Concurrency Admission Control Management in ACCORD*

Sue Nagy†
The Open Group
Research Institute
Cambridge, MA 02142
s.nagy@opengroup.org

Azer Bestavros
Computer Science Department
Boston University
Boston, MA 02215
best@cs.bu.edu

Abstract

We propose and evaluate admission control mechanisms for ACCORD, an Admission Control and Capacity Overload management Real-time Database framework—an architecture and a transaction model—for hard deadline RTDB systems. The system architecture consists of *admission control* and *scheduling* components which provide early notification of failure to submitted transactions that are deemed not valuable or incapable of completing on time. In particular, our Concurrency Admission Control Manager (CACM) ensures that transactions which are admitted do not overburden the system by requiring a level of concurrency that is not sustainable. The transaction model consists of two components: a *primary task* and a *compensating task*. The execution requirements of the primary task are *not* known *a priori*, whereas those of the compensating task are known *a priori*. Upon the submission of a transaction, the *Admission Control Mechanisms* are employed to decide whether to *admit* or *reject* that transaction. Once admitted, a transaction is guaranteed to *finish* executing before its deadline. A transaction is considered to have finished executing if exactly one of two things occur: Either its primary task is completed (*successful commitment*), or its compensating task is completed (*safe termination*). Committed transactions bring a profit to the system, whereas a terminated transaction brings *no* profit. The goal of the admission control and scheduling protocols (*e.g.*, concurrency control, I/O scheduling, memory management) employed in the system is to *maximize* system profit. In that respect, we describe a number of concurrency admission control strategies and contrast (through simulations) their relative performance.

*This work has been partially supported by NSF (grant CCR-9706685).

†This work was conducted as part of the author's Ph.D. thesis at Boston University.

1 Introduction

The main challenge involved in scheduling transactions in a Real-Time DataBase (RTDB) system is that the resources needed to execute a transaction are not known *a priori*. For example, the set of objects to be read (written) by a transaction may be dependent on user input (*e.g.,* in a stock market application) or dependent on sensory inputs (*e.g.,* in a process control application). Therefore, the *a priori* reservation of resources (*e.g.,* read/write locks on data objects) to guarantee a particular Worst Case Execution Time (WCET) becomes impossible—and the non-deterministic delays associated with the on-the-fly acquisition of such resources pose the real challenge of integrating scheduling and concurrency control techniques.

Current real-time concurrency control mechanisms resolve the above challenge by relaxing the *deadline* semantics (thus suggesting best-effort mechanisms for concurrency control in the presence of *soft* and *firm*, but not *hard* deadlines), or by restricting the set of acceptable transactions to a finite set of transactions with execution requirements that are known a priori (thus reducing the concurrency control problem to that of resource management and scheduling).[1]

Consider the huge body of research on real-time concurrency control, where complex time-cognizant concurrency control techniques are proposed for the sole purpose of maximizing the number of transactions that meet their deadlines (or other metrics thereof). A careful evaluation of these elaborate techniques reveals that their superiority is materialized only when the RTDB system is overloaded. However, when the system is not overloaded, the performance of these techniques becomes comparable to that of much simpler techniques (*e.g.,* 2PL-PA). It is important to observe that when a RTDB system is overloaded, a large percentage of transactions end up missing their deadlines. This observation leads to the following question: How better would be the performance of the system if these same transactions (that ended up missing their deadlines) were not allowed into the system in the first place? The answer is obviously "*much better*" because with hindsight, the limited resources in the system would not have been wasted on these transactions to start with. While such a clairvoyant scheduling of transactions is impossible in a real system, admission control and overload management techniques could be used to achieve the same goal. In this paper, we introduce and evaluate such techniques.

Admission control and overload management techniques preserve system resources by minimizing the likelihood of a transaction being accepted for execution, and later not being able to meet its deadline. Obviously, such a situation cannot totally be eliminated in a system where the execution requirements of transactions are not known *a priori*. Therefore, missing a deadline is always a possibility, with which the system must contend. For transactions with firm deadlines, such a situation is tolerable because commitment past a firm deadline is of no value. However, for transactions with hard (soft) deadlines, such an abortion is disastrous because missing a hard (soft) deadline results in an infinite (eventual) loss.[2] Thus, to support transactions with hard deadlines

[1] In this paper, we do not consider approaches that attempt to relax ACID properties—serializability in particular.

[2] Most RTDB systems avoid dealing with the consequences of missing a hard deadline by restricting the class of

without *a priori* knowledge of their execution requirements, there must exist some compensating actions that, when executed in a timely fashion, would allow the system to be "bailed out" from the disastrous consequences of missing a hard deadline.

Our research is motivated by research problems in application areas such as command and control systems, the stock market and robotics. Consider, for example, industrial automation processes which commonly employ robots, typically in a hazardous environment. Here, a real-time database is used to represent the state of the world, *i.e.* the location of the robot arms and of the physical components which are manipulated by the robot's arms. The robot may be required to complete a transaction (an atomic set of actions) by a specified time before proceeding to the next one. Compensating actions are needed, for example, if a transaction that is about to miss its deadline must be terminated safely (requiring the clearing of the workspace, for example).

We start in section 2 with an overview of our transaction processing model and the different components therein. Next, in section 3 we describe the various concurrency admission control mechanisms to be used in our simulations. Then in section 4 we present and discuss our simulation baseline model and results as well as results of our value-cognizant protocol. In section 5, we review previous research work and highlight our contributions. We conclude in section 6 with a summary and a description of future research directions.

2 System Model

Each transaction submitted to the system consists of two components: a *primary task* and a *compensating task*. The execution requirements for the primary task are *not* known *a priori*, whereas those for the compensating task are known *a priori*.[3] Figure 1 shows the various components in our RTDB system.

When a transaction is submitted to the system, an *Admission Control Mechanism* (ACM) is employed to decide whether to *admit* or *reject* that transaction. Once admitted, a transaction is guaranteed to *finish* executing before its deadline. A transaction is considered to have finished executing if exactly one of two things occur: Either its primary task is completed, in which case we say that the transaction has *successfully committed*, or its compensating task is completed, in which case we say that the transaction has *safely terminated*. A committed transaction brings a *positive* profit to the system, whereas a terminated transaction brings *no* profit. The goal of the admission control and scheduling protocols employed in the system is to *maximize* profit.

When submitted to the system, each transaction is associated with a *deadline* and a *value*. The value of a transaction represents the profit that the system makes if the transaction is successfully committed (*i.e.* its primary task is committed by its deadline). In this paper we consider only

transactions they manage to those with either firm or soft deadlines.

[3] While the execution time of a transaction's primary task is not known *a priori*, we assume that this execution time cannot exceed the difference between the transaction's deadline and its submission time.

