# A QoS-Sensitive Approach for Timeliness and Freshness Guarantees in Real-Time Databases *

Kyoung-Don Kang Sang H. Son John A. Stankovic Tarek F. Abdelzaher
Department of Computer Science
University of Virginia
{kk7v, son, stankovic, zaher}@cs.virginia.edu

## Abstract

*The demand for real-time database services has been increasing recently. Examples include sensor data fusion, decision support, web information services, and online trading. In these applications, it is desirable to execute transactions within their deadlines using temporally consistent data. Due to the high service demand, real-time databases can be overloaded. As a result, many transactions may miss their deadlines, or data temporal consistency constraints can be violated. To address these problems, we present a QoS management scheme to support guarantees on deadline miss ratio and data freshness (temporal consistency) even in the presence of unpredictable workloads and data access patterns. Using our approach, admitted user transactions can be processed in time using fresh data. A simulation study shows that our QoS-sensitive approach can achieve a significant performance improvement, in terms of deadline miss ratio and data freshness, compared to several baseline approaches. Furthermore, our approach shows a comparable performance to the theoretical oracle that is privileged by a complete future knowledge of data accesses.*

## 1 Introduction

The demand for real-time information services has been increasing recently. Many real-time applications are becoming very sophisticated in their data needs. Applications such as agile manufacturing have data that span the spectrum from low level control data, typically acquired from sensors, to high level management and business data. Other examples include sensor data fusion, decision support, web information services, and e-commerce.

In real-time databases, it is essential both to minimize the deadline miss ratio of transactions and provide the temporal consistency between the data in databases and continuously changing external environments, e.g., current temperature or stock prices [2, 6]. Due to the high service demand and stringent timing/data freshness (temporal consistency) constraints, real-time databases can be overload. As a result, many deadline misses or data freshness violations can occur. To address this problem, we present a QoS-sensitive approach, called **QMF** (a QoS-sensitive approach for Miss ratio and Freshness guarantees). By using QMF, certain guarantees on miss ratio and freshness can be provided for admitted transactions even in the presence of unpredictable workloads and data access patterns. Our scheme is important to applications which require timely transaction execution using fresh data. For example, transactions in online stock trading and agile manufacturing should be processed within their deadlines using fresh data before the current market conditions or process states change. In these applications, a deadline miss or a stale (temporally inconsistent) data access may lead to a loss of profit or a reduction in the product quality.

Unfortunately, timeliness and data freshness requirements can conflict with each other. The deadline miss ratio of user transactions can be decreased by giving a higher priority to user requests. In contrast, better freshness can be achieved by favoring data updates [2]. Effective balancing between the user transaction and update workloads is the key to provide a satisfactory service, which can meet both the specified deadline miss ratio and data freshness constraints. For this purpose, we propose a *dynamic balancing* scheme to balance the user transaction and update workloads in a flexible manner. Main challenges include the unpredictability of workloads/data access patterns and the potential conflict between the timeliness and freshness requirements. To effectively handle the potential conflicts and varying access patterns, we introduce a cost-benefit model for updates. Using the model, an adaptive update policy is developed to perform the dynamic balancing.

We apply a feedback control scheduling policy [5] to

*Supported, in part, by NSF grants EIA-9900895 and CCR-0098269, and contract IJRP-9803-6 from the Ministry of Information and Communication of Korea.

provide robustness against unpredictable workloads. In a feedback control system, target performance can be achieved by dynamically adjusting the system behavior based on the performance error measured in the feedback loop. In our approach, update workloads are dynamically adjusted by the adaptive update policy as requested by the feedback controller. By leveraging the robustness of feedback control and the flexibility of the adaptive update policy, we can provide the guaranteed real-time database services in terms of the deadline miss ratio and data freshness perceived by users.

To show the effectiveness of QMF, we compare its performance to that of several baselines including a theoretical oracle. Given a complete future knowledge of data accesses, the oracle can update each data only if necessary, i.e., the corresponding data *will* be accessed before the next update. Otherwise, the update will not be scheduled to minimize the deadline miss ratio of user transactions. In this way, the oracle can provide a perfect data freshness and minimal deadline miss ratio.

## 2 Data and Transaction Model

In this section, an overview of our real-time database model is discussed.

