Design Issues for Extensible Concurrency Control Mechanisms

S. Seshadri $*$ Shiby Thomas $*$ Jayant R. Haritsa $**$
CSE, Indian Institute of Technology, Bombay $*$
SERC, Indian Institute of Science, Bangalore $**$

Abstract

Relational database management systems lack the requisite support for specialized applications such as engineering databases, electronic design databases, geographic databases, etc. Extensible database management systems attempt to fill this void by providing capabilities to define query language extensions and specialized storage structures that suit the particular application. However, they do not provide for having multiple concurrency control policies operating simultaneously. This feature appears useful for multi-application workloads since it would be best, from a performance viewpoint, if each application operated under its favourite concurrency control algorithm. The problem, however, is to maintain database integrity in this environment.

In this paper, we address the issue of designing mechanisms that allow extensible database systems to simultaneously operate multiple concurrency control policies without loss of data integrity. The key properties of such a mechanism are: Firstly, to ensure extensibility, the mechanism must be independent of the actual concurrency control policies that are co-existent. More generally, it must be able to support *any* concurrency control policy that adheres to a standard interface. Secondly, the mechanism must minimize the interference between the individual concurrency control policies to ensure that the performance of the individual applications remains good. We present here a database system architecture that attempts to satisfy both these requirements. Preliminary results from a simulation-based study of this architecture indicate promising performance trends.

Keywords: Extensible Database, Concurrency Control

1 Introduction

For a period of almost two decades since its inception in the early 70s, the relational approach [Cod70] to designing database management systems has held sway over the database market arena. Over the last few years, however, it has been widely recognized that while relational database systems are quite appropriate for commercial applications, they lack the required level of support for more specialized applications in which the need for database technology has arisen (e.g. [Hei87]). Examples of such applications include geographic databases, manufacturing databases, electronic design databases, etc. [Cat91b].

Extensible Database Systems

There have been basically two approaches for designing new database systems that adequately serve the above-mentioned types of specialized applications. The first approach, as exemplified by systems such as ObjectStore [LLOW91], Orion [Kim88], etc. is to build completely new database systems which entirely discard the earlier relational approach and adopt a new modeling paradigm, typically the object-oriented paradigm. The second approach, which is seen in systems such as Postgres [SK91], Genesis [BLW88] and Exodus [Car90], is to provide added functionality to a relational core. For example, all of these systems support abstract data types to describe complex objects. Such systems are called **extensible** systems [Car87] or "extended relational database systems" [Cat91a].

While the goals of the various extensible database systems are similar, their design approaches have been quite different. Some have developed entire database management systems (for example, Postgres), while others have followed the so-called "toolkit" approach. In the toolkit approach, used by both Exodus and Genesis, the database implementor is not provided with a complete end-user DBMS. Instead, only a comprehensive set of basic DBMS kernel facilities and software modules are provided. Using these tools, the database implementor can **generate** a database system customized to the application or group of applications that are to be supported. This approach to building DBMS has become popular and several research projects have used Exodus to build their database testbeds (e.g. [LOL92]).

Concurrency Control

Extensible Database Management Systems [CH90] based on the tool-kit approach typically support extensions to the query language, the query processor, and the storage and access methods in order to provide greater modeling power and better performance. They do not however address the issue of simultaneously operating multiple concurrency control (CC) policies in a database management system. For instance, both Exodus and Genesis provide only the well-known two-phase locking (2PL) CC mechanism. The motivation for having multiple concurrency control policies is the following:

- The semantics of the operations of a certain class of applications may make it possible to devise custom-designed concurrency control schemes. For example, semantic mode locks such as *increment* and *decrement* may enhance concurrency [Wei88].
- Specific concurrency control policies may be especially suited for certain classes of applications. For example, optimistic concurrency control (OPT) algorithms [KR88] typically provide better performance than locking-based algorithms for "firm-deadline" real time transactions [HCL90] [1].

In short, the needs of a particular application may be served better by providing it with its "favourite" concurrency control policy.

To motivate the above situation, consider a stock market application in which there are two transaction classes: **real-time** and **standard**. Transactions in the real-time class have associated completion deadlines, that is, the user submitting the transaction would like it to be completed before the stated deadline. Accordingly, a higher quality of service is associated with processing these transactions before their deadlines expire. Such transactions are typically submitted by brokers who are buying and selling shares and need their transactions to complete before the current share prices are updated. In contrast, transactions of the standard class are usually submitted by stock exchange officials who are running various types of accounting transactions. For this class of transactions, the quality of service is measured in terms of the transaction throughput.

[1] Firm deadlines means that transactions which miss their deadlines are considered to be worthless and are immediately discarded from the system without being executed to completion.

Obviously, a database system designed for supporting the above stock market application has to meet the performance requirements of both real-time and standard classes of transactions. Unfortunately, however, concurrency control policies are usually suitable for meeting only a specific class's performance requirements – in this case, 2PL for standard transactions, and OPT for real-time transactions. It is therefore necessary for the DBMS to simultaneously support both types of concurrency control mechanisms in order to achieve the desired performance objectives.

Yet another example for extensible concurrency control policies is the possible exploitation of application-specific semantics. Consider a banking environment where some tellers are depositing money while others are catering to withdrawals, transfers etc. Since some of these operations such as deposit, for example, do not require knowledge of the actual account balance, an increment or decrement lock can be used as suggested in [Wei88] and multiple such locks can be held simultaneously [2]. This could significantly improve the concurrency of deposit transactions. Other transaction classes may wish to use a different concurrency control policy and it may be best to allow each class to operate under its chosen CC policy. In general, the semantics of a transaction class can be exploited to define better concurrency control algorithms for that class [Mol83, Lyn83].

At first glance, a database system which supports multiple concurrency control policies may look similar to a multidatabase system. However, the issues involved in both are different. In a multidatabase, a given transaction is subjected to different concurrency control policies at different sites while we are considering the case where a single transaction is subjected to a single concurrency control policy of its choice. Further, the same data item will not be subjected to different concurrency control policies in a multidatabase system while that is a possibility in our case. The major issue in multidatabase concurrency control is to serialize the global transactions without losing site autonomy which is not an issue in our case.

Goals

In this paper, we address the issue of designing extensible database systems that provide a suite of concurrency control mechanisms at run-time. To the best of our knowledge, there have not been any efforts in this direction so far. One reason may be that it is not obvious how data integrity can be maintained when multiple concurrency control policies co-exist in the same database system.

Multiple concurrency control policies can "peacefully" co-exist in a DBMS if either of the following conditions are satisfied:

1. At any given instant of time, exactly one class of transactions is active, that is, the classes are *temporally disjoint*.

