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 [15, 10, 2, 3]. As an example, a program trading application monitors the prices of various stocks, financial instruments, and currencies, 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. 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. For example, a financial database refreshes its state of the stock market by receiving a "ticker tape" — a stream of price quote updates from the stock exchange.

To better support decision making, the large numbers of base data items are often summa-

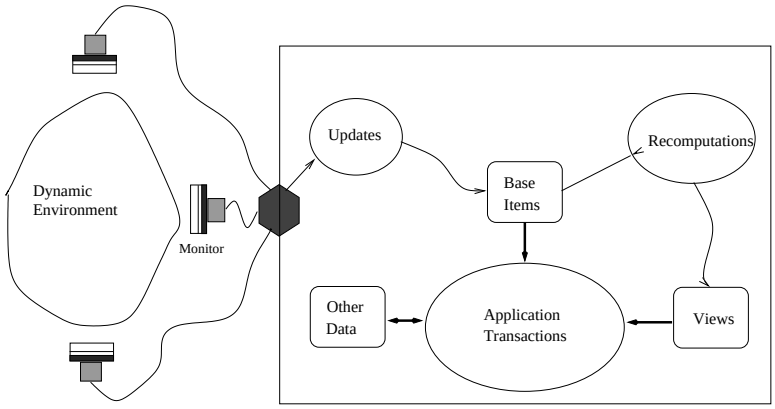

Figure 1: A Real Time Database System

rized into *views*. Some example views in a financial database include composite indices (e.g., S&P 500, Dow Jones Industrial Average and sectoral sub-indices), time-series data (e.g., 30-day moving averages), and theoretical financial option prices, etc. 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.

Besides base item updates and view recomputations, application transactions are executed to generate the ultimate actions taken by the system. These transactions read the base data and views to make their decisions. For instance, application transactions may request the purchase of stock, perform trend analysis, signal alerts, or even trigger the execution of other transactions. Application transactions may also read other static data, such as a knowledge base capturing expert rules.

Figure 1 shows the relationships among the various activities in such a real-time database system. Notice that updates to base data or recomputations for derived data may also be run as transactions (e.g., with some of the ACID properties). In those cases, we refer to them as update transactions and recomputation transactions. When we use the term transaction alone, we are referring to an application transaction.

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, while data timeliness refers to how "fresh" the data read is, or how closely in time the data read by a transaction models the environment. Stale data is considered less useful due to the dynamic nature of the data.

Satisfying the 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. Also, whenever the value of a base data item changes, affected derived views have to be recomputed accordingly. The computational load of applying base updates and performing recomputations can be extremely high, causing critical delays to transactions,

either because there are not enough CPU cycles for them, or because they are delayed waiting for fresh data. Consequently, application transactions may have a high probability of missing their deadlines.

In this paper we study the intricate balance in scheduling the three types of activities: updates, recomputations, and application transactions to satisfy the two timing requirements of data and transactions. Our goals are:

- to define temporal correctness from the perspective of transactions;
- to investigate the performance of various transaction scheduling policies in meeting the two timing requirements of transactions under different correctness criteria;
- to address the design issues of an RTDB such that temporal correctness can be enforced.

To make the right decision, application transactions need to read fresh data that faithfully reflects the current state of the environment. The most desirable situation is that all the data items read by a transaction are fresh until the transaction commits. This requirement, however, could be difficult to meet. As a simple example, if a transaction whose execution time is 1 second requires a data item that is updated once every 0.1 seconds. The transaction will hold the read lock on the data item for an extensive period of time, during which no new updates can acquire the write lock and be installed. The data item will be stale throughout most of the transaction's execution, and the transaction cannot be committed without using outdated data. A stringent data timing requirement also hurts the chances of meeting transaction deadlines. Let us consider our simple example again. Suppose the data update interval is changed from 0.1 seconds to 2 seconds. In this scenario, even though it is possible that the transaction completes without reading stale data, there is a 50% chance that a new update on the data arrives while the transaction is executing. To insist on a no-stale-read system, the transaction has to be aborted and restarted. The delay suffered by transactions due to aborts and restarts, and the subsequent waste of system resources (CPU, data locks) is a serious problem. The definition of data timeliness thus needs to be relaxed to accommodate those difficult situations (e.g., by allowing transactions to read slightly outdated data, probably within a predefined tolerance level). We will discuss a number of options for relaxing the data timing requirement in this paper.