### 2.1 Real-Time Database Model and Data Types

We consider a firm real-time database system, in which tardy transactions – transactions that have missed their deadlines – add no value to the system, and therefore, are aborted. In this paper, we consider a main memory database model. Because of their relatively high performance and decreasing main memory cost, main memory databases have been increasingly applied to real-time data management, e.g., online auction, stock trading, and voice/data networking [2, 7]. In our main memory database model, the CPU is the main system resource for consideration.

Data can be classified into two classes: non-temporal and temporal. A non-temporal data object does not become outdated due to the passage of time, e.g., PIN numbers. In contrast, a temporal data object may change continuously to reflect the real world state, e.g., current temperature or stock prices. Each temporal data object has a timestamp indicating the latest observation of the real-world state.

### 2.2 Temporal Data Management Issues

Temporal consistency was defined using validity intervals in a real-time database to address the consistency issue between the real world state and the reflected value in the database. The absolute validity interval is the length of the time a temporal data object remains fresh (temporally consistent). A temporal data object $X$ is considered fresh, if $(current\ time - timestamp(X) \leq avi(X))$ where $avi(X)$ is the absolute validity interval of $X$ [6].

Temporal data can be further classified into *base data* and *derived data*. Base data objects import the view of the outside environment. A derived data object can be derived from possibly multiple base/derived data. For example, a composite index can be derived from various stock prices. A base data item is directly associated with an absolute validity interval. A derived data object is associated with absolute and relative validity intervals [6]. We focus on the base data freshness issues in this paper. Derived data management is reserved for future work.

In this paper, we consider periodic temporal data updates, which are common in real-time applications. Our approach can also be extended to handle aperiodic updates. For more details, refer to [3].

### 2.3 Transaction Types

Currently, our real-time database model includes two classes of transactions:

• *Update Transactions*: In a real-time database, base data objects should be updated periodically to reflect the current status of the real-world environment, e.g., sensor data updates. Update transactions are write-only transactions specially designed for this purpose. For brevity, we call them *updates*.

• *User Transactions*: A user-level transaction can be submitted to the real-time database with a deadline. User transactions arrive aperiodically. User transactions are not allowed to write any temporal data object, but they can read/write non-temporal data. For example, user transactions are not allowed to change the currently measured temperature/stock prices, but they can change the default temperature metric/PIN numbers. User transactions are called *transactions* for short in the remainder of this paper.

### 2.4 Admission Control, Scheduling, and Concurrency Control

For user transactions, admission control is applied to reduce the chance of potential overload, which can lead to undesirable consequences, e.g., the reduced product quality in agile manufacturing. A newly arriving user transaction can be admitted to the system if the requested CPU utilization is currently available. The current utilization can be examined by aggregating the utilization estimates of the previously admitted transactions. Note that a deadline miss (or a temporal consistency violation) can not be completely prevented by admission control. For example, the CPU utilization estimates may include estimation errors. As a result, too many transactions can be admitted leading to many deadline misses. Further, database applications usually include unpredictable aborts/restarts due to data/resource conflicts, but it is very hard to predict aborts/restarts at the admission control stage, if at all possible. For this reason, we also employ the feedback control and adaptive update policy to support guarantees on miss ratio and freshness.

In our model, each user transaction has a deadline. The deadline of an update is set to the update period. A tardy transaction or an outdated update is aborted upon its deadline miss. For scheduling, we apply earliest deadline first (EDF) algorithm combined with our adaptive update scheduling policy. Scheduling issues will be revisited in Section 4.

For concurrency control, we employ two phase locking high priority (2PL-HP) [1], in which a low priority transaction is aborted and restarted upon a conflict. 2PL-HP is selected, since it is free of priority inversion.

# 3 Metrics and QoS Specification

In our approach, a database administrator (DBA) can explicitly specify the required database QoS in terms of miss ratio and freshness as follows.

## 3.1 Main Performance Metrics

• *User Transaction Deadline Miss Ratio* ($MR$): In a QoS specification, a DBA can specify the target deadline miss ratio that can be tolerated for a specific real-time application.

• *Data Freshness*: We categorize data freshness into "database freshness" and "perceived freshness". Database freshness, also called QoD (Quality of Data) in this paper, is the ratio of fresh data to the entire temporal data in a database. In contrast, perceived freshness ($PF$) is defined for the data accessed by timely transactions − transactions that have met their deadlines − as follows. Let $N_{accessed}$ represent the number of data accessed by timely transactions. Let us call the number of fresh data accessed by timely transactions $N_{fresh}$: $PF = 100 \times \frac{N_{fresh}}{N_{accessed}}$ (%).

