INTEGRATING STANDARD TRANSACTIONS IN FIRM REAL-TIME DATABASE SYSTEMS

SHIBY THOMAS[1], S. SESHADRI[1] and JAYANT R. HARITSA[2]

[1]Department of Computer Science and Engineering, Indian Institute Of Technology, Bombay 400 076, INDIA
[2]Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore 560 012, INDIA

Abstract — Real-time database systems are designed to handle workloads where transactions have completion deadlines and the goal is to meet these deadlines. However, many real-time database environments are characterized by workloads that are a mix of real-time and standard (non-real-time) transactions. Unfortunately, the system policies used to meet the performance goals of real-time transactions often work poorly for standard transactions. In particular, optimistic concurrency control algorithms are recommended for real-time transactions, whereas locking-based protocols are suited for standard transactions. In this paper, we present a new database system architecture in which real-time transactions use optimistic concurrency control and, ***simultaneously***, standard transactions use locking. We prove that our architecture maintains data integrity and show, through a simulation study, that it provides significantly improved performance for the standard transactions without diminishing the real-time transaction performance. We also show, more generally, that the proposed architecture correctly supports the co-existence of any group of concurrency control algorithms that adhere to a standard interface.

Key words: Real-Time Database, Concurrency Control

1. INTRODUCTION

A real-time database system (RTDBS) is a transaction processing system that is designed to handle workloads where transactions have completion deadlines. The objective of the system is to meet these deadlines, that is, to process transactions before their deadlines expire. Research on real-time database systems has focussed almost exclusively on identifying the appropriate choice, with respect to meeting the real-time goals, for the various database system policies such as priority assignment, concurrency control, memory management, etc. (e.g. [4, 26, 22, 36]). In many real-time database environments, however, transaction workloads may have a mix of real-time and standard (non-real-time) transactions. For example, in a stock market environment, real-time transactions are submitted by brokers who are buying and selling shares and need their transactions to complete before the current share prices are updated. In contrast, standard transactions are executed by stock exchange officials running various types of accounting transactions. For such mixed workloads, the *throughput* of the standard transactions is an additional performance metric, and ideally, the RTDBS should meet the performance requirements of both (real-time and standard) transaction classes. However, to the best of our knowledge, no work has been reported so far with respect to designing such "integrated" real-time database systems. We address this issue here and present a system architecture that attempts to cleanly integrate standard transactions into the real-time environment.

The main problem in designing an integrated RTDBS is that the system policies used to meet the performance goals of real-time transactions often work poorly for standard transactions. This is especially true with regard to transaction concurrency control: Real-time variants of optimistic concurrency control, such as OPT-WAIT and WAIT-50 [19], provide significantly better real-time performance than locking-based algorithms in "firm deadline" RTDBS [21] (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). In contrast, locking protocols such as 2PL [15] and WDL [16] provide considerably more throughput than optimistic algorithms for standard transactions in conventional resource-limited DBMS [1, 16]. Therefore, in general,

standard transactions may expect to receive poor service from real-time database systems that have been designed with only real-time transactions in mind.

We focus here on designing a database system that allows each transaction class to execute under the concurrency control mechanism that is best suited for that class. We feel that such a facility would be necessary for a database system that aims to maximise the performance metrics of all classes *simultaneously*. In particular, while integrating standard transactions into a real-time database system, we would like to maximise the throughput of standard transactions to the extent possible while *not* affecting the performance of real-time transactions. At first glance, it may appear that there is a simple solution to the problem: Use optimistic algorithms for resolving conflicts among real-time transactions and use locking algorithms for resolving the conflicts among standard transactions. However, this scheme will not maintain data integrity if there is any overlap between the data accessed by real-time transactions and the data accessed by standard transactions. This is because data conflicts that occur *between* real-time transactions and standard transactions will not be detected by either concurrency control policy. Therefore, an explicit integration mechanism is required to coordinate the activities of the various concurrency control mechanisms.

In this paper, we present a new database architecture that allows different transaction classes to choose from multiple concurrency control schemes and execute simultaneously without loss of 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 **(CCMs)**. The MCC routes the data requests of each transaction to the appropriate CCM (that is, to the CCM assigned to the class to which the transaction belongs). Data conflicts between transactions belonging to the same class, that is, *intra-class* conflicts, are handled by the CCM associated with that class[†]. On the other hand, data conflicts between transactions belonging to different classes, that is, *inter-class* conflicts, are handled by the MCC. We prove that our design of this two-level conflict resolution scheme satisfies the conflict serializability criterion of database correctness.

Apart from maintaining consistency, the integration scheme provides improved performance for the standard transactions and does this at *no cost* to the RTDBS's primary concern: real-time performance. This was confirmed by a detailed simulation study of the MCC architecture implemented in a RTDBS whose input workload has both real-time transactions and standard transactions. In our experiments, which cover a range of workloads and system operating conditions, the real-time transactions used OPT-WAIT while the standard transactions used 2PL. The results of all these experiments indicate that allowing both concurrency control algorithms to "peacefully" co-exist by using MCC yields improved performance for the standard transactions at no expense to the real-time transactions.

A notable feature of our integration mechanism is that it is not tied to a specific set or class of concurrency control algorithms but is able to support *any* CC mechanism that adheres to a standard interface, that is, the algorithms are treated as "black boxes". This is especially important since new flavors of both real-time and conventional concurrency control protocols are constantly being proposed in the literature (e.g. [2, 3, 5, 31]) and RTDBS users may wish to take advantage of these new algorithms. Note also that this "open" feature of our architecture makes the integration scheme applicable in not just the real-time database context but in *any* environment where there are multiple transaction classes with different preferred CC algorithms. This aspect is described in more detail in [40].

Finally, we have also designed algorithms to implement recovery for our RTDBS architecture – the details are omitted here due to space limitations but are described in [41].

1.1. Related Work

From the above discussion, it is clear that supporting multiple CC algorithms at run-time is a complex problem and to the best of our knowledge, there have not been any prior efforts in this direction. On the surface, however, it may seem no different from the situation encountered

[†]If a CCM is chosen by more than one transaction class, then these classes are merged into a single class from the CCM's perspective.

in "multidatabase systems". In reality, though, there are differences: In a multidatabase system, though a transaction may be regulated by different CC policies (at different sites) during the course of its execution, access to a given data item is controlled by a *single* fixed CC policy (the policy operational at the site where the data item is located). In our scenario, however, we wish to allow for the same data item to be simultaneously under the regulation of *multiple* concurrency control policies. So, essentially, our problem is the *converse* of the multidatabase problem. Moreover, the problem we are addressing is valid even for single-site database systems.

As discussed above, our concerns are quite different from those encountered in multidatabase systems. However, we have used some of the ideas presented in the multidatabase literature with regard to maintaining serializability [34, 14] and adapted them to our situation. In fact, our architecture itself can be extended to multidatabase systems – the details are provided in [41].

1.2. Organization

The rest of this paper is organized as follows: We present, in Section 2, the design and proof of correctness of the Master Concurrency Controller. The specific implementation of the MCC concept in real-time database systems is detailed in Section 3. The performance model is described in Section 4, and the results of the simulation experiments are highlighted in Section 5. We conclude in Section 6 and outline future research avenues.

2. SYSTEM ARCHITECTURE

In this section, we describe our system architecture for integrating multiple concurrency control policies and prove that it maintains conflict serializability. We initially present a general integration mechanism and then, in Section 3, discuss the specific implementation to be used in real-time database systems. The performance behavior of the scheme is profiled in Section 5.

Our database system architecture 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. In this scheme, all transaction calls from the Transaction Manager are first sent to the Master Concurrency Controller (MCC) (which, as mentioned in the Introduction, is itself a complete concurrency control algorithm). The MCC then forwards each call to the "slave" concurrency control manager (CCM) associated with the transaction[†]. 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.

As mentioned in the Introduction, two types of data conflicts can occur in the above architecture: *intra-class* and *inter-class*. Intra-class conflicts are data conflicts that occur between transactions of the same class while inter-class conflicts are conflicts that occur between transactions of different classes. The MCC is responsible for handling all inter-class conflicts whereas intra-class conflicts are handled by the respective CCMs. This division of labor makes it necessary for the MCC to also implement a full concurrency control mechanism, thereby ensuring a consistent notion of database correctness.

2.1. Interface Functions

Since we wish to make the integration scheme independent of the implementation of the various concurrency control managers used in the database system, we require that they all adhere to a standard interface. In particular, we assume that the MCC and all CCMs support the following common interface functions: Begin_trans, Commit_Trans, Abort_Trans and Access_Item. The first three functions correspond to starting, committing and aborting a transaction, respectively. The Access_Item function refers to access of data items by a transaction. The arguments to this function specify the item being accessed and the mode of access (such as read or write). In addition to this set

[†]Any number of CCMs can be incorporated. For simplicity, however, we have shown only two CCMs in Figure 1.