2. The data items accessed by transactions of each class are disjoint, that is, the classes are *spatially disjoint*.

The only additional control required in the above cases is a router that directs each transaction to its chosen CC mechanism. However, when transaction classes are neither temporally disjoint nor spatially disjoint, allowing multiple concurrency control policies to operate simultaneously in an unmanaged fashion can easily lead to loss of data integrity. For example, if two transactions access the same data using different concurrency control policies, it is equivalent to executing the transactions without any concurrency control since conflicts are not detected (in the absence of a higher-level controller).

In this paper, we present a scheme which allows transactions to choose from multiple concurrency control schemes and execute simultaneously without loss of

[2] We assume that these lock operations are atomic.

data integrity. This is achieved by interfacing a new software module called **Master Concurrency Controller (MCC)** between the transaction manager and the multiple concurrency control managers. The MCC routes transaction data requests to the appropriate concurrency control manager and ensures that the interaction between the various CC modules is regulated to maintain data integrity.

We have built a simulation-based performance model to evaluate the effectiveness of the MCC architecture in simultaneously providing good performance to a mix of applications. In particular, we have focused on a DBMS that caters to real-time transactions and standard transactions. As mentioned earlier, 2PL is most suitable for standard transactions while OPT is preferred for real time transactions. Preliminary results from our simulation experiments indicate that allowing both algorithms to co-exist by using MCC yields good performance for both application classes.

Organization

The rest of this paper is organized as follows: We present, in Section 2, the design issues involved in developing the Master Concurrency Controller. The performance model is described in Section 3, and the results of the simulation experiments are highlighted in Section 4. We conclude in Section 5 and outline future research avenues.

2 Design Issues

In this section, we outline the various design issues involved in integrating multiple concurrency control policies within a single database management system. The integration mechanism must satisfy two important properties:

- To ensure extensibility, the mechanism must be independent of the actual concurrency control algorithms that are co-existent. More generally, the mechanism must be able to support any concurrency control mechanism that adheres to a standard interface (supports a standard set of interface functions).
- The mechanism must minimize the interference between the individual concurrency control policies to ensure that the performance of the individual classes remains good.

We begin by considering a simple integration mechanism – however, this is found to violate the first requirement mentioned above. Suitable extensions to the mechanism are then made to rectify this shortcoming. The extent to which the second requirement is met by this mechanism is evaluated in the following sections.

A candidate DBMS architecture for integrating multiple concurrency control policies is shown in Figure 1. This architecture differs from that of a DBMS employing a single CC policy in that the concurrency control manager is replaced by the dashed box in Figure 1. Here, all transaction calls from the Transaction Manager are first sent to the Master Concurrency Controller (MCC). The MCC then forwards each call to the appropriate "slave" concurrency control manager (CCM). We assume that the MCC and all CCMs support the following standard interface functions: Begin_trans, Commit_Trans, Abort_Trans and Access_Item. The arguments to the function Access_Item specify the item being accessed and the mode of access (such as read or write). The slave CCMs, in addition to the above set of interface functions, support a function called Prepare_to_Commit. We elaborate on this function later. Each CCM interfaces with the Local Copies Manager (LCM) which in turn interacts with the Buffer Manager (BM) to provide the data asked for by transactions. The exact function of the local copies manager is explained later.