Given a correctness criterion, we need a suitable transaction scheduling policy to enforce it. For example, a simple way to ensure data timeliness is to give updates and recomputations higher priorities over application transactions, and to abort a transaction when it engages in a data conflict with an update or recomputation. This policy ensures that no transactions can commit using old data. However, giving application transactions low priorities severely lower their chances of meeting deadlines. This is especially true when updates (and thus recomputations) arrive at a high rate. We will investigate how transaction should be scheduled to balance the contrary requirements of data and transaction timeliness.

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. In particular, we will discuss the implications of these properties on the design of a transaction scheduler and a concurrency controller. Section 4 proposes three temporal correctness criteria. In Section 5 we list out the options of transaction scheduling and concurrency control that support the 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. In the system model, updates come at a very high rate, while transactions must be committed before their deadlines. The authors propose several heuristics and examine their effectiveness in maintaining data freshness while not sacrificing transaction timeliness. They 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)[1] updates and transactions are studied. It is noted that recomputations often come in *bursts*, obeying the principle of *update locality*. 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 [13], Song and Liu discuss data temporal consistency in a real-time system that executes *periodic* tasks. In their model, tasks are either sensor (write-only) transactions, read-only transactions or update (read-and-write) transactions. Transactions must read temporally consistent data (absolutely or relatively) in order to deliver correct results. Since multiversion databases have been shown to offer a significant performance gain over single-version ones, the authors propose and evaluate two multiversion concurrency control algorithms (lock-based and optimistic) in their studies.

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

In multiversion locking concurrency control, 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. In multiversion optimistic concurrency control, an update goes through three phases: a read phase, a validation phase, and a possible write phase. During the read phase, a transaction reads and writes the most recent versions of data in its own workspace without locking the data. When it is ready to commit, the transaction enters the validation phase. Any conflicting update transactions found are immediately aborted and restarted. If a transaction passes its validation phase, it enters the write phase in which the new version of each object in the transaction's local workspace becomes permanent in the system. Read-only transactions will read the most recent and committed version of data, and go through only one phase — the read phase.

The use of multiversion techniques in both algorithms serve the common 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 closer look at some of the properties of updates, recomputations, and application transactions. We will discuss how these properties affect the design of a real-time database system. In particular, we discuss the concept of update locality, high fan-in/fan-out of recomputations, and the timing requirements of transactions. These properties are common in many real-time database systems such as programmed stock trading.

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, a financial database for program trading applications needs to keep track of more than three hundred thousand financial instruments. To handle the U.S. markets alone, the system needs to process more than 500 updates per second [5]. An update usually affects a single base data item (plus a number of related views).

The high volume of updates and their special properties (such as write-only or append-only) warrant special treatment in an RTDB. In particular, 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. (Recall that a financial database may need to process more than 500 updates per second.) Application transactions will also be adversely affected because of resource conflicts against updates. As is proposed in [3], 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. While an update involves a simple write operation, recomputing a view may require reading a large number of base data items (high *fan-in*),[2] and complex operations[3]. Also, an update can trigger multiple recomputations if the updated base item is used to derive a number of views (high *fan-out*).

One way to reduce the load due to updates and recomputations is to avoid useless work. An update is *useful* only if the value it writes is read by a transaction. So if updates are done in-place, an update to a base item b needs not be executed if no transactions request b before another update on b arrives. Similarly, a recomputation on a view needs not be executed if no transactions read the view before the view is recomputed again. This savings, however, can only be realized if successive updates or recomputations on the same data or view occur closely in time. We call this property *update locality* [3].

Fortunately, many applications that deal with derived data exhibit such a property. Locality occurs in two forms: time and space. Updates exhibit time locality if updates on the same item occur in bursts. Space locality refers to the phenomenon that when a base item b, 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. An example of update locality found in real financial data is reported in [3].

Update locality implies that recomputations for derived data occur in bursts. Recomputing the affected derived data on every single update is probably very wasteful because the same derived data will be recomputed very soon, often before any application transaction has a chance to read the derived data for any useful work. Instead of recomputing immediately, a better strategy is to defer recomputations by a certain amount of time and to batch or coalesce the same recomputation requests into a single computation. 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. Indeed, [4] shows that transaction response time is much improved when *events* and *actions* (in our case updates and recomputations) are

[2]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.

