# Maintaining Temporal Consistency of Discrete Objects in Soft Real-Time Database Systems

Ben Kao, Kam-Yiu Lam, *Member*, *IEEE*, Brad Adelberg,
Reynold Cheng, *Student Member*, *IEEE*, and Tony Lee

**Abstract**—A real-time 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 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 database system, two timing constraints need to be enforced. We require that 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 for discrete data objects. 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.

**Index Terms**—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 [21], [13], [2], [3], [8]. 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. In a dynamic environment, an entity changes its state in either a *continuous* or a *discrete* fashion. Changes to an entity are *continuous* if the state of the entity is constantly changing. One example would be the altitude of a flying aircraft. Base items that model continuous entities must be periodically updated. On the other hand, changes to an entity are *discrete* if the changes occur at distinct time instants. One example is stock prices, which only change when trades are made. Base items that model discrete entities are updated only when changes occur.

To better support decision making, the large number 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 sectoral subindices), 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 or write other static data, such as a knowledge base capturing expert rules.

Fig. 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

- *B. Kao is with the Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong. E-mail: kao@csis.hku.hk.*
- *K.-Y. Lam and T. Lee are with the Department of Computer Science, City University of Hong Kong, 83 Tat Chee Ave., Kowloon, Hong Kong. E-mail: cskylam@cs.cityu.edu.hk.*
- *B. Adelberg is with the Computer Science Department, Northwestern University, Evanston, IL . E-mail: adelberg@cs.nwu.edu.*
- *R. Cheng is with the Department of Computer Science, Purdue University, West Lafayette, IN 47907. E-mail: ckcheng@cs.purdue.edu.*

*Manuscript received 23 Oct. 2000; revised 1 Aug. 2001; accepted 24 Oct. 2001.*
*For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 113038.*

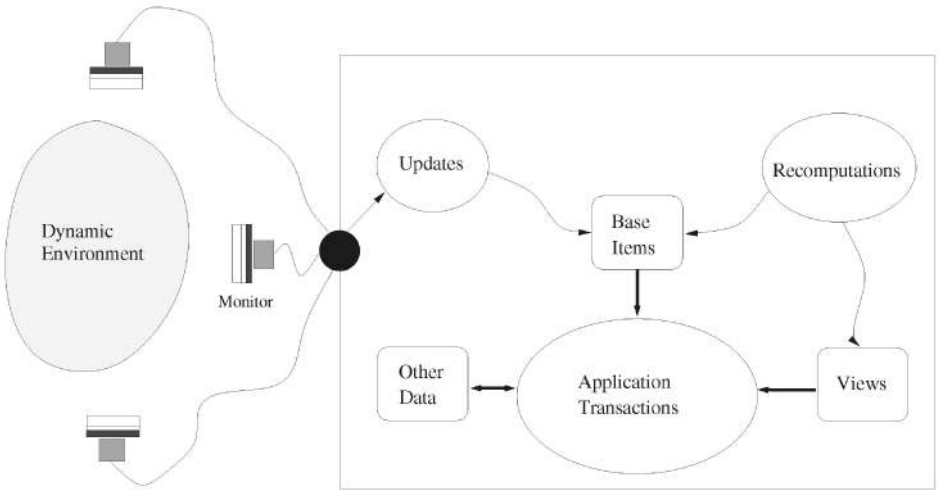


Fig. 1. A real-time database system.

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.

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, consider a transaction whose execution time is 1 second which 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 percent 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) are a serious problem. The definition of data timeliness thus needs to be relaxed to accommodate those difficult situations.

The issue of data timeliness has been previously studied. However, most of these previous works focus on continuous objects. They assume, for example, that updates to base items arrive at *regular*, *periodic* intervals. Moreover, the definition of *staleness* is *time-based*. That is to say, a base item's value is outdated and should not be used if a certain predefined period of time has passed since the item's last update. For many applications, however, discrete objects are the focus (such as stock quotes in a financial database). As we will elaborate further in Sections 2 and 4, the traditional way of enforcing data timeliness (for continuous objects) does not apply naturally to applications with discrete objects. For example, updates to data items that model discrete objects are sporadic and unpredictable; the definition of data staleness should be *event-based*, etc. All these differences lead us to reexamine the issues of how data timeliness should be defined and handled.

Given a data timeliness definition, we need a suitable transaction scheduling policy to meet its requirements. 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 lowers their chances of meeting deadlines. This is especially true when updates (and, thus, recomputations) arrive at a high rate. The challenge is therefore on how the various activities should be scheduled so that the two timing requirements (data and transaction) are satisfied.

To sum up, the goals of our study are:

- to define the temporal correctness of discrete objects 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.

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 and present the simulation results. Section 7 presents some further discussion on the future works. We conclude the paper in Section 8.

## 2 Related Works

### 2.1 Temporal Correctness of Continuous Objects

As we have argued in the last section, data timeliness plays a crucial role in an RTDB. A transaction can make a faulty decision if the data supplied to it is not fresh enough. Here, we briefly describe some previous works in the maintenance of data timeliness for continuous objects in an RTDB. We discuss the concept of *temporal correctness*, a formal definition of data timeliness proposed by Ramamritham [15]. We also describe some works on enforcing the temporal correctness criteria. We then argue that the traditional definition of temporal correctness, assuming a continuous data model, is not suitable for some applications. A discrete data model is needed instead.

In the last section, we used an example to illustrate that maintaining data timeliness is nontrivial. It often leads to transaction aborts and restarts, wastes system resources, and hurts the chances of meeting transaction deadlines. To address these problems, some researchers assert that the data timeliness requirement should be relaxed. In [11], Kuo and Mok propose the concept of *data similarity*, which allows a transaction to read a slightly outdated data object within a predefined tolerance level. The value of the outdated object should be "similar" to the most updated value in the sense that its use will not yield any adverse results. This technique saves a transaction from being aborted, even if the data the transaction has read have become stale. In so doing, hopefully, a transaction would have a higher chance of committing before its deadline.

The data timeliness constraint is more formally defined in [15]. In that paper, Ramamritham defines *data temporal correctness* as how well the data maintained by an RTDB models the actual state of the environment. A *continuous* data model is assumed in which the state of an object is constantly changing. Each data object is associated with an age. As time passes, an object ages and, if its age is larger than a predefined threshold, the object becomes outdated and needs to be refreshed. Based on this assumption, two correctness constraints, namely, absolute consistency and relative consistency, are defined. These constraints are used to ensure that data read by a transaction are fresh and temporally correlated. A data item obeys absolute consistency when its value is updated within a predefined time interval (called the *absolute validity interval* ($avi$)). A set of data items are relatively consistent when their values are updated within a certain time interval, called the *relative validity interval* ($rvi$). In Section 4.1.1, we will discuss the absolute and relative consistency constraints for continuous objects in more detail.

