and information about this transaction, such as deadline and execution time, is sent to the scheduler. It is then determined by the scheduler how and when to run the action, possibly with overload management [14].

4. APPLICATIONS

This section describes the application of events as a service in its own right, the application of rules to various scenarios in engineering, and the use of rules as a distribution mechanism.
4.1 APPLICATION OF EVENT MONITORING

The application of event monitoring is not solely for the use in rules. It should rather be viewed as a separate service, which the rules can use. The reason is that event monitoring is useful for different purposes. In particular, testing of distributed real-time applications require non-intrusive monitoring [19, 6]. This requires that monitoring operations for testing purposes is left in the operational system. Moreover, there may be other services that require event monitoring, such as software tools, statistics, etc.

One problem with making event monitoring a service is to guarantee isolation properties of transactions. If event monitoring follows the transaction concept, then we have problems with external events that do not belong to the same transaction as other events and with locking of events. The latter leads to performance degradation and potential deadlocks. If event monitoring is outside the transaction concept, then care must be taken not to break isolation in an inappropriate way.

4.2 APPLICATION OF RULES

In many industrial applications, the primary controller function is to read the status of a physical entity which the system is intended to manage, and then to act in a predefined manner corresponding to the readings. This behavior can be mapped in a straight-forward way onto ECA rules. As a simple example, a rule set could be defined as follows: The first rule specifies that the pressure of a tank periodically is read and this value is stored in a database. The second rule specifies that when the pressure value is updated, and if that value is above a safety threshold, then an action is started which opens a safety valve, thereby reducing pressure.

As can be seen, such reactive behavior is easily modelled using ECA rules. Of course, time constraints must be added for real-time applications, for example using the extension presented in this chapter. Moreover, the underlying reactive mechanisms and database need to be designed to satisfy timeliness.

4.3 APPLICATION OF RULES AS A DISTRIBUTION MECHANISM

The previous section described an application which periodically reads the value of a physical entity and stores that value in a database. Assume that this database is fully replicated, i.e., a copy of the complete base is present at every node in the system. This means that the newly written value is propagated to all nodes in the system. Assume further that, at one or more nodes, there are rules specified to react to this value being updated. In other words, in response to an event happening in one node, an action is triggered in another node in
the system. Note that no assumptions have been made as to in which node(s) the rules must be located, since the database is fully replicated. Of course, the application needs to be able to tolerate temporary inconsistency while data is being replicated. Refer to [1] for a more detailed description of this rule-based replication mechanism.

5. SUMMARY AND FUTURE WORK

In this chapter, a more detailed view of real-time active databases has been presented. In order not to jeopardize timeliness of the real-time database system, the reactive mechanisms must be carefully designed with predictability and efficiency in mind. Considerable research has been performed in the area and some research prototypes have been briefly described. It has also been shown that by using a replicated active database, an application can be distributed over several nodes, and by placing rules on specific nodes, desired behavior can be obtained.

Future work to perform includes investigation of performance of the distributed system using rules, and determination of the types of applications in which these mechanisms can be used. Another issue to determine is the ability to capture application semantics using rules. Today, most controlling systems are designed in a procedural manner, and moving towards a reactive paradigm brings out the need for new and different design and specification methods.

References


Chapter 14

UPDATES AND VIEW MAINTENANCE

Ben Kao  
*University of Hong Kong*

K.Y. Lam  
*City University of Hong Kong*

Brad Adelberg  
*Northwestern University*

1. INTRODUCTION

A real-time database system (RTDB) is often employed in a dynamic environment to monitor the status of real-world objects and to discover the occurrences of “interesting” events [1, 2, 3, 4]. As an example, a program trading application monitors the prices of various stocks, financial instruments, and currencies, looking for trading opportunities. A typical transaction might compare the price of German Marks in London to the price in New York and if there is a significant difference, the system will rapidly perform a trade. The state of a dynamic environment is often modeled and captured by a set of base data items within the system. Changes to the environment are represented by updates to the base data. For example, a financial database refreshes its state of the stock market by receiving a “ticker tape” — a stream of price quote updates from the stock exchange.

To better support decision making, the large numbers of base data items are often summarized into views. Some example views in a financial database include composite indices (e.g., S&P 500, Dow Jones Industrial Average and sectoral sub-indices), time-series data (e.g., 30-day moving averages), and theoretical financial option prices, etc. For better performance, these views are materialized. When a base data item is updated to reflect certain external activity, the related materialized views need to be updated or recomputed as well.
Besides base item updates and view recomputations, application transactions are executed to generate the ultimate actions taken by the system. These transactions read the base data and views to make their decisions. For instance, application transactions may request the purchase of stock, perform trend analysis, signal alerts, or even trigger the execution of other transactions. Application transactions may also read other static data, such as a knowledge base capturing expert rules. Figure 14.1 shows the relationships among the various activities in such a real-time database system.

Application transactions can be associated with one or two types of timing requirements: transaction timeliness and data timeliness. Transaction timeliness refers to how "fast" the system responds to a transaction request, while data timeliness refers to how "fresh" the data read is, or how closely in time the data read by a transaction models the environment. Stale data is considered less useful due to the dynamic nature of the data.

Satisfying the two timeliness properties poses a major challenge to the design of a scheduling algorithm for such a database system. This is because the timing requirements pose conflicting demands on the system resources. To keep the data fresh, updates on base data should be applied promptly. Also, whenever the value of a base data item changes, affected derived views have to be recomputed accordingly. The computational load of applying base updates and performing recomputations can be extremely high, causing critical delays to transactions, either because there are not enough CPU cycles for them, or because they are delayed waiting for fresh data. Consequently, application transactions may have a high probability of missing their deadlines.

Figure 14.1  A Real Time Database System
To make the right decision, application transactions need to read fresh data that faithfully reflects the current state of the environment. The most desirable situation is that all the data items read by a transaction are fresh until the transaction commits. This requirement, however, could be difficult to meet. As a simple example, if a transaction whose execution time is 1 second requires a data item that is updated once every 0.1 seconds. The transaction will hold the read lock on the data item for an extensive period of time, during which no new updates can acquire the write lock and be installed. The data item will be stale throughout most of the transaction’s execution, and the transaction cannot be committed without using outdated data.

A stringent data timing requirement also hurts the chances of meeting transaction deadlines. Let us consider our simple example again. Suppose the data update interval is changed from 0.1 seconds to 2 seconds. In this scenario, even though it is possible that the transaction completes without reading stale data, there is a 50% chance that a new update on the data arrives while the transaction is executing. To insist on a no-stale-read system, the transaction has to be aborted and restarted. The delay suffered by transactions due to aborts and restarts, and the subsequent waste of system resources (CPU, data locks) is a serious problem. The definition of data timeliness thus needs to be relaxed to accommodate those difficult situations (e.g., by allowing transactions to read slightly outdated data, probably within a predefined tolerance level). We will discuss the options for relaxing the data timing requirement in this chapter.

Given a correctness criterion, we need a suitable transaction scheduling policy to enforce it. For example, a simple way to ensure data timeliness is to give updates and recomputations higher priorities over application transactions, and to abort a transaction when it engages in a data conflict with an update or recomputation. This policy ensures that no transactions can commit using old data. However, giving application transactions low priorities severely lower their chances of meeting deadlines. This is especially true when updates (and thus recomputations) arrive at a high rate. We will investigate how transaction should be scheduled to balance the contrary requirements of data and transaction timeliness.

As an overview, this chapter studies the intricate balance in scheduling the three types of activities: updates, recomputations, and application transactions to satisfy the two timing requirements of data and transactions. The goals are:

- to understand the properties of updates, recomputations, and application transactions;
- to define temporal correctness from the perspective of transactions;
- to investigate the performance of various transaction scheduling policies in meeting the two timing requirements of transactions under different correctness criteria; and
to address the design issues of an RTDB such that temporal correctness can be enforced.

2. **UPDATES, RECOMPUTATIONS, AND TRANSACTIONS**

In this section we take a closer look at some of the properties of updates, recomputations, and application transactions. We will discuss how these properties affect the design of a real-time database system. In particular, we discuss the concept of update locality, the high fan-in/fan-out relationship of updates and recomputations, the idea of recomputation batching, and the timing requirements of transactions. These properties are common in many real-time database systems such as programmed stock trading.

For many real-time database applications, managing the data input streams and applying the corresponding database updates represents a non-trivial load to the system. For example, a financial database for program trading applications needs to keep track of more than three hundred thousand financial instruments. To handle the U.S. markets alone, the system needs to process more than 500 updates per second [5]. An update usually affects a single base data item (plus a number of related views). Three important properties of updates that affect the design of an RTDB are:

**Periodic versus aperiodic updates.** With periodic updates, the current value of a data object is provided at periodic intervals regardless of whether it has changed or not. Examples include sensor readings on an aircraft, whose values change continuously and thus it makes sense to sample the values at regular interval. Aperiodic updates do not occur at times predictable by the database system, and usually only occur when the value of the data object changes. A stock’s price, for example, only changes when there are trading activities on the stock. A major difference between periodic and aperiodic updates is that for the former, the system is better informed (e.g., the system knows when a data item will become stale). The update load is also regular and predictable. On the other hand, for aperiodic updates, the system can assume that a data item remains fresh unless an update on that data item arrives. Therefore, no updates are needed during steady period.

**Update-in-place versus append-only.** For some applications, old values of the base items are useless. So installing an update simply means replacing the old value by the new one carried in the update. For other applications (e.g., a financial database system that performs data charting and trend analysis), historical values are maintained. A data item is thus represented by not a single value but by a time series (or multiversion) of data. In the latter case, updates (and the data) are append-only.
**Staleness.** The application semantics typically determine when a base data item is considered up-to-date or stale. There are actually several options for defining staleness.

One option is to define staleness based on the time when update values were generated. In this case, updates arrive at the RTDB with a timestamp giving the generation time at their external source. A database value is then considered stale if the difference between the current time and its generation timestamp is larger than some predefined maximum age $\Delta$.\(^1\) We call this definition *Maximum Age (MA)*. Notice that with MA, even if a view object does not change value, it must still be periodically updated, or else it will become stale. Thus, MA makes more sense in applications where all view data is periodically updated and/or where data that has not been recently updated is “suspect.” For example, MA may be used in a plant control system where sensors generate updates on a regular basis, or in a military scenario where an aircraft position report is not very useful unless it is quite recent.

Another option is to be optimistic and assume that a data object is always fresh unless an update has been received by the system but not yet applied to the data. We will refer to this definition as *Unapplied Update (UU)*. A problem with UU is that it ignores delays that may occur before an update is handed to the RTDB system. Hence, UU is more useful in applications with fast and reliable delivery of updates. For example, UU may be used in a telecommunications RTDB server because delays outside the system may be irrelevant and because we do not want the extra traffic associated with MA. (Example: if a call is on-going, we do not want to be periodically notified that it is still going on.) Figure 14.2 illustrates the two staleness models.

There are several variations on these two basic definitions. For example, in the MA staleness definition we could replace generation time by arrival time at

---

\(^1\)For simplicity we assume synchronized physical clocks.
the RTDB. This would mean that updates for every database object should be received at least every $\Delta$ units. We could also combine $MA$ and $UU$. Here, an object would be considered stale if it were stale under either definition. Readers are referred to [6, 7] for some other interesting discussion on data timeliness.

The high volume of updates and their special properties (such as write-only or append-only) warrant special treatment in an RTDB. In particular, they should not be executed with full transactional support. If each update is treated as a separate transaction, the number of transactions will be too large for the system to handle. (Recall that a financial database may need to process more than 500 updates per second.) Application transactions will also be adversely affected because of resource conflicts against updates. As is proposed in [3], a better approach is to apply the update stream using a single update process. Depending on the scheduling policy employed, the update process installs updates in a specific order. It could be linear in a first-come-first-served manner, or on-demand upon application transactions’ requests.

When a base data item is updated, the views which depend on the base item have to be updated or recomputed as well. The system load due to view recomputations can be even higher than that is required to install updates. While an update involves a simple write operation, recomputing a view may require reading a large number of base data items (high \textit{fan-in}),\textsuperscript{2} and complex operations\textsuperscript{3}. Also, an update can trigger multiple recomputations if the updated base item is used to derive a number of views (high \textit{fan-out}).

One way to reduce the load due to updates and recomputations is to avoid useless work. An update is \textit{useful} only if the value it writes is read by a transaction. So if updates are done in-place, an update to a base item $b$ needs not be executed if no transactions request $b$ before another update on $b$ arrives. Similarly, a recomputation on a view needs not be executed if no transactions read the view before the view is recomputed again. This savings, however, can only be realized if successive updates or recomputations on the same data or view occur closely in time. We call this property \textit{update locality} [4].

Fortunately, many applications that deal with derived data exhibit such a property. Locality occurs in two forms: time and space. Updates exhibit time locality if updates on the same item occur in bursts. Since base data models the physical environment, an update on a base data item often signals the “movement” of a modeled object. In many cases, this motion generates a burst of updates to the same base item over a certain period of time. As an example, a stock price update usually indicates trading activity on that stock. Further

\textsuperscript{2}For example, the S&P 500 index is derived from a set of 500 stocks; a summary of a stock’s price in an one-hour interval could involve hundreds of data points.

\textsuperscript{3}For example, computing the theoretical value of a financial option price requires computing some cumulative distributions.
related updates are therefore likely to arrive soon. As a representative example, Figure 14.3 shows the inter-arrival distribution of the stock price updates of General Electric Co. on January 3rd, 1994 [4]. In this trace there were 3240 updates on G.E. stock over a trading period of about 31,000 seconds. The average update arrival rate is therefore about one update every 10 seconds. If there were no correlation in update arrivals, the inter-arrival time would be exponentially distributed (memoryless). This hypothetical exponential distribution (with mean 10 seconds) is also shown in Figure 14.3. We see that the actual distribution is much more “skewed” than the exponential one. For example, about 700 out of the 3240 updates occur within 1 second of a previous update, which is about twice as many as the exponential distribution predicts. Also, 2/3 of the updates occur within 4 seconds of a previous one. This is again twice as many as the number one would expect if there were no correlation among update arrivals. Thus, the graph clearly illustrates the time locality of the updates.

The other form of locality is space locality: when a base item $b$, which affects a derived item $d$, is updated, it is very likely that a related set of base items, affecting $d$, will be updated soon. For example, changes in a bank’s stock price may indicate that a certain event (such as an interest rate hike) affecting bank stocks has occurred. It is thus likely that other banks’ stock prices will change too. Each of these updates could trigger the same recomputation, say for the finance sectoral index.
Update locality implies that recomputations for derived data occur in bursts. Recomputing the affected derived data on every single update is probably very wasteful because the same derived data will be recomputed very soon, often before any application transaction has a chance to read the derived data for any useful work. Instead of recomputing immediately, a better strategy is to defer recomputations by a certain amount of time and to batch or coalesce the same recomputation requests into a single computation. We call this technique recomputation batching. In [4], it is shown that batching significantly improves the performance of an RTDB.

Application transactions may read both base data and derived views. One very important design issue in the RTDB system is whether to guarantee consistency between base data and the views. To achieve consistency, recomputations for derived data are folded into the triggering updates. Unfortunately, running updates and recomputations as coupled transactions is not desirable in a high performance, real-time environment. It makes updates run longer, blocking other transactions that need to access the same data. Indeed, [8] shows that transaction response time is much improved when events and actions (in our case updates and recomputations) are decoupled into separate transactions. Thus, we assume that recomputations are decoupled from updates. We will discuss how consistency can be maintained in Section 4.

Besides consistency constraints, application transactions are associated with deadlines. We assume a firm real-time system. That is, missing a transaction’s deadline makes the transaction useless, but it is not detrimental to the system. In arbitrage trading, for example, it is better not to commit a tardy transaction, since the short-lived price discrepancies which trigger trading actions disappear quickly in today’s efficient markets. Occasional losses of opportunity are not catastrophic to the system. The most important performance metric is thus the fraction of deadlines the RTDBS meets. In Section 4 we will study a number of scheduling policies and comment on their performance in meeting deadlines.

3. TEMPORAL CORRECTNESS

One of the requirements in an RTDB system is that transactions read fresh and consistent data. Temporal Consistency refers to how well the data maintained by the RTDB models the actual state of the environment [9, 10, 11, 12, 7, 13]. Temporal consistency consists of two components: absolute consistency (or external consistency) and relative consistency. A data item is absolutely consistent if it timely reflects the state of an external object that the data item models. We can define absolute consistency based on the staleness model (MA and UU) described in Section 2. A set of data items are relatively consistent if their values reflect the states of the external objects at the same time instant.
If a base data item is updated but its associated views are not recomputed yet, the database is not relatively consistent. It is clear that an absolutely consistent database must also be relatively consistent. However, the converse is not true. For example, a relatively consistent database that never installs updates remains relatively consistent even though its data are all stale. An ideal system that performs updates and recomputations instantaneously would guarantee both absolute and relative consistency. However, as we have argued, to improve performance, updates and recomputations are decoupled, and recomputations are batched. Hence, a real system is often in a relatively inconsistent state. Fortunately, inconsistent data do no harm if no transactions read them. Hence, we need to extend the concept of temporal consistency from the perspective of transactions. Here, we formally define our notion of transaction temporal consistency. We start with the definition of an ideal system first, based on which correctness and consistency of real systems are measured.

**Definition 1:** instantaneous system (IS) An instantaneous system applies base data updates and performs all necessary recomputations as soon as an update arrives, taking zero time to do it.

**Definition 2:** absolute consistent system (ACS) In an absolute consistent system, an application transaction, with a commit time $t$ and a readset $R$, is given the values of all the objects $o \in R$ such that this set of values can be found in an instantaneous system at time $t$.

The last definition does not state that in an absolute consistent system data can never be stale or inconsistent. It only states that no transactions can read stale or inconsistent data. It is clear that transactions are given a lower execution priority comparing with updates and recomputations. For example, if an update (or the recomputations it triggers) conflicts with a transaction on certain data item, the transaction has to be aborted. Maintaining an absolute consistent system may thus compromise transaction timeliness. To have a better chance of meeting transactions’ deadlines, we need to upgrade their priorities. A transaction’s priority can be upgraded in two ways, with respect to its accessibility to data and CPU. For the former, transactions are not aborted by updates due to data conflicts, while for the latter, transactions are not always scheduled to execute after updates and recomputations.

**Definition 3:** relative consistent system (RCS) In a relative consistent system with a maximum staleness $\Delta$, an application transaction with a start time $t$ and a readset $R$ is given the values of all the objects $o \in R$ such that this set of values can be found in an instantaneous system at time $t_1$, and $t_1 \geq t - \Delta$.

Essentially, an RCS allows some updates and recomputations to be withheld for the benefit of expediting transaction execution. Data absolute consistency is compromised but relative consistency is maintained. Figure 14.4 illustrates the two correctness criteria.
This figure illustrates the differences between ACS and RCS. Suppose a transaction T reads objects o₁ and o₂ during its execution, with maximum staleness Δ. Let oᵢⱼ denote the jᵗʰ version of object oᵢ. In an ACS, the set of objects read by T must be (o₁₂, o₂₂) because only this set of values can be found in an IS at the commit time of T. In an RCS, the object versions available to T are (o₁₁, o₁₂), (o₂₁, o₂₂) or (o₁₂, o₂₂) as they can be found in an IS at a time not earlier than to.