[3]For example, computing the theoretical value of a financial option price requires computing some cumulative distributions.

decoupled into separate transactions. 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. In arbitrage trading, for example, it is better not to commit a tardy transaction, since the short-lived price discrepancies which trigger trading actions disappear quickly in today's efficient markets. Occasional losses of opportunity are not catastrophic to the system. The most important performance metric is thus the *fraction* of deadlines the RTDBS 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 system 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, 13, 6, 7, 8, 14]. 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.

One option to define absolute consistency (opp staleness) is to compare the *current* time with an update's *arrival time* (a timestamp) which is an indication of which snapshot of the external object the update is representing. A data item is considered stale if the difference of its last update's timestamp and the current time is larger than some predefined maximum age T. (The value T is also called the absolute validity interval.) We call this definition *Maximum Age* (MA) [2]. Notice that with MA, even if a data object does not change value, it must still be periodically updated, or else it will become stale. Thus, MA makes more sense in applications where data items are continuously changing in time.

Another option is to be optimistic and assume that a data object is always fresh unless an update has been received by the system but not yet applied to the data. We will refer to this definition as *Unapplied Update* (UU). UU is more suitable for *discrete* data objects which change at discrete point in time and not continuously [12]. For example, in program trading, stock prices are updated when trades are made, not periodically. In such a context, age has less meaning since a price quote could be old but still be correct. UU is more general than MA, since the arrival times of updates are not assumed known in advance. Figure 2 illustrates the two staleness models.

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 rel-

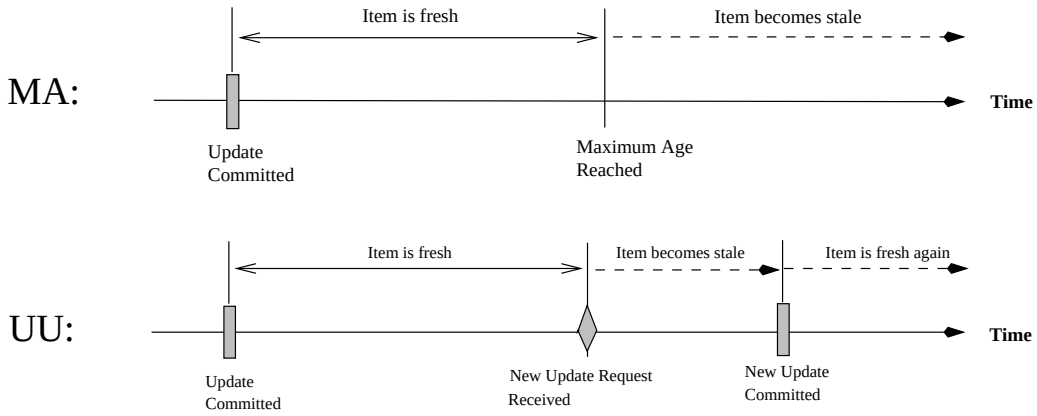

Figure 2: Maximum Age (MA) and Unapplied Update (UU)

atively 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 time 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. To have a better chance of meeting transactions' deadlines, we need to upgrade their priorities. A transaction's priority can be upgraded in two ways, with respect to its accessibility to data and CPU. For the former, transactions are not aborted by updates due to data conflicts, while for the latter, transactions are not always scheduled to execute after updates and recomputations.

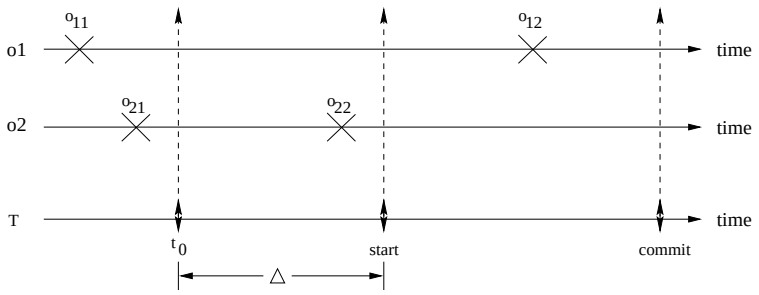

Figure 3: This figure illustrates the differences between ACS, weak ACS and RCS. Suppose a transaction T reads objects o_1 and o_2 during its execution, with maximum staleness Δ. Let o_{ij} denote the j^{th} version of object o_i. In an ACS, the set of objects read by T must be (o_{12}, o_{22}) because only this set of values can be found in an IS at the commit time of T. In a weak ACS, the object versions read can be (o_{11}, o_{22}) and (o_{12}, o_{22}) as they can be found in an IS at a time not earlier than the start time of T. In an RCS, the object versions available to T are (o_{11}, o_{21}), (o_{11}, o_{22}) or (o_{12}, o_{22}) as they can be found in an IS at a time not earlier than t_0.