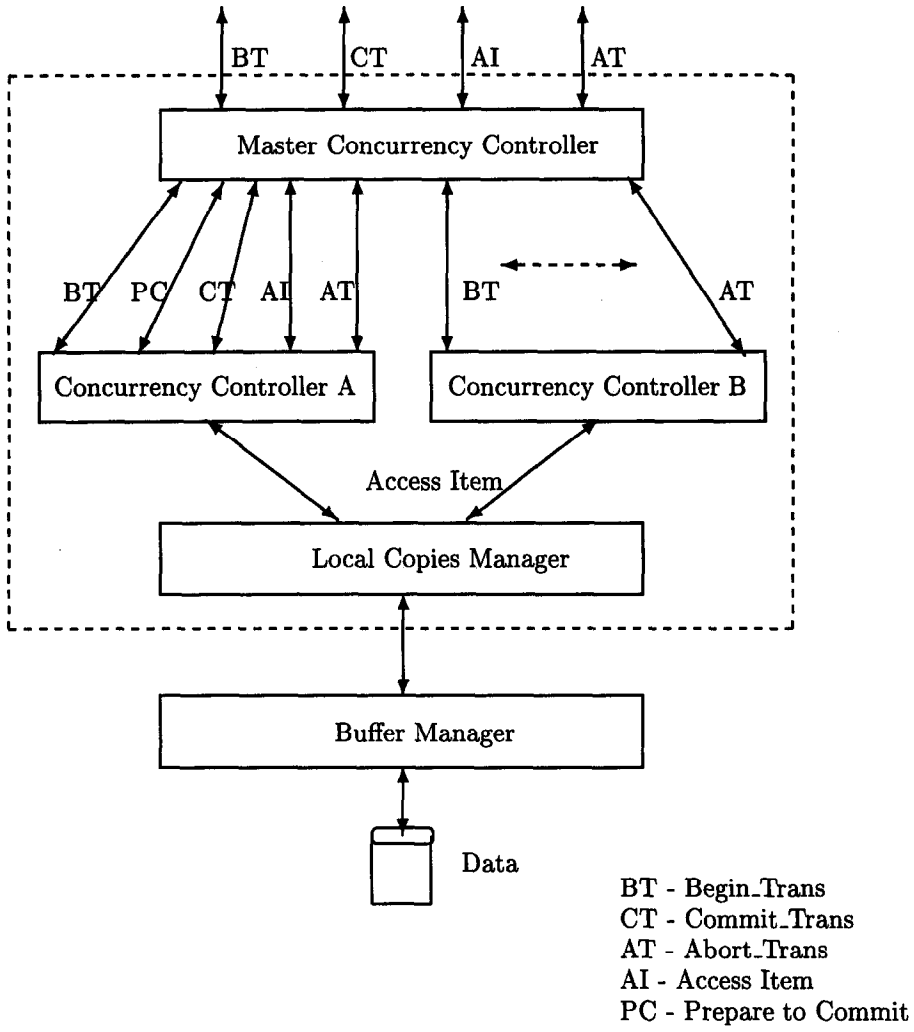

Fig. 1: System Architecture

of interface functions, the slave CCMs are required to support a function called Prepare_to_Commit. This function is elaborated upon later.

We also make the reasonable assumption that all the concurrency control algorithms used in the database system produce only conflict serializable schedules and that all schedules are strict [6]. Note that, apart from this requirement and the interface requirement, the concurrency control managers are effectively treated as "black boxes" by the MCC, thereby providing an open architecture.

2.2. Master Concurrency Controller

The MCC operates in the following manner: On receiving a Begin_Trans call, it performs any action the corresponding concurrency control algorithm demands 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 call is simply handled by aborting the transaction at the MCC and forwarding the abort request to the concerned CCM.

The Access_Item call is processed by first recording the call in the data structure appropriate to that used by the specific CC algorithm implemented at the MCC. (For example, if MCC were

using 2PL, the call would be recorded in a lock table.) Then, if the requested action does *not* result in any *inter-class* conflicts, the requested action is forwarded to the appropriate CCM. Otherwise, the MCC takes action (according to the CC algorithm followed by it) to handle the inter-class conflict. Once again, assuming the MCC were using 2PL, the call would be forwarded to the CCM if there were no inter-class conflicts caused by the call. Otherwise, the call would be ***blocked*** (i.e., not be forwarded to the CCM) until the conflicting transaction(s) terminate. Note that the MCC is able to discover inter-class conflicts because it records all the Access_Item calls that it receives.

In short, the behavior of the concurrency control algorithm implemented at the MCC is identical to that of the same concurrency control manager in a normal database system – the only difference is that it recognizes only inter-class conflicts and ignores intra-class conflicts.

In the above situation, where inter-class conflicts and intra-class conflicts are detected and resolved by the MCC and CCMs, respectively, the following problem may arise: The serialization order for the same set of transactions may be different at the MCC and the CCMs! This scenario is illustrated in the following example (we later explain how our MCC architecture solves this problem):

Example 1 Assume that there are two CCMs, A and B, implementing Basic Timestamp Ordering(BTO) [37] and Optimistic Concurrency Control, respectively, and that the MCC implements 2PL. Consider the following schedule, where T_A^1 and T_A^2 are transactions submitted to CCM-A and T_B is a transaction submitted to CCM-B:

T_A^1	T_A^2	T_B
Begin		
R(X)		
	Begin	
	R(X)	
	W(X)	
	R(Y)	
	W(Y)	
		Begin
		R(Y)
	Commit	
		R(Z)
		W(Z)
R(Z)		
		Commit
Commit		

It is clear that the above schedule is not serializable (due to the $T_A^1 \rightarrow T_A^2 \rightarrow T_B \rightarrow T_A^1$ cycle). It would be permitted, however, by a combination of 2PL-based MCC, BTO-based CCM-A and OPT-based CCM-B. This is explained as follows: CCM-A, which implements BTO, permits the schedule that it locally sees since T_A^1 is older than T_A^2; that is, it enforces a serial order in which T_A^1 is before T_A^2. On the other hand, MCC (which implements 2PL), permits the global schedule assuming that T_A^2 is serialized before T_A^1. Therefore, the serialization orders at MCC and CCM-A are different. □

Since different serialization orders could lead to loss of data consistency, we have devised the following scheme, called Global Ordering Scheme, to ensure that this does not happen.

2.3. Global Ordering Scheme

In this scheme, we ensure that a single serialization order is followed in the entire database system. This is done by requiring the CCMs to keep track of the local serialization order of transactions and to communicate this to the MCC whenever requested. The local CCMs communicate the local serialization order using the notion of serialization numbers which is defined shortly. The MCC then ensures that a single global order is maintained.

Definition 1 A *serialization number sequence*, with respect to a schedule S, for a set of transactions is a sequence of numbers 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 conflict equivalent to S, in which T_i finishes before T_j.

The local serialization order of transactions at a CCM is communicated to the MCC by augmenting 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 with respect to the global schedule restricted to transactions of that class alone is also returned by the CCM of that class. The serialization numbers returned by a CCM are distinct. In other words, serialization numbers are never reused by a CCM, irrespective of the outcome of a transaction.

The MCC invokes the function Prepare_to_Commit when it gets the Commit_Trans call from the Transaction Manager. The serialization number returned by the slave CCM is then checked for validity (explained shortly) and the transaction is committed or aborted based on the outcome of this test. We will need a few definitions before outlining the validity test.

Definition 2 Let S be a schedule of a set of transactions T produced by a DBMS employing precisely one concurrency control policy, say M. A *serialization function, ser,* for M is a function that maps every transaction in T to one of its operations such that the following holds: S is conflict equivalent to S', where S' is the serial schedule in which T_i occurs before T_j if and only if $ser(T_i)$ occurs before $ser(T_j)$ in S.

Our notion of serialization functions is similar to the notion of serialization function and serialization events [34, 14]. All concurrency control algorithms either inherently have serialization functions, or they can be introduced externally [17]. In particular, for locking protocols and optimistic algorithms, the function that maps every transaction to its commit operation is a serialization function, whereas for timestamp algorithms, the function that maps every transaction to the begin operation is a serialization function.

Definition 3 The *serialization time stamp* (STS) of T_i is the time at which $ser(T_i)$ is received by the MCC.

The MCC uses the STS to define the serialization order at the MCC while the CCMs communicate their local order through serialization numbers returned by the Prepare_to_Commit call. The MCC now ensures that the serialization orders at the MCC and the CCMs agree by maintaining the following data structures: The MCC maintains a *serialization time stamp* (STS) for each transaction. 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 determine 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 after T_i in MCC's serial order (i.e., $STS(T_i) < STS(T_j)$) and T_j conflicts with T_i, set $LWM(T_j) = \max(LWM(T_j),$ 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 prior to T_i in MCC's serial order, (i.e., $STS(T_i) > STS(T_j)$) set $HWM(T_j) = \min(HWM(T_j),$ serialization number of $T_i)$.

As promised earlier, we now describe the test used for checking the validity of the serialization number returned in response by a slave CCM to a Prepare_to_Commit call by the MCC. Based on the result of the test, the transaction is either committed or aborted.

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

We will now prove that the commit test ensures that a single global order is maintained.

Lemma 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 some value which is smaller than or equal 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 some value which is greater than or equal 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 inter-class conflicts are regulated by the CC algorithm at the MCC, which will produce serializable schedules only. □

2.4. Local Copies Manager

Most concurrency control algorithms make "in-place" updates to data items, that is, transaction write commands are applied directly to the data items themselves. This is true, for example, of both locking protocols and timestamp algorithms. However, a unique feature of algorithms based on optimistic concurrency control (OPT) is that they make updates to *private local copies* of data items and these private updates are made public only at transaction commit time. If such "private copy" algorithms are used at the MCC, it is possible to have situations wherein transactions perform "dirty reads" [18] even without violation of the rules of any of the concurrency control mechanisms. This situation is illustrated in the following example:

Example 2 Assume that there are two CCMs, A and B, implementing 2PL and BTO, respectively, and that the MCC implements OPT. Consider the following schedule, where T_A and T_B refer to transactions submitted to CCM-A and CCM-B, respectively:

T_A	T_B
Begin	
R(X)	
W(X)	
	Begin
	R(X)
	W(X)
R(X)	

Since T_A and T_B are the only transactions in their respective classes, there are no intra-class conflicts here. Assume that T_A reaches validation phase at the MCC before T_B. In this situation, the MCC, which detects the inter-class conflict, aborts T_B at this time and allows T_A to commit. However, the second read by transaction T_A is now a "dirty read" since it read the value written by T_B. Therefore, it was incorrect to commit transaction T_A. □

The problem is a direct consequence of the fact that we did not satisfy all the preconditions of the algorithm at the MCC which in this case means that OPT expects transactions to write to

local copies whereas the system was not doing so. To address this problem, a *local copies manager* (LCM) (see Figure 1), which interfaces between the buffer manager and the concurrency control managers, is introduced into our architecture. 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. Note that the LCM is needed only if OPT is employed at the MCC, but not for 2PL or BTO.

2.5. Summary

To summarize the above description, we list below the main characteristics and assumptions of our architecture for integrating multiple concurrency control mechanisms.