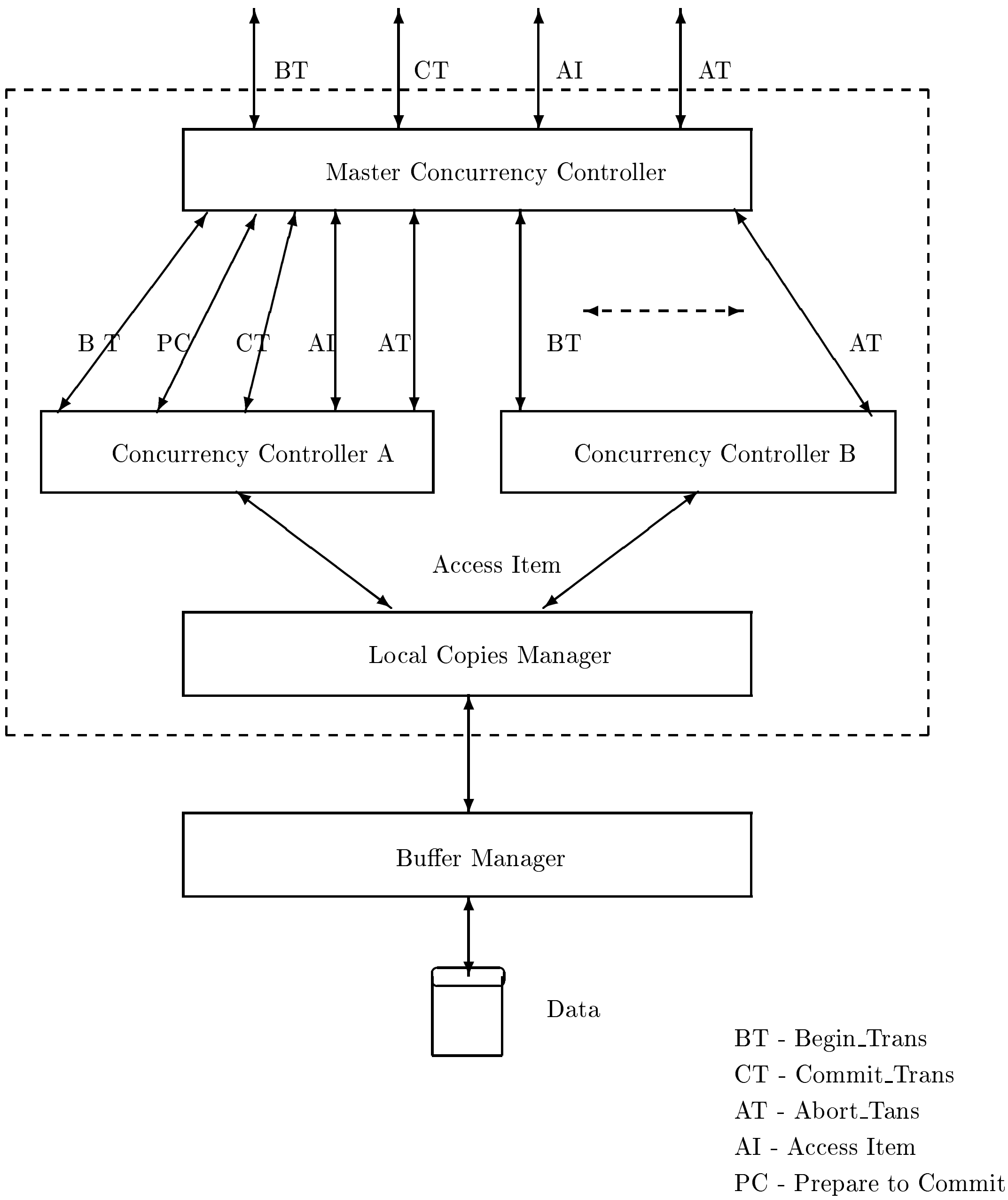

Figure 1: System Architecture

The MCC operates in the following manner: On receiving a Begin_Trans call, it enters the transaction into a private lock table and forwards the call to the appropriate slave CCM. For a Commit_Trans call, it forwards the request to the relevant CCM and either commits or aborts the transaction as per the recommendation of the CCM. An Abort_Trans is simply handled by aborting the transaction at the MCC and forwarding the abort request to the concerned CCM. The Access_Item is processed according to the data accessed and the specific CC policy implemented at the MCC.

The above architecture is applicable to any number of transaction classes and concurrency control algorithms. However, to simplify the discussion, we assume in the remainder of this paper that there are precisely two classes of transactions called Class A and Class B. We denote the chosen concurrency control algorithm for transactions of Class A as CCMA and the algorithm for transactions of Class B as CCMB. Let TiA and TiB denote a generic transaction of Class A and Class B respectively. In the following discussion, we initially assume that two phase locking is used at the MCC and that either two phase locking or optimistic concurrency control is used for CCMA and CCMB.

Consider the situation where TiA requests to read or write item X. If a transaction of Class B is currently holding a conflicting lock on item X, the MCC forces TiA to block (since MCC implements 2PL). Otherwise, the MCC forwards the request to CCMA after locking X in the appropriate mode. This means that *intra*-class conflicts are *not* detected at the MCC and are the responsibility of the respective concurrency control managers, while *inter*-class conflicts are regulated by the MCC.

2.1 Modification One

The above simple algorithm is easily proven to perform correctly under the assumption that the MCC employs 2PL and that CCMA and CCMB are either 2PL or OPT. However, as we will next show, the algorithm may not work under all situations. For example, the algorithm fails if either CCMA or CCMB uses basic timestamp ordering (BTO) [Ree78]. Consider the following schedule

```
T1A                 T1B                 T2A
------------------------------------------

                                        R(X)
R(X)
W(X)
R(Y)
W(Y)
Commit
                    R(Y)
                    W(Z)
                    Commit
                                        R(Z)
                                        Commit
```

It is clear that the above schedule is not serializable (due to the $T2A \rightarrow T1A \rightarrow T1B \rightarrow T2A$ cycle). However, it would be allowed by a combination of 2PL-based MCC and BTO-based CCMA and CCMB if the timestamp of T2A is smaller than T1A (that is, T2A is older than T1A). CCMA permits this schedule assuming that T2A would be serialized before T1A, whereas MCC, on the other hand, permits the schedule assuming that T1A is serialized before T2A. Therefore, the order of serialization at MCC and CCMA are different and our original algorithm has to be modified to ensure that this does not happen.

To rectify the problem, the MCC needs to ensure that if the CCM serializes T_i before T_j, then it also serializes them in the same order. We need to first introduce the following definitions before we can describe how the MCC ensures the above condition.

Definition 2.1 *The CCM assigns a distinct* **serialization number** *to all transactions in its class such that for any pair of transactions T_i and T_j, if m and n indicate the serialization numbers of T_i and T_j respectively, then the following holds: $m < n \Rightarrow \exists$ a serial schedule of transactions in its class in which T_i is serialized before T_j.*

We will now introduce the notion of serialization functions, which is similar to the notion of serialization function and serialization events [Meh92, Elm90].

Definition 2.2 *Let T be a set of transactions and $S(T)$ be an equivalent serial schedule produced by a DBMS employing precisely one concurrency control policy, say, A. A* **serialization function** *for the concurrency control policy A,* ser, *is a function that maps every transaction in T to one of its operations such that for any pair of transactions $T_i, T_j \in T$, if T_i is serialized before T_j in $S(T)$, then $ser(T_i)$ is executed before $ser(T_j)$.*

Definition 2.3 *The* **serialization time stamp** *(STS) of T_i is the time at which $ser(T_i)$ for the concurrency control policy at MCC is received by the MCC.*

For each transaction, the MCC maintains its *serialization time stamp* (STS). For BTO, the function that maps every transaction to its begin operation is a serialization function. For strict 2PL and OPT, the function that maps every transaction to its commit operation is a serialization function. There exist certain concurrency control policies like Serialization Graph Testing (SGT) that do not have serialization functions but can be introduced externally [Geo91]. However, the STS is maintained only to easily find out the set of transactions which are serialized by the MCC before T_i and the set of transactions which are serialized by the MCC after T_i. This can be done for SGT without explicitly having a serialization function. All transactions whose STS is smaller than $STS(T_i)$ are serialized before T_i; i.e., they are senior to T_i. Similarly, all transactions with a larger STS than $STS(T_i)$ are serialized after T_i; i.e., they are junior to T_i.

To ensure that the MCC serializes transactions in the same order as the slave CCMs, we augment the interface presented by the CCMs to the MCC in the following fashion: Each CCM provides a function called Prepare_to_Commit. This function, when invoked with a transaction identifier, responds with an indication of whether or not the transaction can be committed. If it can be committed, the *serialization number* of the transaction (according to the CCM) is also returned. This number indicates the relative order of this transaction in an equivalent serial schedule of the transactions that have been submitted to this CCM.

In addition to STS, the MCC also maintains for each transaction a *low water mark* (LWM) and a *high water mark* (HWM). The LWM and HWM of T_i determines the range in which its serialization number, as returned by the CCM, can lie. When a transaction is initiated, its LWM and HWM are initialized to 0 and ∞ respectively. They are updated as follows whenever a transaction T_i commits:

- For all T_j such that T_j belongs to the same class as T_i and T_j is junior to T_i and T_j conflicts with T_i, set $LWM(T_j)$ = serialization number of T_i.
- For all T_j such that T_j belongs to the same class as T_i and T_j is senior to T_i, set $HWM(T_j)$ = serialization number of T_i.

The MCC allows a transaction to be committed only if its serialization number, as returned by the CCM, lies between its LWM and HWM.