The temporal correctness criteria can be too strict for some real-time database systems where the update rates of data items are high. Enforcing these constraints can lead to frequent transaction aborts and restarts, undermining an RTDB's ability of meeting transaction deadlines. As an example, suppose a transaction $T$ whose remaining execution time is 1 second has just read a data item $d$ with an $avi$ of 0.1 seconds. To enforce absolute consistency, $d$ has to be updated within 0.1 seconds. Once $d$ is updated, the value of $d$ held by $T$ becomes stale and $T$ has to be aborted and restarted. To resolve this problem, different correctness criteria and scheduling methods have been proposed. In [7], DiPippo and Wolfe relax the stringent requirements of temporal correctness by proposing an object-based semantic concurrency control protocol. This protocol allows a transaction to read stale data based on the semantics of objects, such as "imprecision" values. Similar to Kuo and Mok's data similarity concept [11], a transaction is saved from being aborted due to stale data. It is worth noting that the semantic information required in this protocol may not be available in some RTDBs.

Ahmed and Vrbsky suggest another policy for balancing temporal correctness and transaction timeliness by using on-demand (triggered) updates [4]. In their model, there are three types of transactions: sensor transactions, which update base data items, update transactions, which refresh derived items, and user transactions, which read data items. When a user transaction finds that a data item it intends to read is outdated, the system generates an update transaction (called *triggered update*) to refresh the item. The user transaction waits until the triggered update finishes so that it is given the most recent value of the item and temporal consistency is enforced. The drawback of producing on-demand updates is that a transaction will be forced to wait. This increases the chance of missing transaction deadlines. In particular, if a transaction does not have enough slack time to wait for a triggered update, it had better not generate the update. The time saved can be used to serve the transaction, increasing its chance to commit before its deadline. The authors propose an algorithm which uses the information of a transaction, including the slack time and the execution time, to decide whether an update should be triggered. Although it is shown that the algorithm can balance data freshness and deadline miss rates, such information is often unavailable in practical RTDBs.

Notice that the works we have mentioned assume a *continuous* data model. In this model, the states of the entities in the external environment change continuously. To reflect these changes, the data objects that model the entities need to be updated according to the absolute and relative consistency requirements. In many situations, however, it can be difficult to assign suitable values of $avi$ and $rvi$ to the data objects. For example, the price of a stock remains unchanged until an update that changes its value

arrives. Since the arrival time of an update is unpredictable, we cannot assign reasonable values of $avi$ and $rvi$ to the stock price. A small $avi$ may result in many unnecessary updates, while a large $avi$ can cause the price to be stale for an extensive period of time. To cope with situations where $avi$ and $rvi$ are not readily known, we need to consider the *discrete data model* [2], [16]. In this model, a data object's value remains unchanged until an update arrives. Also, the definitions of absolute consistency and relative consistency constraints are no longer based on $avi$ and $rvi$ values. We will discuss the temporal correctness criteria for discrete objects in more detail in Section 4.1.2.

In this paper, we propose new transaction scheduling algorithms which ensure that transactions comply with the temporal constraints of discrete data objects. Our new scheduling algorithms do not make use of the auxiliary information required by other papers, which is not always available. We will explain the various definitions of temporal correctness for continuous and discrete data objects in Section 4 and propose several transaction scheduling algorithms for discrete data objects in Section 5.

### 2.2 Scheduling Updates, Recomputations, and Transactions

Adelberg et al. study the load balancing problems of updates, recomputations, and application transactions [2], [3]. In [2], they study the load balancing issues between updates and transactions in an RTDB. In their system model, updates come at a very high rate and transactions must be committed before their deadlines. The authors point out that updates need not be executed with full transactional support; they can be applied to data by using a single *update process*, which results in great improvement in system performance. The authors also propose several heuristics and examine their effectiveness in maintaining data freshness and transaction timeliness. They find out that the *On-Demand* strategy, with which updates are only applied when required by transactions, gives the best overall performance. In another study [3], they investigate the balancing problems between derived data (views)[1] updates and transactions. They observe that recomputations often come in *bursts*, obeying the principle of *update locality*. They propose the *Forced Delay* approach to delay the triggering of a recomputation for a short period so that recomputations on the same view object can be *batched* into a single computation. Their study shows that batching significantly improves the performance of an RTDB.

The two papers of Adelberg et al. ([2] and [3]) are very closely related. The former investigates the scheduling issues of updates and transactions, while the latter examines the balancing problems of recomputations and transactions. They do not, however, examine a system in which all three activities—updates, recomputations, and transactions—are present. Also, both studies report how *likely* temporal consistency is maintained under different scheduling policies, but do not discuss how to enforce the consistency constraints. For example, [3] uses the percentage of transactions that read stale data to quantify how well absolute consistency (a kind of temporal consistency constraint) is maintained by the scheduling algorithms. It does not, however, discuss how a scheduling policy should be designed to ensure that transactions can always read absolutely consistent data. In this paper, we consider various scheduling policies for enforcing temporal consistency in an RTDB in which updates, recomputations, and transactions coexist.

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

### 2.3 Multiversion Concurrency Control

In [17], 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.

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 serves 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 for resources with write operations. Hence, read-only transactions are never restarted and the costs of concurrency control and restart can be significantly reduced.

The concurrency control protocol described later on in this paper also adapts a multiversion database. However, our protocol is different from the algorithms discussed above. In our model, application transactions are allowed to read old and committed versions of data items. There is no read-write conflict between application transactions and updates/recomputations and an application transaction need not be aborted because of an update/recomputation. Also, application transactions do not write any base items/views and, so, no write-write conflict occurs between application transactions and updates/recomputations. Our multiversion protocols, therefore, do not need a locking/restart mechanism. A more detailed discussion of our protocols is presented in Section 5.

## 3 Updates, Recomputations, and Transactions

For many real-time database applications, managing the data input streams and applying the corresponding database updates represents a nontrivial 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 US markets alone, the system needs to process more than 500 updates per second [6]. 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) warrants 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. 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 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 $b$ need not be executed if no transactions request $b$ before another update on $b$ arrives. Similarly, a recomputation on a view need 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.

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, [5] shows that transaction response time is much improved when *events* and *actions* (in our case, updates and recomputations) are 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 6, we evaluate their performance in meeting deadlines.

## 4 Temporal Correctness

### 4.1 Temporal Consistency

*Temporal Consistency* refers to how well the data maintained by an RTDB models the actual state of the environment [15], [17], [14], [9], [11], [18], [19], [10]. Basically, temporal consistency consists of two components: absolute consistency (or external consistency) and relative consistency. A data item is *absolutely consistent* (fresh) if it timely reflects the state of an external object that the data item models. A set of data items are *relatively consistent* if they are temporally correlated to each other.