To measure the current perceived freshness, we exclusively consider timely transactions. This is because tardy transactions add no value to our firm real-time database model as described in Section 2. Note that only the perceived freshness can be specified in a QoS specification. Our QoS-sensitive approach provides the perceived freshness guarantee while managing the database freshness internally (hidden to the users).

## 3.2 Transient Performance Metrics

Long-term performance metrics such as average deadline miss ratio are not sufficient for performance specification of dynamic systems in which the workloads/system performance can be widely time-varying. For this reason, transient performance metrics are adopted from control theory for real-time system performance specification [5]:

• *Overshoot* ($M_p$) is the worst-case system performance in the transient system state, e.g., the highest miss ratio in the transient state.

• *Settling time* ($t_s$) is the time for the transient overshoot to decay and reach the steady state performance.

For example, a DBA can specify that $MR \leq 5\%$ and $PF \geq 98\%$ in a QoS specification. A DBA can also require $M_p$ and $t_s$ to be below 6.35% and 45sec, respectively.

# 4 QoS Management

In this section, we introduce a cost-benefit model for temporal data updates. Using this model, an adaptive update policy is derived. Update scheduling issues are discussed and our QoS management scheme for miss ratio and freshness guarantees is described. Generally the update policy is not dynamically adaptable in databases. In contrast, we dynamically switch the update policy between an aggressive and a lazy one for certain temporal data objects based on the current workload and system behavior. The immediate and on-demand update policies are selected for aggressive and lazy update policies, respectively.

## 4.1 Cost-Benefit Model for Updates

In a real-time database, the temporal data update workload might be high. For example, in the NYSE trace the update stream can reach up to 696 updates/sec [4]. If updates receive a higher priority, there may not be enough time left to finish transactions in time. In contrast, if the transactions are scheduled in a preferred manner, they may have to read stale data [2]. To balance the update and transaction workload efficiently, we introduce a cost-benefit model for temporal data updates as follows. The cost is defined as the update frequency of a temporal data object. Intuitively, the more frequent is the update, the higher is the cost. We assume that the frequency of periodic updates is known to the database system. To consider the benefit, access frequency is measured for each data object. If a data object is accessed frequently, e.g., a popular stock price, an update of the object can produce a relatively high benefit. To quantify the cost-benefit relationship, we define Access Update Ratio (AUR) for a data object $O_i$ as follows:

$$AUR[i] = \frac{\textit{Access Frequency}[i]}{\textit{Update Frequency}[i]} \qquad (1)$$

Unfortunately, the access frequency may have a large deviation from one sampling period to another. To smooth the potentially large deviation, we take a moving average of the access frequency ($AF$) in the $k_{th}$ sampling period:

$$SAF_k[i] = a \times SAF_{k-1}[i] + (1-a) \times AF_k[i] \quad (2)$$

where $0 \leq a \leq 1$. As the value of $a$ gets closer to 0, only the recent access frequencies are considered to compute the moving average. In contrast, the wider horizon will be considered as $a$ gets closer to 1.

Since the update frequency $UPDATES[i]$ in a sampling period is known for periodic updates of $O_i$, we can compute Access Update Ratio for $O_i$:

$$AUR[i] = \frac{SAF_k[i]}{UPDATES[i]} \quad (3)$$

If $AUR \geq 1$ for a data object, the benefit of the corresponding data update is worth the cost, since it is accessed at least as frequently as it is updated. Otherwise, it is not cost-effective. For simplicity, let us call a data *hot* if its AUR $\geq 1$. Otherwise, we call the data *cold*.

Note that the notion of AUR does not depend on a specific access pattern or popularity model. It can be derived simply from the known update frequency and monitored access frequency. Therefore, it greatly simplifies our cost-benefit model and makes the model robust against potential unpredictability in data access patterns.

## 4.2 Adaptive Update Policy

From the cost-benefit model, we observe that it is reasonable to update hot data immediately. If a hot data is out-of-date when accessed, a multitude of transactions may miss their deadlines waiting for the update. Alternatively, it may not be necessary to immediately update cold data when overloaded. Only a few transactions may miss their deadlines waiting for the update. For cold data, we can save the CPU utilization by applying the lazy (i.e., on-demand) update policy under overload. In fact, cold data could always be updated on demand regardless of the current system load. However, that approach may increase the response time of the accessing transaction.