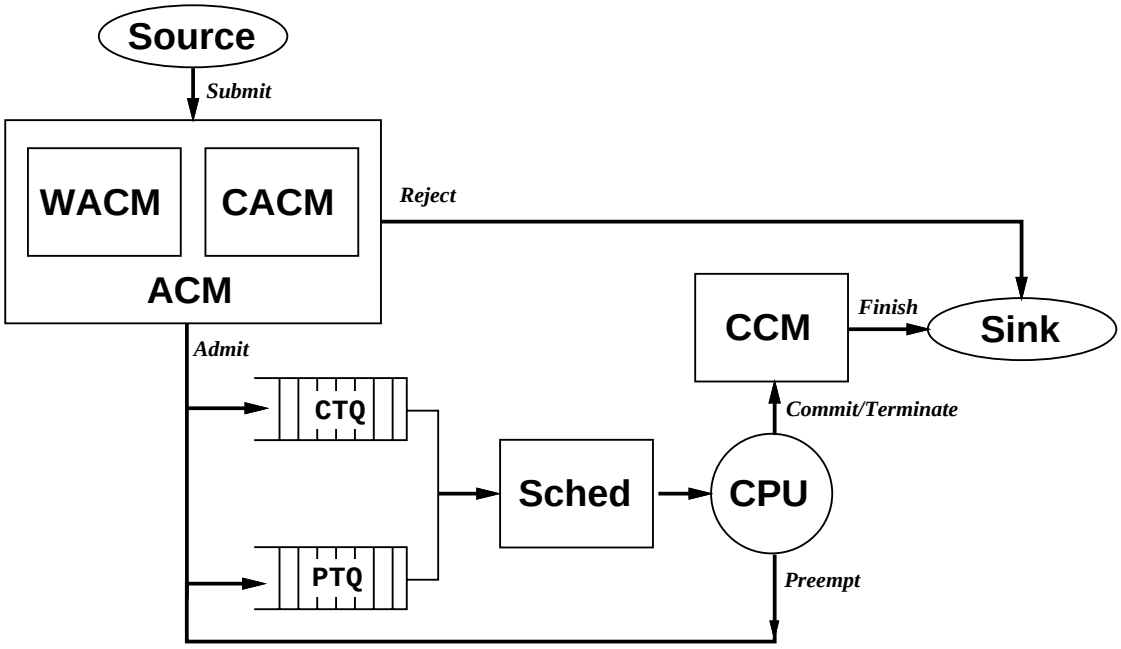

Figure 1: Major System Components

hard deadlines and thus assume that no transaction will finish (*i.e.* successfully commit or safely terminate) past its deadline.[4] We initially assume that all transactions bring in equal profit when committed on time, and then consider the case where the profits of transactions differ. Moreover, once admitted to the system, a transaction is absolutely guaranteed (as opposed to conditionally guaranteed) to finish and cannot now be rejected in order to accommodate a newly submitted transaction.

The ACM consists of two major components: a *Concurrency Admission Control Manager* (CACM) and a *Workload Admission Control Manager* (WACM). The CACM is responsible for ensuring that admitted transactions do not overburden the system by requiring a level of concurrency that is not sustainable. The WACM is responsible for ensuring that admitted transactions do not overburden the system by requiring computing resources (*e.g.,* CPU time) that are not sustainable. Our focus in this paper is on the details of the CACM.

Compensating tasks are executed when a transaction with a hard deadline is deemed incapable of committing by its deadline. Due to the urgency associated with the execution of such compensating tasks, we assume a 2-tier priority scheme for CPU scheduling purposes. In particular, all compensating tasks are assumed to have a higher priority than primary tasks. Thus a primary task may be preempted (or aborted) by a compensating task, whereas a compensating task cannot

[4]Our current research in [31] involves extending our results to soft and firm deadline systems by allowing for a profit/loss past a transaction's deadline.

be preempted by either a primary task or another compensating task under any condition. Notice that this 2-tier priority assumption still allows primary tasks (compensating tasks) to be prioritized amongst themselves.

In this paper we study our admission control mechanism in conjunction with two types of concurrency control protocols, namely Optimistic Concurrency Control with forward validation (such as OCC-BC [29] or SCC-nS [7]), or Pessimistic Concurrency Control (PCC) with Priority Abort (such as 2PL-PA [3]).

2.1 Workload Admission Control Manager

The source contains a set of transactions which are generated off-line. Each enters the system at a random time and is first processed by the ACM. The decision of whether to admit or reject a transaction submitted for execution is based upon a feedback mechanism that takes into consideration the current demand on the resources in the system. This decision is motivated by the overall goal for maximizing profit by maximizing the number—or sum of the values—of successful commitments (when primary tasks finish) and minimizing the number of safe terminations (when compensating tasks finish). For example, if the percentage of the CPU bandwidth already committed to compensating tasks (of admitted primary tasks), within the interval from the current time to the deadline of the submitted transaction is high, it may be prudent for the WACM to reject the submitted transaction. For transactions which successfully pass through the admission control process, the WACM attempts to schedule the compensating task in the Compensating Task Queue (CTQ) whose organization is discussed later in this section. Even if the current demand on the system's resources is low, a transaction is rejected if it is not feasible to schedule its compensating task (*e.g.,* it cannot be accommodated in the CTQ). Details regarding the WACM can be found in [9].

2.2 Concurrency Admission Control Manager

In order to ensure that compensating tasks can execute unhindered (and thus complete within their WCETs) the CACM must guarantee that the admission of a transaction into the system does not result in data conflicts between the compensating task of that transaction and other already admitted transactions. In a uniprocessor system employing an Optimistic Concurrency Control (OCC) algorithm with forward validation (*e.g.,* OCC-BC), compensating tasks (which cannot be preempted) are guaranteed to finish execution without incurring any restart delays. The same is true of a uniprocessor system employing a Pessimistic Concurrency Control (PCC) algorithm with priority abort (*e.g.,* 2PL-PA) because compensating tasks execute at a higher priority than primary tasks and, thus, are guaranteed to finish execution without incurring any blocking delays. This is not true in a multiprocessor system, where multiple compensating tasks may be executing concurrently. In such a system, the CACM ensures that only those compensating tasks that do not conflict with each other are allowed to overlap when executed.

2.3 Processor Scheduling Algorithm

There are two queues managed by the processor scheduler: the Primary Task Queue (PTQ) and the Compensating Task Queue (CTQ). Each admitted transaction contributes one entry in each of these queues. A primary task is ready to execute as soon as it is enqueued in the PTQ, whereas a compensating task must wait for its start time, specified by the ACM. As indicated before, compensating tasks execute at a priority higher than that of the primary tasks. Thus, the scheduling algorithm will always preempt a primary task in favor of a compensating task which is ready to execute.

Since all tasks in the PTQ are ready to execute, a scheduling algorithm must be used to apportion the CPU time amongst these tasks. We use the Earliest Deadline First algorithm (EDF) [26], which is optimal for a uniprocessor system with independent, preemptible tasks having arbitrary deadlines [15].

The CTQ is organized as a series of slots, one for each compensating task. Each slot contains the compensating task id as well as its start and end times. Slots are order according to ascending start time. The CPU continues to service primary tasks until all are finished or a compensating task must begin executing, *i.e.* its start time has arrived. In the later case, the primary task currently using the CPU is preempted and enqueued back into the PTQ where it awaits further processing, if the compensating task is associated with a different primary task. Otherwise, the primary task is aborted and its compensating task executes.

2.4 Concurrency Control Manager

The function of the CCM is to enforce the concurrency control protocol in use. For OCC techniques, this enforcement is done at the time a transaction finishes its execution, either by the commitment of its primary task or by the safe termination of its compensating task. In the case of OCC-BC, conflicting (primary tasks of) transactions are restarted, whereas in the case of SCC-nS, conflicting (primary tasks of) transactions are rolled back to a point preceding the conflicting action. For PCC techniques, this enforcement is done at the time of each read/write request. For compensating tasks, which execute at a higher priority, such a request is always granted. This may result in aborting/restarting conflicting primary tasks. Notice that it is impossible for two compensating tasks to conflict since the processor scheduler guarantees that compensating tasks do not overlap.[5] For primary tasks, such a request may result in blocking (if the read/write lock is not available).

All transactions, whether finished or rejected, are removed from the system and sent to the sink which generates statistical information used to evaluate the system performance.

[5] This condition is true in any uniprocessor system where compensating tasks cannot be preempted.

