# A Study of Concurrency Control in Real-Time, Active Database Systems

Anindya Datta, *Member*, *IEEE*, and Sang H. Son, *Senior Member*, *IEEE*

**Abstract**—*Real-Time, Active Database Systems* (RTADBSs) have attracted a considerable amount of research attention in the very recent past and a number of important applications have been identified for such systems, such as telecommunications network management, automated air traffic control, automated financial trading, process control and military command, and control systems. In spite of the recognized importance of this area, very little research has been devoted to exploring the dynamics of transaction processing in RTADBSs. Concurrency Control (CC) constitutes an integral part of any transaction processing strategy and, thus, deserves special attention. In this paper, we study CC strategies in RTADBSs and postulate a number of CC algorithms. These algorithms exploit the special needs and features of RTADBSs and are shown to deliver substantially superior performance to conventional real-time CC algorithms.

**Index Terms**—Real-time database systems, active database systems, concurrency control, performance evaluation.

## 1 Introduction

Database systems that attempt to model and control external environments have attracted the attention of researchers in recent times. The application domains for such databases are numerous—network management, manufacturing process control, air traffic control, and intelligent highway systems to name a few [5], [38], [40]. The general system model that has been proposed for such systems include the following features:

1. The database is the repository of all system information that needs to be accessed or manipulated.
2. Monitoring tools (commonly known as sensors) are distributed throughout the real system being modeled. These sensors monitor the state of the system and report to the database. Such state reports arrive at the database at a high frequency (e.g., each sensor reports every 60 seconds).
3. The correct operation of the system requires the application of controls, i.e., in the event of semantically incorrect operation of the system, certain actions need to be taken.

These actions, that we call *control actions*, are taken from within the database by automatic control mechanisms and are communicated to the real system using database *executors*. Such systems have been termed ARCS (Active, Rapidly Changing data Systems) [15]. The motivation for designing database management systems (DBMSs) for these systems arise from the fact that such scenarios tend to be extremely data intensive (e.g., a typical network management center handles several gigabytes of data per day [10]). Also, the control decisions in these scenarios are typically data driven, e.g., a control decision in network management may be to compute retransmission rates at specific nodes in response to existing and past state information. In general, it is often necessary to access and manipulate current as well as historical data in order to make control decisions.

An important fact to note is that control actions typically need to be performed within temporal bounds, e.g., *if temperature goes above 80 degrees then reduce pressure within 30 seconds*. Such temporally cognizant processing requires the incorporation of *real-time* features in ARCS databases. Furthermore, the aforementioned control actions are envisioned to be "triggered," reducing human intervention, requiring the provision of *active* capabilities in ARCS databases. Thus, these databases are expected to require an amalgamation of *real-time* and *active* characteristics. In this paper, we study time cognizant concurrency control policies in real-time, active databases (RTADB). An RTADB is a database system where transactions have timing constraints such as deadlines and where transactions may trigger other transactions. Due to such triggering, in a RTADB, *dynamic* work is generated. Such dynamism is a major difference between RTADBs which are classical real-time databases where priority assignment is performed based on a "known" or "static" amount of work that a transaction is supposed to perform. We show that this difference promotes a serious rethinking of transaction management strategies in active systems.

There has been considerable research on active databases (ADBSs) as well as real-time databases (RTDBSs). Below, we briefly summarize the characteristics of ADBSs and RTDBSs. An RTDBS is a transaction processing system that is designed to handle workloads where transactions have completion deadlines. The objective of an RTDBS is to satisfy these deadlines in addition to standard database objectives such as maintaining consistency. The real-time performance of an RTDBS depends on several factors such as database system architecture and underlying processor

- *A. Datta is with the Dupree College of Management, Georgia Institute of Technology, Atlanta, GA 30332-0520. E-mail: anindya.datta@mgt.gatech.edu.*
- *S.H. Son is with the Department of Computer Science, School of Engineering and Applied Science, University of Virginia, Charlottesville, VA 22903. E-mail: son@virginia.edu.*

*Manuscript received 22 Jan. 1996; revised 6 June 1997; accepted 23 Oct. 2000; posted to Digital Library 7 Sept. 2001.*
*For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 105062.*

and disk speeds. For a given system configuration, scheduling and concurrency control policies impact performance. An extensive reference list of RTDBS research is provided below in Section 1.1.

*Active Database Systems* (ADBS) is a database system that is capable of initiation actions. The foundation of an ADBS is the so called Even-Condition-Action (ECA) rule. The semantics of the ECA rule is such that, upon detection of the occurrence of event E and the satisfaction of condition C, the specified action A is executed. Events are happenings of interest in the system, e.g., commitment or abortion of transactions, specified clock tick (temporal event), or the accessing of a data item. Conditions are often specified as predicates evaluable on the database state. An action is often a transaction that is *triggered* in response to the event E occurring and the condition C evaluating to true. References to ADBS literature are provided in Section 1.1.

The work outlined here *does not* propose research on these individual areas in isolation; a large and excellent body of work already exists and we shall borrow extensively from that—the goal is to **synthesize** these characteristics. An attempt at such synthesis, as shown in subsequent sections, exposes new and unique problems providing the potential for substantially, long term research. In other words, such databases must amalgamate real-time characteristics and active characteristics. As a consequence, these databases have been termed real-time, active database systems (RTADBSs).

### 1.1 Related Work

*Active Database Systems* (ADBSs) and *Real-Time Database Systems* (RTDBSs) have gained stature as important areas of research over the last few years, but there is not much reported on RTADBSs. In spite of the paucity of directly related work, we do borrow from the large body of work in two general and one specific area: the two general areas are ADBSs and RTDBSs, while the specific area is concurrency control in RTDBSs.

There have been both theoretical as well as experimental studies of ADBSs. Some notable work includes [9], [16], [17], [19], [35], [11]. Most of this work has either concentrated on data modeling issues (e.g., object-oriented features) or on the specification of ECA rules. In particular, a lot of work has been done on semantics of event and rule specification and evaluation, as well as coupling modes between conditions and actions. None of this work considers a real-time context.

The pioneering work in RTDBS performance evaluation was reported in [1], [2], [3], where the authors simulated a number of CC protocols based on the two-phase locking algorithm. However, this work was not examined in an active context. In [26], the problem of assigning deadlines to subtransactions is studied. This paper, however, does not study concurrency control. It also assumes that the structure of complex transactions is known in advance. In this work, we do not make any such assumption. Furthermore, much research has also been devoted to designing concurrency control (CC) mechanisms geared toward improving the timeliness of transaction processing and their subsequent performance evaluation [1], [2], [3], [22], [21], [25], [24], [33], [32], [45], [7], [31]. Again, all this work has been performed without considering the effects of triggering. An important result that we draw upon in this paper is reported in [22], [21]. In this set of important studies, Haritsa et al. showed that in *firm* or *hard* real-time scenarios (i.e., where late transactions are worthless), optimistic concurrency control (OCC) [28] outperforms locking over a large spectrum of system loading and resource contention conditions. In particular, the *broadcast commit* variant (OCC-BC) of optimistic concurrency control [36] was shown to perform particularly well. The rationale for this behavior was shown as follows: In OCC-BC, transactions at the validation stage are guaranteed to commit. Thus, eventually discarded transactions do not end up restarting other transactions. In locking mechanisms, however, soon-to-be-discarded transactions may block or restart other transactions, thereby increasing the likelihood that these transactions may miss their deadlines as well. Based on the above findings, we start with the basic assumption that optimistic protocols are well suited to RTADBS scenarios.

Another exciting and very recent development was the *International Conference on Active and Real-Time database Systems* (ARTDB95) held in Skövde, Sweden in June 1995. The organization of this workshop is an indication of the timeliness and emerging importance of this area. Eleven papers in all at the workshop [43], [30], [37] were primarily concerned with pure real-time issues, while [14], [13], [44] were primarily concerned with active database issues. In particular, [41], [8] concerned themselves with both active and real-time issues. Especially, [8] set the stage for forming a comprehensive real-time active systems model. These papers, however, are of a very high level—they identify problems and discuss general issues rather than providing solutions for specific problems. Thus, notwithstanding their novelty and significance of these papers, research in real-time, active databases is still in its preliminary stages.

To summarize, it may be stated that there has been some initial work in synthesizing real-time and active database systems. However, there is documented evidence of the necessity of much greater synthesis. In response to this need, in this paper, we design new CC algorithms for RTADBSs. Thus, this paper marks a positive initial step in exploring transaction processing issues in RTADBSs. The rest of the paper is organized as follows: In Section 2, we stipulate a model of execution of real-time active transactions, followed by Section 3 where we analyze the inadequacies of conventional real-time CC algorithms. In Section 4, we state a number of new algorithms in detail. Subsequently, we describe our simulation model in Section 5, show our performance evaluation results in Section 6, and discuss the strengths and weaknesses of our algorithms in Section 7. Finally, we conclude in Section 8.

## 2 A Model of Transaction Execution in RTADBSs

There is a large variety of features proposed for active database systems, including the coupling modes that determine how rules are to be executed relative to the triggering transaction. The problem is that many of those features are not fully understood in terms of their temporal behavior and, hence, they have difficulties being applied directly to real-time applications. It is hard to believe that a full range of active capabilities can be meaningfully supported in real-time, active database systems, at least in

the near future. In this section, we present a model of the execution of transactions in RTADBSs. We try to develop an execution model that can incorporate useful features of active capabilities, while not overly restricting other capabilities. For example, we do not restrict the depth of triggering. One level of triggering may appear to be enough in certain applications [39], [8], while it may be necessary to allow multiple levels of triggering in other applications [42]. Hence, in our execution model, we do not limit the depth of rule triggering.

There are primarily three types of transactions in an RTADBSs: *nontriggering*, *triggering*, and *triggered*. Note that triggering and triggered transaction types are not mutually exclusive, i.e., a triggered transaction may trigger other transactions. Our primary concern is to devise an execution model for triggering and triggered transactions. The first order of business is to identify a proper *coupling mode*.

### 2.1 Coupling Mode

Several *coupling* modes have been suggested between triggering and triggered transactions, e.g., *immediate*, *deferred*, *detached*, etc. [17]. In designing transaction models for RTADBS, it is possible to enforce any of these modes. However, it has been pointed out by several researchers (e.g., [8], [6]) that immediate and deferred coupling modes introduce a number of characteristics that work against the philosophy of deadline sensitive processing of real-time transactions. For instance, some degree of predictability of execution times is a big help in designing scheduling and concurrency control strategies. However, immediate and deferred modes introduce additional unpredictability beyond that already introduced by intrinsic database properties such as blocking and restarts. Hence, in this paper, we only consider the detached coupling mode. The execution model of this paper supports both the parallel detached mode with causal dependencies and sequential detached mode with causal dependencies. In addition, the exclusive detached mode with causal dependencies can be included for handling contingency transactions.

When utilizing the parallel causally dependent detached coupling mode, there are certain restrictions that must be satisfied as listed below:

1. triggered transactions being **serialized after** the triggering transactions,
2. **concurrency** between triggering and triggered transactions,
3. **commit dependency** between triggering and triggered transactions, and
4. **abort dependency** between triggering and triggered transactions (note that the terms commit and abort dependency are borrowed from the ACTA metamodel [12]).

A straightforward implementation of parallel causally dependent detached coupling mode in a database system that uses locking could lead to potential deadlocks because the read and write sets of the triggered rule often intersect the read and write sets of the triggering transaction. For this reason, we enforce restriction 1 above. In our model, we use the optimistic approach. By utilizing the notion of dynamic adjustment of serialization order to be discussed in the following section, we reduce the probability of unnecessary restarts.

A feature of real-time active transactions that sets them apart from "conventional" transactions is the notion of *transaction chaining*. The chaining phenomenon is simple to describe: An real-time active transaction may trigger one or more other active real-time transactions, which may, in turn, trigger others and such triggering may proceed to arbitrary depths. We analyze the effects of transaction chaining using two different structures: a) *triggering graphs* and b) *step diagrams*.

### 2.2 Triggering Graphs

Transaction chaining leads to the generation of *triggering graphs*.

**Definition.** *The triggering graph is a directed acyclic graph, where the nodes represent rule firings and the arcs are transactions invoked by the rule firings.*

The triggering graph is a dynamic structure which captures transaction triggering information in the system. The nodes of a triggering graph are labeled by rule identifiers, while the arc labels are transaction identifiers. If arc $T_i$ is incident upon node $R_j$, it means that transaction $T_i$ was (partially) responsible for the firing of rule $R_j$. If arc $T_i$ is incident from node $R_j$, it means that rule $R_j$ was responsible for invoking transaction $T_i$. Since the rule antecedents could be complex, multiple transactions may be responsible for firing a rule. Since the corresponding action may also be complex, the firing of a rule may invoke multiple transactions.

An example triggering subgraph is shown in Fig. 1.

This graph resulted from the initial firing of rule $R_1$ which generated transactions $T_1$ and $T_2$.

An important property of transactions belonging to the same triggering graph is *dependency*. To illustrate the notion of dependency, we use the following scenario: Let $G$ be a triggering graph. Let $T_i$ and $T_j$ be two transactions (i.e., edges) in $G$. Using graph theoretic terminology, we will sometimes allude to $T_i$ and $T_j$ as the ordered pairs $(R_i, R_j)$ and $(R_k, R_l)$, respectively, where $R_i$, $R_j$, $R_k$, and $R_l$ denote nodes in $G$. This signifies that $T_i$ is directed

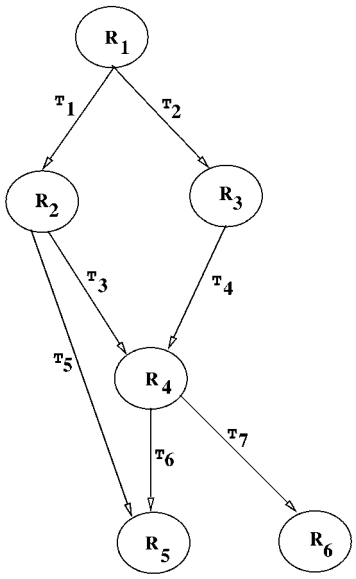



Fig. 1. Triggering graph.