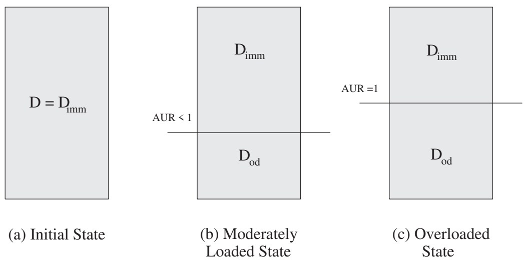


**Figure 1. Update Policy Adaptations**

Consider Figure 1 for example. In the figure, $D$ represents the set of the all temporal base data in the database. $D_{imm}$ is the set of data updated immediately, and $D_{od}$ stands for the set of data updated on demand. Initially, all data are updated immediately. As the load increases, a larger fraction of cold data objects are updated on demand. This is called *QoD degradation*, since lazy updates reduce the QoD (i.e., database freshness).

In Figure 1(c), the update policy degradation stops once it reaches a certain *degradation bound*, namely $AUR = 1$. Further degradation past the degradation bound is meaningless. If a hot data object whose $AUR \geq 1$ is updated on demand, the number of updates may not be reduced, but many transaction may have to miss their deadlines waiting for the on-demand update. Hence, it is necessary to update hot data immediately.

Upon a violation of the perceived freshness specification, the update policy is switched back to the immediate policy for certain cold data objects. This is called *QoD upgrade*.

In our approach, an immediate update receives a higher priority than user transactions and on-demand updates. Conceptually, there are separate scheduling queues for immediate updates and for user transactions/on-demand updates, respectively. In each queue, transactions are scheduled by EDF scheduling algorithm. For transactions belonging to the low priority queue, feedback control is applied to support both the average and transient miss ratio. In this way, a strong freshness guarantee can be supported for hot data that are updated immediately. The freshness of cold data can be relatively low compared to hot data, if the on-demand update policy is applied. However, the update policy can be switched back to the immediate policy to support the specified freshness as described before.

## 4.3 Feedback Control

We apply a feedback control real-time scheduling policy, called FC-UM [5], to control the miss ratio without underutilizing the CPU in the presence of unpredictable workloads. As shown in Figure 2, FC-UM employs two control loops, one for the miss ratio and one for the CPU utilization management to avoid the control saturation problem. A utilization controller may saturate at utilization 100%. In contrast, a miss ratio controller can saturate when the real-time system is underutilized (0 miss ratio as a result). Each control loop generates a control signal to achieve the target utilization or miss ratio based on the current performance error, which is the difference between the target miss ratio (utilization) and the currently measured miss ratio (utilization). Each controller computes the control signal, called requested CPU utilization adjustment $\Delta U$, to achieve the target miss ratio (or target utilization).

We have selected FC-UM, since it can meet our basic

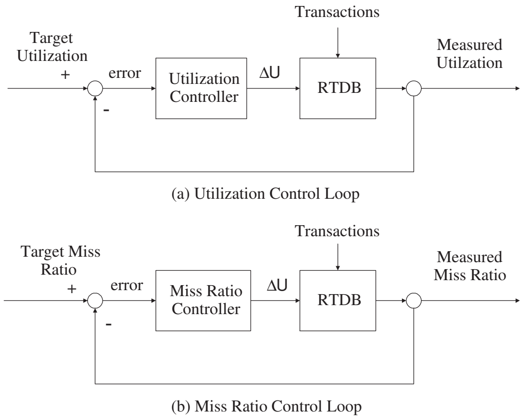


**Figure 2. Miss Ratio/Utilization Controllers**

requirements for real-time scheduling: providing a certain miss ratio guarantee while not underutilizing the CPU given a mix of periodic and aperiodic workloads. Another advantage of FC-UM is that it works well even without a precise workload model. We adapted and tuned the feedback controllers, but we do not include the details in this paper due to space limitations. Interested readers are referred to [3].

In our model, data freshness is managed by an actuator (i.e., QoD manager which performs QoD adaptations) in the real-time database. We do not consider designing a separate feedback controller for freshness management, since timeliness and freshness can pose conflicting requirements leading to a potentially unstable feedback control system.