3 Optimizing Profit through ACM

3.1 Introduction

As described in [9], the motivation for employing an admission control mechanism, especially in situations of overload, is to allocate system resources, such as the CPU, *wisely*, *i.e.* Reject transactions when the processor load exceeds a certain threshold level so that processor cycles can be used for admitted transactions, which if successfully committed, return a profit to the system. In this research, the focus is again on admission control mechanisms, but we shift our attention to conflicts over logical resources (*i.e.* database objects) which are resolved by concurrency control protocols. Specifically, the CACM is responsible for ensuring that admitted transactions do not overburden the system by requiring a level of concurrency that is not sustainable. Two questions immediately come to mind:

1. How do we measure the level of concurrency? And
2. What level of concurrency can be sustained such that the admission of a new transaction is profitable, *i.e.* adds value to the system?

Before addressing these questions, we first review some database nomenclature.

Basic database access operations are traditionally of two types:

1. `read`(T_i, X)
2. `write`(T_j, Y)

where X and Y are database objects (*e.g.,* page, block, record, etc) read by transaction T_i and written by T_j, respectively. Two operations *potentially conflict* if both access the same data object and at least one of the operations is a `write`. Specifically, the potentially conflicting operations are:

- `write`(T_i, X), `read`(T_j, X): Read After Write (RAW)
- `read`(T_i, X), `write`(T_j, X): Write After Read (WAR)
- `write`(T_i, X), `write`(T_j, X): Write After Write (WAW)

Other research [6, 41, 36] has considered semantic-based concurrency control whereby semantic information on database operations is exploited to enhance the degree of concurrency. Objects such as stacks and queues with operations of `push`, `pop`, `top` and `enqueue`, `dequeue`, respectively, are typical. These operations can achieve the same results as `read` and `write`. For example, `enqueue`

writes an object to the queue while `dequeue` reads an object. A higher degree of concurrency is attainable due to the particular data structure used, *i.e.* `enqueue` and `dequeue` can occur simultaneously as long as the two operations access a different element in the queue. Our concurrency admission control mechanism is applicable to these types of objects/operations as well. The use of these operations simply requires the construction of the appropriate conflict (commutativity) table for each pair of operations. In this research, we only consider `read` and `write` operations.

3.2 Concurrency Control Mechanisms

The concurrency control protocol enforced dictates not only the manner in which both potential and materialized conflicts are dealt with but also *when* conflicts are detected and *how* conflicts are resolved. We review two types of concurrency control protocols: pessimistic and optimistic.

3.2.1 Pessimistic Concurrency Control (PCC)

With pessimistic concurrency control techniques, such as 2PL [16], conflicts *never* materialize since potential conflicts are *avoided* by blocking transactions. For example, if `write`(T_i, X) is followed by `read`(T_j, X), transaction T_j is blocked until T_i either commits or aborts (assuming this is the only conflict that T_j has). 2PL has been criticized as being too pessimistic since it blocks transactions often unnecessarily and for potentially unbounded time. Real-time variants of 2PL have been suggested. One such variant is 2PL High Priority (2PL-HP) [3] which augments 2PL with a priority-based conflict resolution mechanism. A higher priority, lock requesting transaction aborts and restarts all lower priority, lock holding transactions which have a lock on the desired object in a conflicting lock mode. Moreover, 2PL-HP prevents deadlocks due to its conflict-based priority mechanism (assuming that static, unique priorities are assigned to all transactions).

3.2.2 Optimistic Concurrency Control (OCC)

Unlike pessimistic techniques, optimistic protocols like OCC [24] ignore potential conflicts by allowing database operations issued by transactions to be performed when requested. Specifically, transactions proceed in three phases: read, validate and write. During the read phase, a transaction both reads and writes data objects to its private workspace, deferring any updates to the database until the write phase. Upon reaching the validation phase, checks are made to ensure that any previously ignored potential conflicts have not materialized. A transaction which successfully passes through the validation phase moves onto the write phase where the transaction's update operations are applied to the database. Transactions failing the validation phase are restarted. With OCC techniques, serializability is guaranteed by the validation phase.

Transaction validation occurs in one of two manners: *forward validation* [18] and *backward validation* [24], depending upon the manner in which conflicts are detected. With backward validation, if the read set of the validating transaction T_i intersects with the write set of any transaction

which committed since T_i started its read phase, T_i is restarted—the potential conflict has materialized. With forward validation, if the write set of the validating transaction T_i intersects with the read set of active, uncommitted transactions, then either T_i is restarted or the conflicting, active transactions are restarted. Since OCC is a restarted-based protocol, deadlocks are not possible.

With classical OCC [24], conflicts are not detected until the validation phase. Resources are wasted by those transactions which reach the validation phase only to be restarted. In order to waste less system resources and restart transactions as early as possible, both of which are important for real-time systems, [29, 34] introduced a variant of forward validation which employs a broadcast commit (BC) mechanism. OCC-BC guarantees that a transaction reaching its validation phase will commit as checks for materialized conflicts are made with *uncommitted* transactions rather than with committed transactions. Uncommitted transactions which conflict with the validating transaction are restarted.

Although OCC-BC detects conflicts earlier than with simple OCC, it suffers from the possibility of unnecessary aborts, *i.e.* potential conflicts may actually *never* materialize. For example, suppose transactions T_i and T_j execute the following sequence of operations:

$$\texttt{read}(T_i, X), \texttt{read}(T_j, X), \texttt{write}(T_j, X), \texttt{read}(T_j, Y), \texttt{read}(T_i, Z), \texttt{validate}_\texttt{j}$$

where $\texttt{validate}_\texttt{j}$ denotes the validation of T_j.

The broadcast of the commitment of T_j ($\texttt{validate}_\texttt{j}$) results in the restart of active, uncommitted transaction T_i which has read an object which T_j has written. However, suppose that T_i had not been aborted but allowed to execute its remaining operation, $\texttt{validate}_\texttt{i}$ (other such scenarios are possible as well). T_i would commit now as well. The restart of a transaction due to the validation of another does *not* necessarily imply the materialization of a potential conflict.

Due to their non-blocking behavior, optimistic concurrency control techniques, such as OCC-BC, are better able to guarantee both absolute and relative consistency [4, 35] requirements. Concurrency control protocols which are blocking-based, such as PCC, lend themselves to using stale rather than recent data.

3.3 Concurrency Admission Control Manager

3.3.1 Introduction

We return now to address the two questions posed in the first part of this section, regarding the level of concurrency sustainable by a system. We begin with the definition of *delay* followed by *conflict probability*. Consider a system with one processor and one transaction T_i^X where T_i^X takes c time units to complete its execution. T_i^X denotes transaction i which is a member of transaction class X. Now further suppose a system with two processors and two transactions, T_i^X as before

and another transaction T_j^Y. If T_i^X now takes e time units to execute, where $e > c$, the additional time needed by T_i^X, $e - c$, is referred to as delay. This additional time is a result of conflicts over data. The same notion applies for systems with an infinite number of processors and transactions.

For restart-based concurrency control protocols, a delay is the consequence of a restart which leads to the following definition:

Definition 1 *For restart-based concurrency control protocols, in systems with infinite processing resources, the* delay *of* T_i^X*, due to the commitment of* T_j^Y *denotes the* restart *of* T_i^X *with probability* p*.*

T_i^X may be restarted once or possibly a multiple number of times. We measure the level of concurrency between two transactions T_i^X and T_j^Y by the conflict probability defined as follows:

Definition 2 *The* Conflict Probability $CP(T_i^X, T_j^Y)$ *of transaction* T_i^X *with respect to* T_j^Y *equals* p*, the probability that the commitment of the* latter *transaction* T_j^Y *results in the* delay of the former *transaction* T_i^X*.*

In a similar fashion, the delay derived from the conflict probability can be defined for blocking-based concurrency control techniques such as PCC to mean the *block* of T_i^X.

As exemplified earlier, OCC-BC has a number of properties which make it an attractive protocol to use in systems with transactions having timing requirements. These properties include:

- allowing data access to occur when requested,
- guaranteeing the commitment of transactions reaching their validation phases,
- equivalence of serialization and commit orders, and
- lending itself improved chance of guaranteeing temporal consistency.