Definition 3: **weak absolute consistent system (weak ACS)** *In a weak absolute consistent system, 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 $t_1 \geq t$.*

A weak ACS is very similar to an ACS in that transactions in both systems read relative consistent data. The major difference is that in a weak ACS, the data that a transaction reads need only be fresh to the point when the transaction reads them, not when the transaction commits (as is in an ACS). The implication is that once a transaction successfully read-locks a set of relatively consistent data, it needs not be aborted by later updates due to data conflicts. The transaction thus has a better chance of finishing before its deadline.

We can further 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 4: **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 $t_1 \geq t - \Delta$.*

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. Note that we can consider weak ACS as a special case of RCS with a zero Δ. Figure 3 illustrates the three correctness criteria, namely, ACS, weak ACS, and RCS.

5 Transaction Scheduling and Consistency Enforcement

In this section we discuss different policies to schedule updates, recomputations, and application 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 simple terms, OD-H switches between URT and OD depending on whether application transactions are running in the system. We will show that OD-H performs better than URT and OD in Section 7. In these policies, we assume that the relative priorities among application transactions are set using the traditional earliest-deadline-first priority assignment. We start with a brief reminder of the characteristics of the three types of activities.

Updates. We assume that updates arrive as a single stream. Under the URT policy, there is only one update process in the system executing the updates in a FCFS manner. For OD, there could be multiple update activities running concurrently: one from the arrival of a new update, and others triggered by application transactions. We distinguish the latters from the formers by labeling them "On-demand updates" (or OD-updates for short).

Recomputations. When an update arrives, it spawns recomputations. Under URT, we assume that recomputation batching is employed to reduce the system's workload [3]. With batching, a triggered recomputation goes to sleep for a short while during which other newly triggered instances of the same recomputation are ignored. Under OD, recomputations are only executed upon transactions' requests, and hence batching is not applied. To ensure temporal consistency, however, a recomputation induced by an update may have to perform some book-keeping processing, even though the real recomputation process is not executed immediately. We distinguish the recomputations that are triggered on-demand by transactions from those book-keeping recomputation activities by labeling them "On-demand recomputations" (or OD-recoms for short).

Application Transactions. Finally, we assume that application transactions are associated with firm deadlines. A tardy transaction is useless and thus should be aborted by the system.

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. CPU scheduling is more complicated due to the various batching/on-demand policies employed. 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. (We consider a WACS as a special case of an RCS and hence do not explicitly discuss it in this section.)

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 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.[4] 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[5];

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

[5] 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.

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.

- 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. Note that if there are multiple updates on x during the interval I, the Version Manager would have a choice of a valid version. We defer our discussion on this *version selection* issue later. 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. We will discuss how the "very-old" versions are pruned away to keep the multi-version database small at the end of this section.

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 checks and 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 $[ts(T) - \Delta, \infty]$.[6] 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:

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

- 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.

- 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.

Here, we summarize the key procedures for handling the various activities in an OD-RCS system.

- On an update arrival. Newly arrived updates have the highest priorities in the system and are handled FCFS. An update u on a base item x is sent to the OD Manager. Each unapplied update is associated with a validity interval. The validity interval of u is set to $[ts(u), \infty]$. If there is a previous unapplied update u' on x in the ODM, the UTB of $VI(u')$ is set to $ts(u)$; otherwise the latest version of x in the database will have its UTB set to $ts(u)$. Similarly, for any view item v that depends on x, if its latest version in the database has an open UTB (i.e., ∞), The UTB will be updated to $ts(u)$. The changes to the data items' UTBs may induce changes to some transactions' validity intervals. The Version Manger is again responsible for updating the transactions' VIs.

- Running an application transaction. A transaction T with a start time $ts(T)$ has its validity interval initialized to $[ts(T) - \Delta, \infty]$. If T reads a base item x, The VM would select a version x_i for T that is valid within $VI(T)$. If such a version is unapplied, an OD-update is triggered by the OD Manager. The OD-update inherits the priority of T. If T reads a view item v, The VM would select a version v_j for T that is valid within $VI(T)$. If no such version in the database is found, an OD-recom r to compute v is triggered. This OD-recom inherits the priority *and* the validity interval of T, and is processed by the system in the same way as for an application transaction.