## 4.4 Feedback Control and QoD Management

Our QoS-sensitive approach for miss ratio and freshness guarantee (QMF) is described in Figure 3.

When the system is in the steady state, i.e., the measured miss ratio is below the target and the perceived freshness specification is satisfied, no QoD adaptation is necessary.

When $\Delta U$ is negative, the QoD can be degraded to improve the miss ratio if two conditions are met, i.e., the current perceived freshness is above the target and the degradation bound is not reached yet.

When the perceived freshness is violated, the QoD will be upgraded as long as $\Delta U > 0$ and a certain upgrade bound is not exceeded. The corresponding QoD upgrade/degradation stops as soon as whichever condition is not satisfied first. These conditions for a QoD degradation/upgrade are required to prevent possible oscillations between many deadline misses and data freshness violations.

Note that the QoD management may not be able to enforce $\Delta U$ entirely. For example, the current freshness is below the specified target or the degradation bound is reached.

1. Collect access statistics and compute AUR.
2. Monitor the miss ratio (MR), CPU utilization, and perceived freshness (PF).
3. At each sampling period, compute the miss ratio and utilization control signals. Set $\Delta U$ to the minimum of the two control signals for a smooth transition from one system state to another. Based on $\Delta U$, perform one of the following alternative actions.
4. If $MR \leq MR_t$ (target miss ratio), i.e., $\Delta U \geq 0$, and $PF \geq PF_t$ (target perceived freshness), admit more transactions to prevent a potential underutilization.
5. If $MR > MR_t$, i.e., $\Delta U < 0$, and $PF \geq PF_t$ , degrade the QoD. Increase $\Delta U$ by the utilization saved from a QoD degradation. Repeat until $\Delta U \geq 0$ or the degradation bound is reached. Inform the admission controller of the new $\Delta U$.
6. If $PF < PF_t$ and $\Delta U > 0$, upgrade the QoD. Subtract the required utilization for an upgrade from $\Delta U$. Repeat until $\Delta U \leq 0$ or a certain upgrade bound is reached. Inform the admission controller of the new $\Delta U$.
7. If $MR > MR_t$ and $PF < PF_t$, do not admit any incoming transaction until $\Delta U$ becomes positive after some of the currently running transactions terminate.

**Figure 3. QoS Management Scheme for Miss Ratio and Freshness Guarantees**

Hence, the QoD degradation is not possible. To enforce the remaining control signal after potential QoD adaptations, we apply admission control to newly incoming user transactions.

## 4.5 Miss Ratio Adjustment

For a QoD degradation, the update policy is switched to the on-demand policy to reduce the number of updates for a certain temporal data object $O_i$ currently in $D_{imm}$ with the lowest $AUR$. An important question is how to estimate the CPU utilization saved from the degradation. The main interest is the difference between the required CPU utilization for purely immediate updates and for adaptive updates. The number of saved updates due to the degradation for $O_i$ is approximately:

$$N = UPDATES[i] - SAF[i] \tag{4}$$

Given the average CPU utilization per single update transaction, $\sigma U$, the saved CPU utilization from the update policy degradation for $O_i$ is approximately:

$$\delta U = N \times \sigma U \tag{5}$$

Average per update utilization, $\sigma U$, can be either pre-profiled before the database service initiation, or measured at run time. Either approach may not introduce a considerable error, since each update transaction is known a priori and fixed in our real-time database model.

After the update policy degradation for a single data object, the new CPU utilization adjustment is:

$$\Delta U = \Delta U_{old} + \delta U \tag{6}$$

The degradation continues for the next data object in $D_{imm}$ with the least $AUR$ until $\Delta U \geq 0$ or the degradation bound is reached as shown in Figure 1(c).

### 4.6 Freshness Adjustment

To ensure that timely transactions access fresh data, the update policy is switched back to the immediate policy for certain data objects upon a violation of the target perceived freshness. A key issue in QoD upgrades is to avoid a potential miss ratio overshoot in the next sampling period, while improving the perceived freshness as needed at the same time. For this purpose, we define the perceived freshness error and derive the upgrade bound.