The formal definitions of absolute and relative consistency depend on the type of temporal objects in the system. A data object is temporal if its value changes with time. Based on how the value changes, we can classify temporal data objects as *continuous* objects or *discrete* objects [10]. Most of the previous studies on temporal consistency maintenance concentrate on systems with continuous

2. For example, the S&P 500 index is derived from a set of 500 stocks; a summary of a stock's price in a 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.

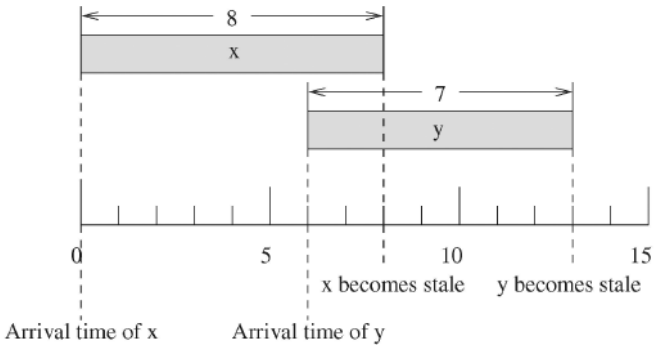


Fig. 2. Absolute and relative consistency.

objects. In the next two subsections, we will explain the nature of these two types of objects and the definitions of absolute and relative consistency for each of them. New correctness criteria are then defined for discrete objects.

#### 4.1.1 Continuous Objects

A *continuous* object represents an entity whose value changes continuously with time in the external environment. The arrival pattern of updates for these objects is usually periodic. They can usually be found in a plant control system, which has sensors to monitor the state of the environment, and in a military system, where positions of aircraft are tracked and reported. The absolute and relative consistencies of a continuous object can be defined as follows:

- **Absolute Consistency.** The *current* time is compared with an update's *arrival time*, which is an indication of which snapshot of the external object the update is representing. A data item is absolutely consistent if the difference of its last update's timestamp and the current time is smaller than some predefined maximum age $T$. (The value $T$ is also called the absolute validity interval, $avi$.) 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.
- **Relative Consistency.** For a set of data items used to derive a view, arrival times should be reasonably close to each other. To define relative consistency formally, consider a set of base items, $R$, which is used to derive a view. $R$ is called a *relative consistent set* and is associated with a *relative validity interval* ($R_{rvi}$). Denote the timestamp (arrival time) of a temporal item $d$ by $timestamp_d$, then $R$ is relatively consistent iff

$$\forall X, Y \in R, |timestamp_X - timestamp_Y| \leq R_{rvi}.$$

That is to say, the timestamp difference between any two objects in $R$ is not larger than $R_{rvi}$ [20], [9], [10], [14].

Fig. 2 illustrates the concepts of absolute and relative consistency for continuous objects. Suppose the maximum ages of $x$ and $y$ are 8 and 7, respectively, and they form a relative consistent set $R$, with $R_{rvi} = 5$. We observe that $x$ is absolutely consistent in the time interval $[0, 8]$, while $y$ is absolutely consistent in $[6, 13]$. $R$ is not relatively consistent because $|timestamp_x - timestamp_y| = |0 - 6| = 6 > R_{rvi}$. Note that the definitions of *MA* for a continuous object and $R_{rvi}$ for a set of related continuous objects are based on the maximum rate of changes of the objects and the correctness requirements of the application transactions.

#### 4.1.2 Discrete Objects

With discrete objects, the value of an entity remains unchanged until the next update arrives. The update can arrive at any discrete point in time and the arrival pattern is sporadic. As an example, a cellular phone network maintains a database that keeps track of the locations of its mobile phone users. The cell that is handling a user's connection remains unchanged until the user of the phone travels to another cell. The data objects that record the locations of users are thus discrete. Unlike continuous objects, it is difficult to define a suitable *MA* for a discrete object since the object changes its state at an unpredictable rate.

To formally define the notion of temporal consistency for discrete objects, we introduce the concepts of *version* and *validity interval*.

**Definition 1 (version).** *A* version $x$ *of a data item* $d$ *is a value of the external object that* $d$ *models. Every time the external object changes its value, a new version of* $d$ *is generated. Each version* $x$ *is thus associated with a time interval that specifies when the version is valid. We call this time interval the* validity interval *of* $x$ *(denoted by* $VI(x)$*). A validity interval* $VI(x)$ *consists of a lower time bound (*$LTB(x)$*) and an upper time bound (*$UTB(x)$*). We consider* $LTB(x)$ *to be the time instant at which an update of* $d$ *with* $x$*'s value arrives. We consider* $UTB(x)$ *to be the time instant at which the next update of* $d$ *arrives.*

The value of a data object thus goes through a series of versions. For notational convenience, we use a numeric subscript to enumerate the versions of an item. For example, $x_i$ represents the $i$th version of the data item $x$.

**Definition 2 (current version).** *The current version of an item is a version* $x_i$ *such that its validity interval contains the current time instant* $t_c$*, i.e.,* $t_c \in VI(x_i)$.

**Definition 3 (absolute consistency).** *A discrete data item* $d$ *is* absolutely consistent *if, at any time instant, a current version for* $d$ *can be found in the system.*

To illustrate the above definition, let us consider the following scenario: Suppose $x_i$ is the current version. After an update $u$ for $x$ arrives, $x_i$ is no longer the current version. This is because $UTB(x_i)$ has been set to the arrival time of $u$ and the current time instant is not contained in $VI(x_i)$ anymore. To ensure that $x$ is absolutely consistent, the system has to create a current version for $x$ immediately. This current version is $x_{i+1}$, which has the data value of $u$ and an *LTB* equal to the arrival time of $u$.

**Definition 4 (Relative consistency).** *Given a set of item versions* $R$*, the versions in* $R$ *are said to be* relatively consistent *if* $\bigcap\{VI(x_i)|x_i \in R\} \neq \emptyset$.

Definition 4 states that a set of relatively consistent item versions should be able to reflect the states of the external objects at the same time.

To illustrate the two consistency constraints for discrete objects, let us consider Fig. 2 again. We assume that $x$ and $y$ are discrete objects and their current versions are $x_m$ and $y_m$, respectively. If $VI(x_m) = [0, 8]$ and $VI(y_n) = [6, 13]$, then $x$ is absolutely consistent during [0, 8] and $y$ is absolutely consistent within [6, 13]. We also notice that $x_m$ and $y_n$ are relatively consistent in the time interval $[0, 8] \bigcap [6, 13] = [6, 8]$.

Under our definitions of temporal consistency, if a set of discrete data items $D$ are absolutely consistent, then their current versions must be relatively consistent. This is because, by definition, all current versions of the items in $D$ are present in the system. All these versions, being current, must have their validity intervals contain the current time. Hence, the intersection of these intervals is nonnull. These current versions are thus relatively consistent.