2.2 Local Copies Manager

Consider the case where the MCC uses OPT, and different flavours of 2PL are employed at CCMA and CCMB. As per OPT philosophy, all operations of transactions have to take place in their own local copies and at commit time they are validated and global copies are updated. The following set of operations describe problems that can arise if we are using OPT at MCC but transactions do not operate on local copies.

```
T1A                T1B
-----------------------

R(X)
W(X)
                   R(X)
                   W(X)
R(X)
```

Since T1A and T1B are the only transactions in their respective classes, CCMA and CCMB will agree to commit them. Assume that T1A reaches the validation phase at the MCC before T1B. In this situation, the MCC will abort T1B at this time. However, the second read by transaction T1A is now a "dirty read" since it reads the value written by T1B. Therefore, the execution of transaction T1A itself is not correct.

However, MCC itself does not operate on data directly and therefore we can not introduce local copies at the MCC. To rectify the above problem, we use the *local copies manager* (LCM) (see Figure 1), which interfaces between the buffer manager and the concurrency control managers. The LCM creates local copies for each item that is accessed by a transaction. At commit time, the local values of a transaction are written into the buffer manager by the LCM.

Correctness

Based on the above discussion, we now list the general characteristics of a database system implementing extensible concurrency control:

Property 2.1 *The characteristics are:*

- *Inter-Class conflicts alone are detected at the MCC.*
- *The operations of TiA are forwarded to CCMA at that point of time when it would have been actually performed in a system in which there is exactly one concurrency control manager (the one that is executing at the MCC) and all transactions from Class B and only TiA from Class A are executing.*
- *The serialization order at the MCC and at the slave CCMs are the same.*
- *The pre-conditions for the algorithm used at the MCC to produce conflict serializable schedules are satisfied*[3].
- *The algorithm at the MCC generates only strict schedules [BHG87]*
- *All concurrency control algorithms used in the system produce only conflict serializable schedules.*

[3]Optimistic algorithms at MCC assumes the presence of LCM whereas BTO requires no LCM

Note that the above characteristics are independent of the actual CC algorithms used at the MCC and at the CCMs. We now prove that database systems which satisfy the above requirements are guaranteed to maintain database consistency.

We will prove that every schedule H produced by any combination of concurrency control algorithms at the MCC and the slave CCMs is serializable if the DBMS satisfies Property 2.1.

Lemma 2.1 *If there exists an edge $T_i \rightarrow T_j$ in the precedence graph corresponding to a schedule produced by any combination of CC algorithms at MCC and the slave CCMs, then $STS(T_i) < STS(T_j)$.*

Proof: We will assume $STS(T_i) > STS(T_j)$ and derive a contradiction.
Case 1 : T_i and T_j belong to the same class.
Case 1.1 : T_i commits before T_j

Since $STS(T_i) > STS(T_j)$, T_j is senior to T_i and hence $HWM(T_j)$ is set to the serialization number of T_i when T_i commits. Since $T_i \rightarrow T_j$, the serialization number of T_j returned by the CCM will be greater than that of T_i. So it exceeds $HWM(T_j)$ and hence the MCC will not allow it to commit.
Case 1.2 : T_j commits before T_i

We know that $LWM(T_i)$ will be set to the serialization number of T_j when T_j commits because by our assumption T_i is junior to T_j and T_i conflicts with T_j. But $T_i \rightarrow T_j$ and hence the serialization number of T_i returned by the CCM is smaller than that of T_j. It is less than $LWM(T_i)$ and MCC will not allow it to commit.

Case 2 : T_i and T_j belong to different classes.

By the definition of STS, if $T_i \rightarrow T_j$, then $STS(T_i) < STS(T_j)$ because interclass conflicts are regulated by the CC algorithm at the MCC, which will produce serializable schedules only. □

Theorem 2.1 *Every schedule H produced by any combination of concurrency control algorithms at the MCC and the slave CCMs is serializable if the DBMS satisfies Property 2.1.*

Proof: Suppose, by way of contradiction, that $SG(H)$ (the serialization graph corresponding to the schedule H) contains a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$, where $n > 1$. By Lemma 2.1 $T_1 \rightarrow T_2 \Rightarrow STS(T_1) < STS(T_2)$. Hence, we have $STS(T_1) < STS(T_2) < \ldots < STS(T_n) < STS(T_1)$ which is impossible. Thus $SG(H)$ has no cycles and hence by the Serializability Theorem [BHG87], H is serializable. □

3 Model and Methodology

To evaluate the effectiveness of the MCC concept, we developed a performance model of a centralized database system that caters to real time and standard applications. In this section, we describe the database model and discuss the experimental methodology and performance metrics used in our study.

3.1 DBMS Model

The organization of our database model is based on the database model of [Har91] and is shown in Figure 2. There are six components in this model, four of which represent the database system itself, while the remaining two capture the applications utilizing the database services. The components, each of which is realized by a separate module in the simulator, are the following:

- **Database**, which models the data and its layout
- **Source**, which generates the transaction workload
- **Transaction Manager**, which models the execution of transactions
- **Resource Manager**, which models the hardware resources
- **Master Concurrency Controller (MCC)** , which controls global access to shared data
- **Concurrency Control Managers (CCM)**, which control access to shared data for specific transaction classes
- **Sink**, which receives exiting transactions

The interaction between these modules primarily consist of service requests and completion replies as shown in Figure 2. In the remainder of this section, we describe each of these modules in detail.

3.1.1 Database

The database is modeled simply as a collection of pages, and all database operations are modeled at the page level of granularity. For example, CPU and disk costs for processing the data are modeled on a per page basis. The database for real time and standard transactions can be identical, partially overlapping or completely disjoint. This is determined by the *Overlap* parameter. A parameter called *DatabaseSize* specifies the number of pages in the database for real time transactions as well as the number of pages in the database for standard transactions. The data itself is modeled as being uniformly distributed across all of the disks.

3.1.2 Source

The source component represents the applications utilizing the database services. In our current implementation, it consists of two sub modules, one generating standard transactions and the other generating real-time transactions. Table 1 summarizes the values of key workload parameters which were kept constant during our experiments.

The real-time workload generator is modeled as an open system and transactions are generated in a Poisson stream at a mean rate specified by the *ArrivalRate* parameter. The standard workload generator is modeled as a closed system where the system load is dictated by the *MPL*(Multiprogramming level) parameter. MPL specifies the number of active transactions (standard) in the system at any given instant.