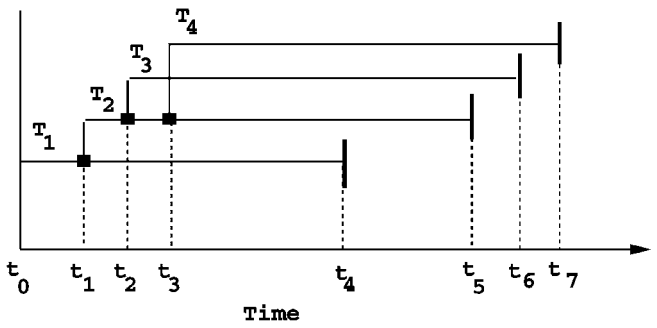



Fig. 2. Step diagram.

from $R_i$ to $R_j$ and $T_j$ is directed from $R_k$ to $R_l$. The node which $T_i$ is incident on will be referred to as $ON(T_i)$. The node which $T_i$ is incident from will be referred to as $FROM(T_i)$. With respect to Fig. 1, $ON(T_1) = R_2$ and $FROM(T_1) = R_1$. This lets us formally define the notion of transaction dependencies.

**Definition.** *Let $G$ be a triggering graph. Let $T_i$ and $T_j$ be two arc labels in $G$. $T_j$ is said to be* **dependent** *on $T_i$, if $FROM(T_j)$ is reachable from $ON(T_i)$.*

If $T_j$ is *dependent* on $T_i$, then we refer to $T_i$ as a *guardian* of $T_j$. By examining the triggering graph shown in Fig. 1, one can easily see the dependencies between the various transactions, e.g., $T_6$ is dependent on $T_1$, as $R_4$ (the node $T_6$ is incident from) is reachable from $R_2$ (the node $T_1$ is incident on). On the other hand, $T_4$ is not dependent on $T_1$, as $R_3$ is unreachable from $R_2$.

### 2.3 Step Diagrams

Another way of looking at real-time active transactions is to observe the precise points at which transactions are triggered, as well as the transaction deadlines. A convenient graphical mechanism that captures the above information is the *step diagram*. For example, consider the step diagram shown in Fig. 2, which marks triggering points with black rectangles.

Transaction $T_1$ arrives at $t_0$ with a deadline of $t_4$. At $t_1$, $T_1$ triggers $T_2$, with a deadline of $t_5$. At $t_2$, $T_2$ triggers $T_3$ with deadline $t_6$. At $t_3$, $T_2$ triggers $T_4$ with deadline $t_7$. This method of representing real-time active transactions through step diagrams allows us to represent transaction chaining, as well as deadline information pictorially. We shall use step diagrams as well as triggering graphs to analyze CC strategies in this paper.

### 2.4 Properties of Real-Time Active Transaction Execution

Based on the preceding discussion, we make the following observations regarding the execution of real-time active transactions in our model:

a. *If a guardian transaction aborts (restarts), a chain of aborts may occur.* Clearly, if a guardian transaction aborts, all its effects are undone, resulting in the abortion of its triggered transactions. If these "triggered" transactions had triggered other transactions, these also need to abort, potentially unleashing a cascade of aborts. In general, the abortion or restart of a guardian transaction will lead to the abortion of all its dependent transactions. For example, in Fig. 2, if $T_1$ aborts, $T_3$, $T_5$, $T_6$, and $T_7$ need to abort as well. In general, one may say that the removal of a node $N$ in the triggering graph would lead to the removal of the subgraph induced by the nodes reachable from $N$. Similarly, in Fig. 2, if $T_1$ were to abort, $T_2$, $T_3$, and $T_4$ must abort as well. Note that, if a guardian transaction is restarted, there is no guarantee that its dependent transactions will be retriggered.

b. *A Triggered transaction will have a deadline later than that of its triggering transaction.* It makes little sense to assign a deadline to a triggered transaction that is earlier than the deadline on its triggering transaction(s). For example, consider transaction $T_X$ as an instance of a triggering transaction and $T_Y$ as an instance of a triggered transaction. Let us assume that $T_Y$'s deadline (say $D_Y$) is earlier than that of $T_X$ (say $D_X$), i.e., $D_Y < D_X$. Now, consider a case when $T_Y$ successfully completes, but, subsequently, $T_X$ is restarted at $t_r$, such that $D_Y < t_r < D_X$. In this case, $T_Y$'s effects on the database would have to be undone, which is clearly an undesirable action to undertake as it means all the processing done on $T_Y$ is wasted. In our model, we assume that the commit of a triggered transaction is deferred until its triggering transaction(s) have committed. Clearly, a corollary of this property may be stated as follows: *A dependent transaction will have a deadline later than the deadlines of all its guardian transactions*.

Note that certain similar properties have been discussed in the parallel rule firing literature (see, e.g., [29]). However, none of this work considers a real-time context.

## 3 The Need for New CC Algorithms

In this section, we explain the need for extending conventional OCC mechanisms to handle the RTADBS scenario, by demonstrating that such protocols, in the RTADBS framework, appear to exhibit erroneous behavior.

The basic OCC protocol [28] lets all transactions proceed unhindered until a transaction wants to commit, at which time it must go through a *certification* or *validation* stage. In this stage, it is checked whether it conflicts with any recently committed transactions. If such conflicts exist, the transaction is restarted. A variant of the basic OCC known as OCC-BC (the *broadcast commit* variant) [36], has been shown to perform particularly well in RTDBSs [21]. In OCC-BC, a validating transaction is always guaranteed to commit. However, all currently running transactions that conflict with the validating transaction are restarted. Another variant of OCC, known as OCC-TI (the *timestamp interval* variant) has been shown to perform even better than OCC-BC [32]. OCC-TI dynamically adjusts the serialization order of transactions to prevent *unnecessary restarts* (explained later in the document). Neither OPT-BC, nor OCC-TI use any priority information in their conflict resolution strategies. Below, we show two examples to demonstrate the difficulties of applying conventional OCC protocols to RTADBSs.

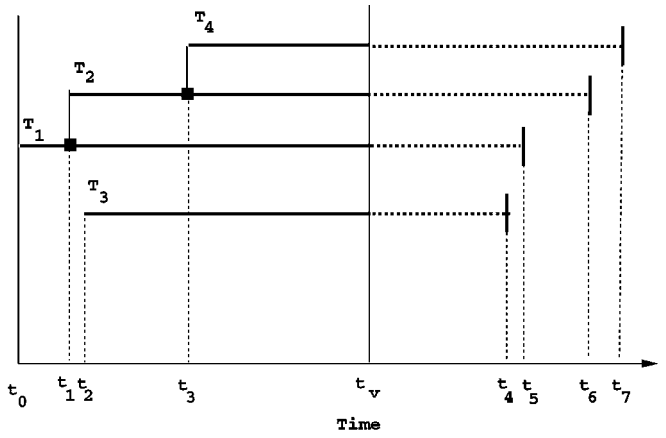



Fig. 3. Problems with optimistic mechanisms: Example 1.

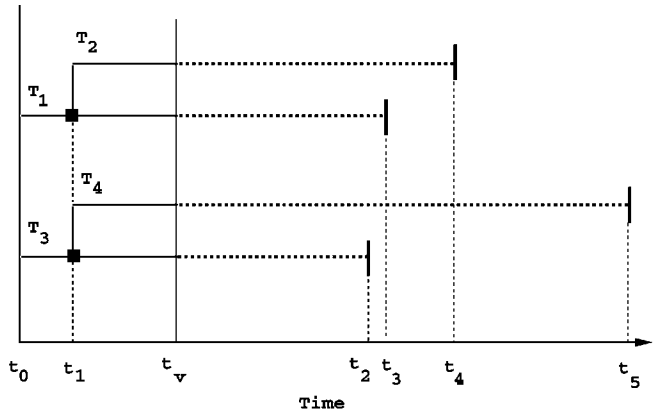



Fig. 4. Problems with optimistic mechanisms: Example 2.

**Example 1.** Consider Fig. 3. Transaction $T_1$ starts at $t_0$ with a deadline of $t_5$. At $t_1$, $T_1$ triggers transaction $T_2$ with deadline of $t_6$. At $t_2$, a new transaction $T_3$ starts with deadline $t_4$. Subsequently, at $t_3$, $T_2$ triggers $T_4$ with deadline $t_7$. At $t_v$, $T_3$ enters the validation stage. Assume that $T_3$ conflicts with $T_1$ and not with $T_2$ and $T_4$. Further, assume that $Priority_{T_1} < Priority_{T_3}$ (as is the case with an *earliest-deadline-first* policy with respect to Fig. 3). In this situation, conventional OCC algorithms (both priority sensitive as well as priority insensitive ones) would restart $T_1$, thereby aborting $T_2$ and $T_4$. One can easily observe that the restarted $T_1$ (regardless of whether it triggers any transactions on this run) has virtually no chance of completing.

From the simple example, one can identify a glaring weakness of conventional OCC algorithms when applied to RTADBS scenarios: One of the guiding principles of real-time transaction processing is reducing the amount of wasted work. However, in the scenario depicted in Fig. 3, by restarting $T_1$, the work done on $T_2$ and $T_4$ is wasted as well. This happens as conventional real-time CC protocols ignore the effect of transaction chaining, i.e., the fact that work is dynamically being generated. Thus, some mechanism is required to take into account this dynamic work being generated through triggering.

**Example 2.** Consider Fig. 4. Here, $T_1$ and $T_3$ both start at $t_0$ and trigger $T_2$ and $T_4$, respectively, at $t_1$. The deadlines for these transactions are easily observed in Fig. 4. Assume $T_2$ and $T_4$ are roughly the same size. At $t_v$, $T_3$ requests validation while conflicting with $T_1$ (but not with $T_2$). From the figure, it is clear that $Priority_{T_1} < Priority_{T_3}$ due to $T_3$'s earlier deadline. In this case, conventional OCC algorithm (regardless of priority sensitivity) would restart $T_1$, thereby aborting $T_2$. The problem with this scenario is that the restart decision ignores the deadlines (i.e., priorities) of $T_2$ and $T_4$. However, were the priorities of the transactions taken into account, a different decision should result in this case. This is illustrated below:

Even though $T_3$ is a higher priority transaction than $T_1$, the deadline of $T_4$ (i.e., $t_5$) is much later than that of $T_2$. Now, consider the two alternatives to resolve this conflict: 1) $T_1$ **is restarted:** in this case, were $T_2$ triggered again, it would have trouble completing within $t_4$ and 2) $T_3$ **is restarted**: in this case, were $T_4$ to be triggered again, it has a better chance of finishing than $T_2$ as its deadline is a long way away (the reader is reminded that $T_4$ and $T_2$ are roughly the same size). Based on the above discussion, it appears that 2 is a better choice than 1, even though it is not the choice OCC would make. The basic point here is that considering transaction priorities takes on special significance in RTADBSs. This means that it is even more important (in comparison to real-time database systems) to provide a mechanism for priority cognizance in CC algorithms.

### 3.1 Summary

Based on the two examples above, we have identified two important drawbacks in using conventional OCC protocols in ARTDBs:

1. Because conventional conflict resolution strategies ignore the effect of transaction chaining, there is a high potential for *wasted processing*, i.e., work done on transactions that are subsequently aborted due to an abort/restart of their guardian transactions. This was shown in Example 1. Thus, some mechanism is needed to account for the dynamic work generated as a result of triggering.
2. Priority cognizance assumes special significance, as shown in Example 2. However, as evidenced in the real-time database literature, incorporating priority cognizance successfully is difficult. This was also exemplified in both examples above where incorrect decisions would have been made by even priority cognizant protocols. Thus, a major rethinking is required for designing successful priority sensitive protocols.

## 4 New CC Algorithms "Tuned" for RTADBSs

In this section, we propose two alternate CC algorithms for RTADBSs. The algorithms are optimistic and based on the notion of *dynamic adjustment of serialization order* [33], [32]. The reason for choosing the notion of *optimism* is explained at the outset of Section 3. We choose to apply the technique of *dynamic adjustment of serialization order* as it has been

```
restart all transactions in CS(T_val)
commit T_val
```

Fig. 5. Conflict set of $T_{val}$.

shown to result in substantial performance gains over other optimistic algorithms. Before stating our algorithms, we give a brief review of 1) OCC algorithms (for an in-depth treatment of OCC, see [28], [36], [20]) and 2) the technique of dynamic adjustment of serialization order (for complete coverage, see [32]).

### 4.1 Optimistic Concurrency Control

Optimistic concurrency control (OCC) consists of three phases: the *read phase*, *validation phase*, and *write phase*. In this paper, which assumes a firm real-time database system, we are primarily interested in the *broadcast commit* variant of OCC known as OCC-BC [36]. In OCC-BC, a transaction is validated only against currently active transactions. The basic assumption in OCC-BC is that the validating transaction is serialized before all other concurrently running transactions and, hence, it is guaranteed to commit. The validation protocol for a transaction $T_{val}$ may be succinctly described by the following procedure, assuming the conflict set of $T_{val}$ is given by $CS(T_{val})$ (Fig. 5).

#### 4.1.1 Dynamic Adjustment of Serialization Order

The basic purpose of this technique is to prevent *unnecessary restarts* that occur on the restart of transactions which could have been serialized successfully with respect to the validating transaction. This can best be explained with the aid of an example. Below, we present an example.

