Updates and View Maintenance in Soft Real-Time Database Systems

Ben Kao‡ K.Y. Lam† Brad Adelberg§ Reynold Cheng‡ Tony Lee†

† Department of Computer Science, City University of Hong Kong. Email: cskylam@cs.cityu.edu.hk
‡ Department of Computer Science, University of Hong Kong. Email: {kao,ckcheng}@cs.hku.hk
§ Computer Science Department, Northwestern University. Email: adelberg@cs.nwu.edu

Abstract

A database system contains base data items which record and model a physical, real world environment. For better decision support, base data items are summarized and correlated to derive views. These base data and views are accessed by application transactions to generate the ultimate actions taken by the system. As the environment changes, updates are applied to the base data, which subsequently trigger view recomputations. There are thus three types of activities: base data update, view recomputation, and transaction execution. In a real-time system, two timing constraints need to be enforced. We require transactions meet their deadlines (transaction timeliness) and read fresh data (data timeliness). In this paper we define the concept of absolute and relative temporal consistency from the perspective of transactions. We address the important issue of transaction scheduling among the three types of activities such that the two timing requirements can be met. We also discuss how a real-time database system should be designed to enforce different levels of temporal consistency.

keywords: updates, view maintenance, transaction scheduling, temporal consistency, real-time database.

1 Introduction

A real-time database system (RTDB) is often employed in a dynamic environment to monitor the status of real-world objects and to discover the occurrences of "interesting" events [14, 10, 2, 3]. As an example, a program trading application monitors the prices of various stocks and financial instruments, looking for trading opportunities. A typical transaction might compare the price of German Marks in London to the price in New York and if there is a significant difference, the system will rapidly perform a trade.

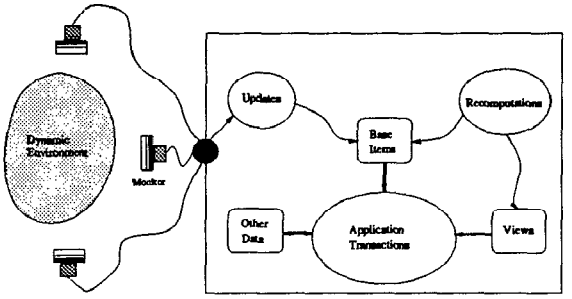

Figure 1: A Real Time Database System

The state of a dynamic environment is often modeled and captured by a set of *base data* items within the system. Changes to the environment are represented by updates to the base data. To better support decision making, the large numbers of base data items are often summarized into *views*. Some example views in a financial database include composite indices (e.g., S&P 500, Dow Jones Industrial Average) and theoretical financial option prices. For better performance, these views are materialized. When a base data item is updated to reflect certain external activity, the related materialized views need to be updated or *recomputed* as well.

These base data and views are accessed by *application transactions* to make decisions and generate the ultimate actions taken by the system.[1] For instance, application transactions may request the purchase of stock, perform trend analysis, or even trigger the execution of other transactions. Figure 1 shows the relationships among the various activities in such an RTDB.

Application transactions can be associated with one or two types of timing requirements: transaction timeliness and data timeliness. Transaction timeliness refers to how "fast" the system responds to a transaction request. A transaction should be fast enough to com-

[1] When we use the term 'transaction' alone, we refer to an application transaction.

plete before its deadline. Data timeliness refers to how "fresh" the data read is. Stale data is considered less useful due to the dynamic nature of the data.

Satisfying these two timeliness properties poses a major challenge to the design of a scheduling algorithm for such a database system. This is because the timing requirements pose conflicting demands on the system resources. To keep the data fresh, updates on base data should be applied promptly, and the views which are derived from the updated base data must be also recomputed. The computational load of applying updates and recomputations can be extremely high, causing application transactions to experience long delays. Consequently, application transactions may have a high probability of missing their deadlines.

In this paper we study the intricate balance in scheduling updates, recomputations and application transactions to satisfy the two timing requirements of data and transactions. Our goals are: (1) to define temporal correctness from the perspective of transactions, and (2) to investigate the performance of various transaction scheduling policies in meeting the two timing requirements of transactions under different correctness criteria.

The rest of this paper is organized as follows. In Section 2 we discuss some related works. In Section 3 we discuss the properties of updates, recomputations, and application transactions which are relevant to the design of an RTDB. Section 4 proposes two temporal correctness criteria. In Section 5 we list out the options of transaction scheduling and concurrency control that support different correctness criteria. In Section 6 we define a simulation model to evaluate the performance of the scheduling policies. The results are presented in Section 7. We conclude the paper in Section 8.