In this research, we restrict our attention to OCC-BC as a concurrency control mechanism representative of restarted-based protocols. Moreover, we treat WAW conflicts using Thomas' Write Rule (TWR) [39] which ignores write operations which arrive too late.

For OCC-BC, $CP(T_i^X, T_j^Y) = 0$ is indicative of the read/write sets of T_i^X and T_j^Y not intersecting, *i.e.* the commitment of T_j^Y *never* restarts T_i^X. On the other hand, $0 < CP(T_i^X, T_j^Y) \leq 1$ suggests that the read/write sets *may* intersect—the commitment of T_j^Y *may* restart T_i^X. Since conflicts may be uni-directional, $CP(T_i^X, T_j^Y)$ is not necessarily equal to $CP(T_j^Y, T_i^X)$.

The definition of Conflict Probability from above can be further refined to be optimistic, pessimistic, or speculative. With an optimistic conflict probability, we specifically examine the conflict probabilities between PT_i^X, the primary task of T_i^X, with respect to PT_j^Y, the primary

task of T_j^Y *i.e.* what is the probability that PT_i^X is restarted due to the commitment of PT_j^Y? Here, we optimistically assume that primary tasks will successfully commit thereby making the execution of the corresponding compensating tasks unnecessary. Similarly, $CP(T_i^X, T_j^Y)$ can be defined *pessimistically* whereby we inspect the conflict probability between PT_i^X with respect to CT_j^Y, *i.e.* we pessimistically assume that primary tasks will abort resulting in the execution of the corresponding compensating tasks. We can also have a *speculative* method for calculating the conflict probabilities in that we utilized both the optimistic and pessimistic conflict probabilities, weighting each one, for example, by the percentage of transactions successfully committed and percentage of transactions safely terminated, respectively, during a specified interval of time. In this research we restrict our focus to optimistically define conflict probabilities.

3.3.2 Generating Conflict Probabilities

There are generally four approaches to obtaining the actual conflict probabilities used by the CACM.

First, with the *dynamic on-line* approach, conflict probabilities can be dynamically calculated based upon trace observations of an on-line system. In particular, as each transaction T_i commits, the transactions which it delays (restarts) as a result of its commitment are noted. Once the system has been observed for a sufficient period of time, conflict probabilities can be determined and continually updated as the system progresses.

Second, with the *dynamic off-line* approach, conflict probabilities can again be dynamically calculated but this time based upon trace observations of an *off-line* system, similar to the determination of the optimal threshold used by the workload admission control manager.

Third, with the *static exhaustive* approach, conflict probabilities can be determined during transaction pre-compilation, a type of analysis typically performed in many real-time systems, given knowledge of the read/write sets of the transaction set.

Fourth, with the *static random* approach, rather than examine *all* possible sequences of transaction traces, a *random sampling* of a subset is taken. For systems with a small number of transactions, the static exhaustive approach is certainly computationally feasible. However, when the number of transactions becomes large, then the static random approach is more reasonable. Consider a system with 100 transactions each with 4 possible traces. The total number of possible traces is: $(100 * 4) * 100!$. A random sampling would be $(100 * i) * j!$ such that $1 \leq i \leq 4$ and $1 \leq j \leq 100$.

3.3.3 Determining the Admission Control Decision

To determine the level of concurrency that can be sustained by the system, as each transaction T_i^X is submitted to the system, we check $CP(T_i^X, T_j^Y)$ for all T_j^Y currently in the system. In particular, we are interested in the average conflict probability that T_i^X has with each T_j^Y currently in the

system as well as the number of these transactions T_j^Y with which T_i^X potentially conflicts with, *i.e.* $CP(T_i^X, T_j^Y) > 0$. We define these terms below.

Definition 3 *The* Transactions In Conflict $TIC(T_i^X)$ *for newly submitted transaction* T_i^X *is defined as:*

$$TIC_\varepsilon(T_i^X) = number\ of\ CP(T_i^X, T_j^Y) > \varepsilon, \quad \forall\ T_j^Y \in\ AT$$

where AT is the set of currently Admitted Transactions and ε *is a constant* ≥ 0.

As mentioned earlier, for OCC-BC, $CP(T_i^X, T_j^Y) = 0$ is indicative of the read/write sets of T_i^X and T_j^Y not intersecting; hence we always exclude from the calculation of $TIC(T_i^X)$ those conflict probabilities whose values equal 0. By appropriately setting the value of ε, we can bias the count of the number of transactions in conflict to be an optimistic one (for $CP(T_i^X, T_j^Y)$ values tending towards 1), a pessimistic one ($CP(T_i^X, T_j^Y)$ values tending towards 0), or somewhere in between the two extremes.

Definition 4 *The* Average Restart Probability $ARP(T_i^X)$ *for newly submitted transaction* T_i^X *is defined as:*

$$ARP_\varepsilon(T_i^X) = \frac{1}{TIC_\varepsilon(T_i^X)} * \sum_{\forall j: CP(T_i^X, T_j^Y) > \varepsilon} CP(T_i^X, T_j^Y), \quad \forall\ T_j^Y \in\ AT$$

where AT is the set of currently Admitted Transactions and ε *is a constant* ≥ 0.

$ARP_\varepsilon(T_i^X)$ represents the average restart probability incurred by T_i^X as a result of having conflict probabilities $CP(T_i^X, T_j^Y) > \varepsilon$ with admitted transactions T_j^Y.

In deciding whether or not to admit T_i^X to the system, we calculate the Conflict Index for Submitted transaction T_i^X which takes into account both $ARP_\varepsilon(T_i^X)$ and $TIC_\varepsilon(T_i^X)$. Specifically,

Definition 5 *The Conflict Index for Submitted transaction* $CIS(T_i^X)$ *for newly submitted transaction* T_i^X *is defined as:*

$$CIS(T_i^X) = \alpha * \frac{TIC_\varepsilon(T_i^X)}{N} + (1 - \alpha) * ARP_\varepsilon(T_i^X)$$

where N is the number of transactions currently admitted to the system and α *is a weight factor.*

The first term in the sum, $\frac{TIC_\varepsilon(T_i^X)}{N}$, represents the fraction of admitted transactions which the submitted transaction T_i^X potentially conflicts with, with probability greater than ε, *i.e.* the percentage of transactions whose commitment may result in the restart of T_i^X. The second term, $ARP_\varepsilon(T_i^X)$, as discussed earlier, is the average restart probability incurred by T_i^X. The importance of these two distinct terms can be seen in the following example. Case 1: $TIC_\varepsilon(T_i^X) = 0.9$, $N = 100$, and $ARP_\varepsilon(T_i^X) = 0.9$, *i.e.* the submitted transaction conflicts with 90% of the admitted transactions and with an average restart probability of 90%. Case 2: $TIC_\varepsilon(T_i^X) = 0.1$, $N = 100$,

and $ARP_\varepsilon(T_i^X) = 0.9$. Here, although the average restart probability is high, the same as in case 1, the number of admitted transactions in conflict is now quite low at 10%. In case 1, we would consider rejecting T_i^X given that both TIC and ARP are high—setting α to 0.5 allows CIS to take into consideration both TIC and ARP. In case 2, we would consider accepting T_i^X since it only conflicts with a small number of admitted transactions. Utilization of α allows the value of CIS to be determined by the percentage of admitted transactions in conflict (when our interest is only in the number of potential conflicts above ε), by the average restart probability (when our interest is only in the average restart probability), or some combination of the two.