4. TRANSACTION SCHEDULING AND CONSISTENCY ENFORCEMENT

In this section we discuss different policies to schedule updates, recomputations, and application transactions to meet the different levels of temporal consistency requirements. As we have argued, data timeliness can best be maintained if updates and recomputations are given higher priorities than application transactions. We call this scheduling policy URT (for update first, recomputation second, transaction last). On the other hand, the On-Demand (OD) strategy [3], with which updates and recomputations are executed upon transactions’ requests, can better protect transaction timeliness. We will therefore focus on these two scheduling policies and compare their performance under the different temporal consistency requirements. Later on, we will discuss how URT and OD can be combined into the OD-H policy. In simple terms, OD-H switches between URT and OD depending on whether application transactions are running in the system. In these policies, we assume that the relative priorities among application transactions are set using the traditional earliest-deadline-first priority assignment. We start with a brief reminder of the characteristics of the three types of activities.

Updates. We assume that updates arrive as a single stream. Under the URT policy, there is only one update process in the system executing the updates in a FCFS manner. For OD, there could be multiple update activities running concurrently: one from the arrival of a new update, and others triggered by application transactions. We distinguish the latters from the formers by labeling them “On-demand updates” (or OD-updates for short).
Recomputations. When an update arrives, it spawns recomputations. Under URT, we assume that recomputation batching is employed to reduce the system’s workload [4]. With batching, a triggered recomputation goes to sleep for a short while during which other newly triggered instances of the same recomputation are ignored. Under OD, recomputations are only executed upon transactions’ requests, and hence batching is not applied. To ensure temporal consistency, however, a recomputation induced by an update may have to perform some book-keeping processing, even though the real recomputation process is not executed immediately. We distinguish the recomputations that are triggered on-demand by transactions from those book-keeping recomputation activities by labeling them “On-demand recomputations” (or OD-recoms for short).

Application Transactions. Finally, we assume that application transactions are associated with firm deadlines. A tardy transaction is useless and thus should be aborted by the system.

Scheduling involves “prioritizing” the three activities with respect to their accesses to the CPU and data. We assume that data accesses are controlled by a lock manager employing the HP-2PL protocol (High Priority Two Phase Locking) [14]. Under HP-2PL, a lock holder is aborted if it conflicts with a lock requester that has a higher priority than the holder. CPU scheduling is more complicated due to the various batching/on-demand policies employed. We now discuss the scheduling procedure for each activity under four scenarios. These scenarios correspond to the use of the URT/OD policy in an ACS/RCS.

4.1 POLICIES FOR ENSURING ABSOLUTE CONSISTENCY

As defined in last section, an AC system requires that all items read by a transaction be fresh and relatively consistent up to the transaction’s commit time. It is the toughest consistency requirement for data timeliness.

4.1.1 URT. Ensuring absolute consistency under URT represents the simplest case among the four scenarios. Since the update process and recomputations have higher priorities than application transactions, in general, no transactions can be executed unless all outstanding updates and recomputations are done. The only exception occurs when a recomputation is forced-delayed (for batching). In this case the view to be updated by the recomputation is temporarily outdated. To ensure that no transactions read the outdated view, the recomputation should issue a write lock on the view once it is spawned, before it goes to sleep. Since transactions are given the lowest priorities, an HP-2PL lock manager is sufficient to ensure that a transaction is restarted (and thus
cannot commit) if any data item (base data or view) in the transaction’s read set is invalidated by the arrival of a new update or recomputation.

### 4.1.2 OD.
The idea of On-Demand is to defer most of the work on updates and recomputations so that application transactions get a bigger share of the CPU cycles. To implement OD, the system needs an On-Demand Manager (ODM) to keep track of the unapplied updates and recomputations. Conceptually, the ODM maintains a set of data items \( x \) (base or view) for which unapplied updates or recomputations exist (we call this set the unapplied set). For each such \( x \), the ODM associates with it the unapplied update/recomputation, and an \textit{OD bit} signifying whether an OD-update/OD-recom on \( x \) is currently executing. There are five types of activities in an OD system, namely, update arrival, recomputation arrival, OD-update, OD-recom, and application transaction. We list the procedure for handling each type of event as follows:

**On an update or recomputation arrival.** Newly arrived updates and recomputations are handled in a FCFS manner, and have higher priorities than OD-updates, OD-recoms and transactions. An update/recomputation \( P \) on a base/view item \( x \) is first sent to the OD Manager. The ODM checks if \( x \) is in the unapplied set. If not, \( x \) is added to the set with \( P \) associated with it, and a write lock on \( x \) is requested\(^4\); Otherwise, the OD bit is checked. If the OD bit is “off”, the ODM simply associates \( P \) with \( x \) (essentially replacing the old unapplied update/recomputation by \( P \)); If the OD bit is “on”, it means that an OD-update/OD-recom on \( x \) is currently executing. The OD Manager aborts the running OD-update/OD-recom and releases \( P \) for execution. In the case of an update arrival, any view that is based on \( x \) will have its corresponding recomputation spawned as a new arrival.

**On an application transaction read request.** Before a transaction reads a data item \( x \), the read request is first sent to the OD Manager. The ODM checks if \( x \) is in the unapplied set. If so, and if the OD bit is “on” (i.e., there is an OD-update/OD-recom being run), the transaction waits; otherwise, the ODM sets the OD bit “on” and releases the OD-update/OD-recom associated with \( x \). The OD-update/OD-recom inherits the priority of the reading transaction.

**On the release of an OD-update/OD-recom.** An OD-update/OD-recom executes as a usual update or recomputation transaction. When it finishes, however, the OD Manager is notified to remove the updated item from the unapplied set.

\(^4\) The write lock is set to ensure AC, since any running transaction that has read (an outdated) \( x \) will be restarted due to lock conflict.
4.2 POLICIES FOR ENSURING RELATIVE CONSISTENCY

The major difficulty in an ACS is that an application transaction is easily restarted if some update/recomputation conflicts with the transaction. An RCS ameliorates this difficulty by allowing transactions to read slightly outdated (but relatively consistent) data. An RCS is thus meaningful only if it can maintain multiple versions of a data item; each version records the data value that is valid within a window of time (its validity interval).

For notational convenience, we use a numeric subscript to enumerate the versions of a data item. For example, \( x_i \) represents the \( i \)th version of the data item \( x \). We define the validity interval of an item version \( x_i \) by \( VI(x_i) = [LTB(x_i), UTB(x_i)] \), where LTB and UTB stand for the lower time bound and the upper time bound of the validity interval respectively. Given a set of item versions \( D \), we define the validity interval of \( D \) as \( VI(D) = \bigcap \{ VI(x_i) \mid x_i \in D \} \). That is, the set of values in \( D \) is valid throughout the entire interval \( VI(D) \). Also, we denote the arrival time of an update \( u \) by \( ts(u) \). Finally, for a recomputation or an application transaction \( T \), we define its validity interval \( VI(T) \) as the time interval such that all values read by \( T \) must be valid within \( VI(T) \).

An RCS needs a Version Manager (VM) to handle the multiple versions of data items. The function of the Version Manager is twofold. First, it retrieves, given an item \( x \) and a validity interval \( I \), a value of a version of \( x \) that is valid within \( I \). Note that if there are multiple updates on \( x \) during the interval \( I \), the Version Manager would have a choice of a valid version. We defer our discussion on this version selection issue later. Second, the VM keeps track of the validity intervals of transactions and the data versions they read. The VM is responsible for changing a transaction’s validity interval if the validity interval of a data version read by the transaction changes. We will discuss the VI management shortly. Finally, we note that since every write on a base item or a view generates a new version, no locks need to be set on item accesses. We will discuss how the “very-old” versions are pruned away to keep the multi-version database small at the end of this section.

4.2.1 URT. Similar to an ACS, there are three types of activities under URT in an RCS:

**On an update arrival.** As mentioned, each version of a data item in an RCS is associated with a validity interval. When an update \( u \) on a data item version \( x_i \) arrives, the validity interval \( VI(x_i) \) is set to \( [ts(u), \infty] \). Also, the UTB of the previous version \( x_{i-1} \) is set to \( ts(u) \), signifying that the previous version is only valid till the arrival time of the new update. The Version Manager checks
and sees if there is any running transaction \( T \) that has read the version \( x_{i-1} \). If so, it sets 
\[
UTB(VI(T)) = \min\{UTB(VI(T)), ts(u)\}
\]

**On a recomputation arrival.** If an update \( u \) spawns a recomputation \( r \) on a view item \( v \) whose latest version is \( v_j \), the system first sets the UTB of \( v_j \) to \( ts(u) \). That is, the version \( v_j \) is no longer valid from \( ts(u) \) onward. Similar to the case of an update arrival, the VM updates the validity interval of any running transaction that has read \( v_j \). With batching, the recomputation \( r \) is put to sleep, during which all other recomputations on \( v \) are ignored. A new version \( v_{j+1} \) is not computed until \( r \) wakes up. During execution, \( r \) will use the newest versions of the data in its read set. The validity interval of \( r \) (\( VI(r) \)) and that of the new view version (\( VI(v_{j+1}) \)) are both equal to the intersection of all the validity intervals of the data items read by \( r \).

**Running an application transaction.** Given a transaction \( T \) whose start time is \( ts(T) \), we first set its validity interval to \([ts(T) - \Delta, \infty]\).\(^5\) If \( T \) reads a data item \( x \), it consults the Version Manager. The VM would select a version \( x_i \) for \( T \) such that \( VI(x_i) \cap VI(T) \neq \emptyset \). That is, the version \( x_i \) is relatively consistent with the other data already read by \( T \). \( VI(T) \) is then updated to \( VI(x_i) \cap VI(T) \). If the VM cannot find a consistent version (i.e., \( VI(x_i) \cap VI(T) = \emptyset \forall x_i \)), \( T \) is aborted. Note that the wider \( VI(T) \) is, the more likely that the VM is able to find a version of \( x \) that is consistent with what \( T \) has already read. Hence, a good choice is to pick the version \( x_i \) whose validity interval has the biggest overlapping with that of \( T \).

4.2.2 OD. Applying on-demand in an RCS requires both an OD Manager and a Version Manager. The ODM and the VM serve similar purposes as described previously, with the following modifications:

- Since multiple versions of data are maintained, the OD Manager keeps, for each base item \( x \) in the unapplied set, a list of unapplied updates of \( x \).

- In an ACS (single version database), an unapplied recomputation to a view item \( v \) is recorded in the ODM so that a transaction that reads \( v \) knows that the current database version of \( v \) is invalid. However, in an RCS (multi-version database), the validity intervals of data items already serve the purpose of identifying the right version. If no such version can be found in the database, the system knows that an OD-recom has to be triggered. Therefore, the ODM in an RCS does not maintain unapplied recomputations.

\(^5\)Recall that \( \Delta \) is the maximum staleness tolerable with reference to a transaction's start time.
• In an ACS, an OD bit of a data item $x$ is set if there is an OD-update/OD-recom currently executing to update $x$. The OD bit is used so that a new update/recomputation arrival will immediately abort the (useless) OD-update/OD-recom. In an RCS, since multiple versions of data are kept, it is not necessary to abort the (old but useful) OD-update/OD-recom. Hence, the OD bits are not used.

• Since different versions of a data item can appear in the database as well as in the unapplied list, the Version Manager needs to communicate with the OD Manager to retrieve a right version either from the database or by triggering an appropriate OD-update from the unapplied lists.

Here, we summarize the key procedures for handling the various activities in an OD-RCS system.

**On an update arrival.** Newly arrived updates have the highest priorities in the system and are handled FCFS. An update $u$ on a base item $x$ is sent to the OD Manager. Each unapplied update is associated with a validity interval. The validity interval of $u$ is set to $[ts(u), \infty]$. If there is a previous unapplied update $u'$ on $x$ in the ODM, the UTB of $VI(u')$ is set to $ts(u)$; otherwise the latest version of $x$ in the database will have its UTB set to $ts(u)$. Similarly, for any view item $v$ that depends on $x$, if its latest version in the database has an open UTB (i.e., $\infty$), The UTB will be updated to $ts(u)$. The changes to the data items’ UTBs may induce changes to some transactions’ validity intervals. The Version Manager is again responsible for updating the transactions’ VIs.

**Running an application transaction.** A transaction $T$ with a start time $ts(T)$ has its validity interval initialized to $[ts(T) - \Delta, \infty]$. If $T$ reads a base item $x$, The VM would select a version $x_i$ for $T$ that is valid within $VI(T)$. If such a version is unapplied, an OD-update is triggered by the OD Manager. The OD-update inherits the priority of $T$. If $T$ reads a view item $v$, The VM would select a version $v_j$ for $T$ that is valid within $VI(T)$. If no such version in the database is found, an OD-recom $r$ to compute $v$ is triggered. This OD-recom inherits the priority and the validity interval of $T$, and is processed by the system in the same way as for an application transaction.

**4.2.3 Pruning the multi-version database.** Our RC system requires a multi-version database and an OD Manager that keeps multiple versions of updates in the unapplied lists. We remark that it is not necessary that the system keeps the full history on-line. One way to prune away old versions is to maintain a Virtual Clock (VC) of the system. We define VC to be the minimum of the start times of all running transactions minus $\Delta$. Any versions (be they in the database or in the unapplied lists) whose UTBs are smaller than the virtual clock can be pruned. This is because these versions are not valid
with respect to any transaction’s validity interval and thus will never be chosen by the Version Manager. The virtual clock is updated only on the release or commit of an application transaction.

4.2.4 A Hybrid Approach. In OD, updates and recomputations are performed only upon transactions’ requests. If the transaction load is low, few OD-updates and OD-recoms are executed. Most of the database is thus stale. Consequently, an application transaction may have to materialize quite a number of items it intends to read on-demand. This may cause severe delay to the transaction’s execution and thus a missed deadline. A simple modification to OD is to execute updates and recomputations while the system is idling, in a way similar to URT, and switch to OD when transactions arrive. We call this hybrid strategy OD-H.

In [15], an extensive simulation study on the performance of the three scheduling policies, under both an ACS and an RCS is reported. The study identifies the various factors that adversely affect the performance of the policies. It is found that different policies coupled with different consistency systems suffer from different combinations of the factors. In general, the URT policy when applied to an ACS gives the worst performance. The pure On-Demand strategy works better unless transactions arrive at a low rate and have very tight deadlines. The hybrid strategy, OD-H, is shown to perform much better than the others. Finally, the study also shows that by allowing transactions to read slightly outdated data (i.e., RCS), transaction miss rate can be significantly reduced.

5. SUMMARY AND FUTURE WORK

In this chapter we discussed the properties of updates, recomputations, and application transactions. We mentioned various techniques, such as the use of a single update process, recomputation batching, and update-recomputation decoupling to improve the system’s performance. Moreover, we defined the important concept of temporal consistency from the perspective of transactions. In an absolute consistent system, a transaction cannot commit if some data it reads become stale at the transaction’s commit time. We showed that this consistency constraint is very strict. It often results in high transaction miss rate. If transactions are allowed to read slightly stale data, however, the system’s performance can be greatly improved through the use of a multi-version database. We defined a relative consistent system as one with which a transaction reads relatively consistent data items and that those items are not more than a certain threshold (D) older than the transaction’s start time. We argued that a relative consistent system has a higher potential of meeting transaction deadlines.

We discussed three scheduling policies: URT, OD, and OD-H in a system where three types of activities: updates, recomputations, and application
transactions are present. We discussed how the policies are implemented in a real-time database system to ensure absolute consistency or relative consistency. We showed that an ACS using URT is the easiest to implement. An HP-2PL lock manager and a simple static priority-driven scheduler suffice. This system, however, could have a very high transaction miss rate. To improve performance, two techniques were considered. One is to perform updates and recomputations on-demand, and the other is to relax the temporal consistency constraint from absolute to relative. Implementing these techniques add complexities to the implementation, though. For example, an on-demand manager is needed for OD: a version manager is needed for an RCS. The relative performance of the various strategies was discussed.

References


PART V

DISTRIBUTED REAL-TIME DATABASE SYSTEMS
This page intentionally left blank.
Chapter 15

DISTRIBUTED CONCURRENCY CONTROL

Özgür Ulusoy
Department of Computer Engineering
Bilkent University
06533 Bilkent, Ankara, TURKEY
oulusoy@cs.bilkent.edu.tr

1. INTRODUCTION

Distributed databases fit more naturally in the decentralized structures of many real-time database system (RTDBS) applications that are inherently distributed (e.g., the stock market, banking, command and control systems, and airline reservation systems). Distributed database systems provide shared data access capabilities to transactions; i.e., a transaction is allowed to access data items stored at remote sites. While scheduling transactions in a distributed RTDBS, besides observing the timing constraints, it should also be provided that the global consistency of the distributed database is preserved as well as the local consistency at each data site. To achieve this goal, it is required to exchange messages that carry scheduling information between the sites where the data items to be accessed by an executing transaction reside. The communication delay introduced by message exchanges constitutes a substantial overhead for the response time of a distributed transaction. Thus, guaranteeing the response times of transactions (i.e., satisfying the timing constraints), is more difficult in a distributed RTDBS than that in a single-site RTDBS.

The results of a considerable amount of research devoted to various issues in distributed RTDBSs have appeared in the literature. These issues include concurrency control (e.g., [10], [11], [12], [13], [14], [15], [17], [18], [21], [23], [24]), deadline assignment (e.g., [8]), replication (e.g., [25]), nested transaction execution ([4], [26]), and commitment (e.g., [5], [22]). Among these issues, concurrency control in distributed RTDBSs is the main theme of this chapter. Concurrency control in database systems is used to control the interaction among concurrently executing transactions in order to maintain the consistency of the database [20]. In a distributed database system, a scheduler at each site is
responsible for controlling concurrent accesses to data items stored at that site. Access requests of both local and remote transactions are ordered together on the basis of the concurrency control protocol being executed. The schedulers at all sites together constitute a distributed scheduler. Implementation of concurrency control protocols in a distributed RTDBS environment is difficult due to the conflicting requirements of meeting timing constraints and maintaining data consistency. Research efforts in this field has basically focused on development and evaluation of protocols that meet soft/firm deadline requirements with the aim of minimizing the number of missed deadlines.

This chapter provides a brief overview of the recent work that has addressed the concurrency control problem in distributed RTDBSs. It also includes the relative performance results of several real-time concurrency control protocols in a distributed database environment. The protocols aim to maximize the satisfaction of real-time requirements while maintaining data consistency via enforcing serializability. Concurrency control protocols are different in the way timing constraints of transactions are involved in controlling concurrent accesses to shared data.

2. RESEARCH EFFORTS

In this section, we describe examples of work that has addressed the concurrency control issue in distributed RTDBSs. The amount of work done in this field is much less compared to the work on concurrency control in single-site RTDBSs.

In an early work on distributed real-time concurrency control, Sha et al. proposed a protocol to overcome the so called priority inversion problem, which is the result of blocking high priority transactions by lower priority ones [17], [18]. The protocol which is called priority ceiling, bounds the blocking time of high priority transactions to no more than one transaction execution time, and also eliminates the deadlock problem of the locking protocols. The 'priority ceiling' of a data item is defined as the priority of the highest priority transaction that may have a lock on that item. In order to obtain a lock on a data item, the protocol requires that a transaction \( T \) must have a priority strictly higher than the highest priority ceiling of data items locked by the transactions other than \( T \). Otherwise, transaction \( T \) is blocked by the transaction which holds the lock on the data item of the highest priority ceiling. Performance of the priority ceiling protocol was examined in [19] using simulation. The results obtained revealed that performance of the protocol is not satisfactory when the database is not memory resident. However, a significant improvement was observed in the performance when intention I/O was used to prefetch data items accessed by transactions. Son and Chang investigated the performance of several different versions of the priority ceiling protocol [21]. They developed
and used a prototype of a distributed RTDBS for the performance evaluation. The priority inversion problem was investigated by Huang et al. as well in a particular real-time environment where a two-phase locking concurrency control protocol was employed [6], [7].