2 Related Works

In [2], the load balancing issues between updates and transactions in a real-time database system are studied. The authors point out that the *On-Demand* strategy, with which updates are only applied when required by transactions, gives the best overall performance. In [3], the balancing problems between derived data (views)[2] updates and transactions are studied. The authors propose the *Forced Delay* approach which delays the triggering of a recomputation for a short period, so that recomputations on the same view object can be *batched* into a single computation. The study shows that batching significantly improves the performance of the RTDB.

The two studies reported in [2] and [3] are very closely related; The former studies updates and transactions, while the latter studies recomputation transactions. However, they do not consider the case when updates, recomputations, and transactions are all present. Also, the studies report how *likely* temporal consistency is maintained under different scheduling policies, but do not discuss how to enforce the consistency constraints. In this paper we consider various scheduling policies for *enforcing* temporal consistency in an RTDB in which updates, recomputations, and transactions co-exist.

In [12], Song and Liu discuss the multiversion locking concurrency control algorithm. In this algorithm, two-phase locking is used to serialize the read/write operations of update transactions, while timestamps are used to locate the appropriate versions to be read by read-only transactions. The use of multiversion techniques serves the purpose of eliminating the conflicts between read-only and update transactions. This is because read-only transactions can always read the committed versions, without contending resources with write operations. Hence read-only transactions are never restarted, and the costs of concurrency control and restart can be significantly reduced.

3 Updates, Recomputations, and Transactions

In this section we take a look at the concept of update locality, high fan-in/fan-out of recomputations, recomputation batching and the timing requirements of transactions, which are common in many RTDBs such as programmed stock trading. These concepts also play a crucial role in the design of an RTDB.

For many real-time database applications, managing the data input streams and applying the corresponding database updates represents a non-trivial load to the system. For example, to handle the U.S. markets alone, the system needs to process more than 500 updates per second [4]. The high volume of updates and their special property (write-only) imply that they should not be executed with full transactional support. If each update is treated as a separate transaction, the number of transactions will be too large for the system to handle. As is proposed in [2], a better approach is to apply the update stream using a single *update process*. Depending on the scheduling policy employed, the update process installs updates in a specific order. It could be linear in a first-come-first-served manner, or on-demand upon application transactions' requests.

When a base data item is updated, the views which depend on the base item have to be updated or *recomputed* as well. The system load due to view recomputations can be even higher than that is required to install updates. Recomputing a view may require reading a large number of base data items (high *fan-in*).[3] Also, an update can trigger multiple recomputations if the

[2] In this paper, we use the terms "views" and "derived items" interchangeably.

[3] For example, the S&P 500 index is derived from a set of 500 stocks; a summary of a stock's price in an one-hour interval could involve hundreds of data points.

updated base item is used to derive a number of views (high *fan-out*).

One way to reduce the heavy load due to updates and recomputations is to avoid useless work. This can be achieved by observing that many applications dealing with derived data exhibit a property called *update locality* [3]. When a base item which affects a derived item d, is updated, it is very likely that a related set of base items, affecting d, will be updated soon. For example, changes in a bank's stock price may indicate that a certain event (such as an interest rate hike) affecting bank stocks has occurred. It is thus likely that other banks' stock prices will change too. Each of these updates could trigger the same recomputation, say for the finance sectoral index. Therefore, update locality implies that recomputations for derived data occur in bursts. It is too wasteful and time-consuming to recompute a view each time its value changes. A better strategy is to defer recomputations by a certain amount of time and to *batch* the same recomputation requests into a single recomputation. We call this technique *recomputation batching.*

Application transactions may read both base data and derived views. One very important design issue in the RTDB system is whether to guarantee consistency between base data and the views. To achieve consistency, recomputations for derived data are folded into the triggering updates. Unfortunately, running updates and recomputations as coupled transactions is not desirable in a high performance, real-time environment. It makes updates run longer, blocking other transactions that need to access the same data. Thus, we assume that recomputations are decoupled from updates. We will discuss how consistency can be maintained in Section 5.

Besides consistency constraints, application transactions are associated with deadlines. We assume a *firm* real-time system. That is, missing a transaction's deadline makes the transaction useless, but it is not detrimental to the system. The most important performance metric is thus the *fraction* of deadlines the RTDB meets. In Section 5 we will study a number of scheduling policies and in Section 7 we evaluate their performance on meeting deadlines.