Property 1 *The characteristics are:*

- *All concurrency control algorithms support the basic set of interface functions.*
- *All concurrency control algorithms produce only conflict serializable strict schedules.*
- *Inter-class conflicts are resolved at the MCC whereas intra-class conflicts are resolved at the corresponding CCMs.*
- *The serialization order at the MCC and at the slave CCMs are the same.*
- *Any preconditions required by the algorithm at the MCC have to be satisfied by the system.*

Note that the above characteristics are independent of the specific CC algorithms used at the CCMs.

2.6. Proof of Correctness

We now prove that database systems which satisfy the above requirements are guaranteed to maintain database consistency. In particular, we prove that every schedule S produced by any combination of concurrency control algorithms at the MCC and the slave CCMs is serializable if the DBMS satisfies Property 1.

Theorem 1 *Every schedule S produced by any combination of concurrency control algorithms at the MCC and the slave CCMs is serializable if the DBMS satisfies Property 1.*

Proof. Suppose, by way of contradiction, that $SG(S)$ (the serialization graph corresponding to the schedule S) contains a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$, where $n > 1$. By Lemma 1 $T_i \rightarrow T_2 \Rightarrow STS(T_1) < STS(T_2)$. Hence, we have $STS(T_1) < STS(T_2) < \ldots < STS(T_n) < STS(T_1)$ which is impossible. Thus $SG(S)$ has no cycles and hence by the Serializability Theorem [6], S is serializable. □

3. RTDBS IMPLEMENTATION

In the previous section, a general scheme for integrating multiple concurrency control algorithms was described. We move on in this section to considering the RTDBS context specifically wherein the workload is a mix of real-time and standard transactions. As discussed in the Introduction, the preferred concurrency control mechanisms for the real-time transactions and standard transactions are OPT-WAIT and 2PL, respectively†. We briefly describe below the essential features of these algorithms.

In the following description we assume that the RTDBS has a priority assignment mechanism for the real-time transactions (for example, *Earliest Deadline*, where transactions with earlier deadlines are assigned higher priority than transactions with later deadlines) and that resources are always assigned based on priority. We also assume that all standard transactions are assigned a lower priority than any real-time transaction.

†In [19], WAIT-50 was found to outperform OPT-WAIT – however, for ease of implementation and explanation, we have used OPT-WAIT in our experiments. The thrust of this paper is unaffected by the change.

3.1. 2PL

In the two-phase locking (2PL) protocol [15], transactions set read locks on objects that they read, and these locks are later upgraded to write locks for the objects that are updated. Multiple transactions can simultaneously share a read lock on a data item, but write locks are exclusive. If a lock request is denied, the requesting transaction is blocked until the lock is released. Locks obtained by a transaction during the course of its execution are held until the transaction commits, at which time it simultaneously releases all of its locks. Deadlocks are possible with the 2PL protocol, and these are resolved by restarting one of the transactions in the cycle of waiting transactions.

3.2. OPT-WAIT

In classical optimistic concurrency control (OPT) [27], transactions read and update data items freely, storing their updates into a private workspace. These updates are made public at commit time. Before a transaction is allowed to commit, however, it has to pass a validation test. This test checks that there is no conflict of the validating transaction with transactions that committed since it began execution. The validating transaction is restarted if it fails this test.

A variant of the above algorithm incorporates the *Broadcast Commit* scheme[†] suggested in [33, 38]. Here, when a transaction commits, it identifies other currently executing transactions that it conflicts with and these conflicting transactions are immediately restarted. Note that there is no need to check for conflicts with already committed transactions since any such transaction would have, in the event of a conflict, already restarted the validating transaction at its (the committed transaction's) own earlier commit time. This also means that a validating transaction is always guaranteed to commit. The broadcast commit variant detects conflicts earlier than the classical OPT algorithm, resulting in fewer wasted resources and earlier restarts. In the rest of this paper, we will refer to this variant as the basic OPT algorithm.

The OPT-WAIT algorithm [19] is a real-time variant of the OPT protocol, which aims to meet more transaction deadlines by preferentially serving urgent transactions. It incorporates a *priority wait* mechanism: A transaction that reaches validation and finds higher priority transactions in its conflict set is "put on the shelf", that is, it is forced 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, it is possible that it is restarted due to the commit of one of the conflicting higher priority transactions.

3.2.1. MCC

In principle, we could choose any CC algorithm for the Master Concurrency Controller. However, we have deliberately chosen OPT-WAIT for the MCC also. This choice was dictated by our desire to ensure that the real-time performance of the RTDBS should not suffer in an attempt to improve the performance of standard transactions. Using OPT-WAIT at the MCC ensures that the real-time transactions, by virtue of their higher priority, never "see" the standard transactions (since all inter-class conflicts are resolved in favor of the real-time transactions). At the physical resources also, since they are assigned on a priority basis, the real-time transactions are virtually unaffected by the standard transactions[‡]. Therefore, the performance of the real-time transactions is practically unaltered with respect to the original RTDBS.

We conducted experiments to evaluate the performance of the MCC implementation described

[†]The Broadcast Commit scheme is also sometimes referred to as Forward optimistic concurrency control [23].

[‡]In our simulation model, the processors and disks are scheduled on a priority head-of-line basis. Since this discipline is non-preemptive, real-time transactions which arrive when a standard transaction has already acquired a resource have to wait for that one transaction to complete before seizing the server. So, in that sense, the real-time transactions are indeed blocked by standard transactions. However, since the service times of requests are typically small and since, more often than not, real-time transactions arrive when other real-time transactions are being served, this has a negligible performance impact. Note that while CPUs are typically preemptible, all currently available disks are non-preemptive devices.

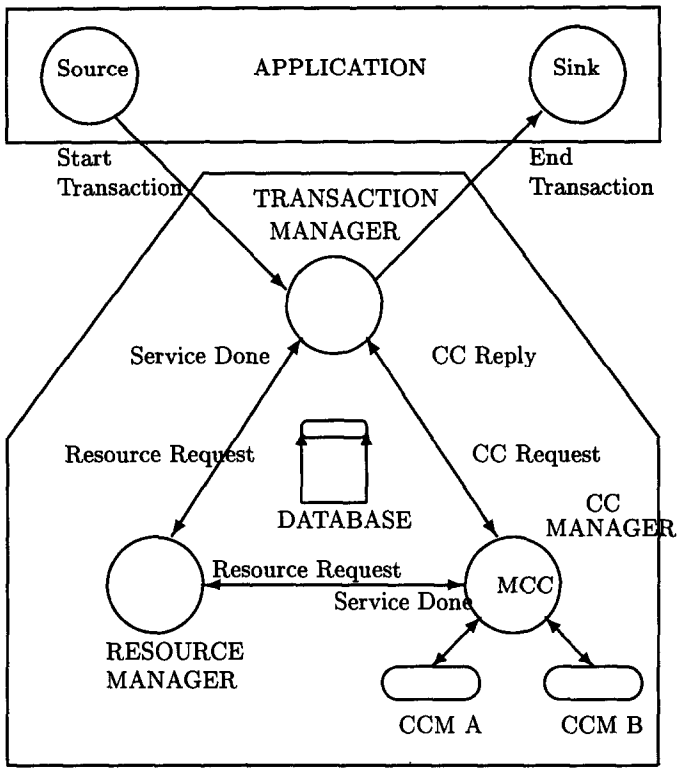

Fig. 2: Database Model

here, and the following sections describe the experimental framework and the results of the experiments.

4. 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.

4.1. DBMS Model

The organization of our database model is based on the database model of [24] 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 (CCMs)**, 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. The list of parameters that we use in our model is shown in Table 1. In the remainder of this section, we describe each of these modules in detail.

4.1.1. 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 parameter *DatabaseSize* specifies the number of pages in the database that are accessible by the real-time transaction class and by the standard transaction class. The databases for real-time and standard transactions can be completely disjoint, partially overlapping or completely identical. The amount of overlap is determined by the *Overlap* parameter. The data itself is modeled as being uniformly distributed across all the system disks.

4.1.2. 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.

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 with a constant population. The number of transactions in the standard population is specified by the *MPL*(Multiprogramming level) parameter.

Parameter	Meaning
DatabaseSize	Number of database pages per class
Overlap	Overlap of class databases
Arrival rate	Real-time transaction arrival rate
MPL	Multiprogramming level of standard transactions
RTMeanTransSize	Average real-time transaction size (in pages)
STDMeanTransSize	Average standard transaction size (in pages)
SizeSprd	Spread in transaction size
RTWriteProb	Write probability per accessed page for real-time transactions
STDWriteProb	Write probability per accessed page for standard transactions
SlackFactor	Deadline tightness factor
NumCPU	Number of processors
NumDisk	Number of disks
PageCPU	CPU service time per data page
PageDisk	Disk service time per data page

Table 1: System and Workload Parameters

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 *RTMeanTransSize* 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" [6]. A transaction that is restarted follows the page access sequence of the original transaction.

The standard transactions are similar to the real-time transactions in their access pattern except that their mean size is determined by *STDMeanTransSize* and their page update probability is given by *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 shared data.

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

$$D_T = A_T + SF * R_T$$

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, and $PageCPU$ and $PageDisk$ are the CPU and disk processing times per data page. 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. †

The slack factor is a constant that provides control over the tightness/slackness of transaction deadlines. Its value is set by the SF workload parameter.

4.1.3. 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 the real-time transactions, the transaction priority assignment scheme is *Earliest Deadline*, which assigns higher priority to transactions with earlier deadlines. All standard transactions have the same priority and this value is set to be less than that of any of the real time transactions (to reflect our goal of not degrading the performance of 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 Master Concurrency Controller. 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.

† 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.

Transactions may sometimes have to be aborted due to data conflicts. In this case, the CCMs (for intra-class conflicts) and the MCC (for inter-class conflicts) decide that the transaction should be aborted and inform 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.

4.1.4. Resource Manager

The Resource Manager represents the operating system and controls access to the physical resources of the database system. The physical resources in our model consist of multiple CPUs and multiple 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 *NumCPU* and *NumDisk* parameters 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.