Lam and Hung introduced two locking-based concurrency control protocols for distributed RTDBSs [11]. The first protocol, which is based on dynamic locking, uses the concept of cautious waiting to resolve lock conflicts among transactions. The performance of this protocol was shown to be better than some other protocols with different conflict resolution strategies. The second protocol proposed by the authors is based on static locking, assuming that lock requirements of transactions are known prior to their execution. It was shown that this protocol performs well for the systems where the majority of lock requests of transactions are on the remote data items. The authors provided three more static locking protocols in a more recent paper [12]. These protocols resolve lock conflicts by reserving the locks for high priority transactions. The protocols are different in their methods of reducing the blocking time of high priority transactions. The results of an extensive simulation study of the protocols presented in the paper show that the relative performance of the protocols depends on the degree of data contention that exists in the system. It was also observed that performance of the protocols is at a higher level when a centralized scheduler is employed.

An optimistic concurrency control protocol with dynamic adjustment of serialization order was adapted to distributed RTDBSs by Lam et al. [10]. With this protocol, it is aimed to take the advantage of dynamic adjustment of serialization order, while trying to minimize the overhead of that adjustment in a distributed environment. Among the factors that can lead to a substantial amount of overhead are the increased complexity of the dynamic adjustment, high communication message volume, and the possibility of distributed deadlocks during the validation of transaction executions. The performance results presented in the paper show that the proposed protocol is effective in reducing the overhead of dynamic adjustment of serialization order. The protocol involves a new deadlock-free distributed validation scheme and a new conflict resolution scheme.

Lee et al. proposed new priority assignment policies for distributed RTDBSs and investigated their impact on some typical real-time concurrency control protocols [16]. It was presented through performance experiments that optimistic concurrency control protocols are more affected by the priority assignment policies compared to locking-based protocols. It was also shown that considering both transaction deadlines and current data contention together in assigning transaction priorities provides the best performance among a variety of priority assignment techniques.
In a recent paper, Lam et al. introduced a protocol for resolving data conflicts among executing and committing real-time transactions in a distributed database system [14]. The proposed protocol integrates concurrency control and transaction commitment management. The protocol aims to reduce the impact of a committing transaction on the executing transactions which depend on it. A deadline-driven approach is used to overcome the dependency problem by reversing the dependencies if the committing transaction has experienced a long commitment delay.

Lam et al. also studied the concurrency control problem of real-time database transactions in a mobile environment [13], [15]. It was argued by the authors that due to limited bandwidth and unpredictable behavior of mobile networks, results of the past research on distributed real-time concurrency control cannot be directly applicable to mobile distributed real-time database systems. The authors proposed a distributed real-time locking protocol which is based on the High Priority scheme [1] and the concept of similarity [9]. The restrictions introduced by mobile networks were considered in developing the new protocol.

Chen and Gruenwald [4], and Ulusoy [26] provided locking-based concurrency control protocols for distributed RTDBSs where transaction scheduling is based on the nested transaction model. Also provided in references [4] and [26] are the performance evaluation results of the protocols under different loading conditions.

In the rest of the chapter, we provide a presentation of our work on concurrency control in distributed RTDBSs, that was previously published in [23] and [24].

3. A DISTRIBUTED REAL-TIME DATABASE SYSTEM MODEL

This section provides the model of a distributed RTDBS that we used in evaluating real-time concurrency control protocols. In the distributed system model, a number of data sites are interconnected by a local communication network. There exists exactly one copy of each data item in the system. Each site contains a transaction manager, a scheduler, a resource manager, and a message server.

The transaction manager is responsible for generating transaction identifiers and assigning real-time priorities to transactions. Each transaction submitted to the system is associated with a real-time constraint in the form of a deadline. Transaction deadlines are soft; i.e., each transaction is executed to completion even if it misses its deadline. Each transaction is assigned a real-time priority based on the earliest deadline first priority assignment policy. A distributed transaction is modeled as a master process that executes at the originating site of the transaction and a collection of cohorts that execute at various sites where
the required data items reside. The master process initiates the execution of each cohort process. Remote cohorts are initiated by sending a message to the appropriate sites. Each cohort is executed with its transaction’s priority.

Each cohort performs one or more database operations on specified data items. The master process commits a transaction only if all the cohort processes of the transaction run to completion successfully, otherwise it aborts and later restarts the transaction.

Concurrent data access requests of the cohort processes at a site are controlled by the scheduler at that site. The scheduler orders the data accesses based on the concurrency control protocol executed. When a cohort completes all its accesses and processing requirements, it waits for the master process to initiate two-phase commit. Following the successful commitment of the distributed transaction, the cohort writes its updates, if any, into the local database.

Each site’s resource manager is responsible for providing I/O service for reading/updating data items, and CPU service for processing data items, performing various concurrency control operations (e.g. conflict check, locking, etc.) and processing communication messages. Both CPU and I/O queues are organized on the basis of the cohorts’ real-time priorities. Preemptive-resume priority scheduling is used by the CPU at each site; a higher-priority process preempts a lower-priority process, and the lower-priority process can resume when there exists no higher-priority process waiting for the CPU. Communication messages are given higher priority at the CPU than other processing requests. Besides preemption, the CPU can be released by a cohort process as a result of lock conflict, for I/O, or communication to other data sites.

The message server at each site is responsible for sending/receiving messages to/from other sites. It listens on a well-known port, waiting for remote messages.

Reliability and recovery issues were not addressed in our work. We assumed a reliable system, in which no site failures or communication network failures occur.

### 3.1 MODEL PARAMETERS

The following parameters were used to specify the system configuration and workload. Parameter $nr\_sites$ represents the number of data sites in the distributed system. $db\_size$ specifies the number of data items stored in the database of a site, and $mem\_size$ specifies the number of data items that can be held in the main memory of a site. The mean interarrival time of transactions to each of the sites is determined by the parameter $iat$. The times between the arrival of transactions are exponentially distributed. The transaction workload consists of both query and update transactions. $tr\_type\_prob$ specifies the update type probability. $access\_mean$ corresponds to the mean number of
data items to be accessed by a transaction. Accesses are uniformly distributed among data sites. For each data access of an update transaction, the probability that the accessed data item will be updated is specified by the parameter \textit{data\_update\_prob}. The CPU time for processing a data item is determined by the parameter \textit{cpu\_time}, while the time to access a data item on the disk is specified by \textit{io\_time}. It is assumed that each site has one CPU and one disk. For each new transaction, there exists an initial CPU cost of assigning a unique real-time priority to the transaction. This processing overhead is simulated by the parameter \textit{pri\_assign\_cost}. Parameter \textit{mes\_proc\_time} corresponds to the CPU time required to process a communication message prior to sending or after receiving the message. The communication delay of messages between the sites is assumed to be constant and specified by the parameter \textit{comm\_delay}.

\textit{slack\_rate} is the parameter used in assigning deadlines to new transactions. The slack time of a transaction is chosen randomly from an exponential distribution with a mean of \textit{slack\_rate} times the estimated processing time of the transaction.

4. REAL-TIME CONCURRENCY CONTROL PROTOCOLS

In this section, we briefly describe the concurrency control protocols whose relative performance was studied by using the distributed RTDBS model presented in the preceding section.

4.1 PRIORITY INHERITANCE PROTOCOL (PI)

‘Priority inheritance’ is one method proposed to overcome the problem of uncontrolled priority inversion [17]. This method makes sure that when a transaction blocks higher priority transactions, it is executed at the highest priority of the blocked transactions; in other words, it inherits the highest priority.

We implemented a distributed version of the priority inheritance (PI) protocol in our distributed RTDBS model as follows: When a cohort is blocked by a lower priority cohort, the latter inherits the priority of the former. Whenever a cohort of a transaction inherits a priority, the scheduler at the cohort’s site notifies the transaction master process by sending a priority inheritance message, which contains the inherited priority. The master process then propagates this message to the sites of other cohorts belonging to the same transaction, so that the priority of the cohorts can be adjusted. When a cohort in the blocked state inherits a priority, that priority is also inherited by the blocking cohort (and its siblings) if it is higher than that of the blocking cohort.
4.2 HIGH PRIORITY PROTOCOL (HP)

This protocol prevents priority inversion by aborting low priority transactions whenever necessary [1]. High priority (HP) protocol was adapted to our model as follows: In the case of a data lock conflict, if the lock-holding cohort has higher priority than the priority of the cohort that is requesting the lock, the latter cohort is blocked. Otherwise, the lock-holding cohort is aborted and the lock is granted to the high priority lock-requesting cohort. Upon the abort of a cohort, a message is sent to the master process of the aborted cohort to restart the whole transaction. The master process notifies the schedulers at all relevant sites to cause the cohorts belonging to that transaction to abort. Since a high priority transaction is never blocked by a lower priority transaction, this protocol is deadlock-free.

4.3 DATA-PRIORITY-BASED LOCKING PROTOCOL (DP)

This subsection presents the concurrency control protocol that we introduced as an improvement over the ‘priority ceiling’ protocol [23]. Similar to the priority ceiling protocol (see Section 2), data-priority-based (DP) locking protocol is based on prioritizing data items; however, in ordering the access requests of the transactions on a data item \( D \), it considers only the priority of \( D \) without requiring a knowledge of the priorities of all locked items.

Each data item carries a priority which is equal to the highest priority of all transactions in the system that include the data item in their access lists. When a new transaction arrives at the system, the priority of each data item to be accessed is updated if the item has a priority lower than that of the transaction. When a transaction commits and leaves the system, each data item that carries the priority of that transaction has its priority adjusted to that of the highest priority active transaction that is going to access that data item. Protocol DP assumes that transaction priorities are distinct.

Lock requests of transactions are handled by this protocol as follows: In order to obtain a lock on a data item \( D \), the priority of a cohort \( C \) must be equal to the priority of \( D \). Otherwise (if the priority of \( C \) is less than that of \( D \)), cohort \( C \) is blocked by the cohort that determines the current priority of \( D \). Suppose that \( C \) has the same priority as \( D \), but \( D \) has already been locked by a lower priority cohort \( C' \) before \( C \) has adjusted the priority of \( D \). \( C' \) is aborted at the time \( C \) needs to lock \( D \). When a cohort is aborted due to data conflict, the aborted cohort’s master is notified to restart the whole transaction.

This protocol is deadlock-free since a high priority transaction is never blocked by lower priority transactions and no two transactions have the same priority.
5. SUMMARY OF PERFORMANCE EVALUATION RESULTS

Performance of the real-time concurrency control protocols was evaluated by simulating them on the distributed RTDBS model described in Section 3. Values of the system parameters were selected to provide a transaction load and data contention high enough to bring out the differences between the real-time performances of concurrency control protocols. The primary performance metric used in the evaluation of the protocols was success ratio; i.e., the fraction of transactions that satisfy their deadlines.

When the performance of protocols was examined under various levels of transaction load by varying parameter iat, it was observed that the high priority (HP) protocol provides consistently better performance than the priority inheritance (PI) protocol under all load conditions tested. Resolving data conflicts by aborting low priority transactions seems to be more desirable in distributed RTDBSs than allowing the low-priority transactions to execute with inherited high priorities. Another observation was that data-priority-based (DP) locking protocol provides a substantial improvement in real-time performance over the other protocols, and the improvement becomes more as the transaction load in the system increases. Similar to HP, protocol DP does not allow the situation in which a high priority transaction can be blocked by a lower priority transaction. The number of transaction restarts is much lower than that of protocol HP since protocol DP restricts the possibility of transaction abort only to the following case. Between any two conflicting transactions, if the lower priority transaction accesses the data item before the higher priority transaction is submitted to the system (before the priority of the data item is adjusted by the entry of the higher priority transaction) the lower priority transaction is aborted when and if the higher priority transaction accesses the data item before the commitment of the lower priority transaction.

Some other experiments were conducted to evaluate the effect of various other system parameters on the protocols’ performance. These parameters included \(nr_{sites}\) (number of data sites), \(comm\_delay\) (message transfer delay), and \(access\_mean\) (average number of data items accessed by a transaction). The results of each of these experiments can be found in [24]. The comparative performance of the protocols was not sensitive to varying the values of these parameters.

We also conducted various experiments by using a distributed transaction execution model which is different than that described in Section 3. In the modified model, the master process of a transaction spawns cohorts all together, and the cohorts are executed in parallel. The master process sends to each remote site a message containing an (implicit) request to spawn a cohort, and the list of all operations of the transaction to be executed at that site. The assumption
here is that the operations performed by one cohort are independent of the results of the operations performed at the other sites. The sibling cohorts do not have to transfer information to each other. A cohort is said to be completed at a site when it has performed all its operations. The master process can start the two-phase commit protocol when it has received ‘completed’ messages from all its cohorts. Results of the experiments conducted by using this model can also be found in [24].

6. SUMMARY AND FUTURE WORK

Crucial to the concurrency control problem in real-time database systems (RTDBSs) is processing transactions within their deadlines, as well as maintaining the consistency of data accessed by transactions. It is difficult to implement concurrency control protocols in a RTDBS due to the conflicting requirements of meeting timing constraints and maintaining data consistency. The communication delay and other constraints of distributed systems make it more difficult to develop distributed concurrency control protocols that can offer good performance in terms of the satisfaction rate of timing constraints.

In this chapter, various approaches to the concurrency control problem in distributed RTDBSs are briefly described. As summarized in the chapter, research efforts in this field has basically focused on development and evaluation of concurrency control protocols. Our work in this direction was chosen as an example to be presented in the chapter.

We believe that the amount of work done in this field is much less compared to the work on concurrency control in single-site RTDBSs and there is much room for further research and experimentation.

References


Chapter 16

DATA REPLICATION AND AVAILABILITY

Özgür Ulusoy
Department of Computer Engineering
Bilkent University
06533 Bilkent, Ankara, TURKEY
oulusoy@cs.bilkent.edu.tr

1. INTRODUCTION

In a replicated database system copies of data can be stored redundantly at multiple sites. The potential of data replication for high data availability and improved read performance is crucial to distributed real-time database systems (RTDBSs). On the other hand, data replication introduces its own problems. Access to a data item is no longer controlled exclusively by a single site, instead the access control is distributed across the sites each storing a copy of the data item. It is necessary to ensure that mutual consistency of the replicated data is provided; in other words, replicated copies must behave like a single copy. This can be made possible by preventing conflicting accesses on the different copies of the same data item, and by making sure that all data sites eventually receive all updates [7]. Multiple copy updates lead to a considerable overhead due to the communication required among the data sites holding the copies.

In this chapter, a brief overview of the research efforts that has addressed the data replication problem in distributed RTDBSs is provided together with our work which investigates the impact of storing multiple copies of data on satisfying timing constraints of real-time transactions. In our work, a performance model of a distributed RTDBS is employed to evaluate the effects of various workload parameters and design alternatives on system performance. The primary performance issue considered is the satisfaction of transaction deadlines; more specifically, an answer to the following question is looked for: ‘does replication of data always aid in satisfying timing constraints of transactions?’. Various experiments are conducted to identify the conditions under which data replication can help real-time transactions satisfy their timing constraints. Different application types are considered in evaluating the effects
of the degree of data replication. Each application is distinguished by the type (query vs update) and data access distribution (local vs remote) of the processed transactions.

2. RESEARCH EFFORTS

Although the majority of the previous works involving distributed database models assumed either no-replication or full-replication, some performance evaluation studies of partially replicated database systems were also provided (e.g., [1], [4], [5], [9]). The impact of the level of data replication on the performance of conventional database systems was examined in those studies considering the average response time of the transactions and the system throughput to be the basic performance measures. It was found in those evaluations that increasing data replication usually leads to some performance degradation due to the overhead of update synchronization among replicated copies of data.

Very little amount of work has appeared in the literature exploring data replication and availability in RTDBSs. In an early work in this field, Lin and Lin proposed some techniques to enhance the availability of replicated real-time databases [8]. They suggested that a transaction characterized by a strict deadline should be able to execute even if the most up-to-date data copies are not available, so that the mutual consistency requirement can be relaxed for distributed RTDBSs that process hard deadline transactions. They also introduced the user quorum scheme to increase the availability in a partitioned RTDBS. The scheme is different from traditional quorum protocols in that it gives access rights to a partition with a majority of users rather than a partition with a majority of data copies.

Son and Kouloumbis proposed a replication control algorithm for distributed RTDBSs [12], [13], [14]. The algorithm integrates real-time scheduling with data replication control. It employs epsilon serializability¹ as the correctness criterion to provide more concurrency to real-time transactions. In the algorithm, real-time scheduling features are involved in responding to timing requirements of transactions, and a token-based synchronization scheme is used to control data replication. Performance issues of the algorithm were investigated using a prototyping environment [12], [14].

In a more recent paper, Son and Zhang provided another data replication control algorithm for distributed RTDBSs [15]. Similar to the algorithm provided in [14], the new algorithm also uses epsilon serializability as the correctness criterion of maintaining mutual consistency of replicated data asynchronously [10]. It allows queries to execute without being blocked or aborted by update transactions. Inconsistent data may be seen by some queries, but data will eventually converge to a consistent state. Also, the degree of inconsistency in query results can be controlled.

¹Epsilon serializability offers the possibility of maintaining mutual consistency of replicated data asynchronously [10]. It allows queries to execute without being blocked or aborted by update transactions. Inconsistent data may be seen by some queries, but data will eventually converge to a consistent state. Also, the degree of inconsistency in query results can be controlled.
criterion. However, unlike the previous algorithm which involves a token-based synchronization scheme, this algorithm controls data replication on the basis of the *majority consensus approach*. It was shown through simulation experiments that the real-time performance of the new algorithm, in terms of the percentage of satisfied transaction deadlines, is better than that of the token-based algorithm under low levels of data conflicts. The implementation overhead of the algorithm is lower, however, the degree of concurrency provided is not as high as that allowed by the token-based algorithm.

Xiong et al. proposed a concurrency control protocol, specially designed for firm-deadline applications operating on replicated real-time databases [17]. The proposed protocol augments a non-real-time concurrency control protocol with a novel state-based conflict resolution method to fine-tune the real-time performance. Compared to locking-based real-time concurrency control protocols, the performance of the new protocol was shown to be the best in replicated environments for real-time applications. It was also shown that the relative performance of replica concurrency control algorithms in real-time environments can be significantly different from that in non-real-time environments.

Our work in this field involves investigation of the impact of storing multiple copies of data on satisfying timing constraints of RTDBS transactions [16]. A brief description of this work is provided in the following.

3. A REPLICATED REAL-TIME DATABASE SYSTEM MODEL

This section provides the model of a RTDBS model that we used in investigation of the effects of data replication on real-time performance of the system. In this model, a number of data sites are interconnected by a local communication network. Each data site contains a transaction manager, a resource manager, a message server, a scheduler, and a recovery manager.

Each transaction is characterized by a *criticalness* and a *deadline*. The criticalness of a transaction is an indication of its level of importance [3]. The most critical transactions are assigned the highest level. Transaction deadlines are *soft*: i.e., each transaction is executed to completion even if it misses its deadline. The transaction manager at the originating site of a transaction $T$ assigns a real-time priority to transaction $T$ based on its criticalness ($C_T$), deadline ($D_T$), and arrival time ($A_T$). The priority of transaction $T$ is determined by the following formula:

$$P_T = \frac{C_T}{D_T - A_T}$$
There is no globally shared memory in the system, and all sites communicate via message exchanges over the communication network. A message server at each site is responsible for sending/receiving messages to/from other sites.

Access requests for data items are ordered by the scheduler on the basis of the concurrency control protocol executed.