Each real time transaction consists of a sequence of pages to be read, a subset of which are also updated. The range of transaction sizes for real time transactions, measured in terms of the number of pages that they access, is determined by the *RT-MeanTransSize* and *SizeSprd* parameters; real time transaction sizes range between $(1 - SizeSprd) * RTMeanTransSize$ and $(1 + SizeSprd) * RTMeanTransSize$ pages. The number of pages accessed by a transaction varies uniformly between these limits. Page requests are generated by randomly sampling (without replacement) from the entire database, that is, over the range $(1, DatabaseSize)$. A page that is read is updated with probability *RTWriteProb*. Therefore, a page write operation is always preceded by a read for the same page; this means that the write set of a transaction is a subset of its read set and that there are no “blind writes” [BHG87]. A transaction that is restarted follows the page access sequence of the original transaction. The standard transactions are also similar in their

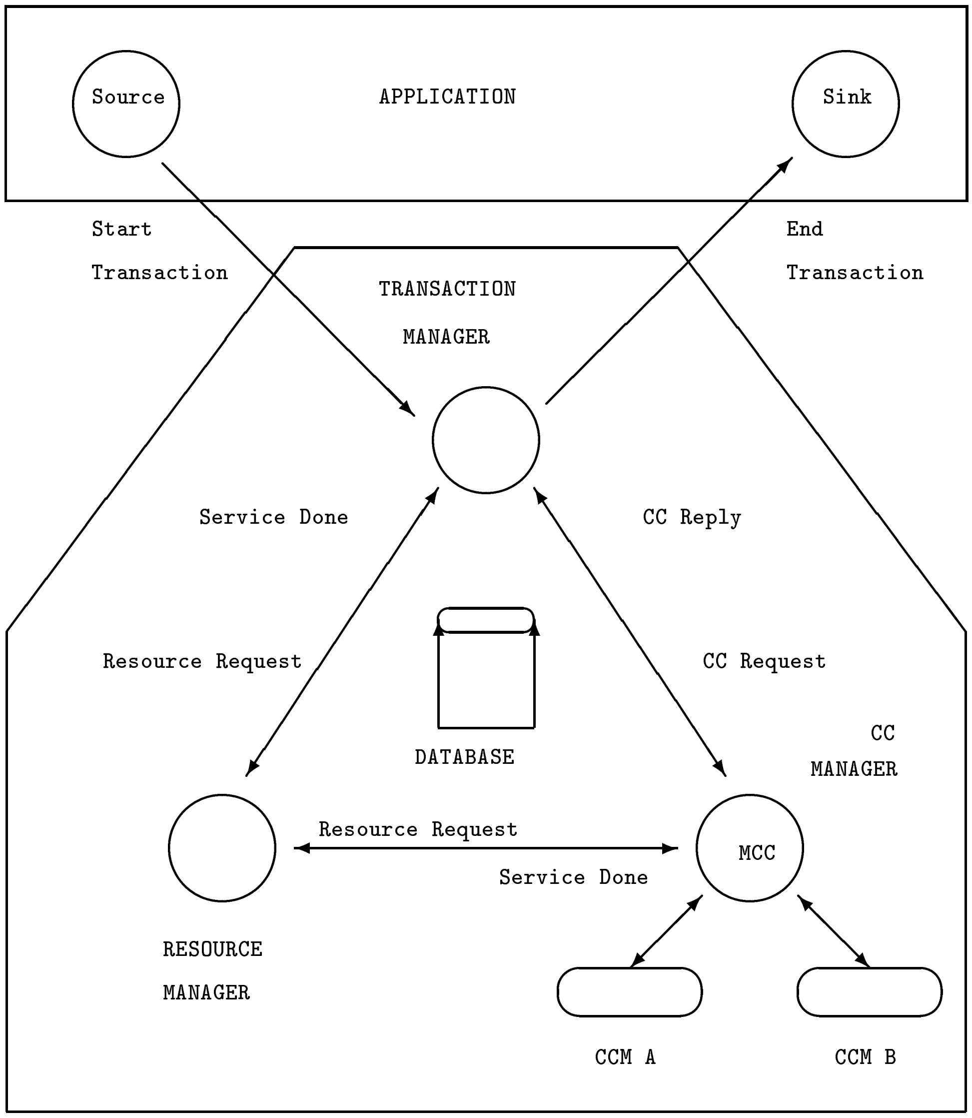

Figure 2: Database Model

DatabaseSize	100 pages
PageCPU	10ms
PageDisk	20ms
MeanRTTransSize	8 pages
MeanSTDTransSize	4 pages
SizeSprd	0.5
RTWriteProb	0.8
STDWriteProb	0.25
NumCPU	5
NumDisk	10
SlackFactor	4

Table 1: Fixed Parameter Settings

access pattern except their mean size is determined by *STDMeanTransSize* and the a page that is read is updated with probability *STDWriteProb*. Further, the page requests for standard transactions are generated uniformly between the range $((1 - Overlap) * DatabaseSize, (2 - Overlap) * DatabaseSize)$. Thus, the database size for both real time and standard transactions is equal to the *DatabaseSize* parameter but the *Overlap* parameter determines the extent of common data accessed by the transactions of the two classes.

Each real-time transaction has an associated deadline. The following formula is used to assign the deadlines:

$$D_T \quad = \quad A_T + SF * R_T$$

where D_T, A_T and R_T denote the deadline, arrival time and resource time of transaction T, respectively. The resource time of a transaction is the total service time at the resources that it requires, that is, it is the response time of the transaction if it were to execute *alone* in the system. It is computed using the following expression $R_T = NumReads_T * (PageCPU + PageDisk) + NumWrites_T * PageCPU$; where $NumReads_T$ and $NumWrites_T$ are the number of pages that are read and updated by the transaction, respectively. The disk time for writing updated data pages is not included in the resource time computation since these writes occur after the transaction has committed. [4]

The *slack ratio* of a transaction is defined to be the ratio$(D_T - A_T)/R_T$. The physical interpretation of this ratio is that it is the number of completion opportunities provided to the transaction by the application. (A slack ratio of less than 1 implies that it is impossible for the transaction to complete before its deadline.) In the above deadline formula, all transactions have the same slack ratio and is equal to the slack factor SF. The setting of SF determines the tightness/slackness of transaction deadlines.

3.1.3 Transaction Manager

All transactions generated by the Source are delivered to the Transaction Manager. The Transaction Manager controls the execution of transactions. It assigns a priority to each transaction upon arrival. For real time transactions, the **Earliest**

[4] We assume sufficient buffer space to allow the retention of updates until commit time. In addition, we assume the use of a log-based recovery scheme where only log pages are forced to disk prior to commit.

Deadline priority mapping is used which assigns higher priority to transactions with earlier deadlines. All standard transactions have less priority than that of any of the real time transactions.