4.1.5. 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)

Two slave concurrency control managers, 2PL and OPT-WAIT, are implemented in our system. OPT-WAIT is used by the real-time transactions whereas the standard transactions use 2PL. The MCC also implements the OPT-WAIT algorithm. For the 2PL CCM, deadlock detection is executed by maintaining a waits-for graph and checking for cycles. The transaction whose data request caused the deadlocked cycle to form is chosen as the victim to be aborted.

4.1.6. 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.

4.2. Performance Metrics

The performance metric for the standard transactions is their *throughput*. In contrast, the performance metric for the 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.

5. EXPERIMENTAL RESULTS

The simulation model described in Section 4 was implemented in the C programming language. In this section, we report on the results of a wide range of experiments that were conducted on this simulator.

A mixed workload that consisted of real-time and standard transactions was generated according to the workload characteristics described in Table 2. On this workload, we consider the performance of DBMS architectures based on the following three algorithms:

DatabaseSize	1000 pages
Overlap	25%
ArrivalRate	7 trans/sec
MeanRTTransSize	8 pages
MeanSTDTransSize	8 pages
SizeSprd	0.5
RTWriteProb	0.5
STDWriteProb	0.5
SlackFactor	3
NumCPU	1
NumDisk	2
PageCPU	10ms
PageDisk	20ms

Table 2: Baseline Parameter Settings

In all the three algorithms, the priority for real-time transactions is assigned as per the earliest deadline policy and the priority of standard transactions is less than the priority of all real-time transactions. The priority of all standard transactions is identical.

1. Pure OPT-WAIT: Here, OPT-WAIT is used for both real-time and standard transactions. The performance of standard transactions in this system is representative of what might happen if standard transactions were submitted to a typical real-time database system.
2. Pure 2PL: Here, the 2PL concurrency control mechanism is used for both standard and real-time transactions. The performance of standard transactions in this system is representative of the best performance that standard transactions could expect to achieve in a real-time database environment (assuming no other changes were made to the RTDBS).
3. MCC: Here, the RTDBS uses the MCC algorithm that was outlined in Section 3. The real-time transactions use OPT-WAIT, the standard transactions use 2PL, and the MCC itself uses OPT-WAIT.

We began our performance evaluation by first developing a baseline experiment. Further experiments were constructed around the baseline experiment by varying a few parameters at a time. In the following discussion we refer to the Pure 2PL, the Pure OPT-WAIT and the MCC architectures as 2PL, OPT-WAIT and MCC, respectively.

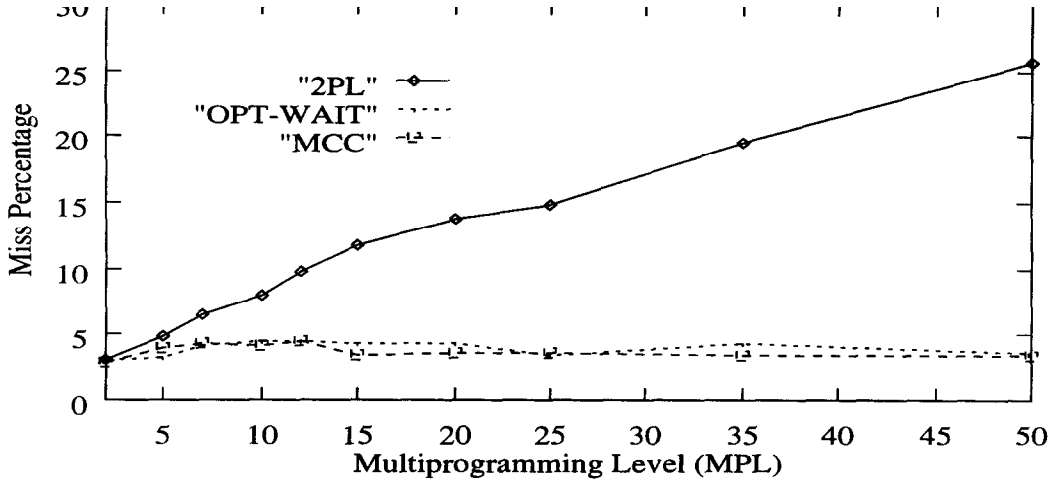

Fig. 3: Baseline Experiment: Miss Percentage Vs. MPL

5.1. Baseline Experiment

The settings of the workload and system parameters for the baseline experiment are listed in Table 2. These settings were chosen with the objective of having significant data and resource contention in the system, thus helping to bring out the performance differences between the various architectures. With these values, the transaction size distribution is the same for both real-time and standard transactions and the real-time transactions share 25% of their database with standard transactions (and vice versa). The transaction write probabilities model medium data contention, while the resource values model a limited resource environment. The arrival rate of real-time transactions is set to maintain their resource utilization to be approximately half the system capacity, thereby ensuring that the performance effects of both transaction classes are highlighted.

Figure 3 shows the percentage of real-time transactions that miss their deadline as a function of the multiprogramming level of standard transactions and in Figure 4, the complementary performance picture of the throughput of standard transactions as a function of MPL is shown. We will discuss the performance of each of the algorithms in turn.

- **OPT-WAIT:** The miss percent of OPT-WAIT remains virtually constant with increasing MPL. For OPT-WAIT, the real-time transactions have priority at the physical resources and the concurrency control manager and therefore, in a sense, do not "see" the standard transactions. This fact can be verified by looking at Figure 5 which shows the total and useful utilizations of resources[†] by the real-time transactions as a function of MPL for the three algorithms. The total utilization of real-time transactions for OPT-WAIT remains virtually constant with MPL confirming our claim that real-time transactions do not "see" the standard transactions. Further, useful utilization of real-time transactions is very very close to the total utilization due to the low miss percent.

- **2PL:** In contrast to OPT-WAIT, the miss percent increases with increasing MPL for 2PL. Figure 5 shows that the total utilization of real-time transactions for 2PL declines steadily with MPL which corroborates the miss percent figure. Although the real-time transactions do not "see" standard transactions at the resources, they may have to wait for standard transactions at the concurrency control manager (as there is no priority at the concurrency control manager). The wait increases with increasing MPL since the data conflicts increase and consequently the total utilization of real-time transactions declines with MPL.

[†]The system parameters are so chosen as to distribute the load evenly on the CPU and the disk. Hence, the utilization figures for both the disks and CPU are identical.

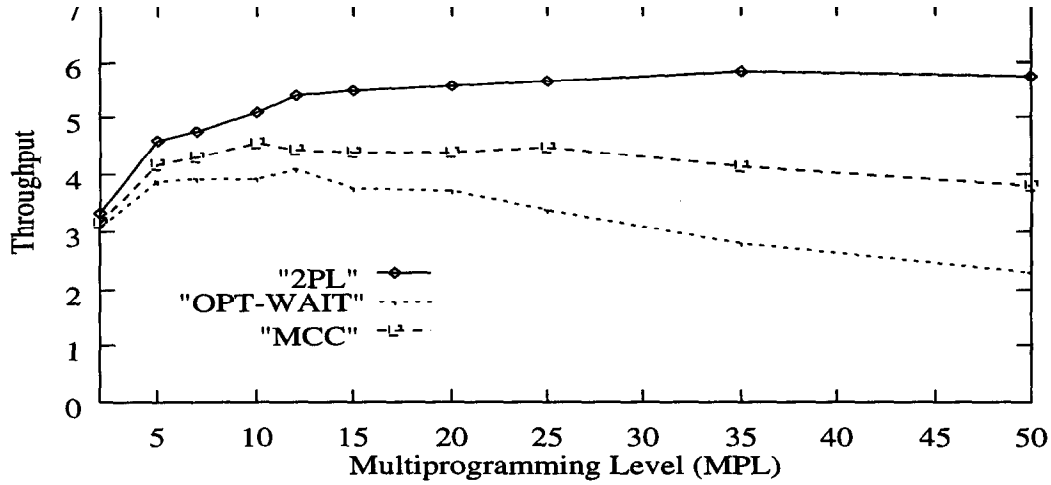

Fig. 4: Baseline Experiment: Throughput Vs. MPL

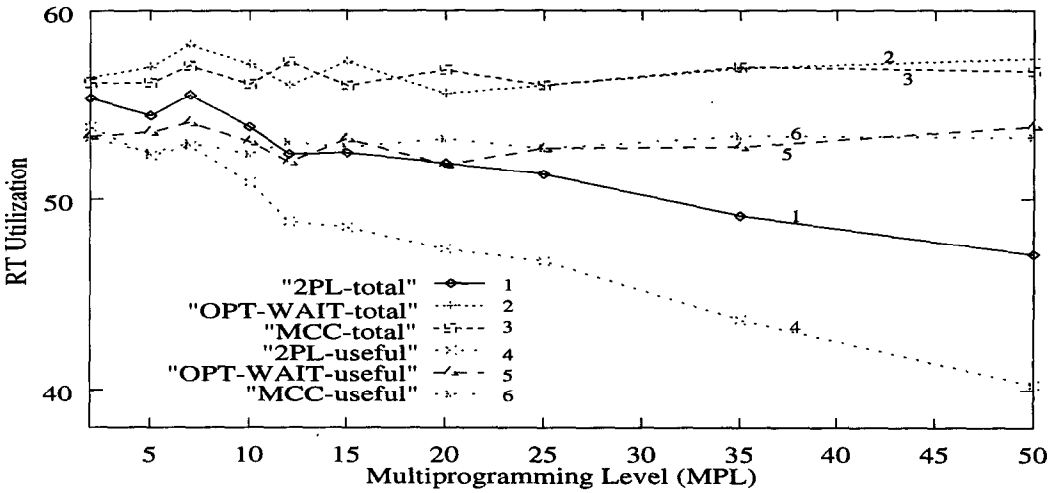

Fig. 5: Baseline Experiment: RT Utilization Vs. MPL

- **MCC:** MCC is identical to OPT-WAIT in all respects in this case since both algorithms have priority at the resources and the concurrency control manager and effctively use the same algorithm (OPT-WAIT) for real-time transactions. (As far as real-time transactions are concerned MCC behaves like OPT-WAIT).