In addressing the question regarding the level of concurrency which can be sustained by the system, we must also determine what the effect of admitting a new transaction T_i^X will be upon the currently admitted transactions. Similar to ARP_ε and TIC_ε, we compute the Admitted transactions Average Restart Probability (AARP) and Admitted Transactions In Conflict (ATIC), respectively, as follows:

$$ATIC_\varepsilon(T_i^X) = number\ of\ CP(T_j^Y, T_i^X) > \varepsilon, \quad \forall\ T_j^Y \in\ AT$$

$$AARP_\varepsilon(T_i^X) = \frac{1}{ATIC_\varepsilon(T_i^X)} * \sum_{\forall j:\, CP(T_j^Y, T_i^X) > \varepsilon} CP(T_j^Y, T_i^X), \quad \forall\ T_j^Y \in\ AT$$

Similar to CIS, we compute the Conflict Index for Admitted transactions (CIA) as follows:

$$CIA(T_i^X) = \beta * \frac{ATIC_\varepsilon(T_i^X)}{N} + (1 - \beta) * AARP_\varepsilon(T_i^X)$$

where like α, β is a weight factor which may be the same as or different from α. In this research we assume that α and β have the same constant value.

To complete the decision regarding the admission/rejection of T_i^X, we compare CIS and CIA as follows:

if (CIS is less than CT_S) **and** (CIA is less than CT_A), *admit* T_i^X; otherwise *reject* T_i^X

CT_S and CT_A are the Conflict Thresholds for the Submitted transaction and Admitted Transactions, respectively, which indicate the level of concurrency maximizing the value-added to (profit of) the system for the particular transaction characteristics at hand and are simulation input parameters. We restrict our attention to the case where CT_S and CT_A are equal.

Each transaction which successfully passes through the CACM is admitted to the system provided that its compensating task can be scheduled according to Latest Fit (LF) scheduling

technique (see [9] for details). Those transactions failing the CACM test are rejected from the system.

The aforementioned CACM procedure does not take into consideration transactions' values when making admission control and scheduling decisions. When transactions return different profits to the system upon their timely completion, the admission control mechanism must be value-cognizant. In the next section, we describe a value-cognizant concurrency admission control protocol.

3.3.4 Value-cognizant CACM

The CACM procedures described in the previous section are not cognizant of transactions' values, *i.e.* all admitted transactions equally contribute to the calculation of TIC, ARP, $ATIC$, and $AARP$. When transactions belong to different classes, distinguished by transaction value, for example, the CACM procedures must be enhanced to take into account the values of transactions when making admission control decisions. In particular, higher-valued transactions should have a greater influence on the admission control process while lower-valued transactions should have a lesser influence. Specifically,

Definition 6 *The* Value of the Admitted Transactions at Risk $VATR(T_i^X)$ *for newly submitted transaction* T_i^X *is defined as:*

$$VATR_\varepsilon(T_i^X) = \sum_{\forall j: CP(T_j^Y, T_i^X) > \varepsilon} V(T_j^Y), \quad \forall\ T_j^Y \in\ AT$$

where AT *is the set of currently Admitted Transactions,* ε *is a constant* ≥ 0*, and* $V(T_j^Y)$ *is the value of transaction* T_j^Y*.*

Definition 7 *The* Weighted Admitted transactions Average Restart Probability $WAARP(T_i^X)$ *for newly submitted transaction* T_i^X *is defined as:*

$$WAARP_\varepsilon(T_i^X) = \frac{1}{VATR_\varepsilon(T_i^X)} * \sum_{\forall j:\, CP(T_j^Y, T_i^X) > \varepsilon} CP(T_j^Y, T_i^X) * V(T_j^Y), \quad \forall\ T_j^Y \in\ AT$$

where AT *is the set of currently Admitted Transactions,* ε *is a constant* ≥ 0*, and* $V(T_j^Y)$ *is the value of transaction* T_j^Y*.*

$VATR(T_i^X)$ represents the sum of the values of the admitted transactions T_j^Y which could be affected by the admission of T_i^X, *i.e.* $CP(T_j^Y, T_i^X) > \varepsilon$. $WAARP(T_i^X)$ is similar to $AARP(T_i^X)$ defined previously. With $AARP$, all transactions contribute equally to the final result; however, with $WAARP$, each admitted transaction contributes relative to its value so that higher-valued transactions have more influence on the final result while lower-valued transactions have less influence.

The final concurrency admission control decision for newly submitted transaction T_i^X is based up the following conditions:

$$\textbf{if } \left(\left(\frac{V(T_i^X)}{VATR(T_i^X)} > 1 \right) \textbf{ or } \left(WAARP(T_i^X) < CT_A \right) \right), \text{ admit } T_i^X \text{ else reject } T_i^X$$

The conditions above determine the profit of admitting a new transaction in relationship to the loss incurred by previously admitted transactions. The question which we seek to answer is, "Do we stand to gain more than we stand to loose in admitting a new transaction?". $\frac{V_i^X}{VATR(T_i^X)}$ is the ratio of the value of the newly submitted transaction T_i^X to the sum of the values of the admitted transactions which could be affected by the admission of T_i^X. A value greater than one is indicative of T_i^X being more valuable than the admitted transactions which potentially conflict with it, so we admit T_i^X, *i.e.* we stand to gain more than we stand to loose. On the other hand, values of $\frac{V(T_i^X)}{VATR(T_i^X)}$ less than or equal to one are indicative of these admitted transactions being at least as valuable, if not more valuable, than T_i^X so we further check that $WAARP$ is below CT_A, the Conflict Threshold for the Admitted transactions as defined earlier. If $WAARP$ exceeds the threshold then we conclude that since the value of the submitted transaction is less than or equal to those which potentially conflict with it, we reject T_i^X.

4 Performance Evaluation

We have implemented the above ACM policies for a uniprocessor system using OCC-BC. In the first part of this section, we show the value of concurrency admission control by comparing the performance achievable through workload admission control and a combination of workload and concurrency admission control. Since we assume that all transactions bring in equal profit when committed before their deadlines, we desire to maximize the number of primary task completions while minimizing the number of compensating task completions (*i.e.* primary task abortions). In the second part of this section, we show the performance of the value-cognizant CACM technique in comparison to non-value-cognizant CACM. The superior results of value-cognizant CACM demonstrate the advantage of utilizing the values of both the submitted transaction as well as admitted transactions in the admission control process.

4.1 Baseline Experiments

Table 1 shows the baseline parameters for our simulations. The RTDB system model used in our experiments consists of a uniprocessor system with a 1000-page, memory-resident database. A second CPU is dedicated to supporting both admission and concurrency control protocols. Transactions in the baseline model are from different transaction classes—`Xclasses` = 3—whereby transactions in each class have similar characteristics such as value, transaction size, etc. as described below. The

primary task of each transaction reads 16 pages selected at random with a 25% update probability. The CPU time needed to process a read or a write is 2.5 ms. Thus, in the absence of any data or resource conflicts, the primary task of each transaction would need a *serial execution time* of 50 ms CPU time.[6] The compensating task of each transaction follows a normal distribution with a mean of 10 ms and standard deviation of 5 ms.[7] Transaction deadlines are related to the *serial execution time* through a *slack factor*, such that *(deadline time - arrival time)* = `SlackFactor` × *serial execution time*.

The transaction inter-arrival rate, which is drawn from an exponential distribution, is varied from 5 transactions per second up to 50 transactions per second in increments of 5, which represents a light-to-medium loaded system. We used two additional arrival rates of 75 and 100 transactions per second to experiment with a very heavy loaded system. Each simulation was run three times, each time with a different seed, for 200,000 ms. The results depicted are the average over the three runs.