Transactions that complete are marked as such and forwarded to the Sink module. The transaction manager executes each standard transaction to completion. Real-time transactions, on the other hand, are "killed" if they do not manage to complete before their deadlines expire. Killing a transaction consists of aborting its execution, marking it as killed and forwarding it to the Sink module.

As described earlier, each transaction execution consists of a sequence of read and write page accesses. For a read page access, the Transaction Manager requests access permission from the Concurrency Control Manager. When permission is received, the Transaction Manager requests the Resource Manager to read the corresponding disk page into memory. After the page has been read in, the Transaction Manager requests CPU time to process the page from the Resource Manager. When page processing is complete, the Transaction Manager then begins executing the next page access. A write page access is executed in similar fashion to a read page access. When all the page accesses of a transaction have been completed, the Transaction Manager initiates commit processing for the transaction.

Transactions may sometimes have to be aborted due to data conflicts. In this case, the Concurrency Control Manager decides that the transaction should be aborted and informs the Transaction Manager. The Transaction Manager then invokes the abort procedure for the transaction. After the abort procedure is completed, the transaction is restarted and follows the same data access pattern as the original transaction. Real-time transactions are also aborted when they are killed due to missing their deadlines. In this case, the Transaction Manager is the initiator of the abort process, and the aborted transaction is not restarted but sent, instead, to the Sink.

3.1.4 Resource Manager

The Resource Manager represents the operating system and controls access to the physical resources of the database system. In our model, the physical resources consist of a NumCPU CPUs and NumDisk disks. The CPUs have a common queue while each disk has a separate queue. These queues are served according to a Head-Of-Line (HOL) policy, with the request queue being ordered by transaction priority. Table 1 summarizes the key parameters of the resource model – the NumCPUs and NumDisks specify the hardware resource composition, while the PageCPU and PageDisk parameters capture CPU and disk processing times per data page.

Memory buffer resources are an integral feature of database systems. For the sake of simplicity, however, we assume that all data is accessed from disk and buffer pool considerations are therefore ignored. While modeling buffering would certainly result in different absolute performance numbers, we do not expect that doing so would significantly alter the general conclusions of our study. We also have not modeled the local copies manager for the same reasons as above.

3.1.5 Concurrency Controllers

The Master Concurrency Controller maintains database consistency by regulating global transaction access to data pages. It implements a concurrency control protocol, servicing concurrency control requests received from the Transaction Manager. The basic set of requests are:

- Begin_Trans(trans_class,trans_id)
- Commit_Trans(trans_class, trans_id)

- Abort_Trans(trans_class, trans_id)
- Access_Item(trans_class, trans_id,page_no,operation,parameters)

Based on the transaction's class, the MCC routes these requests to the appropriate CCM. In addition, it also invokes the following function for a transaction that has finished its data processing and requests permission to commit (as explained in Section 2):

- Prepare_to_Commit(trans_class,trans_id,serialization_number)

We have currently implemented two slave concurrency control managers: 2PL and OPT-BC. The 2PL algorithm is the classical two-phase locking algorithm, while OPT-BC is the *broadcast commit* flavour of the classical optimistic concurrency control algorithm [Rob82]. 2PL is the preferred algorithm for standard transactions while OPT-BC is suited for real-time transactions. For the MCC, the same two algorithms (2PL and OPT-BC) have been implemented.

We note here that specialized optimistic algorithms called OPT-WAIT and WAIT-50 have been described in [Har91] and shown to provide better real-time performance than OPT-BC; however, at this stage, we have only implemented the latter. In our future work, we plan to include OPT-WAIT and WAIT-50 in our suite of CC algorithms.

3.1.6 Sink

The Sink module receives both completed and killed transactions from the Transaction Manager. It gathers statistics on these transactions and measures the performance of the system from the perspective of the various applications using the system.

3.2 Performance Metrics

The performance metric for standard transactions is *throughput*. In contrast, the performance metric for real time transactions is *MissPercent*, the steady-state percentage of input transactions that miss their deadlines. All simulation experiments were run until steady-state performance was observed – only statistically significant differences are discussed here. The simulator was instrumented to generate a host of other statistical information including resource utilizations, number of transaction restarts, etc. These secondary measures help to explain the performance behaviour of the database architectures under various workloads and system conditions.

4 Experimental Results

The simulation model described in Section 3 has been coded in the C programming language. In this section we report the results of a preliminary set of experiments that were conducted on this simulator.

A mixed workload that consisted of real time and standard transactions was generated according to the class characteristics described in Section 3. On this workload, we consider the performance of DBMS architectures based on the following four algorithms:

1. 2PL: Here, the 2PL concurrency control mechanism is used for both standard and real-time transactions. This simulates a database system that supports only two phase locking.

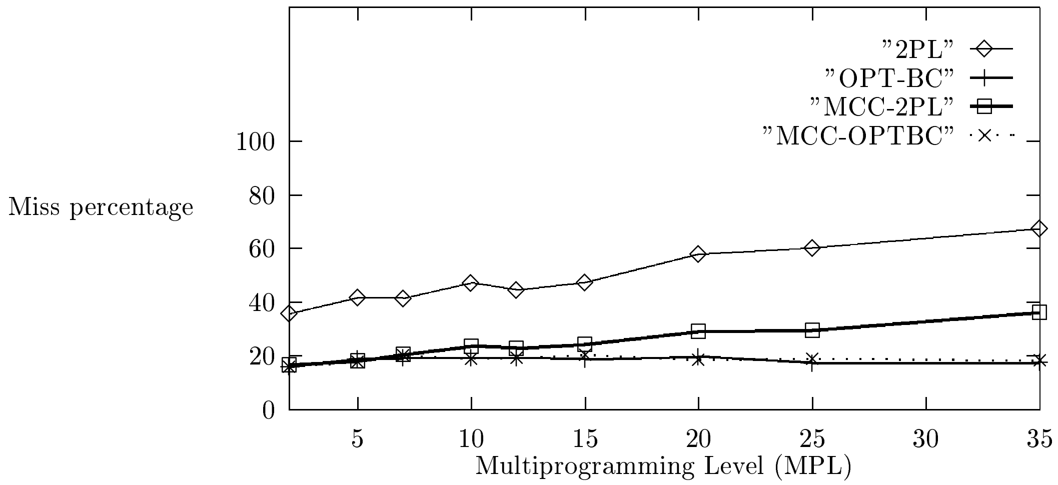

Figure 3: Miss Percentage Vs. MPL (20% Overlap)

2. OPT-BC: Here, OPT-BC is used for both real time and standard transactions. This simulates a database system that supports only optimistic concurrency control.