Figure 4 shows that the throughput of all three algorithms initially increases as the system has enough capacity to absorb more transactions at low MPL. However, the medium-load and heavy-load performance of the algorithms is quite different in that their performance saturates at significantly different MPL values. The reasons for this behavior are explained below:

- **OPT-WAIT:** The throughput for OPT-WAIT saturates around an MPL of 12 since the total utilization (see Figure 6) of standard transactions has reached the maximum possible value (which is close to 100 minus the utilization of real-time transactions). Adding transactions beyond this point is counter productive since the amount of data conflict increases and leads to more restarts of standard transactions. The reason for the throughput dropping after an MPL of 12 is that the useful utilization for OPT-WAIT drops as MPL increases while the total utilization remains constant after it reaches its peak value. The reason for the useful utilization for OPT-WAIT dropping is the increased number of restarts at higher MPL which

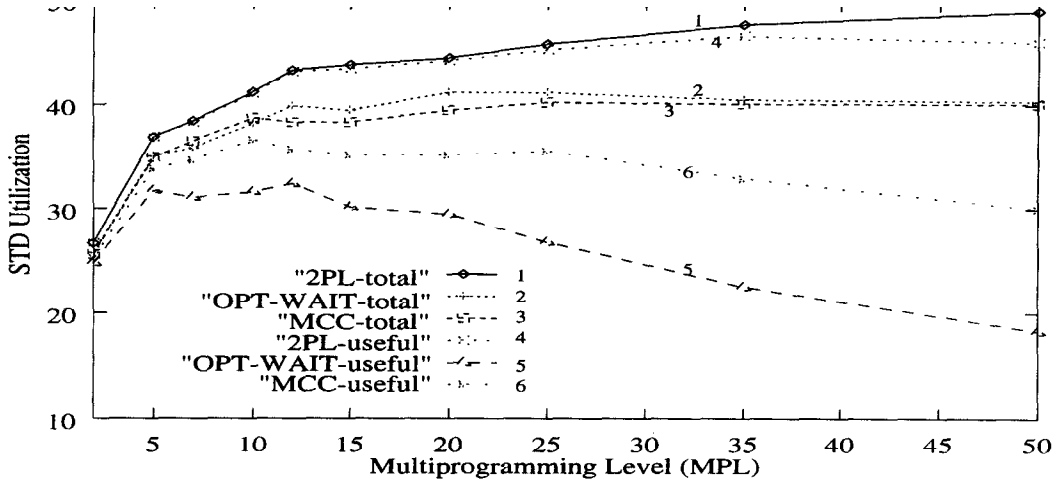

Fig. 6: Baseline Experiment: STD Utilization Vs. MPL

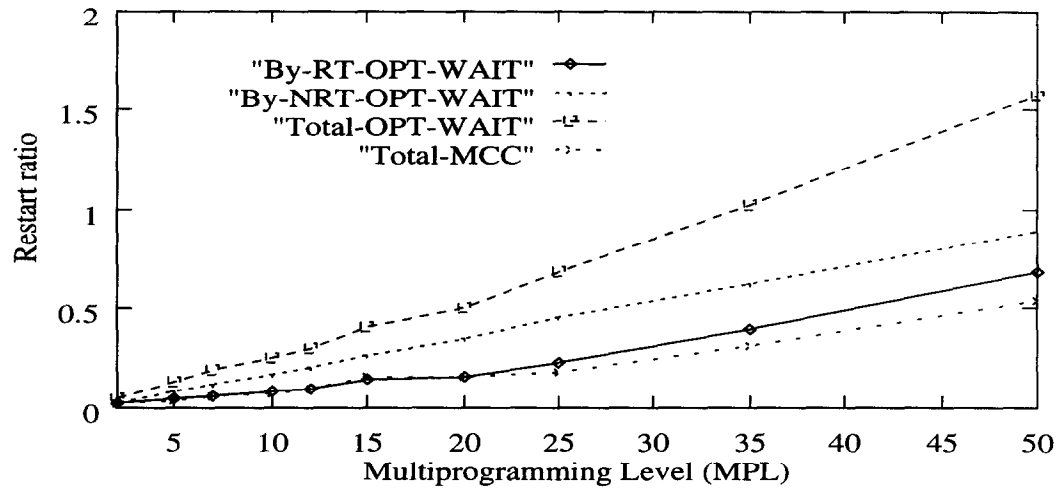

Fig. 7: Baseline Experiment: STD Restart Ratio Vs. MPL

can be verified from Figure 7 which shows the restart ratio (number of times a transaction is restarted in its lifetime) for standard transactions for OPT-WAIT algorithm. As shown in the figure, the restart ratio of standard transaction can be split into restarts caused by real-time transactions ("inter-class" conflicts) and restarts caused by standard transactions ("intra-class" conflicts). The "intra-class" conflicts are higher than "inter-class" conflicts and therefore the restarts are higher for "intra-class" conflicts.

- **2PL:** The throughput of 2PL consistently increases with MPL and flattens out. The total and useful utilization also show a similar trend. Notice that the total utilization for standard transactions increases with MPL since the total utilization for real-time transactions decreases as we noted.

- **MCC:** The throughput of MCC saturates earlier than 2PL but later than OPT-WAIT. The useful utilization of standard transactions for MCC also shows a similar behaviour. The reason useful utilization of standard transactions drops in this case is that at higher MPL's there is more data conflict with real-time transactions and hence more standard transactions get aborted at the master concurrency controller. The restarts are only due to "inter-class" conflicts and are lower than the restarts for OPT-WAIT algorithm (see Figure 7).

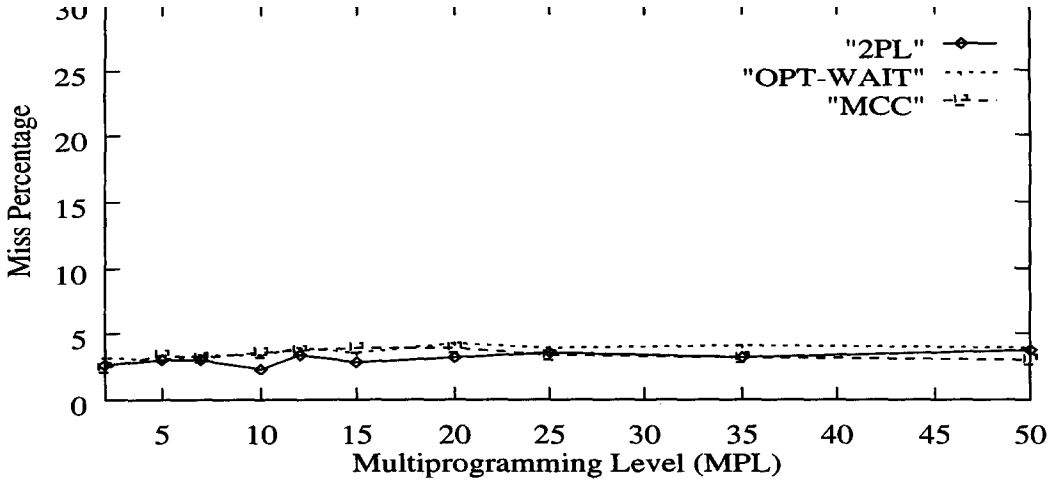

Fig. 8: Miss Percentage Vs. MPL (Overlap = 0%)

Consequently the useful utilization of standard transactions for MCC is higher than that for OPT-WAIT and therefore MCC attains higher throughput compared to OPT-WAIT.

The above baseline experiment demonstrated that there do exist workloads for which MCC provides significantly better throughput than Pure OPT-WAIT without perceptibly affecting the real-time performance. In fact, MCC almost halves the difference in throughput between 2PL and OPT-WAIT. MCC manages to achieve significantly better throughput than OPT-WAIT while maintaining the miss percent of real-time transactions at the same level as OPT-WAIT. We also performed a detailed investigation of the sensitivity of the baseline results. This was done by constructing several experiments around the baseline experiment by varying a few parameters at a time. These experiments evaluate the MCC performance over a wide range of transaction workloads and system operating conditions. The results of these experiments are presented in the remainder of this section.

5.2. Varying Database Overlap

In this set of experiments, the database overlap was varied keeping all other parameters fixed at the values shown in Table 2. Figures 8 and 9 show the miss-percentage of real-time transactions as a function of MPL of standard transactions, when the database overlap was kept at 0% and 100% respectively.

When the standard and real-time transactions were spatially disjoint (i.e., zero overlap), we observe that the miss percentages of all the three algorithms do not vary much as the MPL is varied. This is because the real-time transactions have priority at the resources and therefore do not "see" the standard transactions. Moreover, the two classes of transactions do not overlap in terms of database access and therefore there are no "inter-class" data conflicts. This is the reason for 2PL performing equally well as OPT-WAIT and MCC in terms of miss percent in contrast to the baseline experiment.

- **OPT-WAIT:** When the database overlap is increased, the miss percent of OPT-WAIT remain approximately equal to the corresponding figures in the spatially disjoint case and they are virtually constant as the MPL is varied. This is because OPT-WAIT favor the real-time transactions, which have higher priority than the standard transactions, when there are inter-class data conflicts. As a result, the real-time transactions are unaffected by the presence of the standard transactions and perform as well as in the spatially disjoint case.