**Example 3.** Let $r_i[x]$ and $w_i[x]$ denote the read and write operation, respectively, by transaction $i$ on data item $x$. Further, assume that $v_i$ and $c_i$ denote the validation and commit of transaction $i$, respectively. Consider the following three transactions:

$$\begin{aligned} T_1 &: r_1[x], w_1[x], r_1[y], w_1[y], v_1 \\ T_2 &: r_2[x], w_2[x], \ldots, v_2 \\ T_3 &: r_3[y], \ldots, v_3. \end{aligned}$$

Now, consider the following execution history fragment:

$$H = r_1[x], w_1[x], r_2[x], r_3[y], w_2[x], r_1[y], w_1[y]v_1, c_1.$$

OCC-BC would restart both $T_2$ and $T_3$ in the process of validating $T_1$. However, a careful examination of $H$ shows that $T_2$ clearly needs to restart as it has both write-write and write-read conflicts with $T_1$. However, that is not the case with $T_3$ which only has a write-read conflict on the data item $y$. Thus, as long as we can set the serialization order $T_3 \rightarrow T_1$, $T_3$ does not need to restart. The restart of $T_3$ by OCC-BC is referred to as an *unnecessary restart*.

The technique of dynamically adjusting the serialization order eliminates these unnecessary restarts by adjusting serialization orders of transactions at the validation stage. The authors in [32] differentiate between two classes of conflicting transactions: 1) *irreconcilably conflicting* transactions, which cannot be serialized and, thus, must be restarted, e.g., $T_2$ in the above example and 2) *reconciliably conflicting* transactions whose serialization order can be adjusted and, thus, need not be restarted, e.g., $T_3$ in the above example. The validation process in this case may be expressed, as shown in Fig. 6.

In [32], the OCC-TI protocol based on this technique is shown to perform extremely well as it wastes less effort by restarting transactions fewer times. In fact, to the best of our knowledge OCC-TI is one of the best performing CC algorithms in published literature. In the simulation study reported later in this paper, OCC-TI is used as the primary basis of comparison for our algorithms.

The salient point of OCC-TI is that, unlike other optimistic algorithms, it does not depend on the assumption of the serialization order of transactions being the same as the validation order. Rather, it uses the serialization order that is induced as a transaction progresses and uses restarts only when necessary. Restarts may occur in OCC-TI under two circumstances: 1) While validating $T_{val}$ all conflicting transactions that cannot be successfully serialized (transactions that have bidirectional conflicts with $T_{val}$, i.e., both read-write and write-read) have to restart and 2) while accessing data in the read phase if an unresolvable conflict is detected (e.g., trying to read a data item updated by a transaction serialized after the accessing transaction). The detection of conflicts is achieved through manipulating timestamps—basically, each transaction is allocated a timestamp interval which keeps getting reduced as serialization dependencies are induced. If a transaction's timestamp interval *shuts out*, i.e., there is no possible way to serialize the transaction any more, it must restart. The use of timestamps to record serialization dependencies is well documented in the literature (see e.g., [32]).

```
Validation Algorithm in OCC-TI
foreach transaction T_i in CS(T_val) {
    if T_i irreconciliably conflicts with T_val then
        restart T_i;
    else    \* i.e., if T_i reconciliably conflicts with T_val *\
        adjust the serialization order of T_i w.r.t. T_val;
}
commit T_val
```

Fig. 6. Validation algorithm in OCC-TI.

## 4.2 OCC-APFO

OCC-APFO is an optimistic algorithm that dynamically adjusts the serialization order of transactions. APFO stands for *Adaptive Priority Fan Out*. The specific meanings of these terms will be clear once we explain the algorithm. OCC-APFO has two goals, which correspond to the two shortcomings of conventional OCC algorithms identified in Section 3: 1) to reduce wasted work by being conscious of how much additional work a transaction has generated through triggering and 2) to be priority cognizant. We first discuss how OCC-APFO achieves the two goals separately and then synthesize the two components. Finally, we present the procedure descriptions.

### 4.2.1 Accounting for the Effect of Triggering

OCC-APFO keeps track of the triggering dynamics in order to factor in the effect of triggering in resolving conflicts. The basic goal of keeping track of triggering is to *reduce the amount of wasted work*. Wasted work is the processing done on transactions that are restarted or discarded. In the context of real-time active transactions, wasted work not only involves the processing done on the restarted (or discarded) guardian transaction, but also must include work performed on any transactions that may depend on this transaction.[1] The basic idea therefore is that, as the number of transactions depending on a guardian transactions increases, the cost of restarting the guardian transaction goes up proportionally. In other words, with an increasing number of dependent transactions, a guardian transaction's candidacy for restart or abort goes down. We express this fact, i.e., the attractiveness of a transaction as a candidate for abort/restart, through a property called *fan-out*.

**Definition.** *Let $G$ be a triggering graph and let $T$ be an arc label (i.e., a transaction label) in $G$. Then, the fan-out of the transaction $T$ in $G$ is defined as the number of arcs in the subgraph of $G$ rooted at ON(T).*

Intuitively, fan-out (FO) of a transaction $T$ is the number of transactions that depend on $T$. In Fig. 1, $FO_{T_1} = 4$ and $FO_{T_4} = 2$. FO is used by the OCC-APFO algorithm as a measure of how much additional work a transaction has generated by triggering.

### 4.2.2 Priority Cognizance

This section explains how OCC-APFO attempts to factor transaction priority information into the conflict resolution process. In this algorithm, we only consider the priority of the conflicting transactions, and *do not* consider the priorities of their dependents. In a later algorithm (OCC-APFS), we will consider priority of the entire dependent set.

To discuss how we consider priority, we first show a simple example to illustrate the advantages and disadvantages of priority cognizance. Note that, since we do not consider the priorities of dependent transactions in OCC-APFO, the following example only portrays the conflicting transactions and not their dependents.

**Example 4.** Consider the situation depicted in Fig. 7. This figure portrays the execution profiles of two concurrently active transactions $T_a$ and $T_b$. $T_a$ arrived at time $t_2$ with a deadline of $t_4$, while $T_b$'s arrival time and deadline are $t_1$ and $t_3$, respectively. At time $t_v$, $T_a$ reaches the validation stage. Note that $T_b$'s priority is higher than $T_a$'s, due to $T_b$'s earlier deadline. However, in the process of validating $T_a$, $T_b$ would be restarted using a priority incognizant protocol such as OCC-BC. One can easily see from Fig. 7 that $T_b$ has virtually no chance of finishing again. On the other hand, $T_a$, if restarted, has a better chance of completing successfully. This fact is not recognized by a priority incognizant algorithm. However, if a protocol could be designed that recognizes that $T_b$'s priority is higher and restarted $T_a$ instead, one can see from the figure that $T_a$ would have a fair chance of finishing. Thus, while a priority incognizant protocol would give $T_b$ virtually no chance of finishing, a priority cognizant one gives *both* transactions a chance to commit.

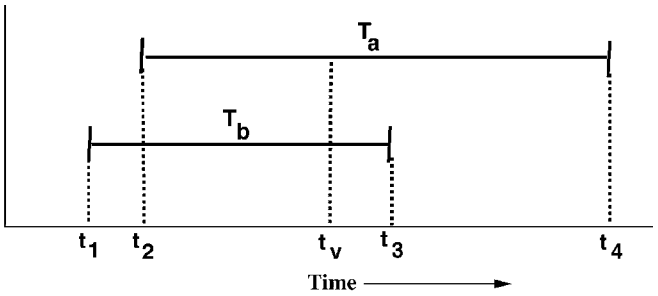



Fig. 7. Using priority information at validation.

The same example can be used to illustrate the weakness of priority cognizant protocols. Clearly, as we have already argued, a priority cognizant protocol will provide *both* $T_a$ and $T_b$ an opportunity to commit. *However, it will not guarantee that either of them does commit*. It is easily seen that even if $T_a$ was restarted and $T_b$ allowed to continue, several events could occur (e.g., $T_b$ may be restarted by some other transaction, the system load may suddenly increase, and $T_a$ may miss its deadline on its restarted run) that could cause *both* $T_a$ and $T_b$ to be lost. On the other hand, OCC-BC (or any other priority incognizant protocol) will *guarantee* that $T_a$ (i.e., the validating transaction) will commit. Thus, a priority cognizant protocol may create a situation that causes more misses than that allowed by priority insensitive algorithms.

The simple example above helps us identify a few key points that need to be considered while designing good priority cognizant CC algorithms. Basically, it has been repeatedly shown that it is the *number of restarts* that determines the performance of real-time CC algorithms. For example, the reason that OCC-TI performs better than OCC-BC is that it reduces the number of restarts by not restarting reconciliably conflicting transactions [32]. It is reasonable to assume therefore that we can expect a priority cognizant protocol to perform better if it can reduce restarts even further. Using the example above, we see that there are situations where a priority conscious protocol *may* offer the potential for reducing restarts. However, the same protocol, as we argued in the previous paragraph, may result in

1. Recall that, when a guardian transaction is restarted, its dependent transactions must be aborted.

larger number of restarts. Therefore, our goal is as follows: **Design an algorithm that offers opportunities of reducing restarts while making sure that chances of increasing restarts are minimal**.

OCC-APFO attempts to satisfy this goal by restarting validating transactions only when it feels that the restarted transaction is very likely to commit eventually. The question is whether it is possible to guarantee the eventual successful completion of restarted validating transactions. Clearly, the answer is no, as system dynamics are unpredictable. The next logical question then is whether it is possible to make it very likely that a restarted validating transaction will successfully complete. The answer to this is yes. One easy way to do that is to see how much time is left before its deadline expires and decide to restart it if sufficient time is left for the transaction to complete. The problem is the determination of *sufficient time*. Clearly, if the system dynamics represent high contention for data and resources, more time is required than if the system dynamics represent very little contention. In other words, the notion of *sufficient time* should be dictated by the condition of the system.

Thus, the basic ideas behind the priority cognizant component of OCC-APFO are:

1. Start out with an estimate of *sufficient time* required to restart validating transactions. As we show later, we make this estimate without making any unrealistic assumptions such as a priori knowledge regarding either transaction or system characteristics.
2. Based on feedback obtained by monitoring system performance, revise estimate of *sufficient time*.

In OCC-APFO, the feedback is in the form of the *Miss Ratio of restarted Validating transactions* (MRV). Based on this feedback, the revision of the sufficient time estimate is performed in an *adaptive* fashion.

#### 4.2.3 How Things are Put Together: Details of OCC-APFO

In this section, we provide the details of the OCC-APFO algorithm. As described before, OCC-APFO is an optimistic, priority cognizant protocol based on the dynamic adjustment of serialization order. Like OCC-TI [32], OCC-APFO uses the notion of timestamp intervals to record and represent serialization orders induced by concurrency dynamics. Timestamps are associated with both transactions and data items, but in different ways.

**Data Items and Timestamps**. Each data item has a *read* and a *write* timestamp, in addition to their usual meanings, i.e., the read and the write time stamps are the largest timestamp of transactions that have read or written the data item, respectively.

**Transactions and Timestamps**. OCC-APFO associates with each active transaction a *timestamp interval* expressed as a [*lower bound (lb), upper bound (ub)*] pair. The timestamp interval denotes the *validity interval* of a transaction. The timestamp intervals are also used to denote serialization order between transactions. For example, if $T_i$ (with timestamp interval $[lb_i, ub_i]$) is serialized before $T_j$ (with timestamp interval $[lb_j, ub_j]$), i.e., $T_i \rightarrow T_j$, then the following relation must hold: $ub_i < lb_j$. Each transaction at the start of execution is assigned a timestamp interval of $[0, \infty]$, i.e., the entire timestamp space. As the transaction proceeds through its lifetime in the system, its timestamp interval is adjusted to reflect serialization dependencies as they are induced.[2] Serialization dependencies may be induced in two ways:

1. by accessing data items in the read phase. In this case, the timestamp interval is adjusted with regard to the read and write timestamps of the data item read or updated and
2. by being in the conflict set of a different validating transaction. In this case, the timestamp interval is modified to dynamically adjust the serialization order.

In the process of adjusting, the timestamp interval may *shut out*, i.e., become null. In that case, the transaction cannot be successfully serialized and needs to be restarted. Note that this is one of the major differences between conventional protocols and protocols based on dynamic adjustment of serialization order. In conventional OCC algorithms, restarts can only occur at validation times. In our case, however (as well as in OCC-TI), transactions can restart at other times if a timestamp interval shut out is detected. The exact mechanics for these adjustments are shown in the procedures given later on in Section 4.2.4.

In this paper, we use the notation $TI(T_i)$ to denote the timestamp interval of transaction $T_i$ and $RTS(D_i)$ and $WTS(D_i)$ to denote the read and write stamps of data item $D_i$, respectively. As a transaction successfully validates, a final timestamp is assigned to it. We assume that this timestamp is equal to the lower bound of its final timestamp interval. The notation $TS(T_{val})$ is used to denote the final timestamp of $T_{val}$ after its validation.

Next, we turn our attention to the adaptive priority cognizance of OCC-APFO, which is one of the strengths of this and distinguishes it from other CC algorithms. The reader is reminded that the goal of priority cognizance is to smartly decide when to sacrifice validating transactions in favor of its usually large conflict set. Priority cognizance in OCC-APFO is designed around a property of real-time active transactions that we define. We call this property the *Concurrency Priority Index* (CPI).

The CPI of a transaction is a measure of its candidacy for restart, i.e., a measure of a transaction's attractiveness for restart. In priority conscious OCC algorithms, a transactions priority determines its candidacy for restart. For example, if there was a choice between restarting transaction $T_X$ (with priority $P_X$) and transaction $T_Y$ (with priority $P_Y > P_X$), a priority conscious OCC would restart $T_X$, owing to its lower priority. In RTADBSs, however, simple priority is not enough as we have to account for dependent transactions as well. In response to this inadequacy, we came up with the notion of CPI to be the determinant of which transaction to restart rather than simple priority. In OCC-APFO, the CPI of a transaction is defined as: where $CPI_T$, $P_T$, and $FO_T$ denote the CPI, priority, and fan out of transaction $T$, respectively.