3. MCC-2PL: Here, 2PL is used at the MCC. The slave CCM for real time transactions uses OPT-BC while the slave CCM for standard transactions uses 2PL.

4. MCC-OPTBC: Here, OPT-BC is used at the MCC. The slave CCM for real time transactions uses OPT-BC while the slave CCM for standard transactions uses 2PL.

We report here on two sets of experiments, one in which there is low overlap of data between the standard and real-time transactions and the other in which there is medium overlap. In these experiments, the arrival rate of real time transactions was kept constant at 10 transactions per second and the multiprogramming level of the standard transactions was varied from 2 to 35. These settings maintained the resource utilization of the real-time class at approximately 50 percent, thereby ensuring that the performance effects of both transaction classes were highlighted.

4.1 Low Overlap

In this set of experiments, the database overlap between the standard and real-time transactions was set to 0.2. In Figure 3, the percentage of real time transactions that miss their deadline as a function of the multiprogramming level of standard transactions is shown. We observe here that the miss percentages of OPT-BC and MCC-OPTBC do not vary much as the MPL is varied. This is because the real time transactions have priority at the resources and therefore, in a sense, do not "see" the standard transactions. Data conflicts with standard transactions may result in restarts for real time transactions but there are sufficient resources (further boosted by the priority) for these transactions to run again and complete. In contrast, the miss percentage of real time transactions for 2PL and MCC-2PL increases consistently with MPL. This happens due to the increased data conflicts at higher MPL. Increased data contention results in significant blocking for the real

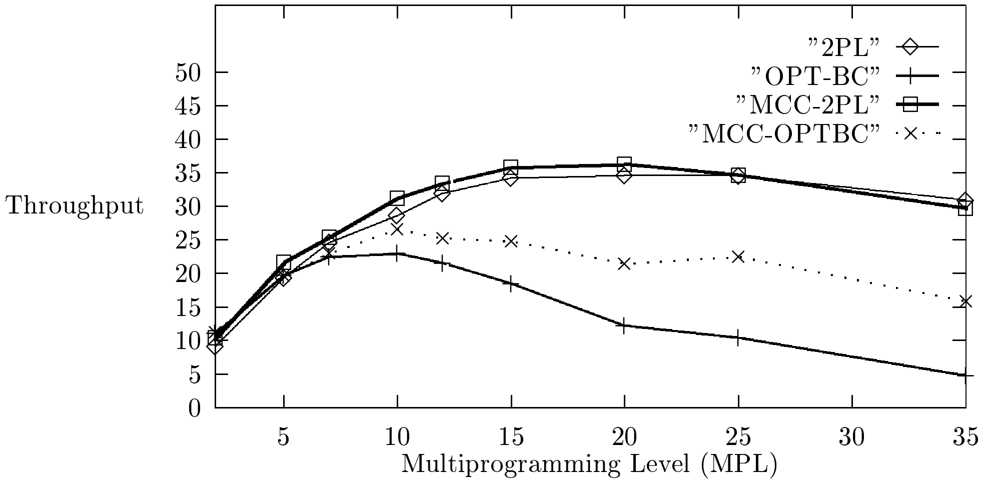

Figure 4: Throughput Vs. MPL (20% Overlap)

time transactions and hence more missed deadlines. MCC-2PL is somewhat better than 2PL since only inter-class data conflicts result in waiting.

In Figure 4, the complementary performance picture of the throughput of standard transactions as a function of MPL is shown. We observe here that the throughput of 2PL and MCC-2PL are fairly close and are higher than that of OPT-BC and MCC-OPTBC. The reason for the inferior throughput of standard transactions for OPT-BC and MCC-OPTBC is their resource-wasting characteristic, which proves detrimental to performance under resource-limited conditions. However, MCC-OPTBC has a better throughput than OPT-BC since 2PL is used for standard transactions and this conserves resources and only inter-class data conflicts result in resource wastage.

4.2 Medium Overlap

In this set of experiments the database overlap of real-time and standard transactions was set at 0.5. Figure 5 shows the percentage of real time transactions that miss their deadline as a function of the multiprogramming level of standard transactions. The shape is very similar to the corresponding curves for overlap of 20%. The miss percentages are slightly higher than the 20% overlap case because of increased data conflicts.

Figure 6 shows the throughput of standard transactions as a function of the MPL. Here too, the behavior of the algorithms are similar to the corresponding curves for 20% overlap except that due to increased data conflict the throughput for all algorithms is reduced.

The above experiments indicate that supporting multiple CC policies using an MCC-based architecture provides better performance with regard to all metrics as compared to providing only a single concurrency control mechanism.

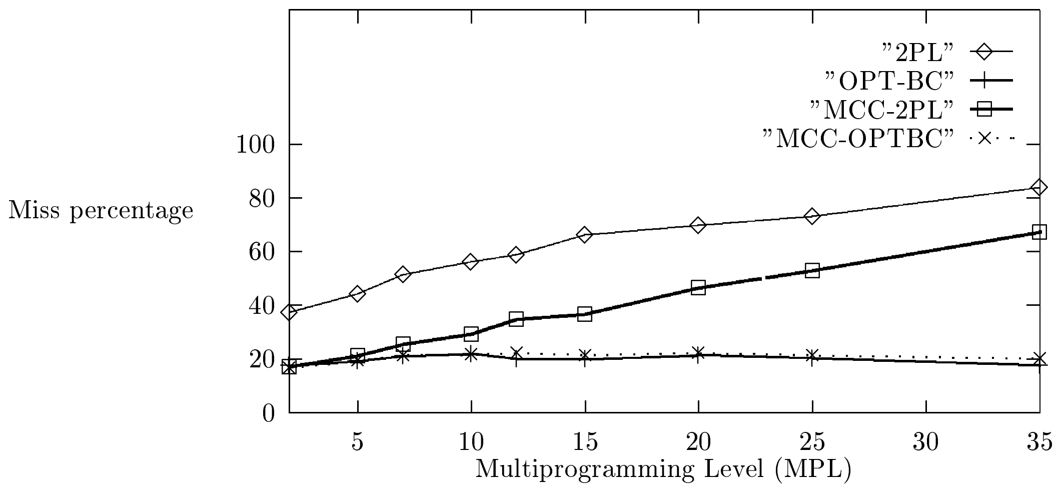

Figure 5: Miss Percentage Vs. MPL (50% Overlap)