I/O and CPU services at each site are provided by the resource manager. I/O service is required for reading or updating data items, while CPU service is necessary for processing data items and communication messages. Both CPU and I/O queues are organized on the basis of real-time priorities, and preemptive-resume priority scheduling is used by the CPUs at each site.

3.1 DISTRIBUTED TRANSACTION EXECUTION

Each distributed transaction exists in the system in the form of a master process that executes at the originating site of the transaction and a number of cohort processes that execute at various sites where the the copies of required data items reside. The transaction manager is responsible for creating a master process for each new transaction and specifying the appropriate sites for the execution of the cohort processes of the transaction. A transaction can have at most one cohort at each data site. For each operation executed, a global data dictionary is referred to find out the locations of the data item referenced by the operation. The coordination of the execution of remote cohorts is provided by the master process through communicating with the transaction manager of each cohort’s site.

When a cohort completes its data access and processing requirements, it waits for the master process to initiate two-phase commit. The master process commits a transaction only if all the cohort processes of the transaction run to completion successfully, otherwise it aborts and later restarts the transaction. A restarted transaction accesses the same data items as before, and is executed with its original priority. The cohorts of the transaction are reinitialized at relevant data sites.

One-copy serializability in replicated database systems can be achieved by providing both concurrency control for the processed transactions and mutual consistency for the copies of a data item. In our model, mutual consistency of replicated data is achieved by using the read-one, write-all-available scheme [2]. Based on this scheme, a read operation on a data item can be performed on any available copy of the data. In order to execute a write operation of a transaction on a data item, each transaction cohort executing at an operational data site storing a copy the item is activated to perform the update on that copy.
3.2 DATA DISTRIBUTION

We used a data distribution model which provides a partial replication of the distributed database. The model enables us to execute the system at precisely specified levels\(^2\) of data replication. Each data item has exactly \(N\) copies in the distributed system, where \(1 \leq N \leq n\). Each data site can have at most one copy of a data item. The remote copies of a data item are uniformly distributed over the remote data sites; in other words, the remote sites for the copies of a data item are chosen randomly. If the average database size at a site is specified by \(db_{\text{size}}\),

\[
db_{\text{size}} = N * local_{\text{db}_{\text{size}}}
\]

where \(local_{\text{db}_{\text{size}}}\) represents the database size originated at each site. Note that \(N = 1\) and \(N = n\) correspond to the no-replication and full-replication cases, respectively.

3.3 RELIABILITY ISSUES

The distributed RTDBS model assumes that the data sites fail in accordance with an exponential distribution of inter-failure times. After a failure, a site stays halted during a repair period, again chosen from an exponential distribution. The means of the distributions are determined by the parameters \(m_{\text{tf}}\) (mean time between failures) and \(m_{\text{tr}}\) (mean time to repair). The recovery manager at each site is responsible for handling site failures and maintaining the necessary information for that purpose. The communication network, on the other hand, is assumed to provide reliable message delivery and is free of partitions. It is also assumed that the network has enough capacity to carry any number of messages at a given time, and each message is delivered within a finite amount of time.

The following subsections details the reliability issues considered in our distributed system model.

3.3.1 Availability. Availability of a system specifies when transactions can be executed [6]. It is intimately related to the replica control strategy used by the system. For the read-one, write-all-available strategy, availability can be defined as the fraction of time (or probability) for which at least one copy of a data item is available to be accessed by an operation [9]. This strategy provides a high level of availability, since the system can continue to operate when all but one site have failed. A transaction that issues an operation that fails is blocked until a copy of the requested data item becomes available.

\(^2\)The level of replication corresponds to the number of copies that exist for each data item.
One method to measure the availability of an executing system is to keep track of the total number of attempted and failed operations experienced over a long period of time. Availability in our model is defined by the following formula:

\[
\text{Availability} = \frac{\text{Total no. of successful (read and write) operations}}{\text{Total no. of (read and write) operation requests}}
\]

This formula is a convenient one to use in RTDBSs since both read and write availabilities are equally crucial to such systems, and thus they can be treated together.

### 3.3.2 Site Failure.
At a given time a site in our distributed system can be in any of three states: *operating*, *halted*, or *recovering*. A site is in the halted state if it has ceased to function due to a hardware or software failure. A site failure is modeled in a fail-stop manner; i.e., the site simply halts when it fails [11]. Following its repair, the site is transformed from the halted state to the recovering state and the recovery manager executes a predefined recovery procedure. A site that is operational or has been repaired is said to be in the operating state. Data items stored at a site are available only when the site is in the operating state.

A list of operating sites is maintained by the recovery manager at each site. The list is kept current by ‘up-state’ messages received from remote sites. An ‘up-state’ message is transmitted periodically by each operating site to all other sites. When the message has not been received from a site for a certain timeout period, the site is assumed to be down.

### 3.3.3 Site Recovery Strategy.
The recovery procedure at a site restores the database of that site to a consistent and up-to-date state. Our work does not simulate the details of site recovery; instead, it includes a simplified site recovery model which is sufficient for the purpose of estimating the impact of site failures on system performance.

The recovery manager at each site maintains a log for recovery purposes, which records the chronological sequence of write operations executed at that site. Whenever a write operation is performed by a transaction, a log record for that write is created before the database is modified. At the commit time of a transaction, a commit record is written in the log at each participating data site. In the case of a transaction abort, the log records stored for that transaction are simply discarded. The recovery manager of a recovering site first performs local recovery by using the local log. Then, it obtains the logs kept at operating sites to check whether any of its replicated data items were updated while the site was in the halted state. It then refreshes the values of updated items using the current values of the copies stored at operational sites.
4. SUMMARY OF PERFORMANCE EVALUATION RESULTS

Details of the replicated database system model described in the preceding section were captured in a simulation program. The performance metric used in the experiments was success-ratio; i.e., the fraction of transactions that satisfy their deadlines.

Different application types were considered in evaluating the effects of level of data replication on satisfying transaction deadlines. Each type considered is characterized by the fraction of update transactions processed and the distribution of accessed data items (local vs remote). In execution environments where queries predominate and the data items at all sites are accessed uniformly, increasing the level of replication helped transactions meet their deadlines. For the application types where the majority of processed transactions are of update type, having many data copies was not attractive. This result was due to the fact that the overhead of update synchronization among the multiple copies of updated data increases with each additional data copy. Concurrency control protocols which employ restarts in scheduling, exhibited better performance than blocking-based protocols in query-based application environments when the level of data replication was low.

With update-oriented applications blocking-based protocols outperformed restart-based protocols, leading to the result that the overhead of executing a blocking-based concurrency control protocol is less than that of a restart-based one when the update transactions dominate in the system. Blocking-based protocols become more preferable as the level of data replication increases, since the performance of restart-based protocols is affected more negatively by the increased overhead of multiple copy updates. Aborting a transaction becomes more expensive as the number of copies of the data items updated by the transaction increases. It was shown that aborting a lock-holding transaction should be considered only in the case that the transaction is at the early stages of its execution.

We also studied the impact of site failures on system performance under different system reliability levels. Investigating the effectiveness of data replication in preventing the effects of site failures, we observed that replication turns out to be more desirable as site failures become more frequent.

The results of our performance experiments led to the following general observation: the optimum number of data replicas to provide the best performance in RTDBSs depends upon the fraction of update operations required by the application, the distribution of accessed data items, and the reliability of data sites. In general, as few as 2 or 3 copies appeared to be a good choice under the parameter ranges explored.
5. SUMMARY AND FUTURE WORK

It might be desirable to replicate data in a distributed real-time database system (RTDBS) due to certain benefits provided, such as high data availability and potentially improved read performance. However, under certain conditions, the overhead of synchronizing multiple copy updates can become a considerable factor in determining performance. Our work on replication in RTDBSs aimed to identify the conditions under which data replication can help real-time transactions satisfy their timing constraints. This chapter provides a summary of this work as well as a brief overview of the recent research in the field. There is a requirement for the development of new techniques to control data replication and to enhance the availability of replicated RTDBSs.

References


This page intentionally left blank.
Chapter 17

REAL-TIME COMMIT PROCESSING

Jayant R. Haritsa
Database Systems Lab, Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore 560012, INDIA.
haritsa@dsl.serc.iisc.ernet.in

Krithi Ramamritham
Dept. of Computer Science and Engineering,
Indian Institute of Technology, Mumbai 400076, INDIA.
krithi@cse.iitb.ernet.in

Ramesh Gupta
Goldencom Technologies, Inc.
392 Acoma Way, Fremont, CA 94539, USA.
ramash@goldencom.com

1. INTRODUCTION

Many real-time database applications are inherently distributed in nature [24, 28]. These include the intelligent network services database described in [5] and the mobile telecommunication system discussed in [29]. More recent applications include the multitude of directory, data-feed and electronic commerce services that have become available on the World Wide Web.

An essential requirement for most distributed real-time database system (DRTDBS) applications is transaction atomicity – this is satisfied by implementing a transaction commit protocol. Commit protocols typically require exchange of multiple messages, in multiple phases, between the participating sites where the distributed transaction executed. In addition, several log records are generated, some of which have to be “forced”, that is, flushed to disk immediately in a synchronous manner. Due to this series of synchronous message and logging costs, commit processing can significantly increase the execution times of transactions [17, 7, 25]. This is especially problematic in the real-
time context since it has a direct adverse effect on the system’s ability to meet transaction timing constraints. Therefore, the choice of commit protocol is an important design decision for DRTDBS. Unfortunately, there is little research literature on this topic, making it difficult for DRTDBS designers to make informed choices. We address this lacuna here with a detailed quantitative evaluation whose results represent the first such work in the area.

Our study is carried out in the context of real-time applications with “firm-deadlines” [15] – for such applications, completing a transaction after its has expired is of no utility and may even be harmful. Therefore, transactions that miss their deadlines are considered to be worthless and are immediately “killed” – that is, aborted and permanently discarded from the RTDBS without being executed to completion. Accordingly, the performance metric is KillPercent, the steady-state percentage of killed transactions.1

2. PERFORMANCE FRAMEWORK

From a performance perspective, commit protocols can be compared with respect to the following issues:

1. **Effect on Normal Processing:** This refers to the extent to which the commit protocol affects the normal (no-failure) distributed transaction processing performance. That is, how expensive is it to provide atomicity using this protocol?

2. **Resilience to Failures:** When a failure occurs in a distributed system, ideally, transaction processing in the rest of the system should not be affected during the recovery of the failed component. With “blocking commit protocols”, failures can lead to transaction processing grinding to a halt (as explained in Section 3.), whereas “non-blocking protocols” prevent such disruptions by incurring additional messages and forced-log-writes.

3. **Speed of Recovery:** This refers to the time required for the database to be recovered to a consistent state when the failed site comes back up after a crash. That is, how long does it take before transaction processing can commence again in a recovering site?

Of the three issues highlighted above, the design emphasis of most commit protocols has been on the first two (effect on normal processing and resilience to failures) since they directly affect ongoing transaction processing. In comparison, the last issue (speed of recovery) appears less critical for two reasons: First, failure durations are usually orders of magnitude larger than recovery

---

1 Or, equivalently, the percentage of missed deadlines.
Real-Time Commit Processing

times. Second, failures are usually rare enough that we do not expect to see a
difference in average performance among the protocols because of one commit
protocol having a faster recovery time than the other. Based on this viewpoint,
our focus here is on the mechanisms required during normal (no-failure) op-
eration to provide for recoverability and resilience to failures, and not on the
post-failure recovery process.

3. TRADITIONAL DISTRIBUTED COMMIT
PROTOCOLS

We adopt the common “subtransaction model” [3] of distributed transaction
execution in our study. In this model, there is one process, called the master,
which is executed at the site where the transaction is submitted, and a set of
other processes, called cohorts, which execute on behalf of the transaction at
the various sites that are accessed by the transaction. Cohorts are created by
the master sending a STARTWORK message to the local transaction manager at
that site. This message includes the work to be done at that site and is passed
on to the cohort. Each cohort sends a WORKDONE message to the master after
it has completed its assigned data processing work, and the master initiates the
commit protocol (only) after it has received this message from all its cohorts.

For the above transaction execution model, a variety of commit protocols
have been devised, most of which are based on the classical two-phase commit
(2PC) protocol [10]. In this section, we briefly describe the 2PC protocol
and a few of its popular variants – complete descriptions are available in
[20, 7, 23]. For ease of exposition, the following notation is used in the sequel
– “SMALL CAPS FONT” for messages, “typewriter font” for log records,
and “sans serif font” for transaction states.

3.1 TWO PHASE COMMIT

In the 2PC protocol, the master initiates the first phase of the commit protocol
by sending PREPARE (to commit) messages in parallel to all its cohorts. Each
cohort that is ready to commit first force-writes a prepare log record to its
local stable storage and then sends a YES vote to the master. At this stage, the
cohort has entered a prepared state wherein it cannot unilaterally commit or
abort the transaction but has to wait for the final decision from the master. On
the other hand, each cohort that decides to abort force-writes an abort log
record and sends a NO vote to the master. Since a NO vote acts like a veto, the
cohort is permitted to unilaterally abort the transaction without waiting for the
decision from the master.

After the master receives votes from all its cohorts, the second phase of the
protocol is initiated. If all the votes are YES, the master moves to a committing
state by force-writing a commit log record and sending COMMIT messages
to all its cohorts. Each cohort, upon receiving the COMMIT message, moves to the committing state, force-writes a commit log record, and sends an ACK message to the master.

On the other hand, if the master receives even one NO vote, it moves to the aborting state by force-writing an abort log record and sends ABORT messages to those cohorts that are in the prepared state. These cohorts, after receiving the ABORT message, move to the aborting state, force-write an abort log record and send an ACK message to the master.

Finally, the master, after receiving ACKs from all the prepared cohorts, writes an end log record and then “forgets” the transaction (by removing from virtual memory all information associated with the transaction).

3.2 PRESUMED ABORT AND PRESUMED COMMIT

Two variants of the 2PC protocol called presumed abort (PA) and presumed commit (PC) were presented in [20]. These protocols try to reduce the message and logging overheads by requiring all participating cohorts to follow certain rules at failure recovery time. The protocols have been implemented in a number of database products and PA is, in fact, now part of the ISO-OSI and X/OPEN distributed transaction processing standards [7].

3.3 THREE PHASE COMMIT

A fundamental problem with all of the above protocols is that cohorts may become blocked in the event of a site failure and remain blocked until the failed site recovers. To address the blocking problem, a three phase commit (3PC) protocol was proposed in [23]. This protocol achieves a non-blocking capability by inserting an extra phase, called the “precommit phase”, in between the two phases of the 2PC protocol. In the precommit phase, a preliminary decision is reached regarding the fate of the transaction. The information made available to the participating sites as a result of this preliminary decision allows a global decision to be made despite a subsequent failure of the master site. Note, however, that the price of gaining non-blocking functionality is an increase in the communication and logging overheads since: (a) there is an extra round of message exchange between the master and the cohorts, and (b) both the master and the cohorts have to force-write additional log records in the precommit phase.

4. INADEQUACIES IN THE DRTDBS ENVIRONMENT

The commit protocols described in the previous section are designed for traditional database systems where transaction throughput or average response
Real-Time Commit Processing

Time is usually the primary performance metric. With respect to meeting (firm) real-time objectives, however, they fail on two related counts: First, by making prepared data inaccessible, they increase transaction blocking times and therefore have an adverse impact on the number of killed transactions. Second, prioritized scheduling policies are typically used in RTDBS to minimize the number of killed transactions. These commit protocols, however, do not take transaction priorities into account. This may result in high priority transactions being blocked by low priority transactions, a phenomenon known as priority inversion in the real-time literature [22]. Priority inversion can cause the affected high-priority transactions to miss their deadlines and is clearly undesirable.

Priority inversion is usually prevented by resolving all conflicts in favor of transactions with higher priorities. Removing priority inversion in the commit protocol, however, is not fully feasible. This is because, once a cohort reaches the prepared state, it has to retain all its update data locks until it receives the global decision from the master – this retention is fundamentally necessary to maintain atomicity. Therefore, if a high priority transaction requests access to a data item that is locked by a “prepared cohort” of lower priority, it is not possible to forcibly obtain access by preempting the low priority cohort. In this sense, the commit phase in a DRTDBS is inherently susceptible to priority inversion. More importantly, the priority inversion is not bounded since the time duration that a cohort is in the prepared state can be arbitrarily long (for example, due to network delays). If the inversion period is large, it may have a significant negative effect on performance.

It is important to note that this “prepared data blocking” is distinct from the “decision blocking” (because of failures) that was discussed in Section 3.. That is, in all the commit protocols, including 3PC, transactions can be affected by prepared data blocking. In fact, 3PC’s strategy for removing decision blocking increases the duration of prepared data blocking. Moreover, such data blocking occurs during normal processing whereas decision blocking only occurs during failure situations.

To address the above-mentioned drawbacks (prepared data inaccessibility and priority inversion) of the classical commit protocols, we have designed a new protocol called PROMPT. The PROMPT design is based on a specific semantics of firm deadlines in DRTDBS, defined in the following section – the description of PROMPT itself is deferred to Section 6..

5. FIRM DEADLINE SEMANTICS IN DRTDBS

The semantics of firm deadlines is that a transaction should be either committed before its deadline or be killed when the deadline expires. To implement this notion in a distributed RTDBS, ideally the master and all the cohorts of a successfully executed transaction should commit the transaction before the
deadline expires or all of them should abort immediately upon deadline expiry. In practice, however, it is impossible to provide such guarantees because of the arbitrary message delays and the possibility of failures [4]. To avoid inconsistencies in such cases, we define the firm deadline semantics in the distributed environment as follows:

**Definition:** A distributed firm-deadline real-time transaction is said to be committed if the master has reached the commit decision (that is, forced the commit log record to the disk) before the expiry of the deadline at its site. This definition applies irrespective of whether the cohorts have also received and recorded the commit decision by the deadline.

To ensure transaction atomicity with the above definition, we require prepared cohorts that receive the final decision after the local expiry of the deadline to still implement this decision. Note that this is consistent with the intuitive notion of firm deadlines since all that happens is that access to prepared data (at cohort’s site) is prevented even beyond the deadline until the decision is received by the cohort; other transactions which would normally expect the data to be released by the deadline only experience a delay. We expect that many real-time database applications, especially those related to electronic commerce (e.g., electronic auctions), will subscribe to these semantics.

Typically, the master is responsible for returning the results of a transaction to the invoker of the transaction. From the above discussion, it is clear that the semantics we prescribe are such that, if a transaction commits, its results will begin to be output before the deadline. Further, the problem of delayed access to data, even after the expiry of the deadline of the cohort holding these data items, applies primarily to the classical protocols – the effect is considerably reduced with PROMPT, as discussed in the following section.

6. **THE PROMPT REAL-TIME COMMIT PROTOCOL**

The main feature of our new PROMPT (Permits Reading Of Modified Prepared-data for Timeliness) commit protocol is that transactions requesting data items held by other transactions in the prepared state are allowed to access this data. That is, prepared cohorts lend their uncommitted data to concurrently executing transactions (without, of course, releasing the update locks). The mechanics of the interactions between such “lenders” and their associated “borrowers” are captured in the following three scenarios, only one of which will occur for each lending:

1. **Lender Receives Decision Before Borrower Completes Data Processing:** Here, the lending cohort receives its global decision before the borrowing cohort has completed its local data processing. If the global decision is to commit, the lending cohort completes in the normal fash-
ion. On the other hand, if the global decision is to abort, the lender is aborted in the normal fashion. In addition, the borrower is also aborted since it has utilized dirty data.