The perceived freshness error is defined as follows. Given a target perceived freshness $F_{target}$, the current perceived freshness $F_{current}$ can be measured in a sampling period. The performance error in terms of perceived freshness in a sampling period is:

$$F_{error} = \begin{cases} 0 & \text{if} \quad F_{current} \geq F_{target}; \\ F_{target} - F_{current} & \text{if} \quad F_{current} < F_{target}. \end{cases} \tag{7}$$

Given a non-zero $F_{error}$, it is clear that on-demand updates have failed to provide the target perceived freshness in the current sampling period. Therefore, some of these data should be updated immediately in the next sampling period. The freshness can be improved approximately in proportion to the number of data moved from $D_{od}$ to $D_{imm}$. We can improve the perceived freshness by moving from $D_{od}$ to $D_{imm}$ a certain number of data with the highest AUR in $D_{od}$.

An *upgrade bound* $K$ is derived in terms of the number of data for update policy upgrades. Let $D'_{imm}$ represent the set of data to be updated immediately in the next sampling period after the upgrades. The upgrade bound is determined in proportion to the current freshness error and the cardinality of the set $D_{od}$:

$$K = |D'_{imm}| - |D_{imm}| \approx F_{error} \times |D_{od}| \tag{8}$$

By moving *one* data object $O_i$ from $D_{od}$ to $D_{imm}$, the number of updates and the corresponding CPU utilization may increase. The potential increase in the number of updates can be approximated from Eq. 4. We can estimate the required extra CPU utilization for the increase by Eq. 5. The QoD upgrade is repeated until the upgrade bound in Eq. 8 is reached, or the available CPU utilization becomes not enough, i.e., $\Delta U \leq 0$.

In our approach, the QoD is dynamically adjusted based on the current miss ratio or perceived freshness. Due to the approximation, unpredictable workloads and access patterns, our QoD adaptation may not be precise. However, the target performance can be achieved by continuously adjusting the QoD based on the performance error measured in the feedback loop. By balancing update and transaction workload efficiently, a target deadline miss ratio and perceived freshness can be achieved at the same time.

## 5 Performance Evaluation

In this section, we analyze the performance of the QoS-sensitive approach in terms of miss ratio and freshness by a simulation study.

### 5.1 Experimental Goals

The main objective of our performance evaluation is to determine whether or not our approach can support the specified miss ratio and freshness even in the presence of unpredictable workloads and access patterns. We consider a QoS specification which limits the average miss ratio below 5%. A miss ratio overshoot shall be lower than 6.35% and the overshoot shall decay within 45sec. At the same time, the perceived freshness of 98% is required. To avoid underutilization, we aim the utilization to be above 80%. Note that our main purpose is providing guarantees on miss ratio and freshness avoiding severe underutilization. Utilization can vary by applying different update policies tested in our experiments. An update can be either scheduled or dropped depending on different update policies and the current system behavior.

### 5.2 Simulation Model

We have developed a real-time database simulator to evaluate the performance of our QoS-sensitive approach. A block diagram of the simulator architecture is shown in Figure 4. It consists of several main components:

*Sources* generate user transactions to be submitted to the system. The transaction inter-arrival time is exponentially distributed. Each *Update Stream* periodically submits an update for a certain temporal data object.

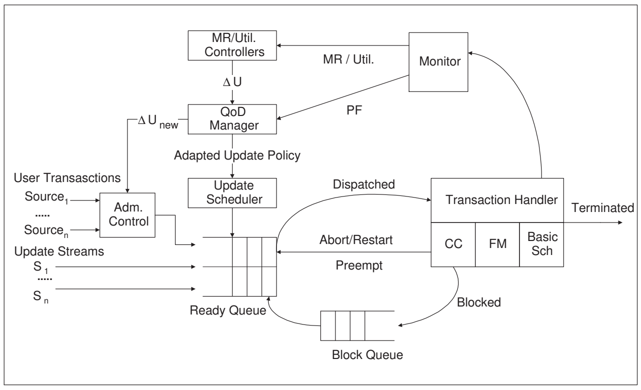


**Figure 4. Simulator Architecture**

*Admission control* is applied to user transactions. It can be turned on/off for performance evaluation. If turned off, all incoming transactions will be simply admitted.