Notice that, since it may be difficult to achieve absolute consistency for systems where the update rates of the objects are very high, relaxing the correctness requirement to relative consistency can increase the chances of transactions reaching their commit states. This advantage, however, is obtained at the expense of data freshness provided to the transactions.

Here, we would like to remind the reader that the reason for defining discrete data objects and their temporal correctness criteria is that the entities mentioned in this subsection cannot be represented by continuous objects. Specifically, we cannot find suitable values of the absolute and relative validity intervals for these entities. For example, we cannot assign $avi$ to a stock price because we do not know when it will become outdated. In this paper, our focus is on the issue of temporal correctness for discrete data objects. Unless stated otherwise, in the rest of this paper, we assume that the definitions of absolute and relative consistency are based on discrete objects.

### 4.2 Transaction Temporal Consistency

If a base data item is updated but its associated views are not recomputed yet, the database is not relatively consistent. We have already proven 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 5 (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 6 (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 transactions must *read* absolutely consistent 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.

**Definition 7 (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 relatively 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 in an ACS). The implication is that, once a transaction successfully read-locks a set of relatively consistent data, it need 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 with respect to the usefulness of the information read by a transaction, this can improve the probability of meeting transaction deadlines.

**Definition 8 (relative consistent system (RCS)).** *In a relative consistent system with a maximum staleness $\Delta$, 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

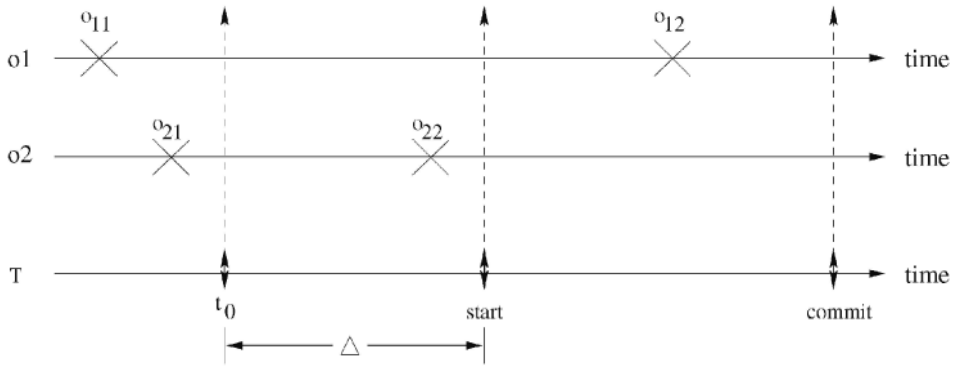


Fig. 3. This figure illustrates the differences between ACS, weak ACS, and RCS. Suppose a transaction $T$ reads objects $o_1$ and $o_2$ during its execution, with maximum staleness $\Delta$. Let $o_{ij}$ denote the $j$th version of object $o_i$. In an ACS, the set of objects read by $T$ must be ($o_{12}$, $o_{22}$) 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 ($o_{11}$, $o_{21}$), ($o_{11}$, $o_{22}$), and ($o_{12}$, $o_{22}$) as they can be found in an IS at a time not earlier than $t_0$.

$\Delta$. Fig. 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 a hybrid policy called OD-H. 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 6. 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 latter from the former 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 bookkeeping processing, even though the real recomputation process is not executed immediately. We distinguish the recomputations that are triggered on-demand by transactions from those bookkeeping 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

#### 5.1.1 URT

Ensuring absolute consistency under URT represents the simplest 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.

**URT/ACS**
Let $u$ be a newly arrived update, $x$ be the item to be updated by $u$, $r$ be a newly arrived recomputation, $T$ be an application transaction, and $B$ be the batching delay time.

**On arrival of** $u$
  For every view $v$ derived from $x$
    **if** $v$ is not write-locked **then**
      Trigger a recomputation for $v$
  Execute $u$

**On arrival of** $r$
  Issue write-lock to view $v$ which will be written by $r$
  Sleep for $B$ time units
  Execute $r$

**On executing** $T$
  For every read/write request
    Use HP-2PL protocol to serve the request
    **if** $T$ misses deadline **then**
      Abort $T$
  Commit $T$

#### 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 are handled in an FCFS manner and have higher priorities than OD-updates, OD-recoms, and transactions. 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;[4] 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.

**OD/ACS**

Let $P$ be an update/recomputation, and $x$ be the base/view item updated by $P$.

**On arrival of** $P$
  Send $P$ to ODM
  **if** $x$ is not in the unapplied set **then**
    Add $x$ to the unapplied set, associating $x$ with $P$
    Request write lock on $x$
  **else**
    **if** OD bit of $x$ is off **then**
      Associate $P$ with $x$
    **else**
      Abort the currently executing OD-update/OD-recom
      Release $P$ for execution
  **if** $P$ is an update **then**
    Spawn recomputations for the views that are based on $x$

**On read request from** $T$
  Send the read request to ODM
  **if** $x$ is in the unapplied set **then**
    **if** OD bit of $x$ is on **then**
      Wait until the OD-update/OD-recom completes
    **else**
      Set OD-bit of $x$ on
      Release the OD-update/OD-recom associated with $x$, inheriting $T$'s priority

**On release of OD-update/OD-recom**
  Notify ODM 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 or recomputation conflicts with the transaction. An RCS ameliorates this difficulty by allowing transactions to read slightly outdated

4. 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.

(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 denote the arrival time of an update $u$ by $ts(u)$. Also, 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 will discuss this *version selection* issue in Section 7. 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 multiversion database small at the end of this section.

#### 5.2.1 URT

Similarly 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 to see 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. Similarly 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 (for a short batching delay time), 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 $[ts(T) - \Delta, \infty]$.[5] 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 it is 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. Recall that $\Delta$ is the maximum staleness tolerable with reference to a transaction's start time.

**URT/RCS**
**On arrival of** $u$
Set $VI(x_i) = [ts(u), \infty]$
Set $UTB(x_{i-1}) = ts(u)$
For every $T$ that has read $x_{i-1}$
  Set $UTB(VI(T)) = min(UTB(VI(T)), ts(u))$
For every view $v$ derived by $x$
  **if** no recomputation for $v$ is sleeping **then**
    Trigger a recomputation for $v$
Execute $u$

**On arrival of** $r$
Set $UTB(v_j)$ to $ts(u)$
For every $T$ that has read $v_j$
  Set $UTB(VI(T)) = min(UTB(VI(T)), UTB(v_j))$
Sleep for $B$ time units
Set $VI(r) = VI(v_{j+1}) = \bigcap\{VI(x_i) | x_i \in$ readset of $r$ and $x_i$ is the newest version of $x\}$
Execute $r$, using the newest versions in its readset

**On executing** $T$
Set $VI(T) = [ts(T) - \Delta, \infty]$
For every read request on data item $x$
  Consult VM to select a version $x_i$ for $T$ such that $VI(x_i) \bigcap VI(T) \neq \emptyset$
  Set $VI(T) = VI(x_i) \bigcap VI(T)$
  **if** $VI(T) = \emptyset$ or $T$ misses deadline **then**
    Abort $T$
Commit $T$

#### 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 $x$.
- 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 (multiversion 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 previously 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., $\infty$), 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 $T$. After the OD-update finishes, $VI(T)$ is updated to $VI(x_i) \cap VI(T)$. 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. After the OD-recom is completed, $VI(T)$ is updated to $VI(v_j) \cap VI(T)$.