5.2.3 Pruning the multi-version database

Our RC system requires a multi-version database and an OD Manager that keeps multiple versions of updates in the unapplied lists. We remark that it is not necessary that the system keeps the full history on-line. One way to prune away *old* versions is to maintain a *Virtual Clock* (VC) of the system. We define VC to be the minimum of the start times of all running transactions minus Δ. Any versions (be they in the database or in the unapplied lists) whose UTBs are smaller than the virtual clock can be pruned. This is because these versions are not valid with respect to any transaction's validity interval and thus will never be chosen by the Version Manager. The virtual clock is updated only on the release or commit of an application transaction.

5.2.4 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, an application transaction may have to materialize quite a number of items it intends to read on-demand. This may cause 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.

Before we proceed to discuss the details of the model, we would like to remark that the purpose of the simulation experiments is not to study the performance of a specific RTDB system when it uses URT or On-Demand. Instead, they are aimed to identify the performance characteristics of the scheduling policies in meeting the different temporal consistency requirements. In practice, an RTDB system can be very complex. Application transactions generated from the users can be extremely varied, ranging from ones with short computation to ones that have thousands of operations; Recomputations can be simple aggregate functions or ones that require complex computational analyses. If we model all this complexity, our results will be obscured by many intricate factors which impair our understanding of the basic tradeoffs of the scheduling policies. Instead, we chose a relatively simple model that captures the essential features of the scheduling problem, so that the observations made are more comprehensible.

In our simulation model, we implemented all the necessary components as described in

Section 5. These include a HP-2PL lock manager, an update installer, a disk manager, a buffer manager, an OD manager (for the On-Demand policy), a version manager (for RCS), and a transaction manager (which handles priority assignment, transaction aborts and restarts, recomputation batching, and transaction scheduling). 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. If the average values of fan-out and fan-in are $\overline{F_o}$ and $\overline{F_i}$ respectively, we have

$$N_b \cdot \overline{F_o} = N_d \cdot \overline{F_i}.$$

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. Under the URT policy, recomputations are batched. A recomputation is delayed t_{FD} seconds before execution, during which all instances of the same recomputation are ignored. Application transactions are generated as another stream of Poisson processes with an arrival rate λ_t. A transaction consists of a number of read/write operations. Each database object has an equal probability of being accessed by an operation. Each transaction performs N_{op} database operations. Each transaction 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[7], $ar(T)$ is the arrival time of T, and *slack* is the slack factor. In the simulation, *slack* is uniformly chosen from the range [S_{min},S_{max}].

The values of the simulation parameters were chosen as reasonable values for a typical financial application. Where possible, we have performed sensitivity analysis of key parameter values. 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.[8]

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

[8] We chose a relatively small database (3,000 base items) to model "hot items". That is, those data items that are frequently updated and those that cause recomputations. In practice, the database would have many other "cold items" as well: those that get updated occasionally and do not trigger recomputations. We have done experiments modeling "cold items". Since the results show similar conclusion as our simple model, we do not explicitly model "cold items" in this paper.

We assume a high-end disk, such as Seagate ST39103LC.

"CPU time per operation" includes the time to perform data locking, memory accesses, CPU computation. We assume transactions perform complex data analysis such as those performed in a financial expert system.

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

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. Also, in the graphs presented below, we consistently use solid lines for ACS and dotted lines for RCS. The three scheduling policies (URT, OD, and OD-H) are associated with different line-point symbols.

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 4 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

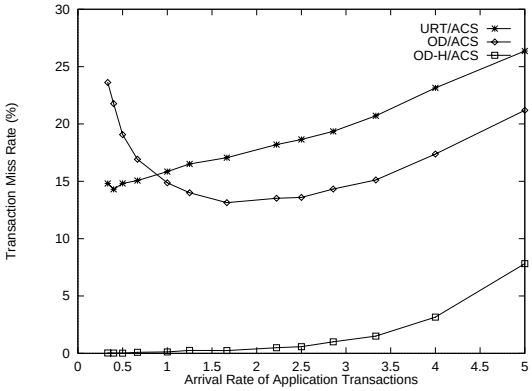

Figure 4: Miss rate vs λ_t (ACS)