2. **Borrower Completes Data Processing Before Lender Receives Decision:** Here, the borrowing cohort completes its local data processing before the lending cohort has received its global decision. The borrower is now “put on the shelf”, that is, it is made to wait and not allowed to send a WORKDONE message to its master. This means that the borrower is not allowed to initiate the (commit-related) processing that could eventually lead to its reaching the prepared state. Instead, it has to wait until either the lender receives its global decision or its own deadline expires, whichever occurs earlier. In the former case, if the lender commits, the borrower is “taken off the shelf” (if it has no other “pending” lenders) and allowed to send its WORKDONE message, whereas if the lender aborts, the borrower is also aborted immediately since it has utilized dirty data (as in Scenario 1 above). In the latter case (deadline expiry), the borrower is killed in the normal manner.

3. **Borrower Aborts During Data Processing Before Lender Receives Decision:** Here, the borrowing cohort aborts in the course of its data processing (due to either a local problem, deadline expiry or receipt of an ABORT message from its master) before the lending cohort has received its global decision. In this situation, the borrower’s updates are undone and the lending is nullified.

In summary, the PROMPT protocol allows transactions to access uncommitted data held by prepared transactions in the “optimistic” belief that this data will eventually be committed. It uses this approach to mitigate the effects of both the data inaccessibility and the priority inversion problems that were identified earlier for traditional commit protocols (Section 4.).

We wish to clarify here that while the PROMPT design may superficially appear similar to that of optimistic concurrency control [16], it is actually quite different since updates are made in-place and not to copies or versions of the data; also, data is lent only by transactions that have completed their data processing and are in the prepared state.

---

1 A similar, but unrelated, strategy of allowing access to uncommitted data has also been used to improve real-time concurrency control performance [2].
6.1 ADDITIONAL REAL-TIME FEATURES OF PROMPT

To further improve its real-time performance, three additional features are included in the PROMPT protocol: Active Abort, Silent Kill and Healthy Lending. These features are described below.

6.1.1 Active Abort. In the basic 2PC protocol, cohorts are “passive” in that they inform the master of their status only upon explicit request by the master. This is acceptable in conventional distributed DBMS since, after a cohort has completed its data phase, there is no possibility of the cohort subsequently being aborted due to serializability considerations (assuming a locking-based concurrency control mechanism).

In a DRTDBS, however, a cohort which is not yet in its commit phase can be aborted due to conflicts with higher priority transactions. Therefore, it may be better for an aborting cohort to immediately inform the master so that the abort of the transaction at the sibling sites can be done earlier. Early restarts are beneficial in two ways: First, they provide more time for the restarted transaction to complete before its deadline. Second, they minimize the wastage of both logical and physical system resources. Accordingly, cohorts in PROMPT follow an “active abort” policy – they inform the master as soon as they decide to abort locally; the subsequent abort process implemented by the master is the same as that followed in the traditional passive environment.

6.1.2 Silent Kill. For a transaction that is killed before the master enters its commit phase, there is no need for the master to invoke the abort protocol since the cohorts of the transaction can independently realize the missing of the deadline (assuming global clock synchronization)\(^3\). Eliminating this round of messages may help to save system resources. Therefore, in PROMPT, aborts due to deadline misses that occur before the master has initiated the commit protocol are implemented “silently” without requiring any communication between the master and the cohort.

6.1.3 Healthy Lending. A committing transaction that is close to its deadline may be killed due to deadline expiry before its commit processing is over. Lendings by such transactions must be avoided since they are likely to result in the aborts of all the associated borrowers. To address this issue, we have added a feature to PROMPT whereby only “healthy” transactions, that is, transactions whose deadlines are not very close, are allowed to lend

\(^3\)Our firm deadline semantics ensure that skew in clock synchronization, if any, only affects performance, but not atomicity. Further, for minor skews, the performance impact is expected to be marginal.
their prepared data. This is realized in the following manner: A health factor, $HFT$, is associated with each transaction $T$ and a transaction is allowed to lend its data only if its health factor is greater than a (system-specified) minimum value $MinHF$. The health factor is computed at the point of time when the master is ready to send the PREPARE messages and is defined to be the ratio $TimeLeft/MinTime$, where $TimeLeft$ is the time left until the transaction’s deadline, and $MinTime$ is the minimum time required for commit processing (recall that a minimum of two messages and one force-write need to be processed before the master can take a decision).

The success of the above scheme is directly dependent on the threshold health factor $MinHF$ — set too conservatively (large values), it will turn off the borrowing feature to a large extent, thus effectively reducing PROMPT to standard 2PC; on the other hand, set too aggressively (small values), it will fail to stop several lenders that will eventually abort. In our experiments, we consider a range of values for $MinHF$ to determine the best choices.

An important point to note here is that the health factor is not used to decide the fate of the transaction but merely to decide whether the transaction can lend its data. Thus, erroneous estimates about the message processing times and log force-write times only affect the extent to which the optimistic feature of PROMPT is used, as explained above.

6.2 ABORTS IN PROMPT DO NOT ARBITRARILY CASCADE

An important point to note here is that PROMPT’s policy of using uncommitted data is generally not recommended in traditional database systems since this can potentially lead to the well-known problem of cascading aborts [1] if the transaction whose dirty data has been accessed is later aborted. However, for the PROMPT protocol, this problem is alleviated due to the following two reasons:

First, the lending transaction is typically expected to commit because: (a) The lending cohort is in the prepared state and cannot be aborted due to local data conflicts, and (b) The sibling cohorts are also expected to eventually vote to commit since they have survived all their data conflicts that occurred prior to the initiation of the commit protocol (given our Active Abort policy).

The only situation where a lending cohort will finally abort is if (a) the deadline expires at the master’s node before the master reaches a decision, or (b) a sibling cohort votes NO. The latter case can happen only if the ABORT message sent by the sibling cohort and the PREPARE message sent by the master to the sibling cohort “cross each other” on the network. As the time during

---

*We assume a locking-based concurrency control mechanism.*
which a message is in transit is usually small compared to the transaction execution times, these situations are unlikely to occur frequently. Hence, a lending transaction is typically expected to commit.\footnote{Aborts could also occur after receiving the PREPARE message due to reasons such as violation of integrity constraints but our experiments have shown that unless the frequency of such aborts is unrealistically high, the improvement offered by PROMPT continues to be significant.}

Second, even if the lending transaction does eventually abort, it only results in the abort of the immediate borrower and does not cascade beyond this point (since the borrower is not in the prepared state, the only situation in which uncommitted data can be accessed). That is, a borrower cannot simultaneously be a lender. Therefore, the abort chain is bounded and is of length one. Of course, if an aborting lender has lent to multiple borrowers, then all of them will be aborted, but the length of each abort chain is limited to one. In short, PROMPT implements a controlled lending policy.

6.3 SYSTEM INTEGRATION

To incorporate the PROMPT protocol in a DRTDBS, a few modifications are required to the standard transaction and lock manager programs. The important point to note about these modifications, which are described in detail in [14], is that they are local to each site and do not require inter-site communication or coordination. Further, they do not appear difficult to incorporate in current database system software. In fact, some of them are already provided in current DBMS – for example, the high-performance industrial-strength ARIES recovery system [6] implements operation logging to support semantically rich lock modes that permit updating of uncommitted data. Moreover, as shown by our experiments, the performance benefits that can be derived from these changes more than compensate for the small amount of run-time overheads entailed by the above modifications and the effort needed to implement them.

6.4 INTEGRATING PROMPT WITH OTHER 2PC OPTIMIZATIONS

A particularly attractive feature of PROMPT is that it can be integrated with many of the other optimizations suggested for 2PC. For example, Presumed Commit and Presumed Abort (Section 3.) can be directly added as a useful supplement to reduce processing overheads. Moreover, the integration may often be synergistic in that PROMPT may retain the good features of the added optimization and simultaneously minimize its drawbacks. This is the case, for example, when PROMPT is combined with 3PC: In its attempt to prevent decision blocking, 3PC suffers an increase in the prepared data blocking period, but this drawback is reduced by PROMPT’s lending feature.
Among additional optimizations [7], PROMPT can be integrated in a straightforward manner with Read-only (one phase commit for read-only transactions), Long Locks (cohorts piggyback their commit acknowledgments onto subsequent messages to reduce network traffic), and Shared Logs (cohorts that execute at the same site as their master share the same log and therefore do not need to force-write their log records). Further, PROMPT is especially attractive to integrate with protocols such as Group Commit [7] (forced writes are batched together to save on disk I/O) and linear 2PC [10] (message overheads are reduced by ordering the sites in a linear chain for communication purposes). This is because these optimizations extend, like 3PC, the period during which data is held in the prepared state, thereby allowing PROMPT to play a greater role in improving system performance.

7. THE PIC PROTOCOL

A plausible alternative approach to that of PROMPT is to use the priority inheritance (PI) mechanism [22]. Here, low priority transactions that block a high priority transaction inherit the priority of the high priority transaction. The expectation is that the blocking time of the high priority transaction will be reduced since the low priority transactions will now execute faster and therefore release their locks earlier.

A positive feature of the PI approach is that it does not run the risk of transaction aborts, unlike the lending approach. Further, a study of PI in the context of (centralized) transaction concurrency control [8] suggested that priority inheritance is useful only if it occurs towards the end of the low priority transaction’s lifetime. This seems to fit well with handling priority inversion during commit processing since this stage occurs at the end of transaction execution.

Motivated by these considerations we now describe Priority Inheritance Commit (PIC), a real-time commit protocol based on the PI approach. In the PIC protocol, when a high priority transaction is blocked due to the data locked by a low priority cohort in the prepared state, the latter inherits the priority of the former to expedite its commit processing. To propagate this inherited priority to the master and the sibling cohorts, the priority inheriting cohort sends a PRIORITY-INHERIT message to the master. The master, in turn, sends this message to all other cohorts. After the master or a cohort receives a PRIORITY-INHERIT message, all further processing related to the transaction at that site (processing of the messages, writing log records, etc.) is carried out at the inherited priority.
8. PERFORMANCE RESULTS

Using a detailed simulation model of a firm-deadline DRTDBS, we evaluated the performance of the various commit protocols described in the previous sections. Our experiments, which covered a variety of transaction workloads and system configurations, demonstrated the following:

1. Distributed commit processing can have considerably more effect than distributed data processing on the real-time performance, especially on systems with slow network interfaces. This highlights the need for developing commit protocols tuned to the real-time domain.

2. The classical commit protocols generally perform poorly in the real-time environment due to their passive nature and due to preventing access to data held by cohorts in the prepared state.

3. Our new protocol, PROMPT, provides significantly improved performance over the standard algorithms. Its good performance is attained primarily due to its optimistic borrowing of uncommitted data and Active Abort policy. The optimistic access significantly reduces the effect of priority inversion which is inevitable in the prepared state. Supporting statistics showed that PROMPT’s optimism about uncommitted data is justified, especially under normal loads. The other optimizations of Silent Kill and Presumed Commit/Abort, however, had comparatively little beneficial effect.

4. By appropriately setting the $MinHF$ parameter, most of the unhealthy lendings in PROMPT can be eliminated. In particular, a threshold of $MinHF = 1$ provided good performance in all the scenarios considered in our study.

5. The gap between PROMPT’s performance and that of the standard algorithms widens with increased data contention due to the corresponding increase in borrowing opportunities. This feature is especially important since while resource contention can usually be addressed by purchasing more and/or faster resources, there do not exist any equally simple mechanisms to reduce data contention.

6. Experiments combining PROMPT with 3PC indicate that the nonblocking functionality can be obtained in the real-time environment at a relatively modest cost in normal processing performance. This is especially encouraging given the high desirability of the nonblocking feature in the real-time environment.

7. PIC’s performance is virtually identical to that of 2PC. The reason is that priority inheritance comes into play only when a high priority transaction
is blocked by a low priority prepared cohort, which means that this cohort has already sent the yes vote to its master. Since it takes two message delays for dissemination of the priority inheritance information to the sibling cohorts, PIC expedites at most the processing of only the decision message. Further, even the minor advantage that may be obtained by PIC is partially offset by the extra overheads involved in processing the priority inheritance information messages.

8. Finally, the difference between the performance of PROMPT (with MinHF = 1) and that of Shadow-PROMPT— an artificially constructed protocol, based on the “shadow transaction” approach of [2], that represents the best on-line usage of the optimistic lending approach – never exceeded two percent.

In summary, we suggest that DRTDBS designers may find the PROMPT protocol to be a good choice for high-performance real-time distributed commit processing. Viewed in toto, PROMPT is a portable (can be used with many other optimizations), practical (easy to implement), high-performance (substantially improves real-time performance) and efficient (makes good use of the lending approach) distributed commit protocol.

9. RELATED WORK

The material discussed in this chapter is based on our earlier papers [11, 12, 13, 14]. Apart from this, the only other work that we are aware of on DRTDBS commit processing is [4, 9, 29]. In this section, we briefly summarize these papers and contrast them with our study.

A centralized timed 2PC protocol is described in [4] that guarantees that the fate of a transaction (commit or abort) is known to all the cohorts before the expiry of the deadline when there are no processor, communication or clock faults. In case of faults, however, it is not possible to provide such guarantees, and an exception state is allowed which indicates the violation of the deadline. Further, the protocol assumes that it is possible for the DRTDBS to guarantee allocation of resources for a duration of time within a given time interval. Finally, the protocol is predicated upon the knowledge of worst-case communication delays.

Our work is different from [4] in the following respects: First, their deadline semantics are different from ours in that even if the coordinator of a transaction is able to reach the decision by the deadline, but it is not possible to convey the decision to all the cohorts by the deadline, the transaction is killed. Second, we do not assume any guarantees provided by the system for the services offered whereas such guarantees are fundamental to the design of their protocol. Note that in a dynamic prioritized system such guarantees are difficult to provide.
and, further, are generally not recommended since it requires pre-allocation of resources, thereby running the risk of priority inversion.

A common theme of allowing individual sites to unilaterally commit is used in [9, 29] — the idea is that unilateral commitment results in greater timeliness of actions. If it is later found that the decision is not consistent globally, “compensation” transactions are executed to rectify the errors. While the compensation-based approach certainly appears to have the potential to improve timeliness, there are quite a few practical difficulties. First, the standard notion of transaction atomicity is not supported — instead, a “relaxed” notion of atomicity [18] is provided. Second, the design of a compensating transaction is an application-specific task since it is based on the application semantics. Third, the compensation transactions need to be designed in advance so that they can be executed as soon as errors are detected — this means that the transaction workload must be fully characterized a priori. Finally, from a performance viewpoint also, there are some difficulties: (a) The execution of compensation transactions imposes an additional burden on the system; (b) It is not clear how the database system should schedule compensation transactions relative to normal transactions.

10. SUMMARY AND FUTURE WORK

In this chapter, we have attempted to characterize the state-of-the-art with regard to real-time commit processing. The few papers on this topic require fundamental alterations of the standard distributed DBMS framework. We have instead taken a different approach of achieving high-performance by incorporating novel protocol features.

We first precisely defined the process of transaction commitment and the conditions under which a transaction is said to miss its deadline in a distributed firm real-time setting. Subsequently, we evaluated the performance of a variety of standard commit protocols, as well as PROMPT, a new protocol that is designed specifically for the real-time environment and includes features such as controlled optimistic access to uncommitted data, Active Abort, Silent Kill and Healthy-Lending. Our performance results showed that PROMPT provides significantly improved performance over the classical commit protocols, and makes extremely efficient use of the lending premise. Finally, we also evaluated the priority inheritance approach to addressing the priority inversion problem associated with prepared data, but found it to be inherently unsuitable for the distributed environment.

We now outline a variety of issues that appear appropriate for further research: First, in all our experiments, update-oriented transaction workloads were modeled. It would be instructive to evaluate the performance for different mixes of read-only and update-oriented transaction workloads. Further, only
a two-level transaction tree was considered in our work. The effects of the commit protocols on the performance of a system with “deep-transactions” (tree of processes architecture [19]) needs to be also evaluated.

Second, we assumed that all transactions have the same “criticality” or “value” [26], and therefore the performance goal was to minimize the number of killed transactions. An important open area is the characterization of commit protocol behavior for workloads where transactions have varying values, and the goal of the RTDBS is to maximize the total value of the completed transactions. These results could also be extended to “soft-deadline” applications, where there is a diminishing residual value for completing transactions even after their deadlines have expired.

Finally, our study was limited to commit protocols operating in two or three phases. A variety of one-phase protocols have also been proposed in the literature. In these protocols, cohorts enter the prepared state at the time of sending the WORKDONE message itself, thereby eliminating the voting phase and retaining only the decision phase. Examples of one-phase protocols include the Unsolicited Vote [27] protocol used in the Distributed Ingres database system and the Early Prepare [21] protocol.

The main attraction of one-phase protocols is that they significantly reduce the overheads of commit processing due to eliminating an entire phase. This reduction makes them especially attractive in the real-time environment. On the flip side, however, they significantly increase the amount of priority inversion due to the fact that cohorts are in a prepared state for extended periods of time. Therefore, a thorough evaluation needs to be made to assess the real-time capabilities of these protocols.

Acknowledgements Ramesh Gupta worked on this project when he was a graduate student at the Database Systems Lab, Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, India. S. Seshadri of Indian Institute of Technology, Mumbai, India, participated in the initial research efforts. Jayant Haritsa’s work was supported in part by research grants from the Dept. of Science and Technology and the Dept. of Bio-technology, Govt. of India. Krithi Ramamritham’s work was supported in part by the US National Science Foundation Grant #IIRL-9619588.

References


Chapter 18

**MOBILE DISTRIBUTED REAL-TIME DATABASE SYSTEMS**

Kam-Yiu Lam  
*Department of Computer Science*  
*City University of Hong Kong, Kowloon, Hong Kong*  
cskylam@cityu.edu.hk

Tei-Wei Kuo  
*Department of Computer Science and Information Engineering*  
*National Taiwan University*  
ktw@csie.ntu.edu.tw

1. INTRODUCTION

Recent advances in mobile communication technology have made mobile computing a reality [5, 6, 12]. Various novel mobile computing applications, such as tele-medicine systems, real-time traffic information and navigation systems, and mobile Internet stock trading systems, are emerging as mobile users require instant access to information using their palmtops, personal digital assistant (PDA) and notebook computers not matter where they are. The realization of "instant" information access over a mobile network relies on real-time processing of transactions. As a result, the research on concurrency control and transaction scheduling for mobile distributed real-time database systems (MDRTDBS) is receiving growing attention in recent years [2, 8, 15, 17].

Although many researchers, e.g., [1, 3, 13, 16], have done excellent research in concurrency control and transaction scheduling for single-site and distributed RTDBS, not much work has been done for MDRTDBS. Owing to the intrinsic limitations of mobile computing systems, such as limited bandwidth and frequent disconnection, the efficient techniques for distributed real-time database systems (DRTDBS) which are supported by wired networks, may not be suitable and useful to MDRTDBS. Compared to wired networks, mobile networks are much slower, unreliable, and unpredictable. The mobility of clients affects
the distribution of workload in the system. Disconnection between the mobile clients and the base stations is common [5]. These unique features of the systems can seriously affect the probability of data conflicts and deadline missing probability. Furthermore, overheads in resolving data conflicts will also be much higher. [15].

In this chapter, we discuss the issues of concurrency control and transaction scheduling for MDRTDBS which is supporting soft or firm real-time transactions. An important consideration in the design of the strategies for transaction processing and concurrency control is how to reduce the impact of the unreliable network on the performance of the system.

2. SYSTEM AND TRANSACTION MODEL

2.1 SYSTEM ARCHITECTURE

Basically, a MDRTDBS consists of four main components: mobile clients (MCs), base stations, mobile network, and main terminal switching office (MTSO) [9], as shown in Figure 18.1. The mobile network is assumed to be a low bandwidth cellular network such as the GSM network. The entire service area is divided into a number of connected cell sites. Within each cell site, there is a base station, which is augmented with a wireless interface to communicate with the MCs within its cell site.