Parameter	Meaning	Value
`CPUTime`	CPU time per page access	2.5 ms
`DBsize`	Database size in pages	1,000
`ArrivalRate`	Transaction arrival rate	5 - 100 TPS
`Xclasses`	Number of transaction classes	1
`Xsize`	Number of reads per transaction	16
`UpdateProb`	Update Probability	0.25
`CTCompTime`	Mean Compensating Task Time	10 ms
`CTStdDev`	St. Dev. of CT Time	0.5 `CTCompTime`
`SlackFactor`	Slack factor	2
`TaskSchd`	Task scheduling protocol	`EDF`
`CTSchd`	CT scheduling protocol	FF, LF, LMF
`Thrsh`	CT computation threshold	0.125
`CCntrl`	Concurrency Control protocol	OCC-BC

Table 1: Baseline Workload Parameters

Each of the three transaction classes makes up 1/3 of the offered load and is equally important, *i.e.* all transactions have value 1. In particular, the relative conflict probabilities for each class of transactions are as follows:

- Class I: read-only. 0 - 30% conflict probability with Classes II and III.
- Class II: update. 70 - 100% conflict probability with Classes II and III.

[6] Notice that these figures (*i.e.* number of pages accessed and serial execution time) are only needed to generate the workload fed to the simulator. They are *not* known to the ACM.

[7] This amounts to an average of 4 page accesses.

- Class III: update. 70 - 100% conflict probability with Classes II and III.

In addition, the following parameter settings were used:

- $\varepsilon = 0$
- $\alpha = \beta = 0.5$
- $CT_S = CT_A$

With these transaction classes and parameter settings we are able to study the effects of two transaction classes having very high conflict probabilities with one another as well as with transactions in the same class. With $\varepsilon = 0$, we take into account *all* conflict probabilities, no matter how small. By setting both α and β equal to 0.5, we equally weigh the two factors used in the calculation of CIS and CIA, respectively.

We conducted two sets of experiments, the first in which the WACM is the only form of admission control employed and the second in which both WACM and CACM are enforced. Testing the performance of the CACM alone necessitates the use of a multiprocessor system which we do not consider here but in our future work. Consequently, we seek to determine the improvement in performance that the CACM affords above and beyond simple WACM. In particular, in the second set of experiments, transactions which are submitted to the system are first processed by the WACM whereby LAF compensating task scheduling is attempted. Those transactions which violate the LAF workload threshold are rejected from the system. However, those successfully passing through the WACM move on to the CACM.

As depicted in figure 2-a, the performance of utilizing *both* workload and concurrency admission controls is better than just workload admission control on its own. Specifically, at 50 TPS, the profit is 10% higher while at 100 TPS it is 14% higher. These results suggest the value in using both the WACM and the CACM—the processor load and the level of concurrency conflict must be monitored in order to prevent the system from thrashing.

4.2 Value-cognizant Results

The baseline parameters used for the value-cognizant CACM simulation experiments are identical to those used in the value-incognizant CACM simulation experiments from the previous section. Each of the three transactions classes, as before, make up 1/3 of the offered load and have the same conflict probabilities. However, transactions in Classes I and II have value 1 (less critical) whereas those in Class III have value 10 (more critical).

Figure 2-b shows the results of our value-cognizant CACM simulations in comparison to our value-incognizant CACM. Two sets of curves are shown. The first shows the Profit Realized using

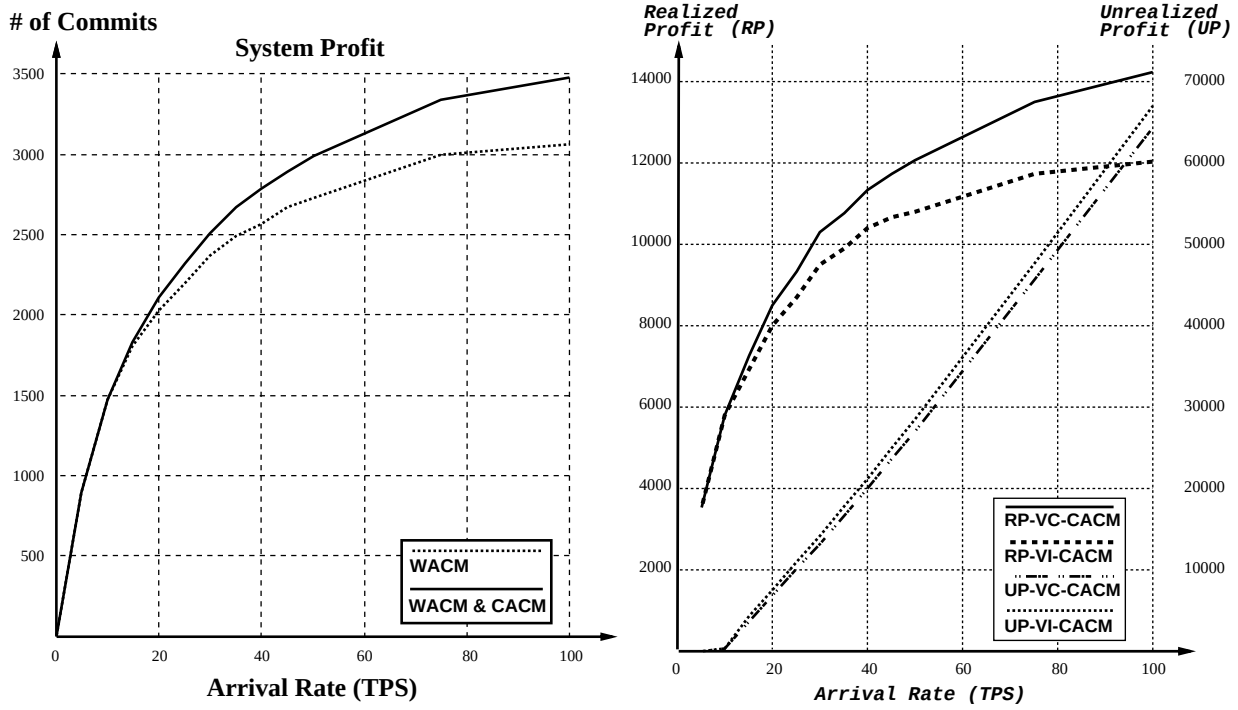

Figure 2: (a) Workload AC vs. Workload/Concurrency AC (b) Basic CACM vs. Value-cognizant CACM

WACM/Value-Incognizant CACM (PR-VI-CACM) and WACM/Value-Cognizant CACM (PR-VC-CACM) while the second shows the Unrealized Profit—profit that had to be given up by the admission control protocol—for WACM/Value-Incognizant CACM (UP-VI-CACM) and WACM/Value-Cognizant CACM (UP-VC-CACM). The results clearly show that the use of a value-cognizant admission control protocol outperforms a value-insensitive one, especially when the system is not under-utilized. For example at an arrival rate of 100 TPS, utilization of WACM/value-cognizant CACM results in 18% more profit when compared to WACM/value-incognizant CACM. The difference between the two unrealized profit curves (UP-VI-CACM and UP-VC-CACM), although not as pronounced as the difference in realized profit, is nevertheless compelling.

5 Related Work

Our work differs from previous research in that our transaction model incorporates not only primary tasks, with unknown WCET, but also compensating tasks. The admission control mechanism used admits transactions into the system with the *absolute* guarantee that either the primary task will successfully commit or the compensating task safely terminate. There have been a number of similar models suggested in the literature. These are contrasted to our model below.

Liu *et al.* [27] developed the *imprecise computation* model which decomposes each task

into two subtasks, a mandatory part and an optional part. Others employing this model include Audsley *et al.* [5] and Davis *et al.* [14]. Our model differs from the imprecise computation model in that the WCET requirements for the mandatory *and* optional parts are assumed in [27, 5, 14], whereas they are assumed only for the compensating tasks in our model. Also, unlike the imprecise computation model, we start off with the execution of the optional component (the primary task), leaving the mandatory component (the compensating task) to a later time (if needed). In a sense, our paradigm is complementary to the imprecise computation paradigm.