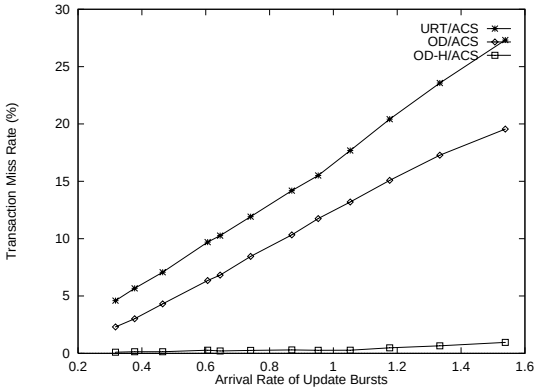

Figure 5: Miss rate vs λ_u (ACS)

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. Similarly, a recomputation can be restarted by an update if they engage in a data conflict. Restarting recomputations means adding extra *high priority* workload to the system under URT. This intensifies the *Low Priority* factor which causes missed deadlines. We call this restart factor ***Transaction Restart***.[9] From our experiment result, we observe that the average restart rate of transactions due to lock conflicts is about 2% to 3%, while that of recomputations is about 0.5%. We remark that even though the restart rate of recomputations is not too high, its effect could be significant, since recomputations are in general numerous and long.

By using the On-Demand approach, 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 4), we see that MD_{OD}^{AC} is smaller than MD_{URT}^{AC} 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_{OD}^{AC}.

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. As evidence, Figures 6 and 7 show the numbers of OD-updates and OD-recoms per transaction respectively. We see that as many as 12 updates and 3.5 recomputations are triggered by (and

[9] "Transaction and Recomputation Restart" would be a more precise term. However, we use the shorter form to save space.

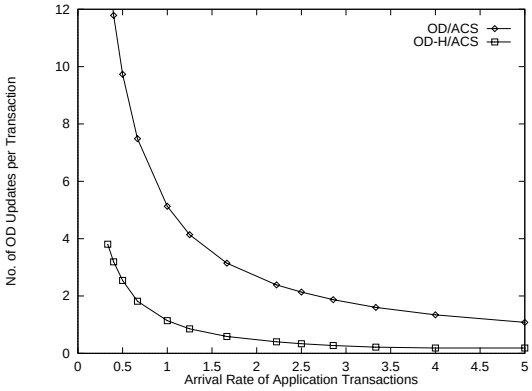

Figure 6: Number of OD-updates per transaction (ACS)

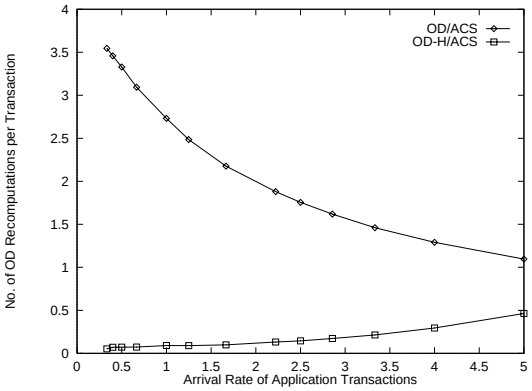

Figure 7: Number of OD-recoms per transaction (ACS)

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. OD-H can thus be considered a hybrid of OD and URT. Figure 4 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 improvement is attributable to a very small number of on-demand requests (Figures 6 and 7). 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.

Effect of update arrival rate In another experiment, we vary the update arrival rate (λ_u). Figure 5 shows the result. We see that a larger λ_u causes more missed deadlines under all the scheduling policies. More updates implies a higher update load and more recomputations. This directly intensifies the effects of *Low Priority*, *Batching Wait*, and *Transaction Restart*. Also, a higher update rate causes data items to become stale faster. This worsen the effect of *OD Wait*. Hence, all policies suffer. Among the three, MD^{AC}_{URT} increases most rapidly with λ_u, since it is affected by three factors. On the contrary, OD-H suffers the least, since it is mainly affected by *Transaction Restart* only.

Effect of slack Our next experiment tests the sensitivity of the three policies against transaction slack. Figure 8 shows the miss rates versus the maximum slack S_{max}. From the figure we see that when slack is tight (e.g., $S_{max} < 2.5$), MD^{AC}_{OD} rises sharply as S_{max} decreases. Recall that OD suffers when a transaction runs into stale data, in which case the transaction has to wait for some OD requests to finish (*OD Wait*). It is thus important that a transaction be given enough slack for it to live through the wait. In other words, OD is very sensitive to the amount of slack transactions have. In order to improve OD's performance, again, the key is to keep the database as fresh as possible (e.g., by OD-H). From Figure 8 we see that OD-H maintains a very small miss rate, and is relatively unrattled even under a small slack situation.