4 Temporal Correctness

One of the requirements in an RTDB is that transactions read fresh and consistent data. *Temporal Consistency* refers to how well the data maintained by the RTDB models the actual state of the environment [11, 12, 6, 7, 8, 13]. Temporal consistency consists of two components: absolute consistency (or external consistency) and relative consistency. A data item is absolutely consistent if it timely reflects the state of an external object that the data item models. A set of data items are relatively consistent if their values reflect the states of the external objects at the same time instant.

If a base data item is updated but its associated views are not recomputed yet, the database is not relatively consistent. It is clear that an absolutely consistent database must also be relatively consistent. However, the converse is not true. For example, a relatively consistent database that never installs updates remains relatively consistent even though its data are all stale. An ideal system that performs updates and recomputations instantaneously would guarantee both absolute and relative consistency. However, as we have argued, to improve performance, updates and recomputations are decoupled, and recomputations are batched. Hence, a real system is often in a relatively inconsistent state. Fortunately, inconsistent data do no harm if no transactions read them. Hence, we need to extend the concept of temporal consistency from the perspective of transactions. Here, we formally define our notion of transaction temporal consistency. We start with the definition of an *ideal* system first, based on which correctness and consistency of real systems are measured.

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

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

The last definition does *not* state that in an absolute consistent system data can never be stale or inconsistent. It only states that no transactions can *read* stale or inconsistent data. It is clear that transactions are given a lower execution priority comparing with updates and recomputations. For example, if an update (or the recomputations it triggers) conflicts with a transaction on certain data item, the transaction has to be aborted. Maintaining an absolute consistent system may thus compromise transaction timeliness.

We can relax the requirement of data freshness by allowing transactions to read slightly stale data. Although this is not desirable in respect to the usefulness of the information read by a transaction, this can improve the probability of meeting transaction deadlines.

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

An RCS is similar to an ACS in that transactions in both systems read relative consistent data. The major

difference is that in an RCS, the data that a transaction reads can be slightly stale, with a maximum staleness Δ, while the data read in an ACS must be fresh up to the point when the transaction commits. The implication is that a transaction needs not be aborted by later updates which conflict with the data already read by the transaction. The transaction thus has a better chance of finishing before its deadline. Essentially, an RCS allows some updates and recomputations to be withheld for the benefit of expediting transaction execution. Data absolute consistency is compromised but relative consistency is maintained.

5 Transaction Scheduling and Consistency Enforcement

In this section we discuss different policies to schedule updates, recomputations, and transactions to meet the different levels of temporal consistency requirements. As we have argued, data timeliness can best be maintained if updates and recomputations are given higher priorities than application transactions. We call this scheduling policy URT (for update first, recomputation second, transaction last). On the other hand, the *On-Demand* (OD) strategy [2], with which updates and recomputations are executed upon transactions' requests, can better protect transaction timeliness. We will therefore focus on these two scheduling policies and compare their performance under the different temporal consistency requirements. Later on, we will discuss how URT and OD can be combined into the OD-H policy. In these policies, we assume that the relative priorities among application transactions are set using the earliest-deadline-first priority assignment.

Scheduling involves "prioritizing" the three activities with respect to their accesses to the CPU and data. We assume that data accesses are controlled by a lock manager employing the HP-2PL protocol (High Priority Two Phase Locking) [1]. Under HP-2PL, a lock holder is aborted if it conflicts with a lock requester that has a higher priority than the holder. We now discuss the scheduling procedure for each activity under four scenarios. These scenarios correspond to the use of the URT/OD policy in an ACS/RCS.

5.1 Policies for ensuring absolute consistency

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

5.1.1 URT

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

5.1.2 OD

The idea of On-Demand is to defer most of the work on updates and recomputations so that application transactions get a bigger share of the CPU cycles. To implement OD, the system needs an On-Demand Manager (ODM) to keep track of the unapplied updates and recomputations. Conceptually, the ODM maintains a set of data items x (base or view) for which unapplied updates or recomputations exist (we call this set the unapplied set). For each such x, the ODM associates with it the unapplied update/recomputation, and an *OD bit* signifying whether an OD-update/OD-recom[4] on x is currently executing. There are five types of activities in an OD system, namely, update arrival, recomputation arrival, OD-update, OD-recom, and application transaction. We list the procedure for handling each type of event as follows:

- On an update or recomputation arrival. Newly arrived updates and recomputations have the highest priorities in the system.[5] An update/recomputation P on a base/view item x is first sent to the OD Manager. The ODM checks if x is in the unapplied set. If not, x is added to the set with P associated with it, and a write lock on x is requested[6]; Otherwise, the OD bit is checked. If the OD bit is "off", the ODM simply associates P with x (essentially replacing the old unapplied update/recomputation by P); If the OD bit is "on", it means that an OD-update/OD-recom on x is currently executing. The OD Manager aborts the running OD-update/OD-recom and releases P for execution. In the case of an update arrival, any view that is based on x will have its corresponding recomputation spawned as a new arrival.

[4] OD-update/OD-recom means an update or a recomputation triggered on-demand by an application transaction.

[5] Newly arrived updates and recomputations are handled in a FCFS manner.

[6] The write lock is set to ensure AC, since any running transaction that has read (an outdated) x will be restarted due to lock conflict.

- On an application transaction read request. Before a transaction reads a data item x, the read request is first sent to the OD Manager. The ODM checks if x is in the unapplied set. If so, and if the OD bit is "on" (i.e., there is an OD-update/OD-recom being run), the transaction waits; otherwise, the ODM sets the OD bit "on" and releases the OD-update/OD-recom associated with x. The OD-update/OD-recom inherits the priority of the reading transaction.
- On the release of an OD-update/OD-recom. An OD-update/OD-recom executes as a usual update or recomputation transaction. When it finishes, however, the OD Manager is notified to remove the updated item from the unapplied set.

5.2 Policies for ensuring relative consistency

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

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

Our RCS needs a Version Manager (VM) to handle the multiple versions of data items. The function of the Version Manager is twofold. First, it retrieves, given an item x and a validity interval I, a value of a version of x that is valid within I. Second, the VM keeps track of the validity intervals of transactions and the data versions they read. The VM is responsible for changing a transaction's validity interval if the validity interval of a data version read by the transaction changes. We will discuss the VI management shortly. Finally, we note that since every write on a base item or a view generates a new version, no locks need to be set on item accesses.

5.2.1 URT

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

- On an update arrival. As mentioned, each version of a data item in an RCS is associated with a validity interval. When an update u on a data item version x_i arrives, the validity interval $VI(x_i)$ is set to $[ts(u), \infty]$. Also, the UTB of the previous version x_{i-1} is set to $ts(u)$, signifying that the previous version is only valid till the arrival time of the new update. The Version Manager sees if there is any running transaction T that has read the version x_{i-1}. If so, it sets $UTB(VI(T)) = \min\{UTB(VI(T)), ts(u)\}$.
- On a recomputation arrival. If an update u spawns a recomputation r on a view item v whose latest version is v_j, the system first sets the UTB of v_j to $ts(u)$. That is, the version v_j is no longer valid from $ts(u)$ onward. Similar to the case of an update arrival, the VM updates the validity interval of any running transaction that has read v_j. With batching, the recomputation r is put to sleep, during which all other recomputations on v are ignored. A new version v_{j+1} is not computed until r wakes up. During execution, r will use the newest versions of the data in its read set. The validity interval of r ($VI(r)$) and that of the new view version ($VI(v_{j+1})$) are both equal to the intersection of all the validity intervals of the data items read by r.
- Running an application transaction. Given a transaction T whose start time is $ts(T)$, we first set its validity interval to $[ts(T) - \Delta, \infty]$.[7] If T reads a data item x, it consults the Version Manager. The VM would select a version x_i for T such that $VI(x_i) \cap VI(T) \neq \emptyset$. That is, the version x_i is relatively consistent with the other data already read by T. $VI(T)$ is then updated to $VI(x_i) \cap VI(T)$. If the VM cannot find a consistent version (i.e., $VI(x_i) \cap VI(T) = \emptyset\ \forall x_i$), T is aborted. Note that the *wider* $VI(T)$ is, the more likely that the VM is able to find a version of x that is consistent with what T has already read. Hence, in our study, we always pick the version x_i whose validity interval has the biggest overlapping with that of T.

5.2.2 OD

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

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

[7] Recall that Δ is the maximum staleness tolerable with reference to a transaction's start time.