2. Note that the notion of using timestamp intervals to record dependencies is an established one (see e.g., [32]). It is just used in this paper as a convenient tool and is not a research contribution of this paper.

```
if lower CPI irreconciliably conflicting transaction in CS(T_val) then
   if T_val likely to complete if restarted then
      restart T_val
```

Fig. 8. Restart rule.

Intuitively, we view CPI as a measure of a transaction's candidacy for restart. In other words, a transaction with a higher CPI is more attractive as a candidate for restart than one with a lower CPI. We illustrate this with the following simple example: Assume there are two conflicting transactions, say $T_R$ and $T_S$, with fan outs $FO_R$ and $FO_S$, respectively. Assume, for simplicity, that the transactions have identical priorities. Further assume that $FO_R > FO_S$. Then, $CPI_R < CPI_S$ (using the CPI expression above), signifying that the transaction manager would rather restart $S$ than $R$, owing to $S$'s higher CPI. One can easily see this makes sense intuitively: Both $R$ and $S$ have similar priorities. However, $FO_R > FO_S$, indicating $R$ has triggered more transactions than $S$. Thus, restarting $R$ has a higher likelihood of wasting more work than that by restarting $S$. Intuitively, therefore, it makes sense to restart $S$.

The basic idea of OCC-APFO is to restart validating transactions when the following two conditions are satisfied:

1. transactions with a lower CPI exist in the *irreconcilably conflicting* set and
2. it is likely that, if restarted, the validating transaction will eventually commit.

The first condition is easily verified by examining the conflict set in the validation phase and comparing the CPIs of the conflicting transaction to that of the validating transaction. The second condition is checked by estimating how long the transaction will take to execute, if restarted. However, it is well known that this is a nontrivial problem that researchers often tackle by assuming prior knowledge of system dynamics. Our goal is to make no assumptions that are even remotely unrealistic.

On the first run of a transaction through the system, we record its data access behavior. Thus, if a transaction reaches validation, we are aware of its complete read and write sets. We denote the read and write sets of a transaction $T$, by $RS(T)$ and $WS(T)$, respectively. From this information, we can make a worst-case estimate (i.e., assuming each access request results in a page fault) of the *isolated running time* of this transaction, i.e., assuming it is running by itself in the system. Denoting isolated runtime (i.e., runtime assuming $T$ is alone in the system) of $T$ as *IRT(T)*, and assuming $WS(T) \subseteq RS(T)$, we can say: where *ProcCPU* and *ProcDisk* are the times needed to process a page at the CPU and disk, respectively, and are system parameters.

Let us now denote the *concurrent run time* (i.e., the run time when $T$ is not alone in the system) of $T$ as *CRT(T)*. Then, we can say:

$$CRT(T) = F(IRT(T), system\ dynamics).$$

The above expression says that the concurrent running time of $T$ depends on the isolated running time of $T$ and the state of the system. We approximate this function by the following expression:

$$CRT(T) = \alpha \times IRT(T),$$

where $\alpha$ is a control variable of the algorithm, $\alpha \geq 1.0$. Note that, strictly speaking, $\alpha < 1.0$ is feasible, as *IRT(T)* denotes the worst-case isolated runtime. In all our experiments, however, we enforce the greater than 1 inequality.

The dynamic variable $\alpha$ represents the state of the system, i.e., the system dynamics. It is easily seen that a smaller $\alpha$ indicates a less contention-oriented system than does a larger $\alpha$. We can also say:

$$\lim_{\alpha \to 1} CRT(T) = IRT(T).$$

The OCC-APFO algorithm uses a feedback mechanism to monitor the performance of the system, and then adjusts the value of $\alpha$ accordingly ($\alpha$ is initialized to 3.0 at system startup). The feedback is in the form of the *miss ratio of restarted validating transaction* denoted by *MRV*. *MRV* is given by:

$$MRV = \frac{\#transactions\ restarted\ in\ validation\ phase\ and\ subsequently\ missed}{\#transactions\ restarted\ in\ validation\ phase}.$$

Recall that the goal of OCC-APFO is to miss as few restarted validating transactions as possible. A high *MRV* indicates that we are *underestimating* the runtimes (i.e., the *CRTs*), and, consequently, missing too many restarted validating transactions. On the other hand, a low value of *MRV* indicates that $\alpha$ is doing a better job in representing system dynamics.

The value of $\alpha$ is dynamically adjusted with the goal of keeping *MRV* below 5 percent. *MRV* is recomputed after every *SampleReq* transactions request validation, where *SampleReq* is a parameter of the system. If *MRV* is greater than or equal to 5 percent, we are underestimating $\alpha$, and its value is increased by 5 percent. On the other hand, if *MRV* is less than 5 percent, then we have the opportunity to relax the restart condition and possibly extract better performance from the system. Therefore, in this case, $\alpha$ is reduced by 5 percent.

Based on the above description of estimating the runtime of transactions, we are now in a position to state our restart condition for validating transactions. Assuming $T_{val}$ requests validation and priority and deadline of transaction $T$ are denoted by $P(T)$ and $D(T)$, respectively, the restart rule may be stated as Fig. 8.

#### 4.2.4 The OCC-APFO Algorithm

OCC-APFO basically has three procedures of importance: 1) *validation* ($\mathcal{VAL}$) which is run at validation, 2) *timestamp-interval adjustment at validation* ($\mathcal{TAV}$), which adjusts the timestamp intervals of conflicting transactions during a validation process, and 3) *timestamp-interval adjustment by data access* ($\mathcal{TADA}$), which adjusts the timestamp intervals

TABLE 1
Notations used in this Paper

| Notation | Description |
|---|---|
| *TI(T)* | Timestamp Interval of $T$ |
| *TS(T)* | Timestamp of $T$, awarded after validation, = lower bound of *TI(T)* |
| *RS(T)* | Read set of $T$ |
| *WS(T)* | Write set of $T$ |
| *CPI* | Concurrency Priority Index |
| *CPI(T)* | CPI of $T$ |
| *D(T)* | Deadline of $T$ |
| *CS(T)* | Conflict set of $T$ |
| *CRT(T)* | Concurrent run time of $T$ |
| *RTS(D)* | Read time stamp for data item $D$ |
| *WTS(D)* | Write time stamp of data item $D$ |

of the transactions in their read phase as they access data items. The notation used is summarized in Table 1.

The first thing procedure $\mathcal{VAL}$ (Fig. 9) does is to check that the validating transaction has no uncommited guardian transactions in the system. Recall from Section 2 that our execution model requires dependent transactions to be serialized after guardian transactions. This requirement cannot be guaranteed if a dependent transaction is allowed to validate while some guardian transaction is still running. In that case, the transaction is made to wait a random amount of time before requesting validation again. If it is decided that the validating transaction has to wait, the procedure terminates.

Once a transaction has been cleared to go through validation, there are essentially two things that can happen to it: 1) it can be restarted, allowing its conflict set to carry on or 2) it can be committed at the expense of some transactions in its conflict set (i.e., the irreconcilably conflicting transactions). This is achieved as follows: First, we go through the conflict set of $T_{val}$, adjusting the timestamps of the conflicting transactions by invoking procedure $\mathcal{TAV}$. Whenever an irreconcilably conflicting transaction is detected by virtue of the fact that its timestamp interval shuts out, we perform the restart test. If this test succeeds, $T_{val}$ is restarted and procedure $\mathcal{VAL}$ terminates. This situation is somewhat tricky to handle, because, by this time, the procedure may have already adjusted the timestamp intervals of several conflicting transactions. However, if $T_{val}$ is to restart, then these prior adjustments are unnecessary. To remedy this situation, we call the $\mathcal{RESET}$ procedure, which unmarks all transactions marked for restart and resets the timestamp intervals of all transactions in the *original* $CS(T_{val})$ to the values before the invocation of the procedure. This is very easily implemented by keeping an image of the original $CS(T_{val})$ until $\mathcal{VAL}$ terminates. This is also the reason why we only mark transactions for restart in $\mathcal{TAV}$ instead of actually restarting them. If the transactions were restarted in $\mathcal{TAV}$, and later $\mathcal{RESET}$ needed to be run, it would be impossible to undo the restarts at that stage. Finally, if $T_{val}$ restarts, $\mathcal{VAL}$ terminates. If the entire conflict set is traversed without the restart test succeeding, $T_{val}$ is guaranteed to commit, the necessary data timestamp adjustments are done, and all marked transactions are restarted.

Next, we turn our attention to the $\mathcal{TAV}$ procedure (Fig. 10). This procedure checks the *type* of conflict between the validating and the conflicting transaction and accordingly adjusts the timestamp interval of the conflicting transaction. Below, we provide the details of this adjustment for each of the three different kinds of conflict:

**Procedure** $\mathcal{VAL}(T_{val})$

*This procedure is invoked when a transaction, $T_{val}$, requests validation*

```
if any guardian transaction of T_val is not committed then
  {
    wait random amount of time before attempting to validate again;
    exit;
  }
foreach T_j ∈ CS(T_val) do
  {
    TAV(T_j);                    \*  adjust timestamp interval of T_j  *\
    if TI(T_j) = [] then          \*T_j shuts out, i.e., it is irreconciliably
                                      conflicting with T_val  *\
        if CPI(T_val) > CPI(T_j) then
              if CRT(T_val) ≤ (D_T_val − Current_Time) then
                              restart T_val;
                              RESET(CS(T_val));
                              exit;
  }
foreach D_i ∈ RS(T_val)
    if RTS(D_i) < TS(T_val) then
        RTS(D_i) = TS(T_val);
foreach D_j ∈ WS(T_val)
    if WTS(D_j) < TS(T_val) then
         WTS(D_j) = TS(T_val);
restart all transactions marked for restart; \* recall that this marking
                                              was done in procedure TAV *\
commit T_val;
exit;
```

Fig. 9. Procedure $\mathcal{VAL}(T_{val})$.

**Procedure** $\mathcal{TAV}(T_j)$

*This procedure is called from inside the* $\mathcal{VAL}$ *procedure, whenever a transaction* $T_j$ *requires adjustment of timestamp intervals by virtue of being in* $T_{val}$*'s conflict set*

```
foreach D_i ∈ RS(T_val)
{
    if D_i ∈ WS(T_j) then                /*  read-write conflict  */
        TI(T_j) ← TI(T_j) ∩ [TS(T_val), ∞];
    if TI(T_j) = [] then
        mark T_j for restart;
}
foreach D_j ∈ WS(T_val)
{
    if D_j ∈ RS(T_j) then                /*  write-read conflict  */
        if T_val is a guardian of T_j then
            mark T_j for restart;
        else TI(T_j) ← TI(T_j) ∩ [0, TS(T_val) − 1];
    if D_j ∈ WS(T_j) then                /*  write-write conflict  */
        TI(T_j) ← TI(T_j) ∩ [TS(T_val), ∞];
    if TI(T_j) = [] then
        mark T_j for restart;
}
```

Fig. 10. Procedure $\mathcal{TAV}(T_j)$.

**Procedure** $\mathcal{TADA}$

*This procedure is called in the read phase of active transactions whenever there is a data access*

```
if transaction wants to execute a read access on data item D_i then
{
    read(D_i);
    TI(T_j) ← TI(T_j) ∩ [WTS(D_i), ∞];
    if TI(T_j) = [] then
        restart T_j;
}
if transaction wants to execute a write operation on data item D_j then
{
    write D_j in local buffer;
    TI(T_j) ← TI(T_j) ∩ [WTS(D_j), ∞] ∩ [RTS(D_j), ∞];
    if TI(T_j) = [] then
        restart T_j;
}
```

Fig. 11. Procedure $\mathcal{TADA}$.

1. **Read-Write Conflict.** This type of conflict occurs when there is some data item in the read set of the validating transaction $T_{val}$ that also exists in the write set of the conflicting transaction $T_j$, i.e., $RS(T_{val}) \cap WS(T_j) \neq \emptyset$. In this case, we adjust $T_j$'s timestamp to induce the serialization order $T_{val} \rightarrow T_j$. This is known as *forward ordering* [32]. The logic behind this ordering is that the writes of $T_j$ should not affect the read phase of $T_{val}$.
2. **Write-Read Conflict.** This type of conflict occurs when there is some data item in the write set of the validating transaction $T_{val}$ that also exists in the read set of the conflicting transaction $T_j$, i.e., $WS(T_{val}) \cap RS(T_j) \neq \emptyset$. In this case, the only possible adjustment is to induce the serialization order $T_j \rightarrow T_{val}$. This is known as *backward ordering* [32] and signifies that the writes of $T_{val}$ have not affected the read phase of $T_j$. In RTADBSs, this ordering is tricky and must be considered in context of the relationship between the validating and the conflicting transaction:
   - **Case 1—$T_{val}$ is a guardian of $T_j$.** In this case, backward ordering is invalid as guardian transactions must be serialized before dependent transactions. Thus, $T_j$ must be restarted.
   - **Case 2—$T_{val}$ is not a guardian of $T_j$.** In this case, we simply perform the backward ordering and set the serialization order as $T_j \rightarrow T_{val}$.
3. **Write-Write Conflict.** This type of conflict occurs when there is some data item in the write set of the validating transaction $T_{val}$ that also exists in the write set of the conflicting transaction $T_j$, i.e., $WS(T_{val}) \cap WS(T_j) \neq \emptyset$. In this case, we forward the order by adjusting $T_j$'s timestamp to induce the serialization order $T_{val} \rightarrow T_j$. The logic behind this ordering is that the writes of $T_{val}$ should not overwrite the writes of $T_j$.

Finally, we state our third procedure $\mathcal{TADA}$ (Fig. 11). This procedure simply adjusts the timestamp intervals with each data access to ensure consistency is not violated. The reader should have no problems understanding the logic behind the adjustments.