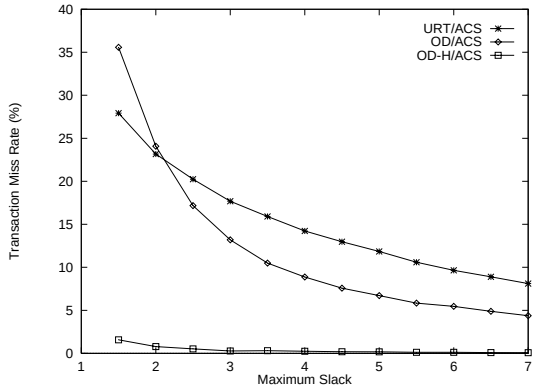

Figure 8: Miss Rate vs S_{max} (ACS)

7.2 Relative Consistent System

Our previous discussion illustrates that in an ACS, URT suffers from three adverse factors, namely *Low Priority*, *Batching Wait*, and *Transaction Restart*. These three factors lead to a high MD^{AC}_{URT}. By switching from URT to OD, we eliminate *Low Priority* and *Batching Wait*, but introduce *OD Wait*. We then show that the hybrid approach, OD-H, can greatly reduce the effect of *OD Wait* (see Figures 6 and 7). Hence, the only culprit left to tackle is *Transaction Restart*.

As mentioned in Section 5.2, an RCS uses a multi-version 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*.

Figure 9 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. Figure 10 magnifies the part containing the curves for $MD^{AC}_{OD\text{-}H}$ and $MD^{RC}_{OD\text{-}H}$ for clarity.

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 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.

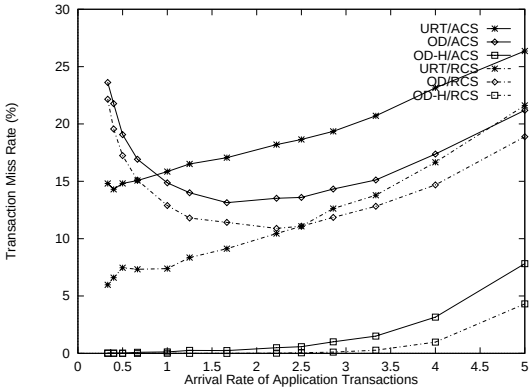

Figure 9: Miss rate vs λ_t (ACS & RCS)

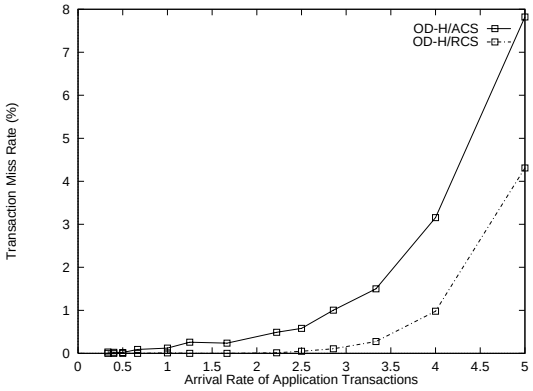

Figure 10: Miss rate vs λ_t ($MD^{AC}_{OD\text{-}H}$ and $MD^{RC}_{OD\text{-}H}$)

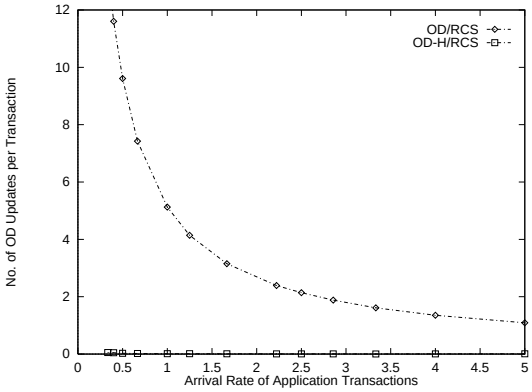

Figure 11: Number of OD-updates per transaction (RCS)

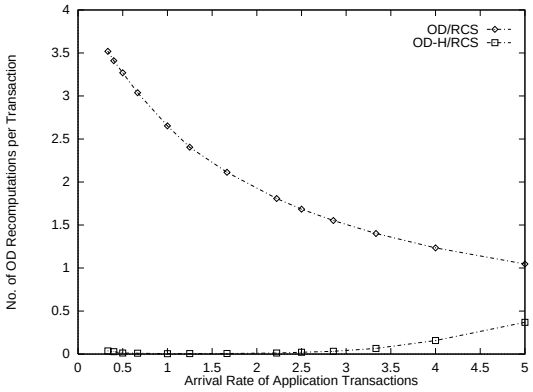

Figure 12: Number of OD-recoms per transaction (RCS)