A number of papers have employed the *primary / alternative* model in which the primary task provides good quality of service and is preferable to the alternative which produces an acceptable quality of service. Alternatives handle timing faults in [25, 13] and processor failures in [30, 32, 23]. Our notion of a compensating task is indeed similar to that of an alternative; execution of a compensating task provides less attractive quality of service in comparison to the execution of the primary task. The similarities end here, however. The alternatives in Liestman and Campbell are not subject to timing failures, whereas in our model compensating tasks may have hard, soft or firm deadlines. Moreover, in Chetto and Chetto, the alternatives are periodic in nature, unlike compensating tasks which are not.

In [38], Tew *et al.* introduce a task model with two components: a *load task* and an *execute task* whereby the load task first loads the task from disk into memory thereby making the execute task eligible to run (*i.e.* there is a precedence relation between the two tasks). The task model of Tew *et al.* is similar to our transaction model. Both models consist of a main task (primary task, execute task). However, the motivation for having the second component differs. Our compensating task is necessitated by the fact that the read/write sets and WCETs of primary tasks are non-deterministic, whereas Tew *et al.* are interested in accounting for loading a task into memory.

Most previous RTDB system studies have assumed that the only possible outcome of a transaction execution is either the *commitment* or the *abortion* of the transaction. In many systems, a third outcome of an outright *rejection* may be desirable. For example, in a process control application, the outright rejection of a transaction may be safer then attempting to execute that transaction, only to miss its deadline. Our work allows the system to reject a transaction, thus making it possible for other actions to be taken in a timely fashion (possibly by the outside mechanism that submitted that very same transaction). Also, this flexibility allows the system to ration its resources in the most *profitable* way, by only admitting high-value transactions when the system is overloaded, while being less choosy when the system is under-loaded.

Admission control protocols and feedback mechanisms have been employed in a variety of RTDB system components: transaction scheduling [19, 20], memory allocation for queries [33], and B-tree index concurrency control [17]. Haritsa *et al.* [19] incorporate a feedback mechanism into an Adaptive Earliest Deadline (AED) and Hierarchical Earliest Deadline (HED) scheduling strategies for transactions in a firm deadline environment. Both AED and HED attempt to stabilize the overload performance of EDF. Goyal *et al.* [17] describe an approach that allows transactions to

be rejected as part of an optimization of the Load Adaptive B-link algorithm (LAB-link), a real-time version of index (B-tree) concurrency control algorithms in firm-deadline RTDB systems. LAB-link ensures that the root of the B-tree (disk) does not become a bottleneck by rejecting transactions when the percentage of transactions missing their deadlines is above a preset threshold. By tuning the system based on the percentage of missed deadlines, their technique does not guarantee a maximum profit. Also, the notion of a guarantee (whether for commitment or safe termination by the deadline) is non-existent in their work.

Hong *et al.* [20] introduce the Cost Conscious Approach (CCA) to scheduling transactions in a soft RTDB system. CCA takes into account both static (*i.e.* deadline) and dynamic (*i.e.* effective service time, restart cost) aspects of a transaction's execution when dynamically computing the priority of a transaction. Chakravarthy *et al.* [12] extend CCA to adapt to the system load—CCA-ALF—Cost Conscious Approach with Average Load Factor. Like CCA, CCA-ALF uses both static and dynamic information in calculating the priority of a transaction. In addition, through a feedback mechanism, CCA-ALF incorporates the average load factor of the most recent N completed transaction. Simulation experiments of a multi-class system are performed in which 3 transaction classes are specified based upon the cpu time needed per page access (*i.e.* transaction length is varied). Since only soft deadline transactions are considered, Chakravarthy *et al.* do not employ an admission control protocol.

The focus of Pang *et al.* [33] is on admission control and memory management of queries requiring large amounts of computational memory in a firm RTDB system. Their Priority Memory Management (PMM) algorithm consists of two components: admission control and memory allocation. The admission control component dynamically sets the target MPL by using a feedback process based upon information from previously completed queries. The memory allocation component also utilized feedback obtained from previously completed queries in order to determine the memory allocation strategy to follow (*i.e.* Max or MinMax).

In all of the above research, the basic system model is one of transactions (or queries) accessing information in the database, after which, each transaction either completes by its deadline or is aborted when its deadline is missed. The only two possible transaction execution outcomes are *commitment* and *abortion*. In [33], when the number of transactions admitted to the system exceeds the MPL, new transactions are made to wait. This non-zero admission waiting time is detrimental to the progress of these transactions completing by their deadlines. The situation is analogous in [17]. When the load control mechanism is active and the utilization of the bottleneck resource is above the preset threshold, new transactions are not allowed to enter the system. Eventually, these transactions are aborted when it is discovered that their deadlines have passed.

The performance objective in most previous RTDB system studies has been to minimize the number of transactions that miss their deadlines in a hard or firm deadline environment, or to minimize tardiness, *i.e.* the time by which late transactions miss their deadlines, in a soft deadline environment. The assumption in these systems is that all transactions are of equal value. In many

systems, this assumption is not valid, making it necessary to consider the worth of a transaction, when making resource allocation and conflict resolution decisions. In such systems, the performance objective becomes that of maximizing the system *profit*.

The notions of transaction values and value functions [22, 28], used to express the value that a transaction has to the system as a function of time, have been utilized in both RTS [10, 11] as well as in RTDB systems [2, 8, 19, 21, 37, 40]. In [10, 11], the value of a task is evaluated during the admission control process. The decision to reject a task or remove a previously guaranteed task is based upon tasks' values. A task which is accepted into the system is *conditionally* guaranteed[8] to complete its execution provided that no higher valued (critical) task (with which it conflicts) arrives. In all cases, the WCET of the tasks is assumed to be known *a priori*.

In the context of RTDB systems, Huang *et al.* [21], continuing with the work of [37], use transactions' values to schedule system resources (*e.g.,* CPU) and in conflict resolution protocols in a soft real-time environment. Abbott and Garcia-Molina [2] also employ transactions' values to assign priorities to transactions for scheduling system resources in a soft RTDB system. Extending their AED scheduling algorithm to be value-cognizant, Haritsa *et al.* [19], developed Hierarchical Earliest Deadline (HED) for a firm RTDB system. All of the aforementioned research (with the exception of [2] which does not have any performance results) make use of transactions' values which are time-invariant.

A different approach is taken by Bestavros and Braoudakis [8] and Tseng *et al.* [40]. In [8] each transaction is characterized by a time-variant value function which is used to specify both the nature of the timing constraint (*i.e.* no deadline, hard, soft, or firm deadline) as well as the transaction's importance to the system relative to other transactions. For a soft RTDB system, Bestavros and Braoudakis introduce the concept of a *penalty gradient*, i.e. the rate at which a transaction looses its value when it commits past its deadline. Transactions which commit by their deadlines return their full value to the system while those that commit past their deadlines return a diminished value, a value specified by the transaction's associated value function. Value functions are specifically used in [8] in order to determine whether it is advantageous to delay the commitment of a transaction which has finished its execution, *i.e.* Will the commitment of this transaction return more profit to the system if it is committed now or delayed to a later point in time?

Like [8], Tseng *et al.* use time-variant value functions in their Highest Reward First (HRF) scheduling algorithm for a firm RTDB System. The priority of a transaction, which in continuously evaluated [1], is based upon the expected value of the transaction at its *completion time* rather than on the current time. The value function of a transaction is such that a transaction has a constant value until a critical point (determined by an simulation input parameter), a linearly decaying value after the critical point until the deadline, and 0 otherwise.

[8]This is in contrast to an *absolute* guarantee, which specifies that once admitted to a system, the task (or its alternative or compensating task) will complete its execution by its deadline.