#### 4.2.5 Correctness of OCC-APFO

To prove that OCC-APFO is correct, we show that it can only produce serializable histories. This is shown using the notion of *precedence graphs* [27]. We use the term $H$ to denote history and the term $PG^H$ to denote the precedence graph corresponding to the history $H$. By definition, if an edge $(T_1, T_2)$ exits in $PG^H$, then $T_1$ precedes $T_2$ in logical order.

**Lemma 1.** *If an edge $(T_1, T_2)$ exists in $PG^H$, where $T_1$ and $T_2$ are two committed transactions, then $TS(T_1) < TS(T_2)$.*

**Proof.** By definition, there can be three kinds of conflicts between $T_1$ and $T_2$:

1. $w_1(Q)$ precedes $r_2(Q)$. This means $T_2$'s read has seen the effect of $T_1$'s write, i.e., $T_1$ committed and executed its write phase before $T_2$ executed $r_2(Q)$ in its read phase. Clearly, during $T_1$'s write phase, the write timestamp of data item $Q$ is updated, using procedure $\mathcal{VAL}$ as follows:
$$WTS(Q) = TS(T_1). \quad (1)$$
Subsequently, when $T_2$ executes $r_2(Q)$ in its read phase, the lower bound of $TI(T_2)$ is set to $WTS(Q) + 1$, using procedure $\mathcal{TADA}$. After this, whatever the final timestamp of $T_2$ turns out to be, the following is always true:
$$TS(T_2) > WTS(Q). \quad (2)$$
Using (1) and (2) above, $TS(T_1) < TS(T_2)$.
2. $r_1(Q)$ precedes $w_2(Q)$. Using similar arguments as in the previous case, it can be shown that $TS(T_1) < TS(T_2)$.
3. $w_1(Q)$ precedes $w_2(Q)$. Using similar arguments as in (1), it can be shown that $TS(T_1) < TS(T_2)$.□

**Theorem.** *OCC-APFO exclusively generates serializable histories.*

**Proof.** Assume that in the $PG^H$ of some history $H$ generated by OCC-APFO, there exists a cycle $(T_1, T_2, T_3, \ldots, T_n, T_1)$. Using the results of Lemma 1, this implies

$$TS(T_1) < TS(T_2) < \ldots < TS(T_n) < TS(T_1).$$

This results in an unresolvable contradiction. Thus, $PG^H$ must be acyclic, implying the history must be serializable. □

### 4.3 OCC-APFS

OCC-APFS (*Adaptive Priority Fan-out Sum*) is a second algorithm that we stipulate. OCC-APFS is exactly the same as OCC-APFO, with one critical difference. Recall that CPI in OCC-APFO (which we will refer to as CPI(FO)) was defined as follows:

$$CPI(FO)_T = \frac{1}{P_T \times FO_T},$$

where $FO_T$ attempted to quantify the amount of work generated by $T$. We felt that one possible refinement of this quantifying measure was to account for the priorities of dependent transactions (instead of simply their number as given by $FO_T$). Thus, we defined a new measure called *Fan-out Sum* (FS).

**Definition.** *The Fan-out Sum of a transaction $T$, denoted by $FS_T$, is the sum of the priorities of all dependent transactions of $T$.*

Thus, while $FO_T$ denotes the *number* of dependent transactions of $T$, $FS_T$ denotes the cumulative importance of these dependent transactions. The idea was that $FS_T$ would be able to capture nuances of subtransaction priorities that would otherwise escape $FO_T$. The new CPI value in OCC-APFS (called CPI(FS)) is defined as:

$$CPI(FS)_T = \frac{1}{P_T \times FS_T}.$$

The reason why OCC-APFS appears to improve on OCC-APFO is easily seen by examining Example 2 and Fig. 4 in Section 3. First, we assign the following priority values to the four transactions depicted in the figure ($P_i$ refers to the priority of transaction $T_i$, the *earliest-deadline* principle is used to assign priorities, and larger priority values denote higher priorities): $P_1 = 3$, $P_2 = 2$, $P_3 = 4$, and $P_4 = 1$. OCC-APFO first computes the fan-out values of the conflicting transactions: $FO_{T_1} = FO_{T_3} = 1$ This implies that $CPI(FO)_{T_1} = \frac{1}{3\times1} = .33$ and $CPI(FO)_{T_3} = \frac{1}{4\times1} = .25$. This makes $T_1$ a more attractive candidate for restart than $T_3$ owing to its larger CPI value. This decision, as argued in Example 2, is undesirable.

Now, let us consider the effect of using FS, instead of FO. It is easily seen that $FS_{T_1} = 3$ and $FS_{T_3} = 1$. Thus, $CPI(FS)_{T_1} = \frac{1}{2\times3} = .17$ and $CPI(FO)_{T_3} = \frac{1}{1\times4} = .25$. This makes $T_3$ a more attractive candidate for restart than $T_1$ owing to its larger CPI value. This decision, as argued in Example 2, is correct.

Aside from this important difference, the OCC-APFS algorithm is the same as OCC-APFO in other respects.

### 4.4 An Important Implementation Issue

With regard to both OCC-APFO and OCC-APFS, a question that will need to be answered frequently is: "Given two transactions $T_i$ and $T_j$, is $T_i$ a guardian of $T_j$?" We hasten to add that this is not particular to our algorithms. Any algorithm (such as OCC-BC, OCC-TI) that allows concurrency between guardian and dependent transaction will need to answer the above question in order to serialize dependent transactions after their guardians. In other words, we feel that the above question represents a very general problem that will need to be addressed in RTADBSs. Below, we provide an efficient implementation scheme in order to address this problem.

Basically, the above problem reduces to designing an efficient storage and access mechanism for the triggering graph structure defined in Section 2. Then, the above question reduces to the following alternate question: "Is *FROM* $(T_j)$ reachable from *ON* $(T_i)$?" One basic assumption that we make is that triggering graphs are *sparse*. This seems reasonable as 1) the triggering graph is acyclic, 2) the triggering graph is directed, and 3) one would expect the number of transactions in the system to be of the order of the number of parent-child relationships. Then, we propose representing the triggering graph with an *adjacency list*. For a triggering graph with $V$ nodes, its adjacency list consists of an array of $V$ lists, $l_1, l_2, \ldots, l_V$, where $l_i$ represents the adjacency structure of vertex $v_i$. Fig. 12a shows a simple triggering graph and Fig. 12b shows its corresponding

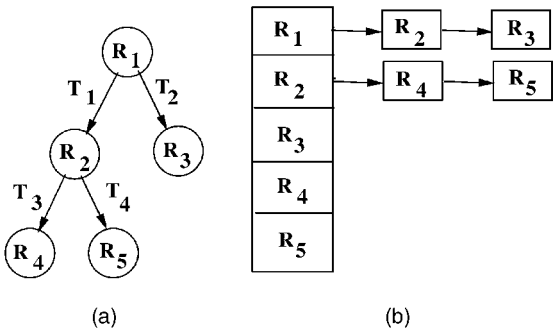



Fig. 12. Implementation of triggering graphs.

adjacency list. Clearly, reachability questions can be very easily answered efficiently from the adjacency list by performing a *depth first search* (DFS). It is well-known that the worst-case time complexity of DFS is $O(V + E)$, where $V$ and $E$ are the number of vertices and edges, respectively.

## 5 Real-Time Active Database System Model

In this section, we present a synopsis of our ARTDB model to aid readers in better understanding the performance analysis results. This model was simulated in SIMPACK [18], a C-based simulation toolkit.

The system consists of shared-memory multiprocessors that operate on disk-resident data. We model the database as a collection of pages. A transaction is nothing but a sequence of read and write page accesses. A read request submits a data access request to the concurrency control (CC) manager, on the approval of which, a disk I/O is performed to fetch the page into memory followed by CPU usage to process the page. Similar treatment is accorded to write requests with the exception that write I/Os are deferred until commit time. Until this point, our model is similar to that in [4]. The rest of this model is our contribution.

An important aspect of our model is the handling of triggered transactions. We partition the data items in the database into two mutually exclusive sets: *reactive* and *nonreactive*. We also maintain a set of condition-action (CA) rules (i.e., rules of the form: *if* **condition** *then* **action**). Each rule *subscribes* to one or more reactive data items. The update of a reactive attribute $X$ immediately raises events, which result in the evaluation of the condition of the rule(s) which subscribe to $X$. Finally, upon the satisfaction of the condition part, the action (A) part is triggered as a transaction. Our RTDBS model is shown in Fig. 13. There are four major components:

1. An *arrival generator* that generates the real-time workload with deadlines.
2. A *transaction manager* that models transaction execution and implements the earliest-deadline (ED) [34] algorithm.
3. A *concurrency controller* that implements the CC algorithm.
4. A *resource manager* that models system resources, i.e., CPUs and disks and the associated queues.
5. A *rule manager* that models the rule base as well as the subscription information. The rule manager is responsible for triggering the active workload.

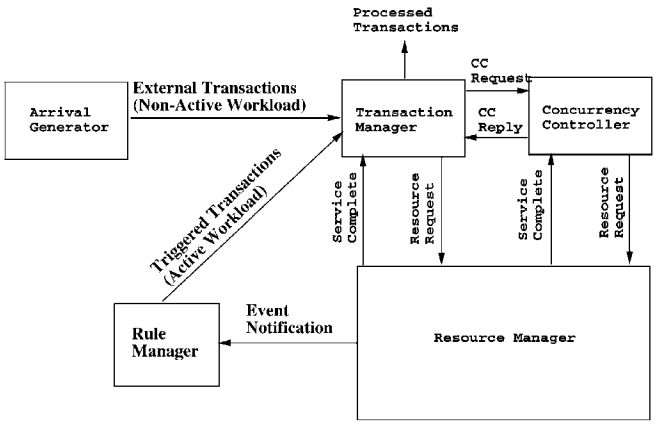



Fig. 13. The RTDBS model.

### 5.1 Resource Model

Our resource model considers multiple CPUs and disks as the physical resources. For our simulations, we assumed the data items are uniformly distributed across all disks. The *NumCPU* and *NumDisk* parameters specify the system composition, i.e., the number of each type of system resource. There is a single queue for the CPUs and the service discipline is assumed to be preemptive-resume based on the transaction priorities. Each individual disk has its own queue with a nonpreemptive, transaction priority-based service discipline. The parameters *ProcCPU* and *ProcDisk* denote the CPU and disk processing time per page, respectively. The total processing time per page is denoted as $Proc_T = ProcCPU + ProcDisk$. These parameters are summarized in Table 2.

Buffers are not modeled explicitly. However, a reasonable buffer management policy is simulated as follows: At any time, all the pages accessed by active transactions are assumed to be available. As soon as a transaction leaves the system, its pages, unless accessed by some other active transaction, are assumed to have left the buffer.

### 5.2 Workload Model

Our workload model consists of modeling the characteristics of transactions that arrive and are processed in the system as well as their arrival rate. In ARCS databases, the workload is mixed, i.e., it consists of triggered transactions and transactions that arrive from outside the system. We refer to the triggered transactions as *active workload* and the external transactions as *nonactive* workload. Below, we

TABLE 2
Input Parameters to our RTDBS Model

| Parameter Type | Notation | Description |
|---|---|---|
| Resource Parameter | *NumCPU* | Number of CPUs |
| | *NumDisk* | Number of disks |
| | *ProcCPU* | CPU time /data page |
| | *ProcDisk* | Disk time/data page |
| Workload Parameter | *ArrivRate* | Arrival Rate of External Transactions |
| | *DBSize* | Number of pages in the database |
| | *NumItemsPerPage* | Number of data items per page |
| | *ReacFrac* | Fraction of data items that are reactive |
| | *TrigProb* | Probability that a transaction will be triggered following the writing of a reactive attribute |
| | *WriteProb* | Write probability/accessed page |
| | *SizeInterval* | Range of the number of pages accessed per transaction |
| | *SRInterval* | Range of slack ratio |

describe how we generate the characteristics of the two different workload types mentioned above.

#### 5.2.1 Nonactive Workload

We consider two broad classes of transaction characteristics: 1) $Size_T$, which denotes the number of pages accessed by $T$ and 2) $D_T$, the deadline of $T$. The service demand of $T$ is denoted as $SD_T = Size_T \times Proc_T$ (recall that $Proc_T$ is the time required to process a page). The arrival generator module assigns a deadline to each transaction using the following expression: $D_T = SD_T \times SR_T + A_T$, where $D_T$, $SR_T$, and $A_T$ denote the deadline, slack ratio, and arrival time of transaction $T$, respectively. Thus, the time constraint of transaction $T$, $C_T = D_T - A_T = SR_T \times SD_T$. In other words, $SR_T$ determines the tightness of the deadline of $T$.

#### 5.2.2 Active Workload

Our active workload consists of triggered transactions. These transactions are triggered with the update of a reactive attribute. The basic characteristics of active workload are the same as those of the nonactive workload with two differences.

First, the deadline assignment policy is somewhat different. Consider a transaction $T$ with deadline $D_T$ that triggers transaction $T_{trig}$. We first compute the deadline of $T_{trig}$ the same way as it is done for an external transaction, as shown in Section 5.2.1, i.e., $D_{T_{trig}} = SD_{T_{trig}} \times SR_{T_{trig}} + A_{T_{trig}}$. However, in the case of active workload, we must ensure that the deadline of a triggered transaction is no earlier than that of any of its guardian transactions (recall the discussion in item 2, Section 2.4). In order to ensure this after we have computed $D_{T_{trig}}$ the usual way, we check the following condition: $D_{T_{trig}} > D_T$ (recall $D_T$ is the deadline of the transaction that triggered $T_{trig}$). If this condition is satisfied, then we do nothing. If this condition is not satisfied, however, a new deadline is assigned to $T_{trig}$ as follows: $D_{T_{trig}} = D_T + \Delta$, where $\Delta$ is a uniform random variable drawn from the range $[\Delta^-, \Delta^+]$.