- In an ACS (single version database), an unapplied recomputation to a view item v is recorded in the ODM so that a transaction that reads v knows that the current database version of v is invalid. However, in an RCS (multi-version database), the validity intervals of data items already serve the purpose of identifying the right version. If no such version can be found in the database, the system knows that an OD-recom has to be triggered. Therefore, the ODM in an RCS does not maintain unapplied recomputations.
- In an ACS, an OD bit of a data item x is set if there is an OD-update/OD-recom currently executing to update x. The OD bit is used so that a new update/recomputation arrival will immediately abort the (useless) OD-update/OD-recom. In an RCS, since multiple versions of data are kept, it is not necessary to abort the (old but useful) OD-update/OD-recom. Hence, the OD bits are not used.
- Since different versions of a data item can appear in the database as well as in the unapplied list, the Version Manager needs to communicate with the OD Manager to retrieve a right version either from the database or by triggering an appropriate OD-update from the unapplied lists.

5.2.3 A Hybrid Approach

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

6 Simulation

To study the performance of the scheduling policies, we simulate an RTDB system with the characteristics described in Sections 1, 3 and 5. This section describes the specifics of our simulation model.

In our model, we implemented all the necessary components as described in Section 5. We simulate a disk-based database with N_b base items and N_d derived items (views). The number of views that a base item derives (i.e., fan-out) is uniformly distributed in the range $[F_{o\_min}, F_{o\_max}]$. Each derived item is derived from a random set of base items. We assume the system caches its database accesses with a cache hit rate p_{cache_hit}.

Updates are generated as a stream of *update bursts*. Burst arrivals are modeled as Poisson processes with an arrival rate λ_u. Each burst consists of *burst_size* updates. The value *burst_size* is picked uniformly from the range $[BS_{min}, BS_{max}]$. To model locality, each update would have a probability of p_{sim} of triggering the same set of recomputations as those triggered by the previous update. Application transactions are generated as another stream of Poisson processes with an arrival rate λ_t. Each transaction T performs N_{op} database operations, and each database object has an equal probability of being accessed by an operation. T is associated with a deadline given by the following formula: $dl(T) = ex(T) \times \text{slack} + ar(T)$ where $ex(T)$ is the expected execution time of the transaction[8], $ar(T)$ is the arrival time of T, and *slack* is the slack factor chosen uniformly from the range $[S_{min}, S_{max}]$.

Description	Parameter	Value
Update burst arrival rate (/sec)	λ_u	1.2
Burst size	$[BS_{min}, BS_{max}]$	[1,12]
Forced delay time (sec)	t_{FD}	1.0
Update similarity	p_{sim}	0.8
Transaction arrival rate (/sec)	λ_t	2.0
# of operations per transaction	N_{op}	50
Slack factor	$[S_{min}, S_{max}]$	[1.3,3.0]
Number of base items	N_b	3000
Number of derived items	N_d	300
Fan-out	$[F_{o\_min}, F_{o\_max}]$	[0,4]
Disk access time (ms)	t_{IO}	5.0
CPU time per operation (ms)	t_{CPU}	1.0
I/O cache hit rate	p_{cache_hit}	0.7
maximum staleness (sec)	Δ	10.0

Table 1: Baseline settings

The values of the simulation parameters were chosen as reasonable values for a typical financial application. The simulator is written in CSIM 18 [9]. Each simulation run (generating one data point) processed 10,000 update bursts. Table 1 shows the parameter settings of our baseline experiment.

7 Results

In this section we present selected results obtained from our simulation experiments. We compare the performance of the various scheduling policies in an ACS and an RCS based on how well they can meet transaction deadlines. To aid our discussion, we use the notation MD_A^B to represent the *fraction of missed deadlines* (or *miss rate*) of scheduling policy A when applied to a B system. For example, $MD_{OD}^{AC} = 10\%$ means that 10% of the transactions miss their deadlines when OD is used in an ACS.

[8]Calculated by multiplying the number of operations by the amount of I/O and CPU time taken by each operation.