For OD, we see that the improvement achieved by an RCS is not as big as in the case of URT. This is because, although *Transaction Restart* is eliminated, the problem of *OD Wait* is not fixed. Figures 11 and 12 show the numbers of OD-updates and OD-recoms per transaction respectively in an RCS. If we compare the curves in Figures 11 and 12 with those in Figures 6 and 7, we see that, under OD, an average transaction triggers more or less the same number of OD requests in the two systems. Recall that a transaction would issue an OD request if it attempts to read a not-yet-materialized data item. In an ACS, each item has only one (the latest) version. A transaction is forced to issue an OD request if the latest version is not yet updated. On the other hand, in an RCS, each item has multiple versions. A transaction can *avoid* issuing an OD request if it can find a materialized version within the transaction's validity interval. So in theory, fewer OD requests are issued in an RCS than in an ACS. Unfortunately, the pure OD policy does not actively perform updates and recomputations. Hence, few of the data versions are materialized before transactions read them. The effect of *OD Wait*, therefore, does not get improved. As we have discussed, the effect of *OD Wait* is the strongest when transactions are scarce. From Figure 9, we see that MD^{RC}_{OD} is much higher than MD^{RC}_{URT} when λ_t is small.

In last sub-section, we explained how OD-H reduces transaction miss rate by avoiding three

of the four adverse factors faced by URT and OD. Figure 10 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 queueing delays. From Figure 10 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. The percentage of saved deadlines by an RCS is even more marked when λ_t is small.

8 Conclusions

In this paper we defined temporal consistency from the perspective of transactions. In an absolute consistent system, a transaction cannot commit if some data it reads become stale at the transaction's commit time. We showed that this consistency constraint is very strict. It often results in high transaction miss rate. If transactions are allowed to read slightly stale data, however, the system's performance can be greatly improved through the use of a multi-version database. We defined a relative consistent system 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 studied three scheduling policies: URT, OD, and OD-H in a system where three types of activities: updates, recomputations, and application transactions are present. We discussed how the policies are implemented in a real-time database system to ensure absolute consistency or relative consistency. We showed that an ACS using URT is the easiest to implement. An HP-2PL lock manager and a simple static priority-driven scheduler suffice. This system, however, could have a very high transaction miss rate. To improve performance, two techniques were considered. One is to perform updates and recomputations on-demand, and the other is to relax the temporal consistency constraint from absolute to relative. Implementing these techniques add complexities to the implementation, though. For example, an on-demand manager is needed for OD; a version manager is needed for an RCS. We showed the pure On-Demand strategy does not perform well in a system where transactions arrive at a low rate and have very tight deadlines. To improve the pure OD policy, a third technique of combining the benefit of URT and OD was studied. The resulting scheduling policy, OD-H, is shown to perform much better than the others.

We carried out an extensive simulation study on the performance of the three scheduling policies, under both an ACS and an RCS. We identified four major factors that adversely affect the performance of the policies. These factors are *Low Priority*, *Batching Wait*, *Transaction Restart*, and *OD Wait*. Different policies coupled with different consistency systems suffer from different combinations of the factors. Table 2 summarizes our result. From the performance

	ACS			RCS		
	URT	OD	OD-H	URT	OD	OD-H
Low Priority	×			×		
Batching Wait	×					
Transaction Restart	×	×	×			
OD Wait		×			×	

Table 2: Factors that cause missed deadlines

study, we showed that OD-H when applied to an RCS results in the smallest miss rate.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: a performance evaluation. In *Proceedings of the 14th VLDB Conference*, August 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. Carey, R. Jauhari, and M. Livny. On transaction boundaries in active databases: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):320–36, 1991.

[5] M. Cochinwala and J. Bradley. A multidatabase system for tracking and retrieval of financial data. In *Proceedings of the 20th VLDB Conference*, pages 714–721, 1994.

[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*, 1(2):199–226, 1993.

[12] A. Segev and A. Shoshani. Logical modeling of temporal data. In *Proceedings of the ACM SIGMOD Annual Conference on Management of Data*, pages 454–466, 1987.

[13] 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, Oct. 1995.

[14] 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.

[15] 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.