The second difference between nonactive and active workloads is the respective arrival patterns. Nonactive workload, i.e., external transactions, arrive according to some predefined distribution pattern. On the other hand, active workloads are generated as follows: Following the update of a reactive attribute, a transaction is triggered probabilistically. This models the fact that every time a reactive attribute is updated (i.e., an event is raised), the condition of the rule which subscribes to this attribute may not be satisfied. The relevant parameters that we use to model this scenario are *ReacFrac* and *TrigProb*. *ReacFrac* is the fraction of data items in the database that are reactive. In other words, whenever a transaction performs a write operation, the data item written is taken to be a reactive data item with probability *ReacFrac*. The parameter *TrigProb* denotes the probability that given an event, the corresponding rule condition will be satisfied. In other words, given that a reactive attribute has been modified, a triggered transaction will be generated with probability *TrigProb*.

Our workload parameters are summarized in Table 2. Table 2 contains some parameters not discussed thus far. The *ArrivRate* and *DBSize* parameters are self explanatory. The value of the *WriteProb* parameter denotes the probability with which each page that is read will be updated. *SizeInterval* denotes the range within which transaction sizes will uniformly belong. In other words, the arrival generator module generates transaction sizes by drawing from a uniform distribution whose range is specified by *SizeInterval*. Similarly, transaction slacks are generated by drawing from the uniform distribution whose range is specified by *SRInterval*.

TABLE 3
Parameter Settings for Baseline Experiments

| Parameter | Value |
|---|---|
| *NumCPU* | 4 |
| *NumDisk* | 8 |
| *ProcCPU* | 10ms |
| *ProcDisk* | 20ms |
| *DBSize* | 1000 pages |
| *NumItemsPerPage* | 4 |
| *ReacFrac* | 1.0 |
| *TrigProb* | 0.03 |
| *WriteProb* | 0.3 |
| *SizeInterval* | [1,30] |
| *SRInterval* | [2,6] |

## 6 Performance Analysis

### 6.1 Performance Metrics

The primary performance metric used is *miss ratio*, or the fraction of transactions that miss their deadlines, calculated as:

$$\text{miss ratio (MR)} = \frac{\text{number of transactions missing deadline}}{\text{total number of transactions arriving into the system}}.$$

In addition to MR, we also measure *Average Restart Count* (ARC). ARC is defined to be the average number of restarts incurred by a transaction before it leaves the system. Note that a transaction could leave the system both for having completed successfully or for having missed its deadline.

All the MR curves presented in this paper exhibit mean values that have relative half-widths about the mean of less than 10 percent at the 90 percent confidence level. Each simulation experiment was run until at least 50,000 transactions were processed by the system. We only discuss statistically significant differences in the ensuing performance reporting section.

### 6.2 Baseline Experiments

We first did a baseline experiment and, subsequently, studied the effects of changing system characteristics by varying one characteristic at a time. The values of input parameters for the baseline experiments are shown in Table 3. The value of the parameter *SampleReq* was set to 100 for all experiments. Note that in all experiments reported, we compare five algorithms: WAIT-50, 2PL-HP (the high priority variant of the two phase locking protocol [23]), OCC-TI, OCC-APFO, and OCC-APFS. OCC-BC was not considered as it has been conclusively shown in [32] that OCC-TI is superior to OCC-BC. **Note:** An important point to note is that WAIT-50, 2PL-HP, and OCC-TI were tuned to our RTADBS execution model, i.e., dependent transactions were serialized after as well as abort and commit dependent

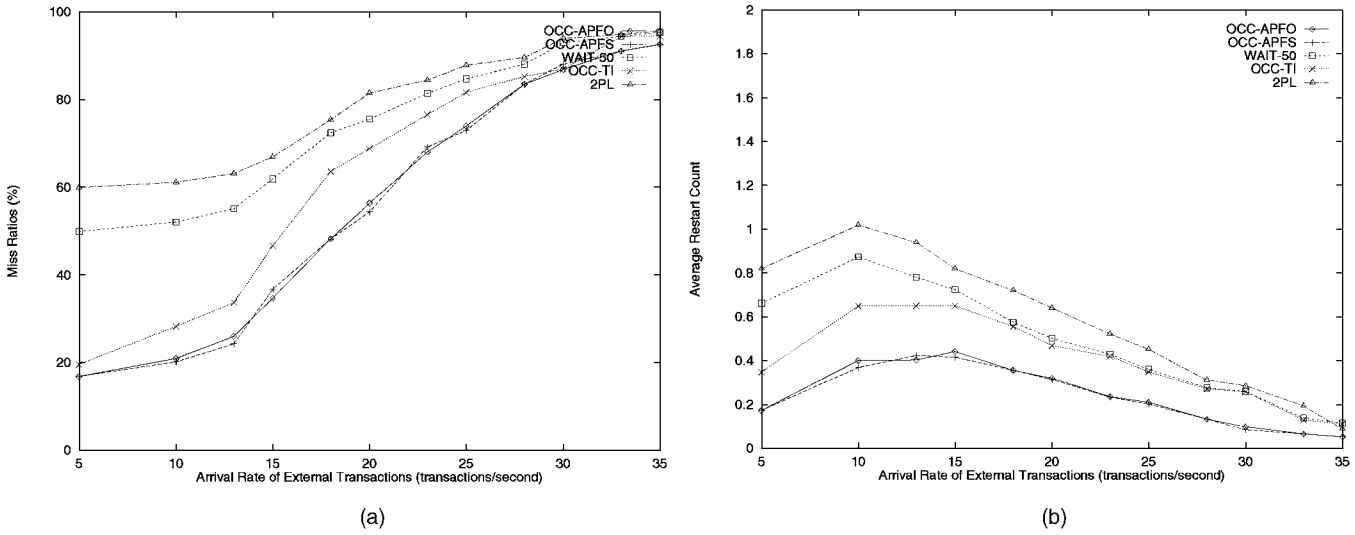



Fig. 14. Miss ratios and average restart counts in the baseline experiment.

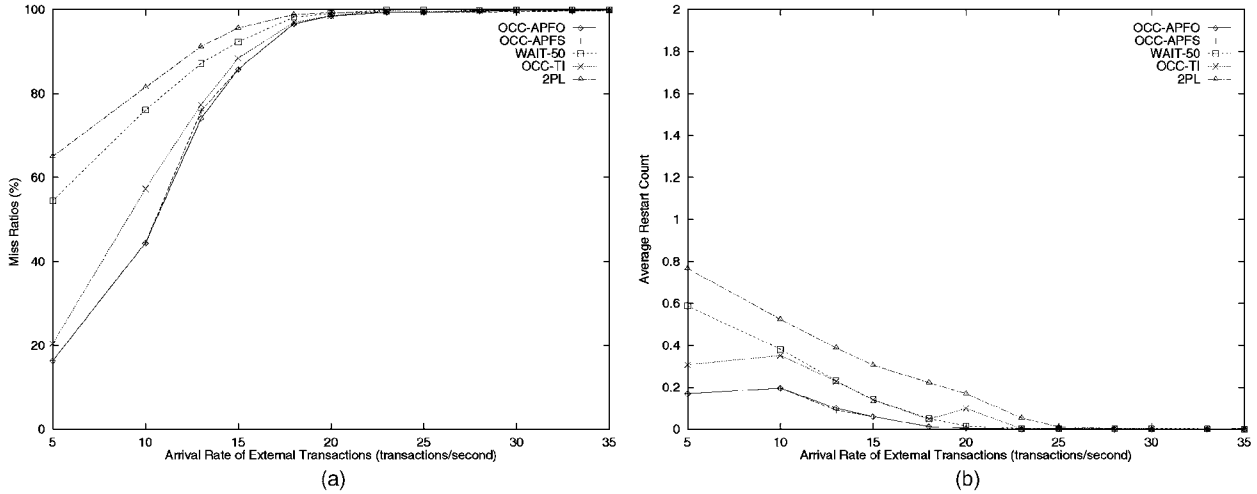



Fig. 15. Miss ratios and average restart counts with high resource contention (two CPUs, four disks).

on guardian transactions. Basically, we are interested in the effect of three specific system features: resource contention, data contention, and degree of triggering. The parameter settings in Table 3 represent moderate conditions in each of these three features. Fig. 14a and 14b, plot the two performance metrics at different system loads. System loading is varied by changing the value of the parameter *ArrivRate*. Note that *ArrivRate* is the arrival rate of external (i.e., nontriggered) transactions. Thus, the total system load is actually a function of both *ArrivRate* and *TrigProb*. Fig. 14a plots miss ratio versus transaction arrival rates. This graph shows that OCC-APFO and OCC-APFS (which we collectively refer to as OCC-APF* in the rest of the paper) perform virtually identically but clearly outperform the other two algorithms. 2PL-HP is clearly the worst performer followed by WAIT-50. OCC-TI and OCC-APF* are close at low system loads (arrival rate around five transactions/sec). However, at higher arrival rates, OCC-APF* performs progressively better, as evidenced by the OCC-APF* curve suffering the least degradation in performance with increasing load. At very high loads, however, the system gets saturated and the curves come close again. Note that the OCC-APF* curve remains below the other curves throughout the entire system loading range.

A very interesting and counterintuitive fact that emerges from Fig. 14a is that OCC-APFS performs the same as OCC-APFO. Recall that in Section 4.3, we argued through an example that OCC-APFS, by virtue of being cognizant of the priorities of dependent transactions, should perform better. On closer scrutiny, however, the reason for this apparent anomaly becomes clear. Even though OCC-APFS encodes more information than OCC-APFO, this information may actually mislead the algorithm. For example, consider a case where $T_X$ with priority $P_X$ has two low priority dependents $T_{xa}$ and $T_{xb}$, with priorities $P_{xa}$ and $P_{xb}$, respectively. Also, consider $T_Y$ with priority $P_Y$ and a high priority dependent $T_{ya}$ with priority $P_{ya}$. Assume $P_X = P_Y$ and $P_{ya} > P_{xa} + P_{xb}$. Further, assume that $W_g$ units of work has been performed on each of $T_X$ and $T_Y$, and $W_d$ units of work has been done on each of $T_{xa}$, $T_{xb}$, and $T_{ya}$. Under these situations, if $T_X$ and $T_Y$ were to conflict and one of these were to request validation, OCC-APFO would restart $T_Y$ while OCC-APFS would restart $T_X$. Note that the decision taken by OCC-APFS (i.e., to restart $T_X$ and, consequently, abort its dependents) results in

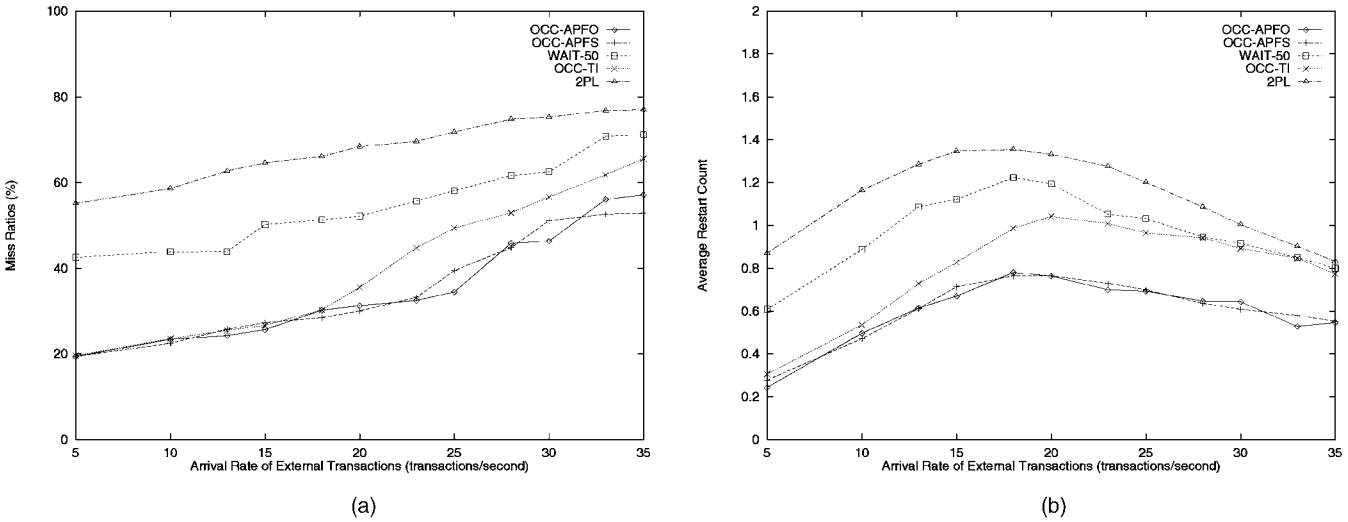



Fig. 16. Miss ratios and average restart counts with low resource contention (eight CPUs, 16 Disks).

greater wasted work. Thus, while in some cases OCC-APFS does make better decisions than OCC-APFO (as shown in Section 4.3), in certain cases, its extra information acts as a disadvantage. For this reason, OCC-APFS does not unilaterally outperform OCC-APFO. It is also easily seen that OCC-APFO has lower cost than OCC-APFS, as the fan-out sum need not be computed in OCC-APFO.

Fig. 14a is explained by Fig. 14b, which plots the average restart count at different system loads. As mentioned previously, restart count is a major determinant of the performance of real-time CC algorithms. It is seen from this graph that the OCC-APF* curves are consistently below the other curves, signifying that OCC-APF* causes fewer restarts than the other protocols. 2PL-HP has the highest restart count followed by WAIT-50. 2PL-HP's high restart counts are a result of the well-known phenomenon of *wasted restarts* [23], i.e., a transaction which is later discarded ends up restarting other transactions. Also, as mentioned earlier, WAIT-50's performance degradation is caused by both *wasted restarts* as well as *unnecessary restarts*. OCC-TI has higher restarts than OCC-APF* as it restarts a larger number of transactions on average at each validation.