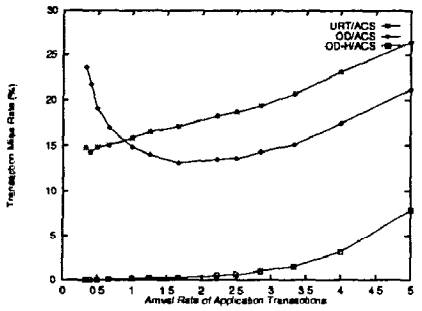

Figure 2: Miss rate vs λ_t (ACS)

7.1 Absolute Consistent System

Effect of transaction arrival rate In our first experiment, we vary the transaction arrival rate (λ_t) from 0.5 to 5 and compare the performance of the three scheduling policies (URT, OD, and OD-H) in an absolute consistent system. Figure 2 shows the result. From the figure, we see that, for a large range of λ_t ($\lambda_t > 1.0$), URT performs the worst among the three, missing 14% to 26% of the deadlines. Three major factors account for URT's high miss rate.

First, since transactions have the lowest priorities, their executions are often blocked by updates and recomputations (in terms of both CPU and data accesses). This causes severe delays and thus high miss rates to transactions. We call this factor ***Low Priority***. Second, under URT with recomputation batching, a recomputation is not immediately executed on arrival. It is forced to sleep for a short while during which it holds a write lock on the derived item (say, v) it updates. If a transaction requests item v, it will experience an extended delay blocked by the sleeping recomputation. We call this factor ***Batching Wait***. Third, in an ACS, a transaction is restarted by an update or a recomputation whenever a data item that the transaction has read gets a new value. A restarted transaction loses some of its slack and risks missing its deadline. We call this restart factor ***Transaction Restart***. From our experiment result, we observe that the average restart rate of transactions due to lock conflicts is about 2% to 3%.

By using OD, transactions are given its fair share of CPU cycles and disk services. Hence, OD effectively eliminates the *Low Priority* factor. Also, recomputations are executed on-demand, hence *Batching Wait* does not exist. This results in a smaller miss rate. In our baseline experiment (Figure 2), we see that MD^{AC}_{OD} is smaller than MD^{AC}_{URT} for $\lambda_t > 1.0$. The improvement (about 5% for large λ_t) is good but is lower than expected. After all, we just argued that OD removes two of the three adverse factors of URT. Moreover, it is interesting to see that when the transaction arrival rate is small ($\lambda_t < 1.0$), reducing transaction workload (i.e., reducing λ_t) actually *increases* MD^{AC}_{OD}.

The reason for the anomaly and the lower-than-expected improvement is that under the pure OD policy, updates and recomputations are executed only on transaction requests. Hence, when λ_t is small, the *total* number of on-demand requests are small. Many database items are therefore stale. When a transaction executes, quite a few items that it reads are outdated and thus OD-updates/OD-recoms are triggered. The transaction is blocked waiting for the on-demand requests to finish. This causes a long response time and thus a high miss rate. In our experiments, we see that as many as 12 updates and 3.5 recomputations are triggered by (and blocking) an average transaction under the OD policy. We call this adverse factor ***OD Wait***.

In order to improve OD's performance, the database should be kept fresh so that few on-demand requests are issued. One simple approach is to apply updates and recomputations (as in URT) when no transactions are present. When a transaction arrives, however, all updates/recomputations are suspended, and the system reverts to on-demand. We call this policy OD-H. Figure 2 shows that OD-H greatly improves the performance of OD. In particular, the anomaly of a higher miss rate at a lower transaction arrival rate exhibited in OD vanishes in OD-H. The effect of *OD Wait* is thus relatively mild. The problem of *Transaction Restart*, however, still exists when OD-H is applied to an ACS.

7.2 Relative Consistent System

As mentioned in Section 5.2, an RCS uses a multiversion database. Each update or recomputation creates a new data item version, and thus does not cause any write-read conflicts with transactions. A transaction therefore never gets restarted because of data conflict with updates/recomputations. The only cases of transaction abort due to data accesses occur under URT, when the version manager could not find a materialized data version that is consistent with the VI of a transaction that is requesting an item. From our experiment, we observe that the chances of such aborts are very small, e.g., only about 0.1% of transactions are aborted in our baseline experiment under URT. The on-demand strategies would not perform such aborts, since any data version can be materialized on-demand. As a result, an RCS effectively eliminates the problem of *Transaction Restart* occurring in an ACS.