**OD/RCS**

**On arrival of** $u$
  Send $u$ to ODM
  Set $VI(u)$ to $[ts(u), \infty]$
  **if** there exists a previous unapplied update $u'$ on $x$ **then**
    Set $UTB(VI(u')) = ts(u)$
  **else**
    Set $UTB(\text{latest version of } x) = ts(u)$
  For every $T$ that has read the latest version of $x$
    Set
    $UTB(VI(T)) = min(UTB(VI(T)),$
    $UTB(\text{latest version of } x))$
  For every view $v$ that depends on $x$
    **if** the latest version of $v$ has $UTB$ of $\infty$ **then**
      Set $UTB(\text{latest version of } v) = ts(u)$
      For every $T$ that has read the latest version of $v$,
        Set
        $UTB(VI(T)) =$
        $min(UTB(VI(T)), UTB(\text{latest version of } v))$

**On executing** $T$
  Set $VI(T) = [ts(T) - \Delta, \infty]$
  For every read request
    **if** the read is base item $x$ **then**
      Consult VM to select a version $x_i$ such that
      $VI(x_i) \bigcap VI(T) \neq \emptyset$
      **if** $x_i$ is unapplied **then**
        Trigger an OD-update for $x_i$, inheriting the priority of $T$
        Wait for the OD-update of $x_i$ to complete
      Set $VI(T) = VI(x_i) \bigcap VI(T)$
    **if** the read is view item $v$ **then**
      Consult the VM to select a version $v_j$ such that
      $VI(v_j) \bigcap VI(T) <> \emptyset$
      **if** $v_j$ is unapplied then
        Trigger an OD-recom for $v_j$, inheriting the priority and the validity interval of $T$
        Wait for the OD-recom of $v_j$ to complete
      Update $VI(T)$ to $VI(v_j) \bigcap VI(T)$
    **if** $T$ misses deadline **then**
      Abort $T$
  Commit $T$

### 5.3 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, 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 and Results

In our simulation model, we implemented all the necessary components as described in Sections 3 and 5. These include an HP-2PL lock manager, an update installer, a recomputation transaction pool, 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 $\lambda_u$. Each burst consists of *burst_size* updates. The value *burst_size* is picked uniformly from the range $[BS_{min}, BS_{max}]$. Within a burst, update arrivals are modeled as Poisson processes with an arrival rate $\lambda_b$. To model spatial locality, each update would have a probability of $p_{space}$ of triggering the same set of recomputations as those triggered by the previous update. To model time locality, each update would have a probability of $p_{time}$ of being generated again. 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 $\lambda_t$. 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. Also, for each read/write operation, a transaction may have to search a number of versions before it can find an appropriate one. To model this searching overhead, we incorporate a penalty of $t_{ver}$ ms in processing each version during a search. 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,[6] $ar(T)$ 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 [12]. Each simulation run (generating one data point) processed 10,000 update bursts. The 95 percent confidence interval of our baseline experiment is $\pm 0.69$ percentage points. Table 1 shows the parameter settings of our baseline experiment.[7]

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

7. 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 conclusions as our simple model, we do not explicitly model "cold items" in this paper. We assume a high-end disk, such as Seagate ST39103LC. We choose a relative cache hit-rate (0.7) in our experiments because, in practice, many RTDB applications, such as financial databases, have relatively large cache memory. Sometimes, hot items were chosen and are placed in memory for fast accesses. "CPU time per operation" includes the time to perform data locking, memory accesses, CPU computation. We assume transactions perform complex data analyses such as those performed in a financial expert system.

TABLE 1
Baseline Settings

| Description | Parameter | Value |
|---|---|---|
| Update burst arrival rate (/sec) | $\lambda_u$ | 1.2 |
| Update arrival rate (/sec) | $\lambda_b$ | 33 |
| Burst size | $[BS_{min},BS_{max}]$ | [1,12] |
| Forced delay time (sec) | $t_{FD}$ | 1.0 |
| Spatial Locality | $p_{space}$ | 0.4 |
| Time Locality | $p_{time}$ | 0.4 |
| Transaction arrival rate (/sec) | $\lambda_t$ | 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) | $\Delta$ | 10.0 |
| Lookup time per version (ms) | $t_{ver}$ | 0.05 |

## 6.1 Results

In this section, we present some representative results obtained from our simulation experiments. 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 percent 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.

### 6.1.1 Absolute Consistent System

**Effect of Transaction Arrival Rate.** In our first experiment, we vary the transaction arrival rate ($\lambda_t$) from 0.5 to 5 and compare the performance of the three scheduling policies (URT, OD, and OD-H) in an absolute consistent system. Fig. 4 shows the result. From the figure, we see that, for a large range of $\lambda_t$ ($\lambda_t > 1.0$), URT performs the worst among the three, missing 14 percent to 26 percent 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. Similarly, a recomputation can be restarted by an update if they engage in a data conflict. Restarting recomputations means adding

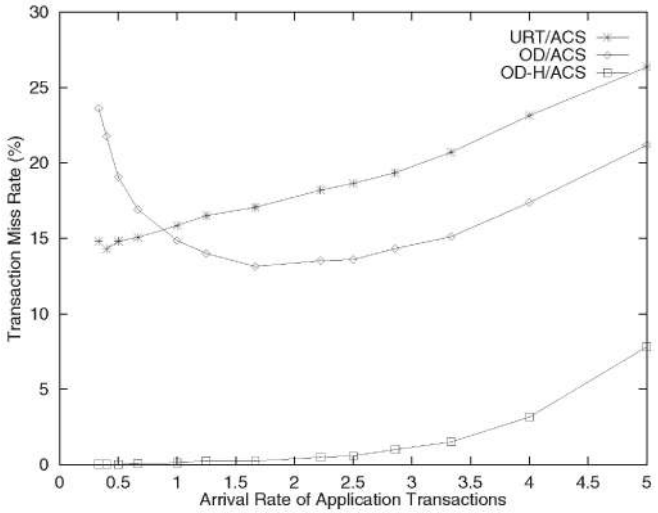


Fig. 4. Miss rate vs $\lambda_t$ (ACS).