Another interesting thing regarding Fig. 14b is the *bell shaped* nature of the restart count curves. This means that restart count goes up to a certain level and then goes down. This happens as a result of the data contention-resource contention trade-off. Up to a certain system loading level (in our case, in the range 10-15 transactions/second), data contention dominates resource contention. As a result, restart counts rise. After this point, resource bottlenecks dominate. This results in transactions spending time in resource queues as a consequence of which, fewer transactions can go through their entire working sets. This means that a lot of transactions miss their deadlines even before reaching the first validation phase. This reduces the restart count. As we shall see further on in this section, when resource contention is decreased by increasing resource levels, this problem is ameliorated and the restart count curves flatten out.

### 6.3 Effect of Varying Resource Contention

The effect of resource contention is studied by varying the *NumCPU* and *NumDisk* parameters. Fig. 15 reports the result of setting *NumCPU* and *NumDisk* to 2 and 4, respectively, and, consequently, increasing resource contention. Fig. 16 reports the result of relaxing resource contention by setting *NumCPU* and *NumDisk* to 8 and 16, respectively. These figures simply reinforce the results already reported. The relative performances on the various algorithms remain the same, with OCC-APF* performing singularly better than the other protocols. Note that, with high resource contention (Fig. 15), while the miss ratios predictably increase with respect to the baseline experiments, the average number of restarts decrease for all algorithms compared to the baseline results. This may seem counterintuitive. However, note that the effect of increased resource contention is to make jobs wait *longer* at resource queues, which means that jobs have less time to actually execute. This means that jobs get killed even before they can reach the point of being restarted. This reduces restart counts.

With increased resources (Fig. 16), resource contention dominates at much higher system loads. This explains why the restart count curves display less of a bell shape than the baseline curves. If one compares Fig. 16b with Fig. 14b, this fact is clearly evident. In fact, under low resource contention, as shown in Fig. 16b, the curve flattens out significantly.

### 6.4 The Effect of Varying Data Contention

Data contention levels are varied by changing the value of the *WriteProb* parameter. Below, we report the results of two experiments. Fig. 17 depicts the effects of reducing data contention from baseline levels by setting *WriteProb* to 0.1. Fig. 18 depicts the effects of increasing data contention from baseline levels by setting *WriteProb* to 0.5. In this figure, we note that the performance difference of 2PL and WAIT-50 undergoes a marked change from that reported so far. Unlike in all previous experiments, there is a performance crossover between these two algorithms. This is due to the fact that the number of restarts of WAIT-50 increases dramatically due to the high data contention levels, while 2PL is relatively immune to the same effects. A similar relative trend was observed in [23]. Aside from the abovementioned phenomenon, the other curves follow a similar relative trend as the previous experiments.

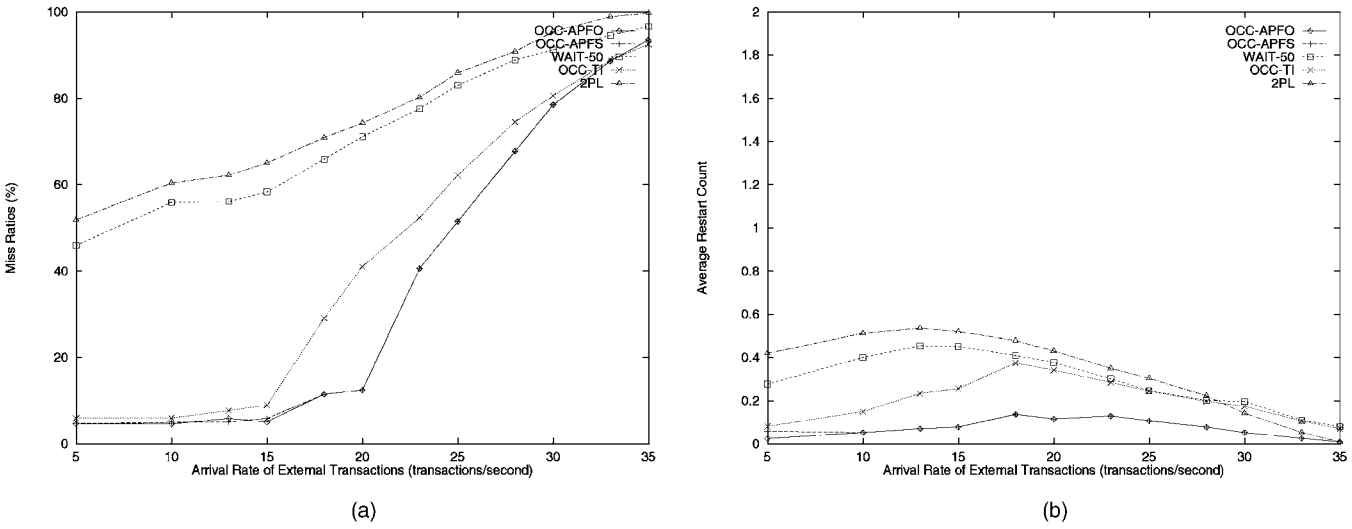



Fig. 17. Miss ratios and average start counts with low data contention ($WriteProb = 0.1$).

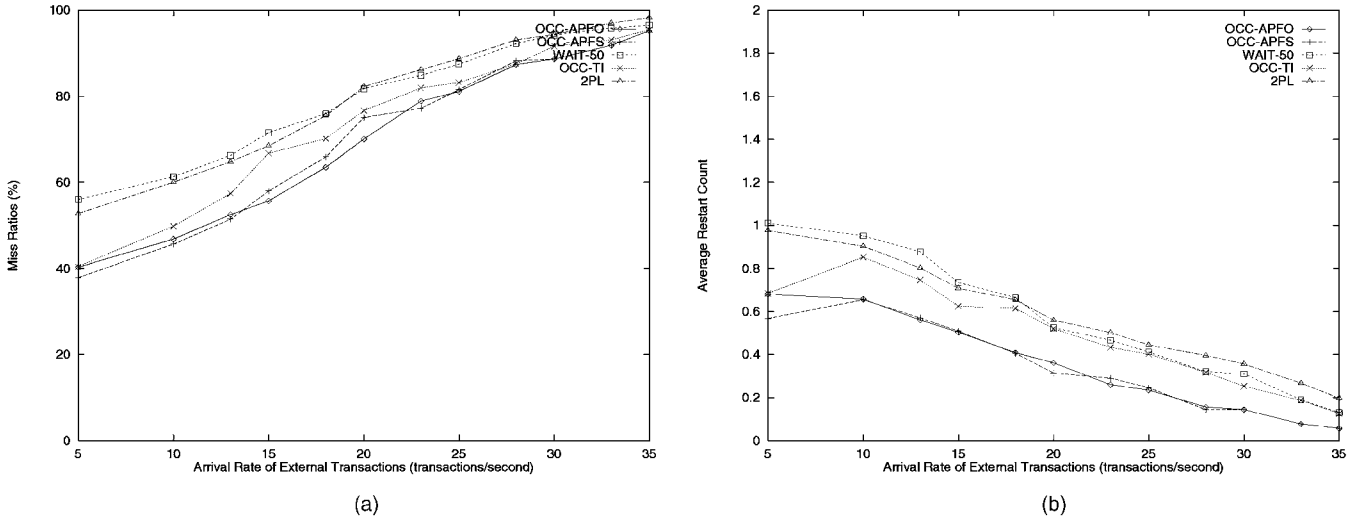


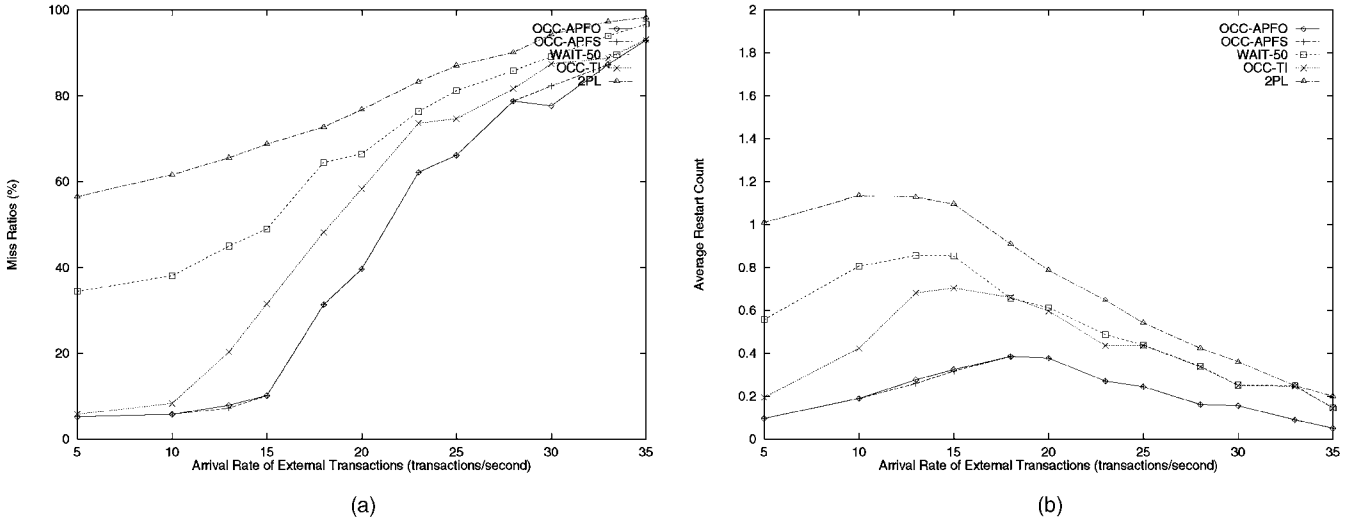

Fig. 18. Miss ratio and average restart counts with high data contention ($WriteProb = 0.5$).

### 6.5 Effect of Varying Transaction Load Through Triggering

In this experiment, we study the effects of varying the parameter *TrigProb*, which results in increasing or decreasing system load even more than that achieved by simply varying *ArrivRate*. Fig. 19 depicts the effects of reducing overall system load from baseline levels by setting *TrigProb* to $0.01$. Fig. 20 depicts the effects of increasing system loading from baseline levels by setting *TrigProb* to $0.05$. We do not expand on these results, as they are expected for brevity as well. Finally, since a number of examples in the literature assume single-level triggering, we ran an experiment to examine how our algorithms perform under this assumption. This was achieved by setting *TrigProb* to .2 and enforcing the rule that nontriggered transactions could further trigger other transactions. The results of this experiment are reported in Fig. 20. Note that since triggering is restricted to a single level, we had to substantially increase the trigger probability to get an appreciable active load in the system. While the relative trends of the curves remain the same, the interesting feature of this experiment was that OCC-TI performs almost identically to the OCC-APF* algorithms at high loads. We ran some additional experiments to examine why this was the case. While we do not report the results of these experiments here, the reason for this phenomenon was that, at high loads, the number of dependent transactions were virtually the same for single-level triggering with *TrigProb* of $0.2$ and multilevel triggering with a *TrigProb* of $0.03$. Thus, OCC-TI and OCC-APF* performed virtually identically patterns.

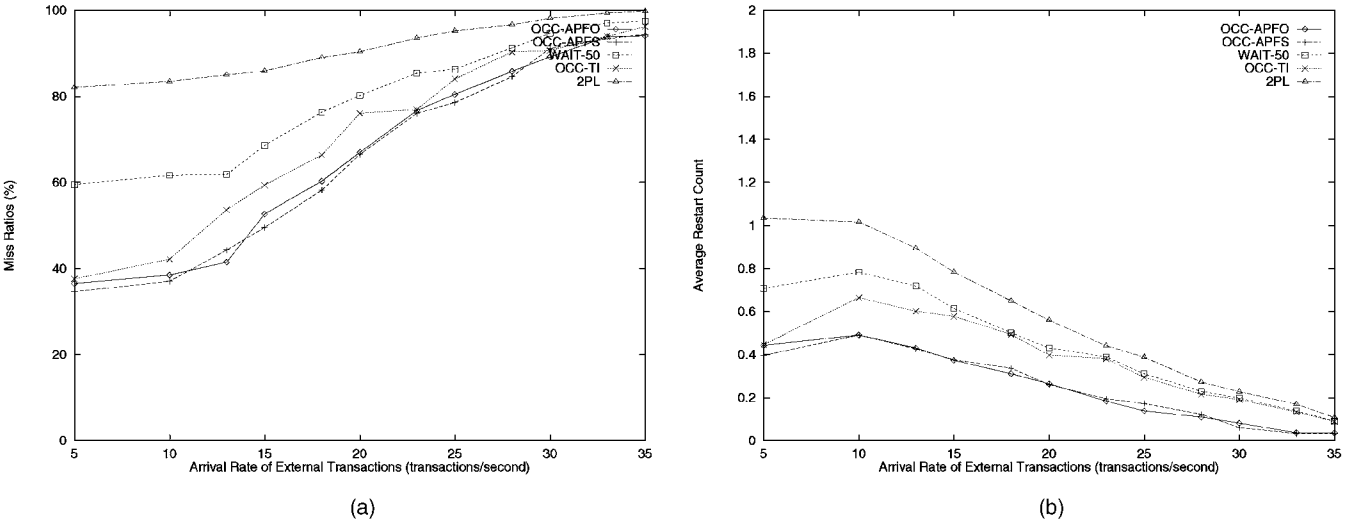
## 7 Discussion

In this section, we discuss the strengths and weaknesses of the OCC-APF* class of algorithms. OCC-APF* possesses all the advantages of OCC-BC such as the high degree of concurrency, freedom from deadlock, early detection and resolution of conflicts, and avoidance of wasted restarts. To the advantages of OCC-BC, it adds the major advantage of OCC-TI, i.e., the reduction of restarts by dynamically adjusting the serialization order. This results in the opportunity of not having to restart all transactions in the conflict set. In addition to all these advantages, OCC-APF* has two other unique characteristics: 1) it attempts to




(a) (b)

Fig. 19. Miss ratios and average restart counts with low system loads ($TrigProb = 0.01$).