Figure 3 shows the miss rates of the three scheduling policies in an RCS (dotted lines). For comparison, the miss rates in an ACS (solid lines) are also shown. From the figures, we see that fewer deadlines are missed in an RCS than in an ACS across the board. This is because the problem of *Transaction Restart* is eliminated in an RCS. Among the three policies, URT registers the biggest improvement. This is because a transaction that reads a derived item can choose an old, but

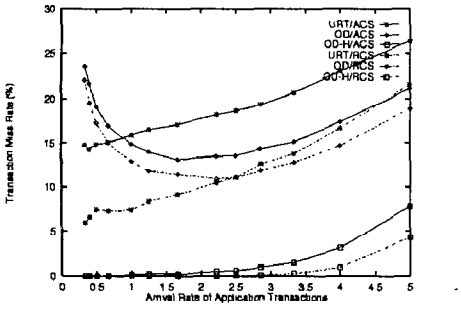

Figure 3: Miss rate vs λ_t (ACS & RCS)

materialized version. It thus never has to wait for any sleeping recomputation to wake up and to calculate a new version of the item. *Batching Wait* therefore does not exist in an RCS. Hence, two of the three detrimental factors that plague URT are gone, leading to a much smaller miss rate.

Figure 3 also shows that the performance of OD-H can be further improved in an RCS by eliminating *Transaction Restart*. Essentially, by applying OD-H to an RCS, the system is rid of any of the adverse factors we discussed. $MD^{RC}_{OD\text{-}H}$ is close to 0 except when λ_t is big. When the transaction arrival rate is high, missed deadlines are caused mainly by CPU and disk queuing delays. From Figure 3 we see that the improvement of $MD^{RC}_{OD\text{-}H}$ over $MD^{AC}_{OD\text{-}H}$ is very significant. For example, when $\lambda_t = 5.0$, about *half* of the deadlines missed in an ACS are salvaged in an RCS.

In the course of our study, we also evaluated the scheduling policies under other scenarios. Due to space limitations, these results are not reported here. Readers are referred to [5] for a more complete discussion on these experiments.

8 Conclusions

In this paper we defined temporal consistency from the perspective of transactions. In an ACS, a transaction cannot commit if some data it reads become stale at the transaction's commit time. We also defined an RCS as one with which a transaction reads relatively consistent data items and that those items are not more than a certain threshold (Δ) older than the transaction's start time. We argued that a relative consistent system has a higher potential of meeting transaction deadlines. We also studied three scheduling policies: URT, OD, and OD-H. We carried out an extensive simulation study on the performance of the these policies, under both an ACS and an RCS. We found that OD-H when applied to an RCS results in the smallest deadline miss rate.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: a performance evaluation. In *Proceedings of the 14th VLDB Conference*, 1988.

[2] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying update streams in a soft real-time database system. In *Proceedings of the 1995 ACM SIGMOD*, pages 245–256, 1995.

[3] B. Adelberg, H. Garcia-Molina, and B. Kao. Database support for efficiently maintaining derived data. In *Advances in Database Technology - EDBT 1996*, pages 223–240, 1996.

[4] M. Cochinwala and J. Bradley. A multidatabase system for tracking and retrieval of financial data. In *Proceedings of the 20th VLDB Conference*, 1994.

[5] B. Kao et. al. Updates and view maintenance in soft real-time database systems. Technical Report TR-99-06, University of Hong Kong, 1999. URL: http://www.csis.hku.hk/publications.

[6] B. Purimetla et al. Real-time databases: Issues and applications. In *Advances in Real-Time Systems*. Prentice-Hall, 1995.

[7] Y.-K. Kim and S. H. Son. Predictability and consistency in real-time database systems. In *Advances in Real-Time Systems*. Prentice-Hall, 1995.

[8] T. W. Kuo and A. K. Mok. SSP: A semantics-based protocol for real-time data access. In *IEEE Real-Time Systems Symposium*, pages 76–86, 1993.

[9] Mesquite Software, Inc. *CSIM 18 User Guide*. URL: http://www.mesquite.com.

[10] G. Ozsoyoglu and R. Snodgrass. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, 1995.

[11] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, Vol.1(No.2), 1993.

[12] Xiaohui Song and Jane W.S. Liu. Maintaining temporal consistency: Pessimistic vs. optimistic concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, pages 787–796, 1995.

[13] M. Xiong, R. Sivasankaran, J.A. Stankovic, K. Ramamritham, and D. Towsley. Scheduling transactions with temporal constraints: Exploiting data semantics. In *Proceedings of 1996 Real-Time Systems Symposium*, Washington, Dec. 1996.

[14] P.S. Yu, K.L. Wu, K.J. Lin, and S.H. Son. On real-time databases: Concurrency control and scheduling. *Proceedings of the IEEE*, 82(1):140–157, 1994.