![System Architecture of a MDRTDBS](image)

*Figure 18.1  System Architecture of a MDRTDBS*
The base stations at different cell sites are connected to the MTSO by a point-to-point wired network. Thus, the communications between the base stations and the MTSO are much more efficient and reliable than the communications between the base stations and the mobile clients. The MTSO is responsible for active call information maintenance, performance of handoff procedure, channel allocation and message routing. Attached to each base station is a real-time database system containing a local database which may be accessed by transactions issued by the MCs within the cell or from other cells via the MTSO.

An MC may move around within the same cell or cross the cell border into another cell. Periodically, it sends a location signal to its base station through an up-link channel. When an MC is crossing the cell border, the MTSO will perform a handoff procedure to assign the MC to the new base station.

The transactions generated from MCs may need to access databases located at several base stations. The MC first issues a call request to the base station of its current cell. A channel is granted to the MC after the completion of a setup procedure. Since the number of channels between a base station and its MCs is limited, it is possible that the channel request may be refused due to unavailability of free channels. Due to channel contention and slow (and unreliable) communication, the time required to establish a channel is unpredictable. Once a channel has been established, the transaction will be sent out through the RF transmitter from the MC to the base station. Due to noise and interference, the signal, which carries data, may be corrupted while it is being transmitted. In this case, the data will be re-transmitted. If the transmission of signals is corrupted consistently after several times, a disconnection may have occurred. For that case, the transaction may need to wait until a new channel is granted before it can proceed. Because of high error rate and non-stability of signal transmission, the effective data transmission rate is also unpredictable.

2.2 DATABASE AND TRANSACTION MODELS

The entire database is partitioned into local databases and distributed at different base stations. The databases consist of two types of data items: temporal and non-temporal [14, 18]. Transactions from the MCs are assumed to be simple flat transactions. Each transaction consists of a set of read and write operations. They are associated with a soft or firm deadline. In between the operations of a transaction, control statements may be defined to control the logic flow of the transaction. Examples of such application systems are jockey-club betting systems, Internet programmed stock trading systems, traffic navigation and information systems, etc. Each transaction is given a criticality. The priority of a transaction is then derived based on its deadline and criticality.
It is assumed that the EDF algorithm is used for scheduling the transactions in using the CPU. Operations may access data objects residing at different base stations. Thus, a transaction may have several processes, called transaction processes, at different base stations for its execution. When an operation of a transaction accesses a data object residing at another base station, the operation will be routed to the base station via the MTSO and a new transaction process will be created if there is no process at that site for the transaction. When all the operations of a transaction have been processed, a commit protocol will be performed to ensure the failure atomicity of the transaction.

3. CONCURRENCY CONTROL PROTOCOL FOR MDRTDBS

Concurrency control is important to ensure consistency of database and data items observed by transactions. Due to the unique properties of a MDRTDBS, in the design of concurrency control protocols for MDRTDBS, there are two important considerations:

1. how to minimize the cost and overheads for resolving data conflicts; and
2. how to minimize the impact of mobile network on the performance of the protocol.

In the last decade, many efficient real-time concurrency control protocols have been proposed. Usually, they use transaction restart or priority inheritance to resolve the problem of priority inversion. However, restarting a transaction is highly expensive in a mobile environment due to the slow and unreliable mobile network. Although priority inheritance [4] is effective in managing the priority inversion problem in single-site RTDBS, a direct application of it may not be effective in a MDRTDBS as it may take a long time before the priority of a transaction is "inherited". Furthermore, deadlock is possible when priority inheritance is used. Deadlock is highly undesirable to real-time systems, especially in a distributed environment. The detection and resolution of a deadlock in a distributed environment may also consume a lot of resources. Thus, a deadlock-free protocol is highly desirable. Although the real-time optimistic concurrency control protocols have been shown to give a good performance in single-site RTDBS [3], they may not be suitable to MDRTDBS. The validation test required in optimistic concurrency control protocols can be very complex in a distributed environment [10] especially in mobile networks.

In the following, we introduce a real-time concurrency control protocol, called Distributed Higher Priority Two Phase Locking [1], for MDRTDBS. Although the basic methods used to resolve data conflicts are still based on transaction restart and priority inheritance, the protocol consists of two main features which are specially design for mobile environment. Firstly, it adopts a
state conscious method to resolve data conflicts, e.g., the method used to resolve a data conflict depends on the status of conflicting transactions. Secondly, methods are included to reduce the cost of transaction restarts and prevent the possibility of deadlock when priority inheritance is used.

3.1 THE PROTOCOL

The definition of the DHP-2PL is as follows, where $T_r$ and $T_h$ are the lock-requesting transaction and the lock-holding transaction, respectively:

\[
\text{LockConflcit} \ (T_r, T_h)
\]

\[
\begin{align*}
\text{Begin} & \\
\quad & \text{If Priority}(T_r) > \text{Priority}(T_h) \\
\quad & \text{If } T_h \text{ is not committing} \\
\quad & \text{If } T_h \text{ is a local transaction} \\
\quad & \quad \text{Restart } T_h \text{ locally} \\
\quad & \text{Else} \\
\quad & \quad \text{Restart } T_h \text{ globally} \\
\quad & \text{Endif} \\
\quad & \text{Else} \\
\quad & \quad \text{Block } T_r \text{ until } T_h \text{ releases the lock} \\
\quad & \quad \text{Priority}(T_h) := \text{Priority}(T_h) + \text{fixed priority level} \\
\quad & \text{Endif} \\
\quad & \text{Else} \\
\quad & \quad \text{Block } T_r \text{ until } T_h \text{ releases the lock} \\
\quad & \text{Endif} \\
\text{End}
\]

A transaction is local if it only accesses data objects resided at one base station. Otherwise, it is a global transaction. Restarting a local transaction is simply done by restarting the transaction process at the conflicting base station. To restart a global transaction, restart messages are sent to the base stations where some operations of the global transactions are executing or have executed. Global restart takes a much longer time and requires a much higher overhead. Thus, the number of transaction restarts, especially a global one, should be minimized and only non-committing transactions will be restarted. Committing transactions are allowed to hold locks until they have finished the commit procedure even though higher-priority transactions are requesting the locks. Although this approach may create the priority inversion problem, the blocking time of the higher-priority transaction will not be long if the committing transaction is assigned a sufficiently high priority by using priority inheritance. The priority of the committing transaction will be raised up by two factors. Firstly, its priority will be at least as high as the highest priority of all
of its blocked transactions, and, secondly, a fixed priority level should be added to its priority to make it even higher than all other executing transactions. The purpose is to finish the committing transaction as soon as possible. The time required to raise the priority of a transaction should be short, as the conflicting processes are located at the same site. No deadlock is possible for the priority raising of any committing transaction. It is because the committing transaction will not be blocked by any other executing transactions, as the committing transaction will not make any lock request during its commitment.

A common characteristic of mobile network is that disconnection between a mobile client and its base station is frequent. In processing a transaction, the control of a transaction may flow between its processes at the base stations and the process at its originating mobile client. In case a disconnection occurs while a transaction is locating at the mobile client (the control flow of the transaction is at the mobile client), the impact of the disconnection on the system performance can be very serious due to chain of blocking. It does not only greatly increase the deadline missing probability of the disconnected transaction, other transactions, which are directly or transitively blocked by the disconnected transaction, will also be affected. The result may be fruitless blocking which is a situation where a blocked transaction is finally aborted due to deadline missing. In the protocol, the problem of fruitless blocking may occur when a lower-priority transaction is blocked by a higher-priority transaction, which is a disconnected transaction. To minimize the impact of disconnection and the probability of fruitless blocking due to disconnection, we may use a cautious waiting scheme in which a higher-priority transaction is restarted by a lower-priority transaction due to data conflict if the higher-priority transaction is suspected to be a disconnected transaction.

Let each executing transaction process be associated with a location indicator. When a transaction is at the mobile client or it is waiting to move back to the mobile client, the location indicator of its processes will be set as "mobile client". Otherwise, it is set as "base station". When the location indicator of a transaction process is "mobile client", the transaction is vulnerable to disconnection. The following summarizes how the cautious waiting scheme is incorporated into the DHP-2PL:

If (the priority of the lock-requester > the priority of the lock-holder)
    and (the lock-holder is not committing)
        Restart the lock-holder (globally or locally, depending on the type of the transaction);
Else
    If the location indicator of the lock-holder is "mobile client"
        If the time already spent at the client side > threshold
            Ping the mobile client where the lock-holder is residing;
        /* The base station sends a message to the mobile client to
test if the mobile client is disconnected or not. */
If no response from the mobile client
    Restart lock-holder;
Else
    Block the lock-requester;
    /* repeat the checking after another threshold */
    Endif
Else
    Block the lock-requester;
    /* the checking will be performed again when the time 
    already spent at the client side is greater 
    than the threshold value */
    Endif
Else
    Block the lock-requester;
Endif
Endif

The \textit{threshold} is a tuning parameter. It is a function of the average performance of the mobile network under normal situations. If a transaction has been at the mobile client for a long time, e.g., greater than the threshold value, and its base station cannot communicate with the mobile client currently, disconnection is assumed. The lock-holding transaction will be restarted even though its priority is higher than the priority of the lock-requesting transaction. Although restarting the lock-holding transaction will make it have a high probability of missing deadline, the restart will not significantly affect the system performance, as the lock-holding transaction is likely to miss its deadline due to disconnection. On the other hand, restarting the lock-holding transaction can increase the chance of meeting the deadline of the lock-requesting transaction. Otherwise, it is highly possible that both of them will miss their deadlines. Note that simply making the assumption of disconnection based on pinging the mobile client may not be sufficient as mobile networks are subjected to different transient communication failures, which are much less harmful than disconnection. A transient communication failure will only last for a very short time and can usually be solved by data re-transmission.

4. STRATEGY FOR PROCESSING OF TRANSACTIONS IN MDRTDBS

Although the DHP-2PL can significantly reduce the cost for resolving data conflicts, the cost of transaction restart can still be substantial if conventional transaction processing strategies, e.g., query shipping or data shipping, are used
to process the transactions. In this section, we discuss a new strategy, called transaction shipping to process the transactions from mobile clients. The goal of the transaction shipping approach is to further reduce the impact of a mobile network on the performance of the DHP-2PL and to further improve the system performance.

4.1 TRANSACTION SHIPPING APPROACH

Conventional methods designed for processing transactions in distributed systems supported by wired network, such as data shipping or query shipping, may not suitable to the systems designed on mobile network as the communication overhead can be very high. This is highly undesirable especially when it is combined with a restart based concurrency control protocol. To reduce the overhead, we may "ship" the entire transaction to the database server (base station) for processing instead of shipping every operation or data request to the database server. The new method is called transaction shipping. Although the idea of shipping the whole transaction is simple, there exist many practical problems when it is applied to a MDRTDBS such as how to identify the execution path and the required data objects of a transaction before its execution, and how to deal with the dynamic properties in transaction execution. In the transaction shipping approach, the problems are resolved by a pre-analysis scheme in which the characteristics of a transaction will be defined and estimated before its start of execution. The practicality of the pre-analysis comes from the observation that the behavior of many real-time transactions is more predictable comparing to the transactions in conventional database systems. To deal with the dynamic properties of the real-time transactions, a data pre-fetching mechanism is included in the transaction shipping approach to reduce the cost of incorrect prediction in the pre-analysis.

4.1.1 Transaction Predictability. Transactions in a real-time database application can be classified into different types based on their pre-defined behavior and characteristics. Different transaction types will have different pre-defined behavior and critically [7]. For example, in a medical information system, periodic hard real-time transactions are used for monitoring the physical status of a critical patient using various sensor devices, such as blood pressure, heart beat rate, and body temperature. In some real-time database applications, e.g., programmed stock trading, although the transaction arrival pattern may be sporadic, their data requirements can be predicted with a high accuracy. For example, in a programmed stock trading, each investor may have a pre-defined investment plan, e.g. what their interested stocks are and how to make the trading analysis under different conditions. In a traffic navigation system, the physical connections of the roads are pre-defined. When searching the best path
to a destination from the current position based on the current road conditions, the set of roads to be search is also pre-defined.

4.1.2 Pre-analysis Phase. In the transaction shipping approach, the execution of a transaction is divided into two phases: the pre-analysis phase and the execution phase. Figure 18.2 shows the transaction architecture when it is processed under the transaction shipping approach. Once a transaction is initiated at a mobile client, a coordinator process, called master coordinator, will be created at the mobile client. Before shipping a transaction to the base station, the system will perform a pre-analysis on the transaction to derive its characteristics, e.g., what the operations of the transaction are and what the execution path of the transaction is.

![Figure 18.2 Transaction Architecture under the Transaction Shipping Approach](image)

The pre-analysis of a transaction consists of two stages. In the first stage, the set of operations in the transaction will be identified. It is usually not difficult to identify the operations in a transaction. For example, if SQL statements are used to access the database, the SELECT statements are read operations and the INSERT statements are write operations. At this stage, it may not be necessary to identify the set of data objects required by the operations. Actually, it may not be easily done at the mobile client as it only contains limited information about the database system and the location of the data objects in the system. The
required data objects of an operation will be determined while the transaction is executed at the base stations.

In the second phase, the execution path of the transaction, e.g., the precedence relationships of the operations, will be determined. However, for some transactions, the whole execution path may only be determined until the data objects required by the transactions have been identified. For example, some conditional statements are based on the values of the data objects. For such transactions, the pre-analysis may identify the transaction type first and then make the prediction based on the pre-defined characteristics of that transaction type.

After the completion of the pre-analysis, a signature of the transaction will be created. A signature transaction \( S_i \) for transaction \( T_i \) consists of a 4-tuple:

\[
S_i = (O_i, D_i, C_i, <_i)
\]

where \( D_i \), \( C_i \), and \( <_i \) are the deadline of \( T_i \), a subset of the operations in \( T_i \), and the partial order relationship among operations in \( O_i \), respectively. Note that \( <_i \) defines the precedence relationship among the operations. If \( Op_j <_i Op_k \), then \( Op_k \) can start its execution only after the completion of \( Op_j \), where \( Op_j \) and \( Op_k \) are operations of \( T_i \).

### 4.1.3 Execution Phase

The signature transaction is forwarded to the base station of the MC through the mobile network. Once the server at the base station receives the transaction signature, it will create a process, called image coordinator, for the transaction. The image coordinator will take over the job from the master coordinator to process the transaction. Other transaction processes for the executions of the transaction will be created at other base stations if the operations of the transaction require to access the data objects located at that base stations.

The benefit of defining an image coordinator at the base station is that the connection between base stations is much better than the connection between the MC's and their base stations. Thus, putting the image coordinators at the base stations can facilitate the management of the transactions and improve the performance of the atomic commitment protocol. Whenever a transaction has to be restarted, all its processes (excluding the master coordinator and the image coordinator) will be destroyed after the completion of undo operations. The image coordinator is responsible for restarting the transaction from its beginning if its deadline has not been missed.

### 4.1.4 Data Pre-fetching Mechanism

Although the transactions in real-time database applications are more well-defined, they may still have some dynamic properties. As a result, the pre-analysis of a transaction may need to
be re-done while it is executing. For that case, it has to go back to its originating MC. Before a transaction goes back to the mobile client, the system may pre-fetch the data objects possibly needed by the next operation of the transaction, and then ship the data objects with the transaction back to the mobile client. The purpose of the data pre-fetching mechanism is to process the next operation of the transaction at the mobile client so that the total number of communications between the mobile client and the base station can be reduced.

The assumption of the data pre-fetching mechanism is that data requirements of the next operation can be predicted with a high accuracy. Again, the validity of the assumption is based on the well-defined behavior of real-time transactions. Suppose the execution path of a transaction \( T \) is originally predicted to be Path 1 (operations: \( Op_1 \), \( Op_2 \) and then \( Op_3 \)) in the first pre-analysis as shown in Figure 18.3. After the completion of operation \( Op_2 \), it finds out that the original prediction is incorrect, and \( T \) has to go back to the mobile client to "re-define" the execution path which is either Path 2 or Path 3 according to the pre-defined behavior of the transaction. Before, \( T \) is shipped back to the mobile client, the system pre-fetches the data objects required by \( Op_4 \) and \( Op_5 \). Later at the mobile client, \( T \) can process the next operation \( Op_4 \) or \( Op_5 \). After that, \( T \) is sent to the base station to commit or continue its following operations. Note that under the data pre-fetching mechanism, \( T \) has to set a lock on the pre-fetched data objects. The purpose to is to ensure the serializability of transaction execution.

Note that the transaction shipping approach does not require the transmission of a large number of data objects to the clients as required in the data shipping approach. This is an attractive feature to MDRTDBS as the communication bandwidth is low. Although some data objects are required to be shipped with a transaction in the data pre-fetching mechanism, the data volume should be much smaller than that required in the data shipping approach as the data pre-fetching mechanism is required only when the predicted execution path is incorrect. Furthermore, only those data objects, which are required by the next operation, are shipped.

It is obvious that the performance gained from the pre-analysis depends on its accuracy. In the best case, one single communication between the MC and the base station is required for transaction execution. In the worst case, the number of communications between the mobile client and the base station for each transaction execution is equal to the number of operations in the transaction. If the system can make a good estimation in the pre-analysis, a lot of communications can be saved.
5. RELATED WORK

Although MDRTDBS are important, not much work has been done until recent years. The impact of mobile network on the performance of the Higher Priority Two Phase locking is studied in [11]. In [8], the various issues to process real-time transactions in a MDRTDBS are examined. Two execution strategies, called Execution Site is a Fixed Host (ESFH) and Execution Site is a Mobile Host (ESMH), are proposed for mobile transactions. The tradeoff of the two strategies is on the processing power of the mobile hosts. The ESFH is similar to the transaction shipping approach except that it uses the mobile host to handle the user requests instead of using an pre-analysis approach as used in the transaction shipping approach.

6. SUMMARY AND FUTURE WORK

The design of mobile distributed real-time database systems (MDRTDBS) is receiving growing interests. Due to the poor quality of services provided by a mobile network, it is not easy to meet the deadlines of transactions in a MDRTDBS. In this chapter, we define a detailed model for MDRTDBS, in which the mobility of the mobile clients and characteristics of the mobile network are modeled explicitly. We introduce a distributed real-time locking protocol, called Distributed High Priority Two Phase Locking (DHP-2PL) for
MDRTDBS, where the characteristics of the mobile network and the status of the conflicting transactions are considered in resolving the conflicts in data accesses. Then, we introduce a new strategy, called transaction shipping with a data pre-fetching scheme, to process the transactions to improve the system performance and to reduce the impact of the mobile network on the system performance.

Due to the advances in mobile communication technology and the need to manage real-time information over a mobile network, it is a trend to "mobilize" the database applications and include various real-time features into the mobile computing systems. Thus, the importance of MDRTDBS is growing. We can see that the technologies from the research in MDRTDBS will be useful in many related research areas such as data broadcast of real-time data, tracking of moving objects and the support of continuous queries.

Acknowledgements The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China [Project No. CityU 1076/00E].

References

vol. 42, no. 6, 1999.


PART VI

PROTOTYPES AND FUTURE DIRECTIONS
This page intentionally left blank.
Chapter 19

PROTOTYPES: PROGRAMMED STOCK TRADING

Brad Adelberg
Northwestern University

Ben Kao
University of Hong Kong

1. INTRODUCTION

We believe that the greatest growth potential for soft real-time databases is not as isolated monolithic databases but as components in open systems consisting of many heterogeneous databases. In such environments, the flexibility to deal with unpredictable situations and the ability to cooperate with other data sources (often non-real-time) is just as important as the guarantee of stringent timing constraints.