(a) (b)

Fig. 20. Miss ratios and average restart counts with high system loads ($TrigProb = 0.05$).

account for the effect of triggering and 2) it attempts to reduce restarts even further by restarting the validating transaction (and thereby not restarting the members of the usually large conflict set) when it feels that the validating transaction would have a high likelihood of making it on its restarted run. This likelihood is judged by using a system state parameter $\alpha$, which is dynamically adjusted based on feedback from the system regarding its state. As we show in Section 6, these two unique characteristics have a substantial effect on the performance of OCC-APFO.

Aside from the two clear advantages mentioned in the previous paragraph, OCC-APF* enjoys one more subtle advantage over OCC-TI. As the authors themselves discuss in [32], OCC-TI remains effective as long as data contention is low (i.e., write probabilities are low). This happens because at low data contention levels, *reconciliable conflicts* outnumber *irreconcilable conflicts*. At high data contention levels, most conflicts turn out to be *irreconcilable*, eroding the effectiveness of OCC-TI. OCC-APF*, on the other hand, is much more immune to this problem. As data contention goes up, OCC-APF* is disadvantaged the same way as OCC-TI is, but this disadvantage is offset by the advantage reaped from larger conflict sets. It is easily seen that, at higher data contention levels, the size of the conflict set grows. Thus, for each restart of the validating transaction, a proportionally *larger* number of conflicting transactions are given an opportunity to survive. This advantage masks, to a large degree, the reduction in effectiveness by more *irreconcilable conflicts*.

Another very attractive feature of OCC-APF* is that it is not any more expensive than its closest competitor, i.e., OCC-TI, in terms of time cost per successful transaction. To see this consider the following argument: The only difference between the two is in the validation stage.[3] OCC-TI, in its validation stage, goes through each transaction in the conflict set and adjusts corresponding timestamp intervals. In other words, OCC-TI *mandates* that the entire conflict set be traversed for *each validation*. OCC-APFO, on the other hand, will only go through the set until the first lower CPI irreconcilably conflicting transaction is found. In

3. In the ensuing overhead comparison, we exclude the cost of traversing the triggering graph as this cost is common to all algorithms.

many cases, such transactions will exist and the validating transaction will be restarted. In such cases, the entire conflict set will not need to be traversed (unless there is only one irreconcilably conflicting lower CPI transaction and it is the last item in the conflict set). Thus, computation *per validation* will be expected to be less in OCC-APF*. However, there will be more validations in OCC-APF* than OCC-TI. Taking these two facts in conjunction, our estimate is that the total validation stage work turns out approximately the same assuming the first higher priority irreconcilable transaction is randomly distributed in the conflict set. However, since OCC-APF* commits significantly more transactions than OCC-TI (as shown in Section 6), the per transaction time cost is *less*. There is, however, more additional space overhead in OCC-APF* than in OCC-TI. This overhead is in terms of the image of $CS(T_{val})$ that needs to be maintained during validation to perform $\mathcal{RESET}$ if necessary. This overhead, however, is not significant as all that needs to be stored is a set of transaction ids with their timestamp intervals.

## 8 Conclusion

In this paper, we explored execution dynamics of real-time, active transactions and suggested new ways of controlling concurrency in RTADBSs. The new mechanisms were motivated by the fact that existing real-time CC algorithms appear to be lacking when applied to RTADBSs. We also reported about a thorough performance evaluation of our suggested algorithms and they performed substantially better than conventional protocols. The contribution of this paper is a small but positive step toward understanding transaction processing in real-time, active database systems.

## References


[1] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions," *ACM SIGMOD RECORD,* 1988.

[2] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data," *Proc. 15th Very Large Data Base Conf.,* 1989.

[3] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions: Performance Evaluation," *ACM Trans. Database Systems,* 1992.

[4] R. Agrawal, M.J. Carey, and M. Livny, "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Trans. Database Systems,* vol. 12, no. 4, pp. 609-654, 1987.

[5] I. Ahn, "Database Issues in Telecommunications Network Management," *ACM SIGMOD,* pp. 37-43, May 1994.

[6] M. Berndtsson and J. Hansson, "Issues in Active Real-Time Databases," *Proc. Int'l Workshop Active and Real-Time Database Systems,* June 1995.

[7] A. Bestavros and S. Braoudakis, "SCC-nS: A Family of Speculative Concurrency Control Algorithms for Real-Time Databases," *Proc. Third Int'l Workshop Responsive Computer Systems,* 1993.

[8] H. Branding and A. Buchmann, "On Providing Soft and Hard Real-Time Capabilities in an Active DBMS," *Proc. Int'l Workshop Active and Real-Time Database Systems,* June 1995.

[9] M.J. Carey, R. Jauhari, and M. Livny, "On Transaction Boundaries in Active Databases: A Performance Perspective," *IEEE Trans. Knowledge and Data Eng.,* Sept. 1991.

[10] Sprint Network Management Center, site visit, Apr. 1992.

[11] S. Chakravarthy and D. Mishra, "Snoop: An Expressive Event Specification Language for Active Databases," Technical Report UF-CIS-TR-93-007, Univ. of Florida, Computer and Information Sciences Dept., 1993.

[12] P. Chrysanthis and K. Ramamritham, "ACTA: The SAGA Continues," *Advanced Transaction Models for New Applications,* A. Elmagarmid, ed., 1992.

[13] C. Collet and J. Machado, "Optimization of Active Rules with Parallelism," *Proc. Int'l Workshop Active and Real-Time Database Systems,* June 1995.

[14] S. Comai, P. Fraternali, G. Psaila, and L. Tanca, "A Uniform Model to Express the Behavior of Rules with Different Semantics," *Proc. Int'l Workshop Active and Real-Time Database Systems,* June 1995.

[15] A. Datta, "Research Issues in Databases for Active Rapidly Changing Data Systems (ARCS)," *ACM SIGMOD RECORD,* vol. 23, no. 3, pp. 8-13, Sept. 1994.

[16] U. Dayal, B. Blaustein, A.P. Buchmann, U.S. Chakravarthy, M. Hsu, R. Ledin, D.R. McCarthy, A. Rosenthal, S.X. Sarin, M.J. Carey, M. Livny, and R. Jauhari, "The HiPAC Project: Combining Active Databases and Timing Constraints," *ACM SIGMOD RECORD,* vol. 17, no. 1, pp. 51-70, Mar. 1988.

[17] U. Dayal, M. Hsu, and R. Ladin, "Organizing Long-Running Activities with Triggers and Transactions," *Proc. ACM SIGMOD Conf. Management of Data,* 1990.

[18] P.A. Fishwick, "Simpack: Getting Started with Simulation Programming in C and C++," Technical Report TR92-022, Computer and Information Sciences, Univ. of Florida, Gainesville, Florida, 1992.

[19] N.H. Gehani and H.V. Jagadish, "Ode as an Active Database: Constraints and Triggers," *Proc. Very Large Data Base Conf.,* Sept. 1991.

[20] T. Haerder, "Observations on Optimistic Concurrency Control Schemes," *Information Systems,* vol. 9, no. 2, 1984.

[21] J.R. Haritsa, M.J. Carey, and M. Livny, "On Being Optimistic about Real-Time Constraints," *Proc. ACM Symp. Principles of Database Systems (PODS),* 1990.

[22] J.R. Haritsa, M.J. Carey, and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control," *Proc. IEEE Real-Time Systems Symp.,* 1990.

[23] J.R. Haritsa, M.J. Carey, and M. Livny, "Data Access Scheduling in Firm Real-Time Database Systems," *The J. Real-Time Systems,* vol. 4, pp. 203-241, 1992.

[24] J. Huang, J. Stankovic, K. Ramamritham, and D. Towsley, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes," *Proc. 17th Int'l Conf. Very Large Data Bases,* 1991.

[25] J. Huang, J. Stankovic, D. Towsley, and K. Ramamrithnam, "Experimental Evaluation of Realtime Transaction Processing," *Proc. IEEE Real-Time Systems Symp.,* 1989.

[26] B. Kao and H. Garcia Molina, "Subtask Deadline Assignment for Complex Distributed Soft Real Time Tasks," Technical Report STAN-CS-93-1491, Stanford Univ., 1993.

[27] H.F. Korth and A. Silberschatz, *Database System Concepts,* second ed. New York: MMcGraw Hill, 1991.

[28] H.T. Kung and J.T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. Database Systems,* vol. 6, no. 2, June 1981.

[29] S. Kuo and D. Moldovan, "Implementation of Multiple Rule Firing Production Systems on Hypercube," *J. Parallel and Distributed Computing,* vol. 13, no. 4, pp. 383-394, Dec. 1991.

[30] K. Lam, K. Lam, and S. Hung, "An Efficient Real-Time Optimistic Concurrency Control Protocol," *Proc. Int'l Workshop Active and Real-Time Database Systems,* June 1995.

[31] J. Lee and S.H. Son, "Currency Control Algorithms for Real-Times Database Systems," *Performance of Concurrency Control Mechanisms in Centralized Database Systems,* V. Kumar, ed., Prentice Hall, 1996.

[32] J. Lee and S.H. Son, "Using Dynamic Adjustment of Serialization Order for Real-Time Database Systems," *Proc. IEEE Real-Time Systems Symp.,* Dec. 1993.

[33] Y. Lin and S.H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," *Proc. IEEE Real-Time Systems Symp.,* 1990.

[34] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM,* Jan. 1973.

[35] D. McCarthy and U. Dayal, "The Architecture of an Active Data Base Management System," *ACM,* 1989.

[36] D. Menasce and T. Nakanishi, "Optimistic Versus Pessimistic Concurrency Control Mechanisms in Database Management," *Information Systems,* vol. 7, no. 1, 1982.

[37] M. Perkusich, A. Perkusich, and U. Schiel, "Object-Oriented Real-Time Database Design and Hierarchical Control Systems," *Proc. Int'l Workshop Active and Real-Time Database Systems,* June 1995.

[38] B. Purimetla, R.M. Sivasankaran, and J. Stankovic, "A Study of Distributed Real-Time Active Database Applications," *Proc. IEEE Workshop Parallel and Distributed Real-time Systems,* Apr. 1993.
[39] B. Purimetla, R.M. Sivasankaran, J.A. Stankovic, K. Ramamritham, and D. Towsley, "Priority Assignment in Real-Time Active Databases," technical report, Computer Sciences Dept., Univ. of Massachusetts, 1994.
[40] R.M. Sivasankaran, B. Purimetla, J. Stankovic, and K. Ramamritham, "Network Services Databases—A Distributed Real-Time Active Database Applications," *Proc. IEEE Workshop Real-Time Applications,* May 1993.
[41] R.M. Sivasankaran, K. Ramamritham, J.A. Stankovich, and D. Towsley, "Data Placement, Logging, and Recovery in Real-Time Active Databases," *Proc. Int'l Workshop Active and Real-Time Database Systems,* June 1995.
[42] N. Soparkar, H.F. Korth, and A. Silberschatz, "Databases with Deadline and Contingency Constraints," *IEEE Trans. Knowledge and Data Eng.,* vol. 7, no. 4, pp. 552-565, Aug. 1995.
[43] O. Ulusoy, "An Evaluation of Network Access Protocols for Distributed Real-Time Database Systems," *Proc. Int'l Workshop Active and Real-Time Database Systems,* June 1995.
[44] T. Weik and A. Heuer, "An Algorithm for the Analysis of Termination of Large Trigger Sets in an OODBMS," *Proc. Int'l Workshop Active and Real-Time Database Systems,* June 1995.
[45] P.S. Yu, K-L. Wu, K.-J. Lin, and S.H. Son, "On Real-Time Databases: Concurrency Control and Scheduling," *Proc. IEEE,* special issue on Real-Time Systems, vol. 82, no. 1, pp. 140-157, 1994.

**Anindya Datta** received the undergraduate degree from the Indian Institute of Technology, Kharagpur and the doctoral degree from the University of Maryland, College Park. He is an associate professor in the DuPree College of Management at the Georgia Institute of Technology and founder of the iXL Center for Electronic Commerce. Previousy, he was an assistant professor of MIS at the University of Arizona, His primary research interests lie in studying technologies that have the potential to significantly impact the automated processing of organizational information. Examples of such technologies include electronic commerce, data warehousing/OLAP, and workflow systems. He has published more than 15 papers in refereed journals such as *ACM Transactions on Database Systems*, *IEEE Transactions on Knowledge and Data Engineering*, *INFORMS Journal of Computing, Information Systems*, and *IEEE Transactions on Systems, Mans, and Cybernetics*. He has also published more than 35 conference papers and has chaired as well as served on the program committees of reputed international conferences and workshops. He is a member of the IEEE.

**Sang H. Son** received the BS degree from Seoul National University, the MS degree from Korea Advanced Institute of Science and Technology, and the PhD degree from the University of Maryland, College Park. He is a professor in the Department of Computer Science at the University of Virginia. He has held visiting positions at the Korea Advanced Institute of Science and Technology (KAIST) and City University of Hong Kong. His current research interests include real-time computing, database systems, QoS management, and information security. He has written or coauthored more than 100 papers and edited/authored three books in these areas. The goal of his research is to design and evaluate models and methodologies for the development of robust and responsive computer systems and databases for complex real-time applications. Dr. Son is an associate editor of *IEEE Transactions on Parallel and Distributed Systems*. He has also served as the guest editor for numerous journals such as *ACM SIGMOD Record* and *IEEE Transactions on Software Engineering*, and has chaired as well as served on the program committees of reputed international conferences and workshops. He is a senior member of the IEEE.