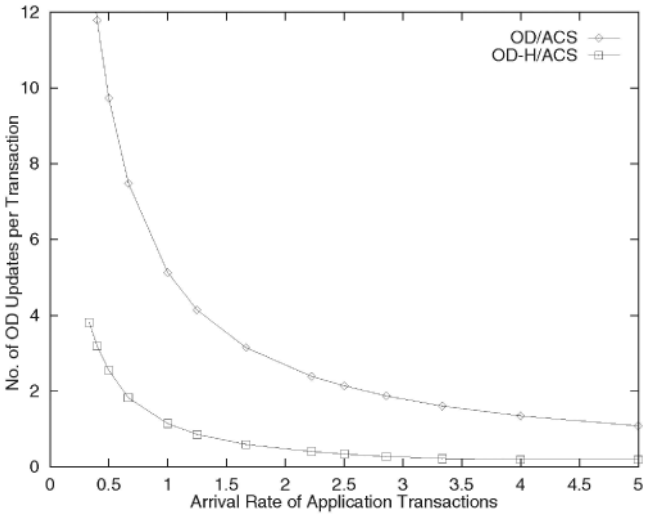


Fig. 5. Number of OD-updates per transaction (ACS).

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***.[8] From our experiment results, we observe that the average restart rate of transactions due to lock conflicts is about 2 to 3 percent, while that of recomputations is about 0.5 percent. We remark that, even though the restart rate of recomputations is not high, its effect could be significant, since recomputations are in general numerous and long.

By using the On-Demand approach, transactions are given their 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 (Fig. 4), we see that $MD_{OD}^{AC}$ is smaller than $MD_{URT}^{AC}$ for $\lambda_t > 1.0$. The improvement (about 5 percent for large $\lambda_t$) 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 $\lambda_t$) 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 $\lambda_t$ 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, Figs. 5 and 6 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 blocking) an average transaction under the OD policy. We call this adverse factor ***OD Wait***.

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

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. Fig. 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 (Figs. 5 and 6). 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.

### 6.1.2 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

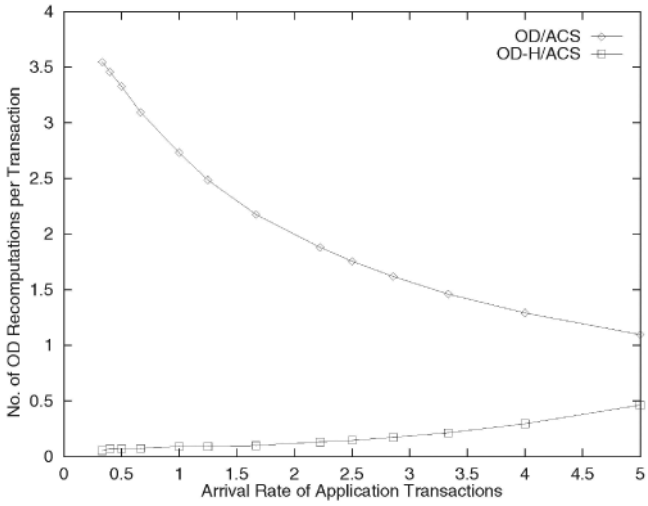


Fig. 6. Number of OD-recoms per transaction (ACS).

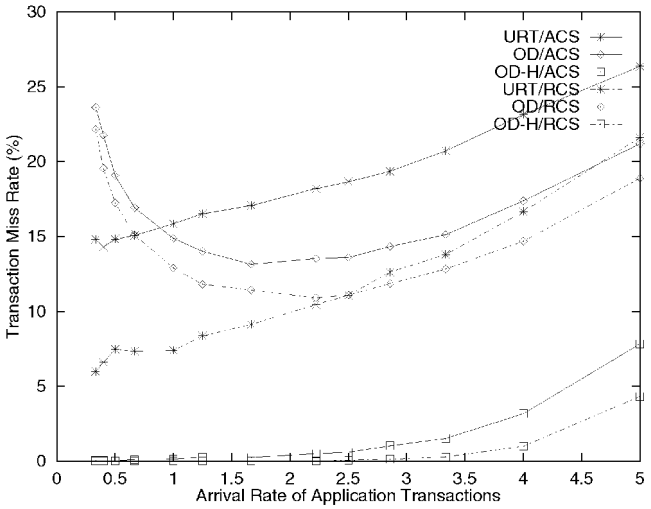


Fig. 7. Miss rate vs, $\lambda_t$ (ACS and RCS).

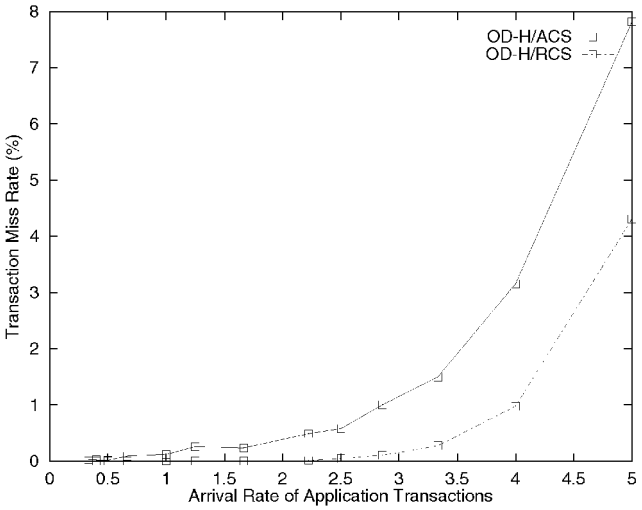


Fig. 8. Miss rate vs. $\lambda_t$ ($MD^{AC}_{OD\text{-}H}$ and $MD^{RC}_{OD\text{-}H}$).

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 Figs. 5 and 6). Hence, the only culprit left to tackle is *Transaction Restart*.

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 percent 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*.

Fig. 7 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. Fig. 8 magnifies the part containing the curves for $MD^{AC}_{OD-H}$ 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.

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. Figs. 9 and 10 show the numbers of OD-updates and OD-recoms per transaction, respectively, in an RCS. If we compare the curves in Figs. 9 and 10 with those in Figs. 5 and 6, 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 Fig. 7, we see that $MD^{RC}_{OD}$ is much higher than $MD^{RC}_{URT}$ when $\lambda_t$ is small.