The focus of our research was to design a database system explicitly for heterogeneous environments, the STanford Real-time Information Processor (STRIP). STRIP, which runs on standard Posix Unix, is a soft real-time main memory database with special facilities for importing and exporting data as well as handling derived data. We chose program trading as the motivating application for the work. A program trading system monitors the prices of stocks or other financial instruments and looks for trading opportunities based on market inefficiencies or short term market models.

1.1 PROGRAM TRADING APPLICATION

In this section we describe the simplified program trading application (PTA) used to drive the STRIP research. In practice, program trading systems are custom built by each trading firm and their market models and trading algorithms are closely held secrets. Thus the example we present here is simplified both out of necessity, since very little information is publicly available, and also to focus our attention on the important issues of data management without getting
lost in the details of financial modeling. Still, we feel that our application model captures the important features of the real problem.

The PTA requires the database to maintain three types of prices: stock prices, composite index prices, and theoretical (call) option prices. The stock prices are the base data of the system, and are updated in the database according to the market feed. The composite and option prices, however, are derived data that must be computed from the stock prices. In actuality, the current trend of feed providers is to send more than stock prices with the feeds, including popular composite prices (e.g., Dow Jones Industrial Average (DJIA)) and other derived values. Still, additional derived data, such as that related to proprietary market models, will always need to be computed by the database system. Because composite averages and theoretical option prices have known functions, are easy to understand, and reasonably reflect the types of data that need to be computed, we choose to compute them as part of the PTA as representative of the proprietary derived data.

The database for the PTA contains the following six tables:

stocks (symbol, price) - contains the current price of every stock as reported by the market feed.

stock-stdev (symbol, stdev) - contains the standard deviation of the annualized rate return of every stock. The standard deviation is usually computed from the daily closing prices of the stock over a period of years. While periodic recomputation is supported by STRIP, we do not model the computation of standard deviation in this study since we are focusing on the behavior of the PTA during trading hours. For our purposes, this table is base data.

comp-prices (comp, price) - contains the computed price of every composite average (e.g., DJIA). This table is a materialized view (defined below) and hence corresponds to derived data.

comps-list (comp, symbol, weight) - describes the many-many relationship between stocks and composites. This table is entered by the trading firm and changes very infrequently.

option-prices (option-symbol, stock-symbol, strike, expir) - contains the computed price of every listed option. The data in this table is derived and its materialized view definition is given below.

options-list (option-symbol, stock.symbol, strike, expir) - describes the one-many relationship between stocks and options. This table must be updated once every three months when the option exchanges create new options and expunge expired options. In this example we only consider call options so the attribute strike is the price
at which the option holder can buy the stock stock-symbol prior to expiration. We consider this table base data since it reflects the listed options currently being traded on the exchanges.

The tables comp-prices and option-prices are materialized views with the following definitions:

```sql
create view comp-prices as
  select comp, sum(price*weight) as price
  from stocks, comps-list
  where stocks.symbol = comps-list.symbol
  group by comp

create view option-prices as
  select option-symbol, f_BS(price, strike, expir, stdev) as price
  from stocks, stock-stdev, options-list
  where stocks.symbol = options-list.stock-symbol and
    stocks.symbol = stock-stdev.symbol
```

The function $f_{BS}$ computes the theoretical price of an option based on the Black-Scholes pricing model [3], which although known to undervalue options, is still commonly used and reported.

### 1.2 INTERESTING PROBLEMS

In the process of both designing the architecture of STRIP and implementing it, a number of interesting and important research issues were encountered. The rest of this section outlines these issues.

#### 1.2.1 Temporal Consistency

Traditionally, databases have been tightly coupled to the external environments that they model. For example, in a banking setting, the amount of money in a customer’s account is exactly the value stored in the database. When funds are transferred, the transaction itself must involve the database system so that there is never a delay between the financial transaction and its reflection in the database. With applications such as program trading, however, the external environment being modeled is completely separate from the database. The stock market changes without approval from the database, forcing the application to monitor the environment and then notify the database about changes. Hence, due to delays in sampling, transmission, and updating, the values in the database will always lag the environment being modeled.
1.2.2 Update Scheduling. In a standalone real-time database system (RTDB), the only source of work is the user transactions, which are typically scheduled based on deadlines or value functions. In a system that imports or exports data, however, there are additional sources of work. If the volume of importing/exporting is relatively low, it can be given higher priority than user transactions without greatly affecting response time. In applications with a heavy import/export load, however, new scheduling algorithms must be designed to balance the two classes of work. For example, in the program trading application described in the last section, rates as high as 500 updates per second have been reported just for the security price feed alone. If all of these updates are performed before user transactions, the missed deadline percentage of the system will increase.

Thus the basic tradeoff is between transaction response time and data freshness, a measure of temporal consistency. How freshness is defined can vary by application. In cases where the value stored in the database is a sample of a continuously changing real world variable, such as the temperature of a solution in a factory control setting, the data starts becoming stale immediately. In cases where the real world variable changes at discrete points in time, such as the price of traded securities, the data is fresh until another trade occurs. The behavior of different applications in the presence of stale data can vary as well: less critical applications may choose to use whatever is available while other applications may rather choose to abort the transaction if the data is not fresh. This topic was discussed in Chapter 14.

1.2.3 Derived Data. Many applications, such as the program trading application require STRIP to maintain substantial amounts of derived data. When the base data changes frequently, maintaining the derived data can require a significant portion of the CPU capacity of the machine. Often, however, we can use the locality of the updates to reduce the recomputation workload. Update locality means that when a base item is updated, it is very likely that the same item or a related one will be updated soon thereafter. For example, a stock price update indicates that there is an interest in its trading. The same stock is therefore likely to be the subject of further trading activities and have its price changed again. Update locality implies that recomputations for derived data occur in bursts. Recomputing the affected derived data on every single update is probably very wasteful because the same derived data will be recomputed very soon, often before any application transaction has a chance to read the derived data. Instead of recomputing immediately, a better strategy might be to defer a recomputation by a certain amount of time and coalesce the same recomputation requests into a single computation. We have developed a variant of this strategy called forced delay which is described later in this chapter. Simulation studies show that forced delay greatly reduces the
recomputation cost while only modestly diminishing data timeliness across a wide range of parameter values.

1.2.4 Rule System Support. Given the positive simulation results for the forced delay algorithm we clearly wanted to implement it in STRIP. Rather than creating a special component, however, we chose to extend STRIP’s rule system. By adding a new feature called unique transactions, it is possible to write rules to maintain derived data that batch recomputation as suggested by the forced delay algorithm. In this chapter we will describe the rule system extensions and provide an example of their use. Performance experiments on STRIP indicate that the forced delay algorithm, as implemented using unique transactions, is as promising in the implemented system as it was in the simulation experiments.

1.3 OTHER PROTOTYPES

Most of the previous research in real-time databases has been based on analytical or simulation studies, not on experimentation on running systems. There are a few notable exceptions, however. One of the first real-time database implementations was the disk-based transaction processing testbed, RT-CARAT [6]. RT-CARAT was used to study the performance of real-time algorithms for CPU scheduling and conflict resolution. Although RT-CARAT implemented all of the major transactional components of a database, such as concurrency control and log management, it did not implement a full relational query interface.

A more recent paper [2] describes a commercial database that has been augmented with real-time functionality, RT-Genesis. The result is a disk-based relational database with an SQL interface that also supports deadlines on queries and transactions. Other components of the system were also made cognizant of deadlines, such as the scheduler, the concurrency control manager, and buffer manager. The authors use the new database to study previously proposed algorithms for each of the modified components.

The system most closely related to ours is StarBase [8] [7]. StarBase is a main memory relational database system with native real-time support for deadlines and temporal consistency. StarBase supports a richer subset of the relational operators than does STRIP but does not provide support for rules or data importation and exportation. To date, Starbase has been used primarily to compare scheduling and concurrency control algorithms.

2. EFFICIENTLY MAINTAINING DERIVED DATA

In the presence of high update rates, the DBMS can spend a significant fraction of its time recomputing derived data. In the PTA, maintaining the two materialized views is expensive for different reasons. The comp_prices
view has a very high fan-in. A change in any of the possibly hundreds of stocks (i.e. SP500) that comprise it requires a recomputation of the composite price. The option_prices view, in contrast, has a high fan-out. A stock will typically have many different options traded for it, each with different strike prices or expirations, so the change of a single stock price will force the relatively expensive recomputation of many option prices.

In previous work, we studied the benefit of allowing changes to collect over a short window before recomputing derived data. The specific algorithm, called forced delay, initiates a recomputation task for each derived datum after the first update to any of its base data arrives. The task is delayed, however, for a user-defined fixed length of time before it is allowed to run. Simulation experiments verify that over a wide range of cases, this algorithm significantly reduces recomputation cost with a tolerable reduction in temporal consistency.

Rather than creating a special component in STRIP to implement the forced delay algorithm, we have chosen to extend its rule system to achieve the same ends. In this chapter we describe the resulting rule system, which is an extension of SQL3-type triggers. The major new feature of the STRIP rule system over previous active rule systems is its unique transaction facility. This facility allows for very efficient incremental maintenance of derived data. The two benefits provided by unique transactions are:

- they allow rule actions to act on database changes batched across transaction boundaries, not just within one transaction, and
- they allow the batches to be partitioned in any way that reduces the cost of the derived data computation.

In this section, we describe the STRIP rule system and its unique transaction facilities. We also demonstrate how to use them to maintain the derived data in the program trading application.

2.1 SUPPORTING PROGRAM TRADING

In this section, we again consider the program trading application but focus on how the STRIP rule system can be used to maintain the derived data it requires, the tables comp-prices and option-prices. In this discussion we only focus on the rules that maintain these two materialized views as the stock prices change. In practice, we would need additional rules to handle changes to the other tables, e.g., comps-list or options-list. Since these other tables change very infrequently, we will ignore the additional rules although they can be easily expressed in STRIP’s rule system.

A straightforward way to handle stock price changes, and the only way available in most rule systems, is through a rule similar to the one shown in Figure 19.1. The rule do_comps1 states that when the price attribute of
create rule do-compsl on stocks
when updated price
if
    select comp, comp-list.symbol as symbol, weight, old.price as oldprice, new.price as newprice
    from comp-list, new, old
    where comp-list.symbol = new.symbol
    and new.execute.order = old.execute.order
    bind as matches
then
    execute compute-compsl

define function compute-compsl
  real composite.change;
  foreach row r in matches
    composite.change =
      r.weight * (r.new.price - r.old.price);
    update comp.prices
    set price += composite.change
    where comp = r.comp;
end function

Figure 19.1 Standard rule to maintain comp-prices for stock price changes

any stock changes, and that stock is used to compute at least one composite,
the transaction compute-compsl should be run to update comp.prices. When compute-compsl runs, it will need to know information about the
stocks that changed and how they impact the composite prices; this information
is assembled when the condition query is checked. Note that the condition
query must equate the execute.order of the old and new transition tables
to be sure that if the triggering transaction changed a stock price more than
once, the correct old and new tuple images are being considered together. To
allow comp.prices to use the data computed in the condition, we bind the
results as the temporary table matches which is then passed to the action
transaction. Using matches, the transaction running compute-compsl can update comp.prices without any other database reads.

Consider the example shown in Figure 19.2 in which the tables are populated
and two simple transactions that change the stock prices are shown. Transaction
T1 changes the prices of S1 and S2 which triggers rule do.compsl. The
condition of the rule is satisfied since computing the condition query returns
three rows. The result of the query is temporarily stored in table matches, and
a new transaction, \( T_{1a} \) is triggered immediately after \( T_1 \) commits. Transaction \( T_{1a} \) will execute the function \texttt{compute-comps1}, and when it does, it will be able to read the result of the condition query as the table \texttt{matches}.

Suppose that before \( T_{1a} \) runs, Transaction \( T_2 \) executes. Since \( T_2 \) also changes the prices of stocks that influence composite prices, it also triggers a new transaction, \( T_{2a} \). Transaction \( T_{2a} \) will also have access to a table called \texttt{matches} although it will see the second matches table shown in Figure 19.2. Although \( T_{1a} \) and \( T_{2a} \) update the same tuples in \texttt{comp-prices}, they remain in the system as two distinct transactions since \texttt{compute-comps1} has not been defined as \texttt{unique}. Figure 19.3(a) shows the two transactions that are enqueued and the bound tables that they will be able to access when they execute.

As mentioned previously, base data often changes in bursts. Hence the situation of Figure 19.3(a) with multiple \texttt{matches} tables can occur frequently. Thus we would like to batch the changes to a particular composite to reduce the amount of recomputation required. This is accomplished using a unique transaction within the rule, along with a delay window of 1 second, as shown in Figure 19.4 (changes from the previous rule and user function are boxed). Since \texttt{compute-comps2} is defined as unique, only one transaction of this type can be queued in the system at once. Let us contrast the behavior of this
rule to that of the previous rule. As before, transaction $T_1$ enqueues transaction $T_{la}$ with a three row bound table. Now, however, assuming that $T_1$ executes within 1 second of $T_a$, when $T_1$ commits it does not enqueue a new transaction. Instead, the rows of the bound table that would have been passed to the new transaction are appended to the bound table of transaction $T_{la}$. This is shown in Figure 19.3(b). Thus when transaction $T_{la}$ finally runs, it will have the combined bound information of both triggering transactions. Note that this is similar to the behavior described in [5] for deferred rules except that unique transactions allow changes to be batched across transaction boundaries.

Batching changes across transaction boundaries can greatly improve performance but it does require that the user rewrite the action function for maximum benefit. Consider the difference between `compute.comps1` and `compute.comps2` (Figures 19.1 and 19.4). The code `compute.comps1` steps through the `matches` table one row at a time, reading the affected composite, incrementally recomputing it, and then rewriting it. This same code
create rule do-comps2 on stocks
when updated price
if
  select comp, compslst.symbol as symbol, weight,
       old.price as old-price, new.price as new-price
  from compslst, new, old
  where compslst.symbol = new.symbol and
        new.execute.order = old.execute.order
  bind as matches
then
  execute compute-comps2
  unique
  after 1.0 seconds

define function compute-comps2
  real composite.change = 0.0;
  foreach row r in
    (select comp, sum ((new-old) *weight)
      as diff from matches groupby comp)
  update comp-prices
  set price += r.diff
  where comp = r.comp;
end function

Figure 19.4 Unique transaction approach to maintain comp-prices for stock price changes

would still work with the rule do-comps2 but it wouldn’t take advantage of the fact that the same composite will probably appear many times in the matches table. In contrast, the code in compute-comps2 groups the changes according to the composite they affect. The select statement aggregates the incremental changes to each composite so that each only has to be read, recomputed, and written once.

Rule do-comps2 is a big improvement over do-comps1 since it allows batching, but the unit of batching is quite coarse: all of the changes that affect any tuple in comp-prices are applied together. A problem with performing all of the changes together is that the recompute transaction can be very long. Longer transactions have two drawbacks:

- They hold locks longer which leads to increased conflicts. More lock conflicts increase both response time and the variance in response time,
neither of which is desirable in a real-time environment. The problem of choosing transaction boundaries in active databases was studied in [4] where similar results were found.

- In a real-time system, transactions may have to be restarted either because they miss their deadlines or because a high priority transaction is blocked waiting for a lock that they hold. Longer transactions result in both more restarts and more lost work when a transaction is restarted.

Hence the goal in choosing a unit of batching should be to balance the performance benefits of combining like work and the scheduling flexibility of smaller transactions.

With this goal in mind, we note that grouping all of the changes that affect a single composite price is useful because the old value can be read once, updated, and then written once. If the same changes were in separate transactions it would result in many more reads and writes of the composite. Any larger unit of batching does not seem to generate any further efficiency. This suggests that we batch on attribute \texttt{comp}, which would yield the grouping shown in Figure 19.3(c).

The rule to batch on composite name is shown in Figure 19.5. When \texttt{compute comps3} runs, the \texttt{matches} table it sees will only contain changes for a single composite. This results in simpler code than that used in the function \texttt{compute comps2}: The for loop accumulates all of the weighted price changes from stock changes and then the total change is applied to the composite price.

We have now seen three rules to incrementally maintain the \texttt{comp-prices} table.

In addition to maintaining the view \texttt{comp-prices}, we also want to maintain the view \texttt{option-prices} that contains theoretical option prices. When a stock price changes, the prices of all of the options that are based on it will change as well. A simple rule to recompute option prices when a stock price changes is shown in Figure 19.6. Note that since option prices cannot be computed incrementally, the transition table \texttt{old} is not bound as part of the condition.

As with composite prices, it may be possible to improve on the performance of this rule using unique transactions, but we must choose a unit of batching. The efficiency to be gained by batching is that if an underlying stock price changes more than once in a delay window, we can use its last value to compute the theoretical option price only once. This is different from the previous example in which we had to consider every change. Hence, batching based on option symbol might be a good choice. On the other hand, there are often many more options in the system than stocks, so batching on such a small unit may lead to a huge number of transactions in the system. Hence it may be
create rule do-comps3 on stocks
  when updated price
  if
    select comp, comps.list.symbol as symbol, weight,
    old.price as old.price, new.price as new.price
    from comps.list, new, old
    where comps.list.symbol = new.symbol and
      new.execute.order = old.execute.order
    bind as matches
  then
    execute compute-comps3
    unique on comp
    after 1.0 seconds

define function compute-comps3
  real composite-change = 0.0;

  foreach row r in matches
    composite-change +=
      r.weight * (r.new-price - r.old-price);
  update comp.prices
  set price += composite-change
  where comp = r.comp;
end function

Figure 19.5  Unique transaction approach (on comp) to maintain comp.prices for stock price changes

wiser to batch on stock symbol, collecting all of the options related to a single stock together: this unit of batching may also allow some partial results used for every option to be computed only once.

3. PROTOTYPE ARCHITECTURE

This section gives a very brief overview of the STRIP architecture. The interested reader is referred to [1] for more details. The block diagram of STRIP is not different enough from a typical active DBMS to include here. Instead, we focus on the process architecture as it relates to emulating real-time scheduling on Unix and handling import/export streams in addition to application transactions.
create rule do-options1 on stocks
when updated price
if
    select option-symbol, stock_symbol, strike, expiration,
      new.price as new.price
    from options.list, new
    where options.list. stock_symbol = new. symbol
    bind as matches
then
    execute compute-options1

define function compute-options1
    real last-price, stdev;
    foreach row r in matches
        stdev = select sd from stock-stdev
               where symbol = r. stock_symbol
        update option-prices
        set price = f(r.new-price, r.strike, r.expiration)
        where option-prices . option-symbol
                     = r. option-symbol;
end function

Figure 19.6 Standard rule to maintain option.prices for stock price changes

3.1 SCHEDULING ON UNIX

Since STRIP is intended for use in open systems, we decided to build it on top of a standard operating system, Posix Unix, instead of a specialized real-time OS. Since standard operating systems do not provide real-time scheduling, this approach is not acceptable for hard real-time systems. However, it is acceptable in a soft real-time system like STRIP, as long as we somehow make the system adhere as closely as possible to the timing constraints of data and transactions.

Our approach to scheduling is to run every task in the system as a separate process and have the processes work cooperatively to schedule themselves. At any particular time, at most one of these processes will be trying to execute and the rest will be blocked. By not competing with each other for OS time slices context switch overhead is reduced.