- **2PL:** In contrast, the miss percentage of real-time transactions for 2PL increases consistently with MPL, as the database overlap is increased. This happens due to the increased data

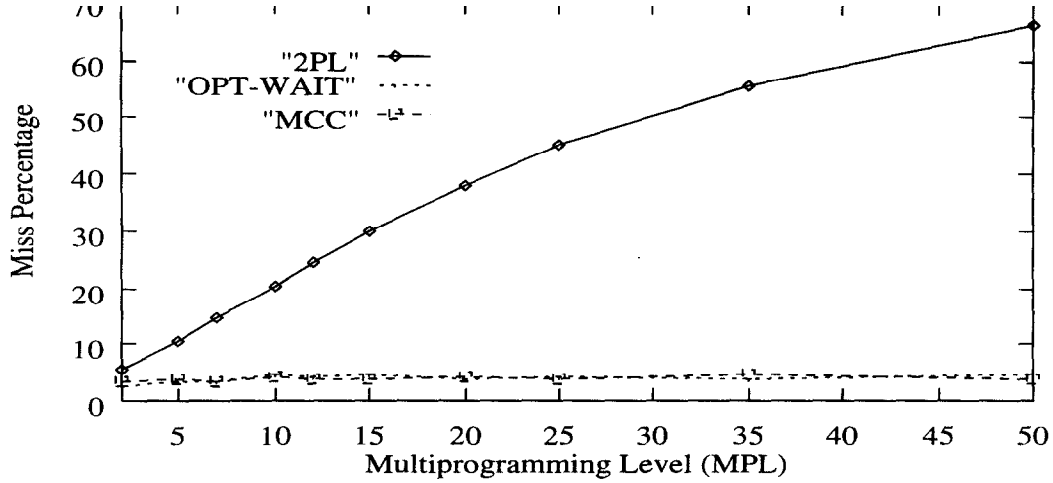

Fig. 9: Miss Percentage Vs. MPL (Overlap = 100%)

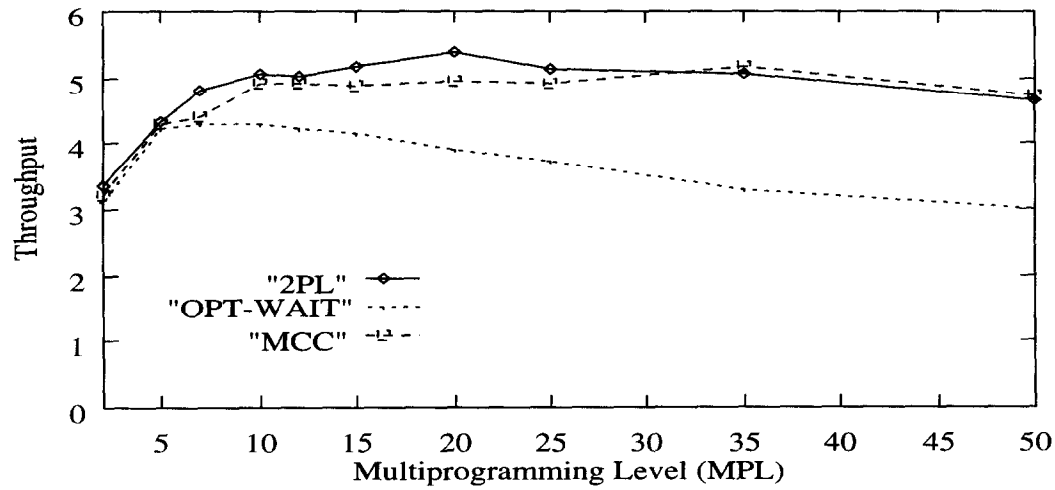

Fig. 10: Throughput Vs. MPL (Overlap = 0%)

conflicts with standard transactions at higher MPL. Increased data contention results in significant blocking for the real-time transactions and hence more missed deadlines.

- **MCC:** When the database overlap is increased, the behavior of MCC is identical to that of OPT-WAIT, since both the algorithms give priority to real-time transactions when there are inter-class data conflicts.

Figures 10 and 11 show the throughput of standard transactions as a function of MPL, when the database overlap was kept at 0% and 100% respectively.

- **OPT-WAIT:** The throughput of standard transactions decreases as the database overlap increases, in the case of OPT-WAIT. The reason is OPT-WAIT victimizes the standard transactions when there are inter-class data conflicts, in order to give better performance for the real-time transactions.

- **2PL:** When the database overlap is increased, 2PL gives higher throughput for standard transactions than in the spatially disjoint case. 2PL algorithm does not enforce priority and hence the real-time transactions have to wait for the standard transactions for conflicting

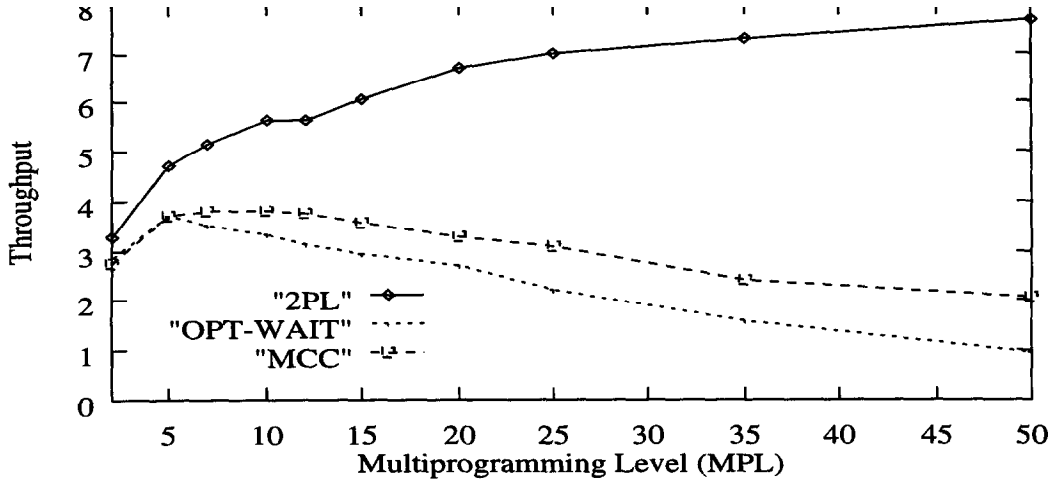

Fig. 11: Throughput Vs. MPL (Overlap = 100%)

data accesses. This results in significant blocking of the real-time transactions at the data and consequently standard transactions have more resources and hence higher throughput.

- **MCC:** At 0% overlap, the throughput of MCC was identical to that of 2PL, and was much higher than that of OPT-WAIT. On the other hand, the throughput of MCC for standard transactions decreases as the database overlap increases. MCC victimize the standard transactions when there are inter-class data conflicts, in order to give better performance for the real-time transactions. However, MCC gives much better throughput than OPT-WAIT. In MCC, 2PL is used for standard transactions which conserves resources when there are intra-class conflicts. Standard transactions are restarted only when they conflict with the real-time transactions. This resource conserving characteristic helps MCC to give better throughput than OPT-WAIT.

5.3. Varying Write Probability

In this set of experiments, the write probability was varied keeping all other parameters fixed at the values shown in Table 2. Figures 12 and 13 show the miss-percentage of real-time transactions as a function of MPL of standard transactions, when the write probability was kept at 0.25 and 0.75 respectively.

- **OPT-WAIT:** When the write probability is increased, there will be more data conflicts which results in more transaction restarts. However, the miss percentage of real-time transactions does not change appreciably since there are sufficient resources (further boosted by priority) to run again and complete. As the graphs show, the variation of miss-percentage with MPL remains identical to the baseline experiment.

- **2PL:** The blocking characteristic of 2PL makes it more sensitive to variation in write probability in terms of miss percent. When the write probability is high, there will be more waiting for data access which results in more real-time transactions missing their deadlines as can be ascertained from the graphs.

- **MCC:** The behavior MCC is identical to OPT-WAIT in all respects in this case.

Figures 14 and 15 show the throughput of standard transactions as a function of MPL, when the write probability was kept at 0.25 and 0.75 respectively.

- **OPT-WAIT:** As the write probability increases the resource utilization of real-time transactions for OPT-WAIT increases as a result of more restarts due to higher data conflicts.

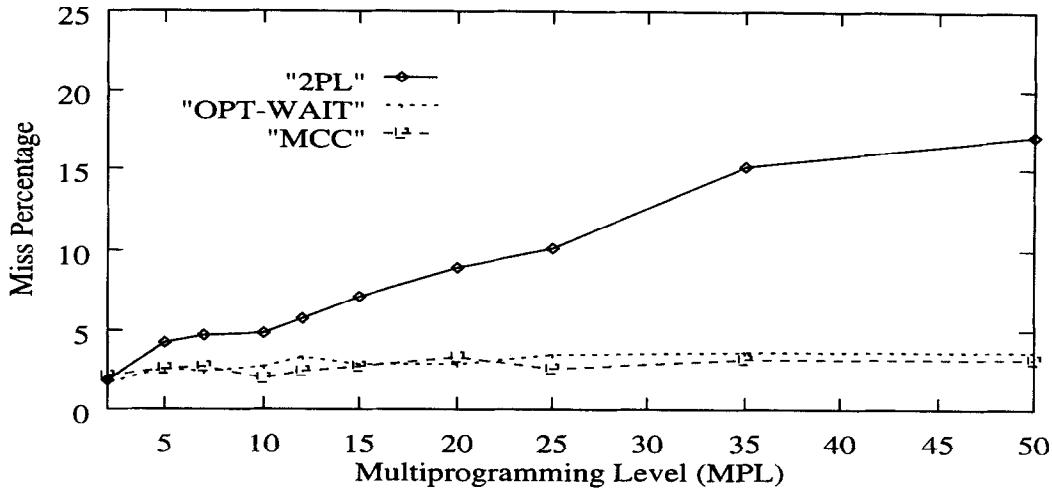

Fig. 12: Miss Percentage Vs. MPL (WriteProb = 0.25)

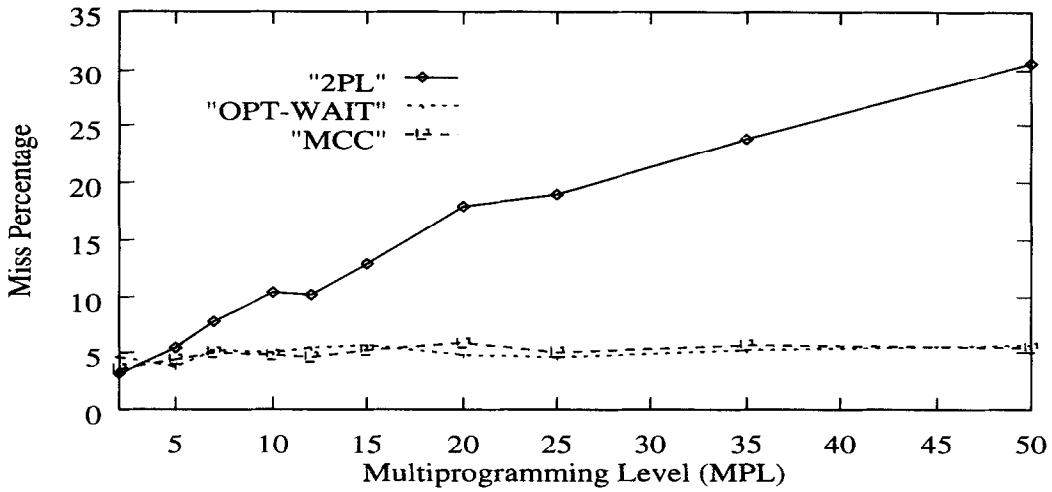

Fig. 13: Miss Percentage Vs. MPL (WriteProb = 0.75)