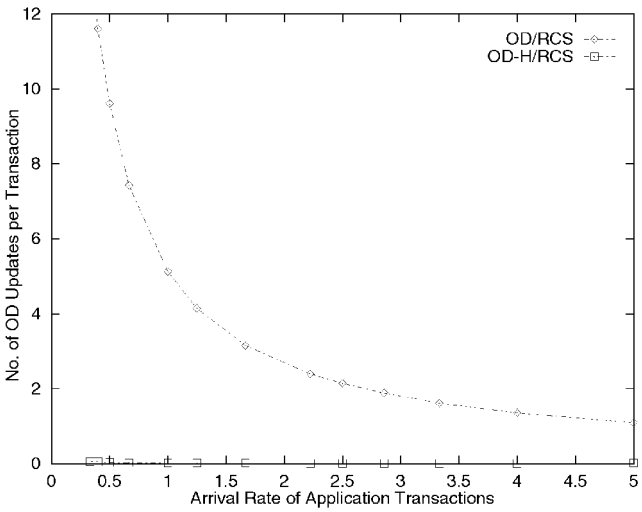


Fig. 9. Number of OD-updates per transaction (RCS).

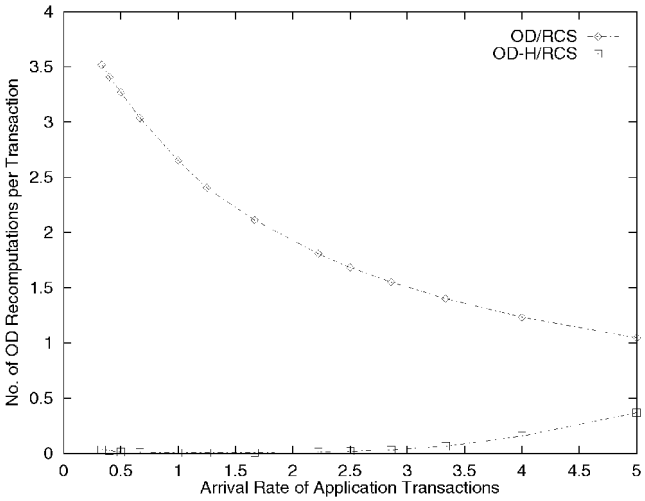


Fig. 10. Number of OD-recoms per transaction (RCS).

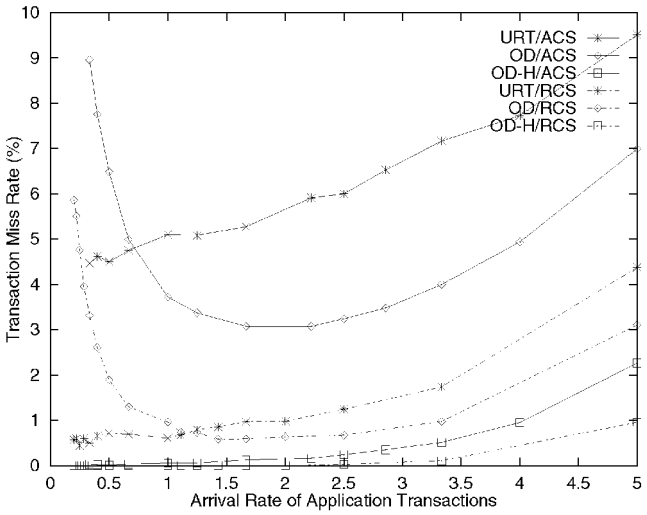


Fig. 11. Miss rate vs. $\lambda_t$ ($\lambda_u = 0.4/sec$).

In the last subsection, we explained how OD-H reduces the transaction miss rate by avoiding three of the four adverse factors faced by URT and OD. Fig. 8 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. $MD^{RC}_{OD-H}$ is close to 0 except when $\lambda_t$ is big. When the transaction arrival rate is high, missed deadlines are caused mainly by CPU and disk queuing delays. From Fig. 8, we see that the improvement of $MD^{RC}_{OD-H}$ over $MD^{AC}_{OD-H}$ 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 $\lambda_t$ is small.

We run the experiment again using a much smaller update arrival rate ($\lambda_u$). Fig. 11 shows the miss rates of the three scheduling policies with $\lambda_u = 0.4/sec$, one-third of the value used in our previous experiment. We can see that the trends of the performance of the scheduling policies are the same as those observed in our previous experiment (Fig. 7). We conclude that the comments made for the scheduling policies are valid under a wide range of $\lambda_u$.

We also examine how $\Delta$ affects the transaction miss rates of RCS. We notice that, as the value of $\Delta$ increases, the transaction miss rates of all RCS policies are reduced. This drop is attributed to the fact that a larger value of $\Delta$ allows an application transaction to have a higher chance of finding a materialized version that has a nonnull intersection with its validity interval. Since the relative performance of the scheduling policies stays the same over a wide range of $\Delta$ value, we do not show the result of the sensitivity study in this paper.

## 7 Further Discussions

With our current RTDB model, we assume that the system does not know what data items a transaction will read before its execution. In certain RTDBs, however, some transactions have fixed access patterns and the data they read are already known before execution. this information can be used to reduce the number of missed deadlines in our scheduling protocols.

- **OD/ACS.** In an ACS, a transaction holds a lock on any item it needs to read. When it triggers an OD-update or OD-recom, the items locked by the transaction before the OD request will be locked for an extra amount of time—the execution time of an OD request. If a transaction knows what items will be required for materialization prior to its execution, it can materialize those items first. The transaction can then start its normal operations and the items locked by the transaction will not suffer extra locking time due to OD requests. As a result, transaction blocking is less severe.
- **URT/RCS.** In our current URT/RCS protocol, a transaction is aborted if the system fails to find a valid materialized version to service its read request. This situation can be improved if the readset of a transaction is completely known before its execution. When a transaction starts, the system determines what versions of items to provide for the transaction's readset in order to minimize the probability that the transaction is aborted. It tries its best to provide a transaction with versions that are all materialized and valid within the transaction's validity interval. The chance of a transaction abort is thus reduced, resulting in a lower deadline miss rate.
- **OD/RCS.** The system can make use of this information to provide a transaction with versions that require the minimum number of OD-updates or OD-recoms. In general, an OD-recom, which involves multiple reads (of base items) and single write (of views), requires many more system resources than an OD-update that only needs to write to a base item. If the system determines that at least one item of the transaction readset has to be materialized anyway, it should try to minimize the number of OD-recoms at the expense of more OD-updates. By reducing the number of

OD-updates and OD-recoms, the *OD Wait* problem is alleviated and the system performance can be improved.

The schemes described above can only be useful in reducing the number of missed deadlines if the overhead for determining the versions of a transaction readset is small. We may need to design special data structures and efficient algorithms to support the new protocols.

## 8 Conclusions

Most of the previous studies on temporal consistency for RTDBs concentrate on continuous data objects. Few studies consider the temporal correctness requirements of discrete data objects. In this paper, we defined temporal consistency of discrete objects 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 a 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 multiversion database. We defined a relative consistent system as one that allows a transaction to read relatively consistent data items. The only requirement is that those items are not older than the transaction's start time by a certain threshold value ($\Delta$). We argued that a relative consistent system has a higher potential of meeting transaction deadlines.