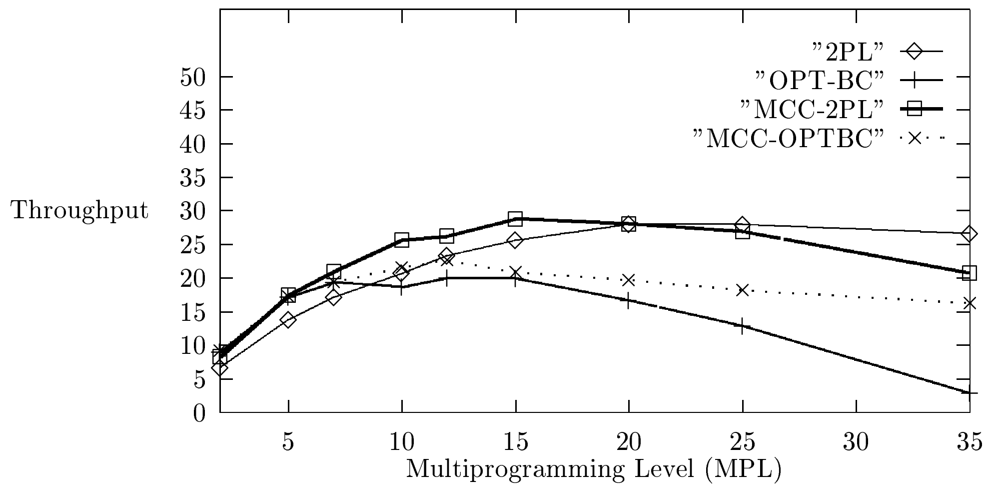

Figure 6: Throughput Vs. MPL (50% Overlap)

5 Conclusions and Future Work

In this paper, we addressed the issue of developing extensible concurrency control mechanisms. We outlined a design for a Master Concurrency Controller (MCC) that allows applications to choose from a suite of concurrency control algorithms at runtime. The MCC tries to maximize the performance benefits to the individual classes of applications. The design of the MCC has to be carefully done to ensure overall global serializability and retain data integrity.

In the context of a mixed workload of real time and standard transactions, we simulated the behaviour of a system that supports only one algorithm (either two phase locking or optimistic concurrency control) and a system that uses MCC and allows multiple concurrency control algorithms to co-exist. The performance metric for the real time transactions was the percentage of transactions that miss their deadlines while the performance metric for standard transactions was the throughput. Our preliminary experimental results indicate that the single-CC systems do well on (at most) one of the performance metrics but not the other (two phase locking has a good throughput but a high percentage of transactions that miss their deadlines while optimistic concurrency control meets the deadlines of most transactions but exhibits a poor throughput). In contrast, the MCC-based system is able to obtain a reasonable throughput as well as minimize the percentage of transactions that miss their deadlines.

The preliminary performance results are encouraging since they indicate that the overall performance of a system is enhanced by allowing applications to use the concurrency control algorithm of their choice. We are currently conducting extensive experimentation to confirm this feature under a wide range of workloads and operating conditions. The MCC design currently handles only conflict serializable CC algorithms. We are researching into the possibility of extending this idea to algorithms that are not conflict serializable.

References

[ACL87] R. Agrawal, M. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Trans. On Database Systems*, 12(4), December 1987.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BLW88] D. Batory, T. Leung, and T. Wise. Implementation concepts for an extensible data model and data language. *ACM Trans. on Database Systems*, September 1988.

[Car87] M. Carey ed. *Special Issue on Extensible Database Systems, Database Engineering*, June 1987.

[Car90] M. Carey et al. The exodus extensible dbms project: An overview. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases.* Morgan Kaufmann, 1990.

[Cat91a] R. Cattell. *Object Data Management: Object-oriented and Extended Relational Database Systems.* Addison-Wesley, 1991.

[Cat91b] R. Cattell. What are next-generation database systems? *Comm. of ACM*, October 1991.

[CH90] M. Carey and L. Haas. Extensible database management systems. *SIGMOD Record*, 19(4), December 1990.

[Cod70] E. F. Codd. A relational model for large shared data banks. *Comm. of ACM*, June 1970.

[Elm90] A. K. Elmagarmid and W. Du. A paradigm for concurrency control in heterogeneous distributed database systems. In *Proc. of Sixth Intl. Conf. on Data Engineering*, 1990.

[Geo91] D. Georgakopoulous, M. Rusinkiewicz and A. Sheth. On serializability of multidatabase transactions through forced local conflicts. In *Proc. of Seventh Intl. Conf. on Data Engineering*, 1991.

[Har91] J. R. Haritsa. *Transaction scheduling in firm real time database systems.* PhD thesis, Univ. of Wisconsin (Madison), 1991.

[HCL90] J. R. Haritsa, M. Carey, and M. Livny. On being optimistic about real-time constraints. In *Proc. of 1990 ACM Symp. on Principles of Database Systems*, April 1990.

[Hei87] S. Heiler et al. An Object-Oriented Approach to Data Management: Why Design Databases Need It. In *Proc. of 24th ACM/IEEE Design Automation Conference*, 1987.

[Kim88] W. Kim et al. Features of the orion object-oriented dbms. In W. Kim and E. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications.* Addison Wesley, 1988.

[KR88] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. In M. Stonebraker, editor, *Readings in Database Systems.* Morgan Kaufmann, 1988.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Comm. of ACM*, October 1991.

[LOL92] C. C. Low, B. C. Ooi, and H. Lu. H-trees: A dynamic associative search index for oodb. In *Proc. of ACM SIGMOD Conf.*, June 1992.

[Lyn83] N. A. Lynch. Multilevel atomicity – a new correctness criterion for database concurrency control. *ACM Trans. on Database Systems*, 8(4), December 1983.

[Meh92] S. Mehrotra, R. Rastogi, Y. Breitbart, H. F. Korth and A. Silberschatz. The concurrency control problem in multidatabases: characteristics and solutions. In *Proc. of ACM SIGMOD Conf.*, June 1992.

[Mol83] H. G. Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. on Database Systems*, 8(2), June 1983.

[Ree78] D. Reed. Naming and Synchronization in a Decentralized Computer System. *Ph.D. Thesis*, Massachusetts Institute of Technology, 1978.

[Rob82] J. T. Robinson. Design of Concurrency Control Protocols for Transaction Processing Systems. *Ph.D. Thesis*, Carnegie Mellon Univ., 1982.

[SK91] M. Stonebraker and G. Kemnitz. The postgres next-generation database management system. *Comm. of ACM*, October 1991.

[Wei88] W. E. Weihl. Commutativity based concurrency control for abstract data types. In *Proc. of 21st Annual Hawaii Intl. Conf. on System Sciences*, 1988.