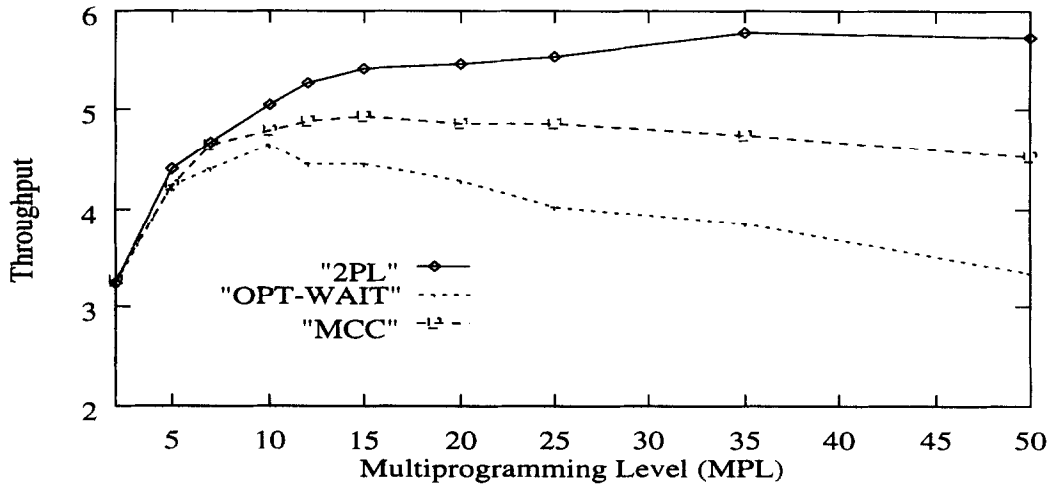

Fig. 14: Throughput Vs. MPL (WriteProb = 0.25)

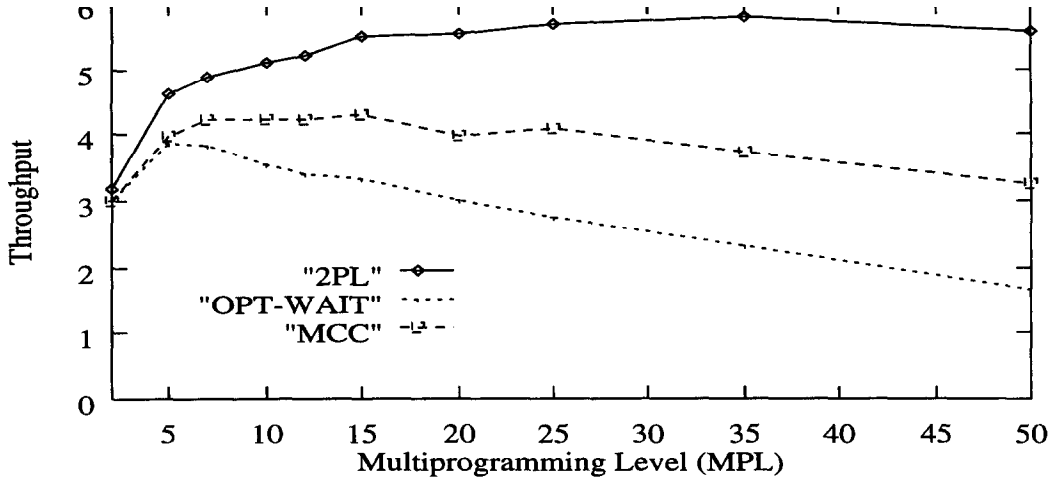

Fig. 15: Throughput Vs. MPL (WriteProb = 0.75)

Therefore, the standard transactions get less resources. Further, the higher data conflict amongst standard transactions also result in more restarts of standard transactions. Therefore, the throughput of standard transactions drops considerably as the write probability is increased.

- **2PL:** In the case of 2PL, as the write probability is increased, standard transactions will have more resources to use as more real-time transactions wait for standard transactions. However, the standard transactions also have to wait more for data access. As a result, the throughput does not vary appreciably as the write probability is varied.
- **MCC:** In the case of MCC also, standard transactions get less resources because of higher utilization by real-time transactions, as the write probability is increased. However, they do not waste resources because of intra-class conflicts since MCC uses 2PL for standard transactions. Therefore, the drop in throughput is less than that of OPT-WAIT as the write probability is increased.

5.4. Varying Resources

In this set of experiments we varied the number of CPU's and disks to study the effect of increased resources on the algorithms. The write probability and overlap were kept fixed at 0.5 and 25% respectively. The number of CPU's was kept at 10 and the number of disks at 20. All other parameters were fixed as per Table 2.

In the first experiment, the arrival rate of real-time transactions was maintained at 7 transactions per second, as in the baseline experiment. Figure 16 shows the miss percent of real-time transactions as a function of MPL of standard transactions. We observe that the qualitative behaviour of the three algorithms remains the same as in the baseline experiment. However, the absolute values of miss percent decreased drastically as compared to the baseline experiment as the system has more resources.

Figure 17 shows the throughput of standard transactions as a function of MPL. We observe here that OPT-WAIT outperforms 2PL and MCC in terms of throughput. The reason is that the load on the system is so little that it behaves like infinite resources. Hence, OPT-WAIT can afford to waste resources whereas, in 2PL and MCC, data conflicts will pose a bottleneck. Note that this behavior can be predicted from previous studies that show that optimistic algorithms achieve higher throughput than 2PL when there is very little resource contention [1].

In the next experiment, we scaled the arrival rate to 70 (by the same factor by which the resources were increased). Figure 18 shows the miss percent of real-time transactions as a function of the MPL of standard transactions. Again the qualitative behaviour remains the same as in

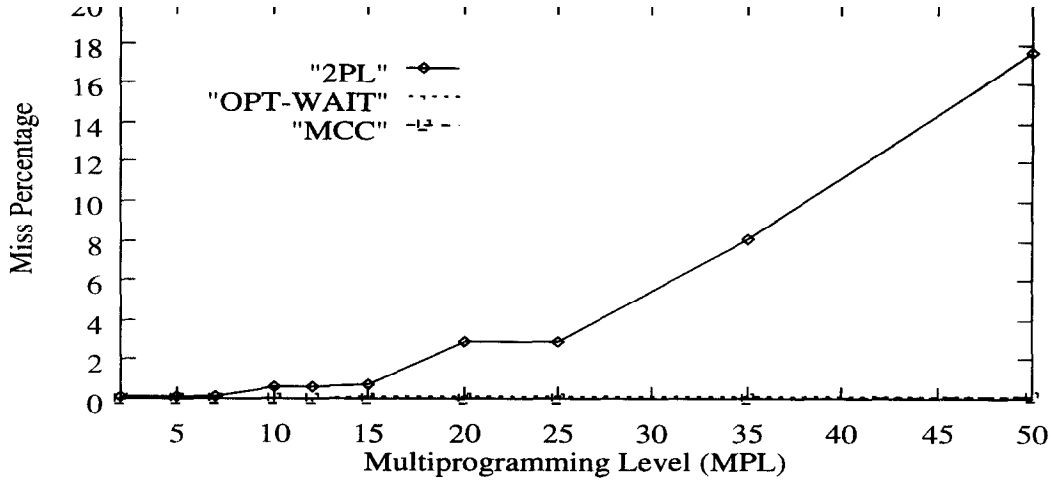

Fig. 16: Miss Percentage Vs. MPL (10 CPU, 20 Disk, Arrival rate = 7)

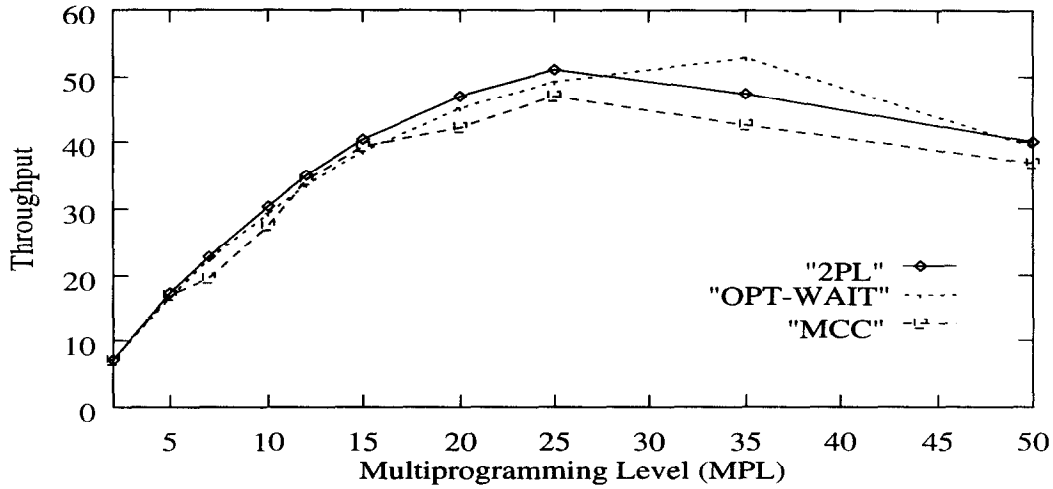

Fig. 17: Throughput Vs. MPL (10 CPU, 20 Disk, Arrival rate = 7)

the baseline experiment. However, the absolute values of miss percent increased drastically. The reason is, the number of concurrently executing transactions is more which causes higher data conflicts. Hence the increase in resources cannot maintain the same performance as in the baseline experiment.