We carried out an extensive simulation study on the performance of the three scheduling policies: URT, OD, and OD-H, 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. From the performance study, we showed that OD-H when applied to an RCS results in the smallest miss rate.

## Acknowledgments

The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CityU 1061/98E).

## References

[1] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *Proc. 14th VLDB Conf.,* Aug. 1988.
[2] B. Adelberg, H. Garcia-Molina, and B. Kao, "Applying Update Streams in a Soft Real-Time Database System," *Proc. 1995 ACM SIGMOD,* pp. 245-256, 1995.
[3] B. Adelberg, H. Garcia-Molina, and B. Kao, "Database Support for Efficiently Maintaining Derived Data," *Advances in Database Technology—EDBT 1996,* pp. 223-240, 1996.
[4] Q.N. Ahmed and S.V. Vrbsky, "Triggered Updates for Temporal Consistency in Real-Time Databases," *Int'l J. Time-Critical Computing Systems,* vol. 19, no. 3, pp. 209-243, Nov. 2000.
[5] M. Carey, R. Jauhari, and M. Livny, "On Transaction Boundaries in Active Databases: A Performance Perspective," *IEEE Trans. Knowledge and Data Eng.,* vol. 3, no. 3, pp. 320-336, 1991.
[6] M. Cochinwala and J. Bradley, "A Multidatabase System for Tracking and Retrieval of Financial Data," *Proc. 20th VLDB Conf.,* pp. 714-721, 1994.
[7] L.B.C. DiPippo and V.F. Wolfe, "Object-Based Semantic Real-Time Concurrency Control," *Proc. IEEE 14th Real-Time Systems Symp.,* pp. 87-96, Dec. 1993.
[8] B. Kao, K.Y. Lam, B. Adelberg, R. Cheng, and T. Lee, "Updates and View Maintenance in Soft Real-Time Database Systems," *Proc. 1999 ACM CIKM,* pp. 300-307, 1999.
[9] Y.K. Kim and S.H. Son, "Predictability and Consistency in Real-Time Database Systems," *Advances in Real-Time Systems,* Prentice-Hall, 1995.
[10] C.M. Krishna and K.G. Shin, "Real-Time Databases," *Real-Time Systems,* chapter 5, pp. 185-222, McGraw-Hill, 1997.
[11] T.W. Kuo and A.K. Mok, "SSP: A Semantics-Based Protocol for Real-Time Data Access," *Proc. IEEE Real-Time Systems Symp.,* pp. 76-86, 1993.
[12] Mesquite Software, Inc., "CSIM 18 User Guide,"http://www.mesquite.com, 2002.
[13] G. Ozsoyoglu and R. Snodgrass, "Temporal and Real-Time Databases: A Survey," *IEEE Trans. Knowledge and Data Eng.,* vol. 7, no. 4, pp. 513-532, 1995.
[14] B. Purimetla, R. M. Sivasankaran, K. Ramamritham, and J.A. Stankovic, "Real-Time Databases: Issues and Applications," *Advances in Real-Time Systems,* chapter 20, pp. 487-507, Prentice-Hall, 1995.
[15] K. Ramamritham, "Real-Time Databases," *Distributed and Parallel Databases,* vol. 1, no. 2, pp. 199-226, 1993.
[16] A. Segev and A. Shoshani, "Logical Modeling of Temporal Data," *Proc. ACM SIGMOD Ann. Conf. Management of Data,* pp. 454-466, 1987.
[17] X. Song and J.W.S. Liu, "Maintaining Temporal Consistency: Pessimistic vs. Optimistic Concurrency Control," *IEEE Trans. Knowledge and Data Eng.,* pp. 787-796, Oct. 1995.
[18] M. Xiong, R. Sivasankaran, J. A. Stankovic, K. Ramamritham, and D. Towsley, "Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics," *Proc. 1996 Real-Time Systems Symp.,* Dec. 1996.
[19] M. Xiong and K. Ramamritham, "Deriving Deadlines and Periods for Update Transactions in Real-Time Databases," *Proc. 20th IEEE Real-Time Systems Symp. (RTSS '99),* Dec. 1999.
[20] M. Xiong, R. Sivasankaran, J. A. Stankovic, K. Ramamritham, and D. Towsley, "Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics," *Proc. 17th IEEE Real-Time Systems Symp.,* Dec. 1996.
[21] P.S. Yu, K.L. Wu, K.J. Lin, and S.H. Son, "On Real-Time Databases: Concurrency Control and Scheduling," *Proc. IEEE,* vol. 82, no. 1, pp. 140-157, 1994.

**Ben Kao** received the BSc degree in computer science from the University of Hong Kong in 1989, the PhD degree in computer science from Princeton University in 1995. He is an assistant professor in the Department of Computer Science and Information Systems at the University of Hong Kong. From 1989-1991, he was a teaching and research assistant at Princeton University. From 1992-1995, he was a research fellow at Stanford University. His research interests include database management systems, data mining, real-time systems, and information retrieval systems. Dr. Kao has published more than 50 technical research papers in various international journals, conference proceedings, and books. He has also served as a program committee member in a number of international computer conferences. He is the principal investigator of a number of government-funded research projects. He is currently also an adjunct fellow at the E-Business Technology Institute (ETI), a research and development institute jointly funded by IBM China/Hong Kong and the University of Hong Kong.

**Kam-Yiu Lam** received the BSc (Hons) degree in computer studies with distinction and the PhD degree from the City University of Hong Kong in 1990 and 1994, respectively. He is currently an associate professor in the Department of Computer Science, City University of Hong Kong. He has served as a paper reviewer for conferences and journals on real-time systems and databases, including SIGMOD, VLDB Conference, RTSS, RTAS, *IEEE Transactions on Knowledge and Data Engineering*, *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, and *Real-Time Systems Journal*. His current research interests are in location-dependent services, continuous query processing, mobile real-time information management, real-time mobile computing, sensor database, and real-time database systems. Dr. Lam is a member of the IEEE, ACM, and ACM SIGMOD.

**Brad Adelberg** Biography and photograph unavailable.

**Reynold Cheng** received the BEng degree in computer engineering and the MPhil degree in computer science from the University of Hong Kong, in 1998 and 2000, respectively. He is currently a PhD student at Purdue University. His research interests are in the area of transaction processing and concurrency control in real-time database systems. He is a student member of the IEEE and IEEE Computer Society.

**Tony Lee** received the BS degree in computer studies from the University of Hong Kong in 1990. He is currently an MPhil student in the City University of Hong Kong. His research interests are in the areas of transaction processing and concurrency control in real-time system and time critical applications.