The utility of taking into consideration transactions'/tasks' values when making such decisions as admission control and scheduling has been shown in the above mentioned research as well as in this research. However, unlike all other work in value-cognizant protocols, our transactions model consists not only of primary tasks but also *compensating tasks* which deal with timing faults of primary tasks.

6 Conclusion and Future Work

In this paper, we proposed an admission control mechanism based upon the level of concurrency conflict in the system. For each transaction submitted, we determined the worthiness of admitting the new transaction, *i.e.* Will the admission of the new transaction cause a level of concurrency that is not sustainable by the system thereby preventing the primary tasks of previously admitted transactions from completing on time? If the admission control mechanism deemed that the admission of the submitted transaction caused more harm than good—potentially more value was lost than was gained—the submitted transaction was reject, otherwise, it was accepted. In addition, we developed a value-cognizant, concurrency admission control protocol which takes into consideration transaction's values when making the admission control decision.

Our current research efforts focus on evaluating the performance of pessimistic as well as speculative CACM techniques. Moreover, our work to date has concentrated on uniprocessor systems. We are currently investigating the extension of our admission control and scheduling protocols to multiprocessor systems. A number of challenging questions arise. How are transactions, both their primary tasks and compensating tasks allocated to processors? What type of CPU scheduling discipline should be used? How valuable is the use of the WACM in a multiprocessor system?

References

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transaction: A performance evaluation. *ACM Transactions on Database Systems*, 17(3):513–560, September 1992.

[2] Robert Abbott and Hector Garcia-Molina. Scheduling real-time transactions. *ACM, SIGMOD Record*, 17(1):71–81, 1988.

[3] Robert Abbott and Hector Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 1–12, Los Angeles, Ca, 1988.

[4] N. Audsley, A. Burns, M. Richardson, and A. Wellings. A database model for hard real-time systems. Technical report, Real-Time Systems Group, University of York,, U.K., 1991.

[5] N. C. Audsley, R. I. Davis, and A. Burns. Mechanisms for enhancing the flexibility and utility of hard real-time systems. In *Proceedings of the Real-Time Systems Symposium*, pages 12–21, December 1994.

[6] B. R. Badrinath and Krithi Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, March 1992.

[7] Azer Bestavros and Spyridon Braoudakis. Timeliness via speculation for real-time databases. In *Proceedings of RTSS'94: The 14th IEEE Real-Time System Symposium*, San Juan, Puerto Rico, December 1994.

[8] Azer Bestavros and Spyridon Braoudakis. Value-cognizant speculative concurrency control. In *Proceedings of VLDB'95: The International Conference on Very Large Databases*, Zurich, Switzerland, Spetember 1995.

[9] Azer Bestavros and Sue Nagy. Value-cognizant admission control for rtdb systems. In *RTSS'96: The 17th Real-Time Systems Symposium*, pages 230–239, Washington, D.C., December 1996.

[10] Sara Biyabani, John Stankovic, and Krithi Ramamritham. The integration of deadline and criticalness in hard real-time scheduling. In *Proceedings of the 9th Real-Time Systems Symposium*, December 1988.

[11] G. Buttazzo, M. Spuri, and F. Sensini. Value vs. deadline scheduling in overload conditions. In *Proceedings of the 16th Real-Time Systems Symposium*, December 1995.

[12] S. Chakravarthy, D. Hong, and T. Johnson. Incorporating load factor into the scheduling of soft real-time transactions. Technical Report TR94-024, University of Florida, Department of Computer and Information Science, 1994.

[13] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, October 1989.

[14] R. I. Davis, S. Punnekkat, N. Audsley, and A. Burns. Flexible scheduling for adaptable real-time systems. In *Proceedings of the Real-Time Technology and Applications Symposium*, pages 230–239, May 1995.

[15] M. L. Dertouzos. Control robotics: The procedural control of physical processes. In *Proceedings IFIP Congress*, pages 807–813, 1974.

[16] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[17] B. Goyal, J. Haritsa, S. Seshadri, and V. Srinivasan. Index concurrency control in firm real-time dbms. In *Proceedings of the 21st VLDB Conference*, pages 146–157, September 1995.

[18] T. Haerder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, October 1984.

[19] Jayant R. Haritsa, Miron Livny, and Michael J. Carey. Earliest deadline scheduling for real-time database systems. In *Proceedings of the 12th Real-Time Systems Symposium*, December 1991.

[20] R. Hong, T. Johnson, and S. Chakravarthy. Real-time transaction scheduling: A cost conscious approach. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pages 197–206, December 1993.

[21] J. Huang, J. A. Stankovic, D. Towsley, and K. Ramamritham. Experimental evaluation of real-time transaction processing. In *Proceedings of the 10th Real-Time Systems Symposium*, December 1989.

[22] E. Jensen, C. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *Proceedings of the 6th Real-Time Systems Symposium*, pages 112–122, December 1985.

[23] C. M. Krishna and K. G. Shin. On scheduling tasks with a quick recovery from failure. *IEEE Transactions on Computers*, 35(5):448–455, May 1986.

[24] H. Kung and John Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2), June 1981.

[25] A. Liestman and R. Campbell. A fault-tolerant scheduling problem. *IEEE Transaction on Software Engineering*, SE-12(11):1089–1095, November 1986.

[26] C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in hard real-time environments. *Journal of the Assocation of Computing Machinery*, 20(1):46–61, January 1973.

[27] J. W.-S. Liu, K. J. Lin, and S. Natarajan. Scheduling real-time, periodic jobs using imprecise results. In *Proceedings of the 8th IEEE Real-time Systems Symposium*, December 1987.

[28] C. Locke. *Best Effort Decision Making for Real-Time Scheduling.* PhD thesis, Carnegie-Mellon University, Department of Computer Science, May 1986.

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

[30] D. Mosse, R. Melhem, and S. Ghosh. Analysis of a fault-tolerant multiprocessor scheduling algorithm. *IEEE Fault Tolerant Computing*, pages 16–25, 1994.

[31] Sue Nagy and Azer Bestavros. Admission control for soft-deadline transactions in accord. In 3^{rd} *IEEE Real-time Technology and Applications Symposium*, Montreal, Canada, June 1997.

[32] Y. Oh and S. Son. An algorithm for real-time fault-tolerant scheduling in multiprocessor systems. In *Fourth Euromicro Workshop on Real-time Systems*, 1992.

[33] H. Pang, M. J. Carey, and M. Livny. Managing memory for real-time queries. In *Proceedings of the 1994 ACM SIGMOD Conference on Management of Data*, pages 221–232, 1994.

[34] John Robinson. *Design of Concurrency Controls for Transaction Processing Systems.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1982.

[35] X. Song and J. Liu. How well can data temporal consistency be maintained? In *Proceedings IEEE Symposium on Computer-Aided Control Systems Design*, pages 275–284, March 1992.

[36] M. Squadrito, B. Thuraisingham, L. DiPippo, and V. Wolfe. Towards priority ceilings in object-based semantic real-time concurrency control. In *Proceedings of RTDB'96: The 1996 Workshop on Real-Time Databases*, pages 127–134, Newport Beach, California, March 1996.

[37] John Stankovic and Wei Zhao. On real-time transactions. *ACM, SIGMOD Record*, 17(1):4–18, 1988.

[38] Ken Tew, Panos K. Chrysanthis, and Daniel Mosse. Emperical evaluation of task and resource scheduling in dynamic real-time systems. In *Proceedings of RTSS'96 WIP Session: The 17^{th} IEEE Real-Time System Symposium*, pages 35–38, Washington, D.C., December 1996.

[39] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, 1979.

[40] S.-M. Tseng, Y.H. Chin, and W.-P. Yang. Scheduling real-time transactions with dynamic values: a performance evaluation. In *Proceedings Second International Workshop on Real-Time Computing Systems and Applications*, pages 60–67, October 1995.

[41] W. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, 1988.