Figure 19 shows the throughput of standard transactions as a function of MPL. The throughput of MCC and OPT-WAIT is almost identical. The reason OPT-WAIT has a throughput identical to MCC is that there are enough resources for OPT-WAIT to fritter away and yet achieve the throughput of MCC. The MCC, in contrast, blocks a standard transaction whenever there are "intra-class" data conflicts (which is pretty high given the arrival rate) or "inter-class" data conflicts. In the case of 2PL, real-time transactions have no priority at the data. As a result the throughput is much higher than in the baseline experiment. However, the throughput is not scaled as much as the resources because real-time transactions get priority at the resources.

5.5. Other Experiments

We also conducted several other experiments to evaluate sensitivity of results to changes in database size, real-time transaction deadline formula, different sizes for real-time and standard transactions, slack factor, etc..

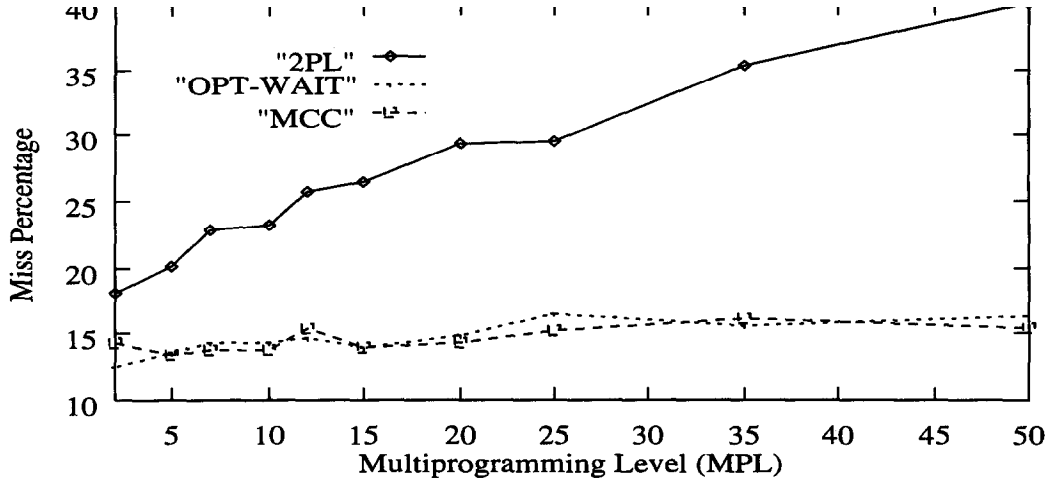

Fig. 18: Miss Percentage Vs. MPL (10 CPU, 20 Disk, Arrival rate = 70)

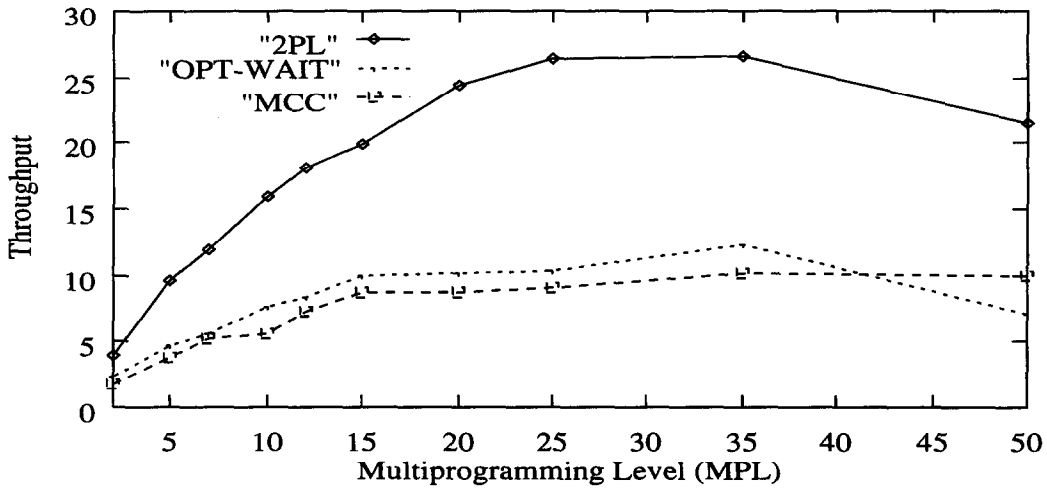

Fig. 19: Throughput Vs. MPL (10 CPU, 20 Disk, Arrival rate = 70)

In all these experiments, the various algorithms showed similar behavioral patterns to those described in the above experiments.

6. CONCLUSIONS

In this paper, we have addressed the issue of designing integrated real-time database systems that can cater to transaction workloads which have a mix of firm-deadline real-time transactions and standard transactions. In particular, we have shown that it is possible to provide improved performance to standard transactions without sacrificing the interests of the real-time transactions. This was achieved by designing a new database system architecture wherein an additional module called Master Concurrency Controller (MCC) was incorporated into the system. The MCC, which itself is a complete concurrency control algorithm, allows different transaction classes to choose from multiple concurrency control schemes and execute simultaneously without loss of data integrity. With this facility, it is possible to construct an RTDBS where each transaction class is regulated by its preferred CC algorithm: OPT-WAIT for the real-time transactions, and 2PL for the standard transactions.

Using a detailed simulation model, we compared the performance of the new MCC-based RT-

DBS architecture against those of RTDBSs which support only OPT-WAIT or only 2PL. 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. The performance of the algorithms was evaluated over a wide range of workloads and system operating conditions. Our experimental results indicate that the single-CC systems do well on (at most) one of the performance metrics but not the other: 2PL provided a good throughput for standard transactions but missed the deadlines of a high percentage of real-time transactions, while OPT-WAIT met the deadlines of most real-time transactions but exhibited poor throughput for standard transactions. In contrast, the MCC-based system was able to deliver a reasonable throughput for the standard transactions (in most cases it was able to bridge the gap in throughput between OPT-WAIT and 2PL by a factor of 40%) and at the same time maintain the miss percentage to be virtually identical to that of OPT-WAIT.

While we have focussed primarily on the real-time database environment in this paper, the MCC architecture is applicable to any environment that has multiple transaction classes having different preferred concurrency control algorithms. In our future work, we plan to evaluate the performance gains obtainable by using MCC in other such environments.

REFERENCES

[1] R. Agrawal, M. Carey, and M. Livny. Concurrency control performance modeling: alternatives and implications. In *ACM Trans. on Database Systems*, **12**(4) (1987).

[2] D. Agrawal and A. El Abbadi. Locks with constrained sharing. In *Proc. of Ninth ACM Symp. on Principles of Database Systems* (1990).

[3] D. Agrawal, A. El Abbadi and R. Jeffers. Using delayed commitment in locking protocols for real-time databases. In *Proc. of ACM SIGMOD Conference* (1992).

[4] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: a performance evaluation. In *ACM Trans. on Database Systems*, September (1992).

[5] A. Bestavros and S. Braoudakis. Timeliness via speculation for real-time databases. In *Proc. of IEEE Real-Time Systems Symposium* (1994).

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

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

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

[9] M. Carey ed. In *Special Issue on Extensible Database Systems, Database Engineering* (1987).

[10] 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).

[11] R. Cattell. In *Object Data Management: Object-oriented and Extended Relational Database Systems.* Addison-Wesley (1991).

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

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

[14] 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).

[15] K. Eswaran et al. The notions of consistency and predicate locks in a database system. In *Comm. of ACM*,November (1976).

[16] P. Franaszek, J. Robinson and A. Thomasian. Concurrency control for high contention environments. In *ACM Trans. on Database Systems*,June (1992).

[17] 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).

[18] J. Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course*, eds. R. Bayer, R. Graham and G. Seegmüller, Springer-Verlag (1979).

[19] J. R. Haritsa, M. Carey, and M. Livny. Dynamic real-time optimistic concurrency control. In *Proc. of IEEE Real-Time Systems Symp.* (1990).

[20] J. R. Haritsa, M. Carey, and M. Livny. Earliest deadline scheduling for real-time database systems. In *Proc. of IEEE Real-Time Systems Symp.* (1991).

[21] J. Haritsa, M. Carey and M. Livny. Data access scheduling in firm real-time database systems. In *Journal of Real-Time Systems*, September (1992).

[22] D. Hong, T. Johnson and S. Chakravarthy. Real-time transaction scheduling: a cost conscious approach. In *Proc. of ACM SIGMOD Conference* (1993).

[23] Theo Härder. Observations on optimistic concurrency control schemes. In *Information Systems*, **9**(2).

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

[25] 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).

[26] Special Issue on Real-Time Database Systems. *Journal of Real-Time Systems*, September (1992).

[27] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. In *ACM Trans. on Database Systems*, **6**(2) (1981).

[28] 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).

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

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

[31] J. Lee and S. Son. Using dynamic adjustment of serialization order for real-time database systems. In *Proc. of IEEE Real-Time Systems Symposium* (1993).

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

[33] D. Menasce and T. Nakanishi. Optimistic versus pessimistic concurrency control mechanisms in database management systems. In *Information Systems*, **7**(1) (1982).

[34] 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.* (1992).

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

[36] H. Pang, M. Carey and M. Livny. Managing memory for real-time queries. In *Proc. of ACM SIGMOD Conference* (1994).

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

[38] J. Robinson. Design of Concurrency Controls for Transaction Processing Systems. *Ph.D. Thesis*, Carnegie Mellon University (1982).

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

[40] S. Seshadri, S. Thomas and J. Haritsa. Design of extensible concurrency control mechanisms. In *Proc. of Sixth Intl. Conf. on Management of Data*, Bangalore, India (1994).

[41] S. Thomas. Extensible Concurrency Control Mechanisms. *Master's Thesis*, Dept. of Computer Sc. and Engg., Indian Inst. of Technology, Bombay (1994).

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