Every process links in the scheduler code which maintains global scheduling information in a shared memory segment. When a process wants to start a unit of work, it calls a library routine to register itself and be scheduled.
If no other process is currently executing the routine returns immediately. Otherwise, it blocks. When a process finishes the task it’s working on it picks the waiting process with the highest priority and wakes it. Longer running tasks or tasks with low priority can also make additional calls to the library routines to allow for intermediary scheduling points. Hence, STRIP supports only cooperative multi-tasking in the standard configuration. Preemptive scheduling has been implemented as well using the real-time priority levels of HP-UX (Posix standard) but this component is less portable. A similar approach is described in [2].

3.2 PROCESSES

The process structure used to implement STRIP is shown in Figure 19.7 and is similar to other high performance and real-time database systems such as [6]. All of the data access and modification is done by a single pool of execution processes. These processes both apply updates to import views and run transactions. The interface to the process pool is a collection of queues: a transaction queue and one update queue for each import view. The transaction queue contains both application requests and internally triggered transactions. The updates are divided into separate queues so that the system can maintain some views more diligently than others in the case of different timing constraints or different values.

The other processes serve to connect the database system to the outside world, such as stock ticker data. The import and export processes decouple the network interface from the database system core and perform the conversion from the internal format to the stream format through which STRIP communicates with other systems. The request process conveys remote requests to the database system and sends the results back to the originator. It supports both TCP and UDP protocols. In addition, local applications can access the queues directly using a client library for greater efficiency.

A simplified view of the flow of work in the system from the perspective of the execution processes is shown in Figure 19.8. The queues in this figure are all composed of tasks from the update queues and the transaction queue of Figure 19.7. Every new task to the system, either from the update or transaction queues, has associated with it a release time specifying the delay before it can be scheduled. Many user and update tasks have immediate release times and are placed in the ready queue upon arrival. Other tasks, though, in particular those that will run unique transactions with delays, will have a future release time. These tasks are placed in the delay queue until their release time.

When an execution process becomes free, it first chooses a task from the ready queue to execute. We call the choice of which ready task to choose the external scheduling problem. The chosen task is then moved to the running
Figure 19.8 The flow of tasks within STRIP

queue and the execution process attempts to run its code. If other execution processes are running, however, the current process must decide whether to preempt one of them or whether to wait. We call this the internal scheduling problem. If, while running, a task must block waiting for a lock, it is placed in the blocked queue until the lock is granted and its execution process waits on a semaphore. When the lock is released the semaphore is released as well and so the process can continue executing. When a task finally finishes, results are returned to the user if appropriate and the execution process looks to the ready queue to find another task. Note that the blocked queue shown in Figure 19.8 is only for tasks waiting on a lock and is not related to the blocking described in Section 3.1 which is used to ensure that only one process is trying to execute at any given time. The cooperative scheduling algorithm is being performed by the processes in the running queue only.
3.3 INTERNAL AND EXTERNAL SCHEDULING

To implement this task flow we need to build two schedulers to solve the external and internal scheduling problems. The two schedulers could use the same algorithm but their problems are different in two ways. First, the external scheduler may need to choose from among far more candidates than the internal scheduler since the internal scheduler is limited to the number of execution processes. (If there are more execution processes than work waiting to be done the internal scheduling problem is trivial.) This is especially true when the application rules trigger many new tasks. This argues for a simpler algorithm in the external scheduler than in the internal scheduler.

On the other hand, STRIP is usually run with only a few execution processes and so the choice of internal scheduler may not be very important. In addition, a poor choice by the external scheduler may effectively “jam” an execution process. This occurs when the external scheduler chooses a low priority task which is then never scheduled by the internal scheduler. In such a case, the execution process is blocked and completely useless to the system.

Clearly, the choice of which scheduler to use for each problem will depend on the characteristics of the application being run. Therefore, STRIP v2.0 allows each scheduler to use a different algorithm that can be chosen at startup. The supported choices are first in first out (FIFO), earliest deadline first (ED) and highest value first (HV). Other schedulers, particularly those for value function scheduling, are planned for the next version. More detailed examination of the performance of the three algorithms is a subject of future work, but preliminary results indicate that in very high performance applications, such as the program trading application, the FIFO and HV algorithms outperform ED since they require less overhead.

4. SUMMARY AND FUTURE WORK

The extended rule system described in this chapter was implemented in STRIP, and yielded significant performance improvements [1]. Unique transactions reduced the CPU load for recomputing comp-prices by 50% with a half second delay window and by more than 67% with a 3 second delay window. Unique transactions were also effective in reducing the CPU load for recomputing the option-prices table: a delay window of 3 seconds reduced the load by 20%.

In general, by building a program trading testbed on STRIP, we were able to verify that for real-time database systems that need to monitor a real world system, the costs of updating base data and maintaining derived data can be as great as that of executing user transactions. Only through careful and explicit treatment of these other tasks can one build a system that satisfies the timing and timeliness constraints of a real class of applications.
To support future applications, we would like to extend STRIP with two new features. First, STRIP provides timing constraints (in the form of value functions) on individual tasks but does not support end-to-end timing constraints. Even if we can assume that the end-to-end constraints only encompass work done in STRIP, we need to develop an algorithm that decomposes a global value function into the individual value functions for each subtask of the global task. Second, although STRIP supports timestamps it does not provide true temporal support. Many of the applications that require real-time constraints must also be able to access the value of external variables over time. In the program trading application, temporal data is used to analyze price trends to predict future changes. If STRIP has to store the full historical record of values, however, its storage requirements will be unbounded which conflicts with the assumption of main memory residency. To solve this problem, STRIP’s exportation facilities can be used to migrate old data to a remote machine for archiving. This raises a number of issues related to both performance and answering distributed queries that must be studied further.

References

This page intentionally left blank.
Chapter 20

FUTURE DIRECTIONS

Tei-Wei Kuo  
Department of Computer Science and Information Engineering  
National Taiwan University  
ktw@csie.ntu.edu.tw

Kam-Yiu Lam  
Department of Computer Science  
City University of Hong Kong, Kowloon, Hong Kong  
ckylam@cityu.edu.hk

1. INTRODUCTION

Although RTDBS has been widely considered as a promising direction for database systems research, and a lot of research effort has been devoted, there still exist many challenging and unresolved issues [9]. As we are approaching new application domains, how to apply RTDBS technology to these new applications is not trivial. In this chapter, we shall address important research topics which have been ignored in the past, practical issues in the development of a RTDBS, and new challenges which are resulted from the emerging of new real-time database applications. In the last part of the chapter, we shall review some important real-time database prototypes and commercial products.

2. DESIGN ISSUES

Although a lot of research efforts have been devoted to the design of efficient RTDBS, many important issues remain unresolved. Some of the issues are on data management in large-scale RTDBS, checkpointing and recovery, and concurrency control for mixed RTDBS.
2.1 DATA MANAGEMENT IN LARGE-SCALED RTDBS

Different from conventional real-time systems, a RTDB may need to manage a large database. In order to satisfy the real-time requirements of the systems, it is highly important to adopt efficient methods to process, organize, and retrieve data items. Consider a large-scaled RTDBS, such as an intelligent transportation system. The amount of data received by a particular traffic management center in an hour, or even in a minute, could be tremendous. It might be very likely over the physical capacity of any ordinary computer system. How to design a highly scalable real-time information server could be essential to the success of such large-scaled data processing systems. A multi-processor architecture may become a must in the system design, and new system implementation constraints and their solutions, such as the criteria in organizing the computations across the processors, should be introduced and explored.

Furthermore, many RTDBS applications, such as programmed stock trading systems, might also need to organize their data items in a truly on-line fashion. Unfortunately, there is little work being done on the on-line organization and retrieval of a tremendous amount of data. It is also very important to explore the relationship between an access structure and a physical data organization on storage media. With a real-time performance requirement, the worst-case system behavior of a RTDBS must be fully understood and quantified. More research in these directions is critical in building large-scaled real-time information systems.

Index is essential in providing efficient data access in a large database. Most of previous work in index-based real-time data access is focused on the B-tree index structure. It is completely lack of any study on the design of real-time index-based access with other access methods, such as hashing. At the same time, it is also important to consider temporal properties of real-time data in the design of an index structure. How to apply different temporal indexing techniques to the index structures of a real-time database to improve the system performance is a possible direction.

2.2 CHECKPOINTING AND RECOVERY IN RTDBS

Checkpointing and recovery are important research topics for traditional database systems. Researchers have proposed various highly efficient mechanisms, such as fuzzy checkpointing, to guarantee various degrees of database consistency, such as action-consistent (AC) or transaction-consistent (TC) checkpoints. However, little work has been done for RTDBS in this aspect. With the increasing of CPU speed and the dropping of RAM cost, more and more real-time applications, especially those in the industrial domain,
may move to memory-resident databases, partially or even entirely. Memory-resident databases may become further attractive when transactions with highly restrictive response time requirements, such as sensor transactions, are considered. While issues in real-time concurrency control are better understood, the demand of system reliability is increasing. How to seamlessly integrate real-time concurrency control protocols and recovery algorithms will be essential in the design of high performance reliable RTDBS.

2.3 CONCURREN CY CONTROL IN MIXED RTDBS

Most of the previous studies on concurrency control were concentrated on systems with a single type of real-time transactions. However, due to the growth of RTDBS application domains and its integration with other non-real-time systems, it becomes more and more common to have mixed RTDBS where different types of real-time and non-real-time transactions may co-exist in a system. How to apply various value-based transaction scheduling algorithms and real-time concurrency control protocols to a mixed RTDBS will be an important topic. Due to the different performance requirements of the transactions, methods proposed for RTDBS with a single type of real-time transactions may not be suitable to mixed RTDBS. For example, many methods proposed for concurrency control of soft real-time transactions are based on transaction restart. They may seriously affect the performance of non-real-time transactions because non-real-time transactions may be repeatedly restarted by soft real-time transactions. It is also important to maintain a balance between meeting the real-time requirements of a system and satisfying the performance requirements of the entire system.

3. PRACTICAL ISSUES

How to apply RTDBS techniques to real commercial application systems could be one of the major challenges in the next decade. Up to now, not many real-time techniques developed in the research community are really adopted in the design of practical commercial RTDBS. It is mainly due to three reasons: Firstly, how to provide real-time performance cost-effectively in a traditionally non-real-time environment is still a challenging problem. The conflict thinking in priority inversion management (from the real-time system community) and high concurrency degree requirements (from the database community) surely introduces dilemma in the mixed scheduling of transactions with both deadline and throughput considerations.

Secondly, there are several essential system components in realizing an "ordinary" RTDBS, which includes those for disk scheduling, buffer management, concurrency control, CPU scheduling, etc. Any component that fails to deliver a real-time performance can result in the failure of providing real-time
performance for the entire system. In order to achieve the required real-time performance, various real-time techniques have to be applied to all necessary components. Most of the time, we might need a proper real-time operating system. However, moving an entire application onto a real-time operating system may not be easy for many soft RTDBS. Although many general-purpose operating systems are now added with different real-time extensions, e.g., priority-driven scheduling, main memory locking, and real-time clock, it is important to identify the performance tradeoff of the adopted real-time database techniques while they are used in an imperfect real-time environment. Techniques might need to be redesigned so that they can be easily integrated to various real-time and non-real-time environments, i.e., Polaris, OSE soft kernel and CORBA.

Thirdly, when it comes to the delivery of a RTDBS, there is simply little work addressing the benchmark issues, which are important in measuring the performance of a database system and for performance comparison. Most of the previous study in RTDBS used simulation in their performance evaluation. In their simulation experiments, different assumptions on the systems were usually made to simplify the analysis. To better illustrate the performance of the real-time database techniques, it is important to use a benchmark program to test the system performance. Although many well-known benchmarks have been developed for conventional database systems and specific database systems, such as TCP benchmarks [8] and Wisconsin Benchmark, it is completely lack of any benchmarks for RTDBS. The design of benchmarks for RTDBS could be based on the TCP-C benchmark, which is designed for traditional on-line transaction processing systems.

4. NEW REAL-TIME DATABASE APPLICATIONS

Due to the advances in Internet technologies and the growth of E-Commerce, the demand on "real-time" applications are growing tremendously. Various innovative applications are emerging, such as "real-time" stock trading systems, systems for dissemination of real-time information, E-Commerce applications, and various mobile computing applications. Unfortunately, most of these new applications are developed using non-real-time techniques. For example, most of the "real-time" stock trading systems are, in fact, "non-real-time" systems. There is no way to guarantee the system performance and the recency of stock information. Another example is on the design of On-Line Analytical Processing (OLAP) systems. Engineers often try to minimize the average response time although the ultimate objective of the systems is to provide timely information to any changes in the external environment. Investigation on how to apply RTDB techniques in the design of these new real-time database applications will be very critical to the usefulness of these applications.
We often need to determine which RTDB technology is most helpful to which real-time application system. A compromise between real-time requirements and other system requirements of a real-time application system must be reached. For example, in a stock trading system, the status of stocks is highly dynamic. Serializability criteria may be better weakened to tolerate certain data imprecision so that sell/buy orders can be matched in the most efficient way. Thus, application semantics must be explored, and simplified data access mechanisms must be proposed to meet the degree of system integrity needed by these applications.

5. RESEARCH PROTOTYPES AND COMMERCIAL PRODUCTS

Various RTDBS prototypes are developed or under development. Some of the early prototype projects are the REACH (Real-time Active and Heterogeneous mediator systems) project [3, 4], the STRIP (Stanford Real-time Information Processor) project [1] and the StarBase project [12,10]. Two of the most recent RTDBS prototype projects are the DeeDS project at the University of Skovd and the BeeHive project at the University of Virginia. In the DeeDS project, a distributed active real-time database system prototype is developed. It is a main-memory-resident fully-replicated real-time database system. In the system, the event-condition-action (ECA) rules are used to monitor the system status. A dedicated deadline-driven scheduler is adopted to guarantee the deadline of hard real-time transactions [2].

The BeeHive project addresses virtual global, multimedia, object-oriented databases. It is enterprise-specific and offers features along real-time, fault tolerance, quality of service for audio and video, and security dimensions [13]. Support of all these features and tradeoffs among them might provide significant improvement in performance and functionality over browsers, where browsers are connected to databases and, in general, today's distributed databases. Such results may be applicable to sophisticated real-time control applications as well as the next generation Internet. Currently, various component technologies are implemented into the prototype, i.e., real-time concurrency control, transaction scheduling, security tradeoffs, resource models and associated admission control, and scheduling algorithms. A cogency monitor has also been developed as an interface from BeeHive (an enterprise system) into the open Internet. The success of these prototypes is important for demonstrating the usefulness of the research findings in the area and for identifying the future research directions. Once we have some RTDBS benchmarks, more reliable performance data can then be obtained by performing experiments with these prototypes.

At the same time, commercial "real-time" database system products have started to appear in the market such as Eaglespeed-RTDB [11], Polyhedra [14],...
Clustra Parallel Data Server [6], Timesten [15], Cascade [5] and Mnesia [7]. Although these products may not be considered "true" RTDBS from the viewpoints of many researchers in the RTDB community since most of them only have very limited real-time features, they represent a significant step toward the success of RTDB. Most of these products basically use main-memory-database techniques to achieve a better real-time performance. Some of them include features for real-time transaction management. For example, the Eaglespeed-RTDB, which is originally developed for defense systems, provides priority inheritance for transaction management. Polyhedra is an active, object-relational database for those demanding high-performance embedded database applications. It uses event queues, which is an active feature, to monitor the values of selected data items. Although it is still lack of a "ture" RTDBS, these commercial products are important systems in demonstrating the usefulness of the RTDBS techniques for practical real-time database applications.

Acknowledgements  We would like to express our gratitude to the Distributed Real-Time Systems (DRTS) Research Group of the University of Skövde and the Distributed Real-Time Multimedia Databases of the University of Virginia for providing us valuable information regarding their real-time database prototypes.

References


This page intentionally left blank.
Index

Δ-serializable, 53
2PL High Priority (2PL-HP), 147
Absolute consistent system (ACS), 193
Action execution, 162
Active Abort, 234
Active database systems, 77
Admission control, 125
Air Traffic Control, 17
Anticipatory Prefetching, 88
Aperiodic updates, 188
Authentication, 141
Automatic distribution mechanism, 174
Avionics and Space, 17
B-link Protocols, 62
B-tree, 60
Backward validation, 38
Bayer-Schkolnick Protocols, 62
Buffer allocation, 78
Buffer management, 78
Buffer replacement, 78
Cautious waiting, 250
Cellular network, 246
Command and Control, 17
Composite event instance, 173
Condition evaluation, 162
Condition-Action Coupling, 166
Conflict serializability, 45
Conservative algorithms, 29
Contingency transaction, 127
Control action transactions, 78
Coupling modes, 165
Crash Recovery, 119
Cyclic executive, 45
Data pre-fetching mechanism, 254
Data similarity, 46
Data-handling mechanisms, 25
Data-Priority-Based Locking Protocol (DP), 211
Database integrity, 3
Deadline tolerance, 127
Deferrable Group, 103
Deferred coupling, 163
Deferred disk writes, 83
Deferred-deferred, 166
Deferred-detached, 166
Derived data, 264
Detached coupling, 163
Detached-detached, 166
Detached-immediate, 166
Disk-stripping-replication, 105–106
Dispatcher, 128
Distributed commit, 155
Distributed RTDBS, 205
Dual-CC, 148
Durability, 22
Dynamic adjustment of serializability order, 40
Earliest deadline first algorithm, 5
Early-Loose-Late-Tight, 100
Early-Tight-Late-Loose, 100
ECA model, 163
Elevator algorithm, 13
Ephemeral transaction logs, 113
Epsilon serializability, 218
Epsilon-serializability, 45
Event detection, 162
Event instance, 173
Event monitoring, 175
Event subscription, 177
Event-Condition Coupling, 166
Event-Condition-Action, 162
Exclusive detached coupled actions with causal dependencies, 167
Execution Site is a Fixed Host (ESFH), 256
Execution Site is a Mobile Host (ESMH), 256
External consistency constraint, 5
External interrupts, 13
Fan-in, 190
Fan-out, 190
FD-SCAN, 102
Final-state serializability, 45
Firm real-time transactions, 4
Secure Real-Time Database Systems (SRTDBS), 142
Security mechanisms, 142
Security, 141
Sensor data, 9
Sequential detached coupling with causal dependency, 167
Shortest Seek and Earliest Deadline by Value (SSEDV), 103
Shortest Seek Time First, 101
Shortest-Seek-Time-First (SSTF), 13
Silent Kill, 234
Similarity bound, 49
Similarity Stack Protocol (SSP), 49
Similarity, 47
Similarity-based concurrency control, 45
Soft real-time transactions, 3
Solid State Disk, 110
Space shuttle, 24
Space threshold, 99
Spatio-temporal characteristics, 59
SPLIT Algorithm, 117
Stepdeadline, 100
Temporal attributes, 173
Temporal consistency constraint, 5
Temporal Consistency, 192
Temporal consistency, 263
Temporal correctness, 192
Temporal databases, 13
Temporal depth, 50
Temporal validity, 5
Thomas Write Rule, 33
Three phase commit, 230
Timesharing systems, 11
Timestamp-interval adjustment scheme, 39
Top-Down protocols, 62
Transaction shipping, 252
Transient overload, 125
Triggering graph, 80
Triggering phase, 162
Turing machine, 12
Two-phase commit, 229
Two-Version PCP (2VPCP), 34
Unapplied Update, 189
Update scheduling, 264
Urgent Group and Shortest Seek Time First (UG-SSTF), 103
Urgent Group, 103
URT, 195
User quorum, 218
Value function of time, 4
Value function, 127
View serializability, 45
View Similar, 52
W-recency bounds, 51
Worst-case execution time, 127
Write-all-available, 220
Write-locked, 33