*Transaction handler* consists of a concurrency controller (CC), a freshness manager (FM) and a basic scheduler. As described in Section 4, immediate updates are scheduled in the high priority ready queue while user transactions and on-demand updates are scheduled in the low priority queue. In each queue, transactions are scheduled in EDF manner. A transaction can be aborted and restarted by CC. It can also be preempted by a higher priority transaction. Freshness manager (FM) checks the freshness of temporal data before the initiation of a user transaction. FM blocks the corresponding transaction if an accessing data is currently stale. The blocked transaction(s) will be transferred from the block queue to the ready queue as soon as the corresponding update commits.

*Monitor* periodically measures miss ratio, utilization, and perceived freshness and reports the statistics to the feedback controllers and QoD manager. *MR/Utilization Controllers* compute the miss ratio and utilization control signals based on the current performance error. Sampling period for feedback control is set to 5sec. This sampling period is selected to collect enough access statistics needed for potential QoD adaptations. Without reliable statistics, QoD management could be jittery.

*QoD Manager* adapts the update policy, if necessary. It informs the admission controller of $\Delta U_{new}$ after potential QoD adaptations. It can be turned on/off for performance evaluation. If turned off, the update policy is not adapted based on the current system behavior.

*Update scheduler* decides whether or not to schedule an incoming update depending on the selected update policy. Immediate updates will always be scheduled. On-demand updates will be scheduled, if any transaction is blocked for the update to access the fresh data.

### 5.3 Simulation Set-Up

**Set-Up for Data and Updates**: The modeled database includes 1000 temporal data objects. We assume user transactions access only the temporal data to focus on the trade-off issues between the data freshness and timeliness. Non-temporal data updates are assumed infrequent and do not affect the performance considerably. Non-temporal data reads/writes can be easily included, if necessary.

An update stream, $S_i$, is dedicated to a certain temporal data object $i$ in the modeled database. $S_i$ is defined by $\{$*update period ($P_i$), estimated execution time ($EET_i$), actual execution time*$\}$. Update period ($P_i$) and $EET_i$ are uniformly distributed in the range (100ms, 50s) and (1ms, 8ms), respectively. Upon a periodic generation of an update, $S_i$ gives the update an actual execution time by applying the normal distribution, $Normal(EET_i, \sqrt{EET_i})$, to introduce execution time variances. The simulator is only aware of the estimated execution time, but not actual execution time. For an update, we set *Deadline = Next Update Period*. Aggregate periodic load is manipulated to require approximately 50% of the total CPU utilization.

**Set-Up for User Transactions**: For user transactions, accessing data sets and execution times may vary. Neither the accessing data set nor actual execution time is known to the simulator. To model the user transaction workload, $Source_i$ generates a group of transactions with an average number of data accesses and average execution time. By generating multiple sources, we can derive transaction groups with different average execution time and average number of data accesses in a statistical manner. Each $Source_i$ has an estimated execution time ($EET_i$), which is uniformly distributed in the range (20ms, 80ms), and an *average* execution time ($AET_i$). $AET_i = (1 + EstErr) \times EET_i$, in which $EstErr$ is used to introduce the execution time estimation error.

Upon the generation of a transaction, $Source_i$ associates an *actual* execution time to the transaction, which is $Normal(AET_i, \sqrt{AET_i})$. This is to introduce the variance of the execution time among transactions in a transaction group. We set $Deadline = Arrival\ Time + Execution\ Time \times Slack\ Factor$ for a user transaction. A slack factor is uniformly distributed in a range (20, 40).

The number of data accesses ($N_{DATA_i}$) for $Source_i$ is set to $0.5 \times EET_i$. As a result, longer transactions access more data in general. Upon a transaction generation, $Source_i$ associates the actual number of data accesses to the transaction, which is $Normal(N_{DATA_i}, \sqrt{N_{DATA_i}})$, to introduce the variance.

### 5.4 Baselines

For completeness of performance analysis, we compare the relative miss ratio and perceived freshness received by *QMF* with the several baselines as follows:

- `Open-IMU`: Temporal data are always updated immediately and all incoming transactions are admitted. The closed loop scheduling by feedback control are turned off.

- `Open-IMU-AC`: This is a variant of Open-IMU, for which admission control is applied.

- `Open-ODU`: This is similar to Open-IMU, but the on-demand policy is applied for all updates.

- `Open-ODU-AC`: This is a variant of Open-ODU, for which admission control is turned on.

- `Theoretical Oracle`: For the purpose of performance evaluation, we assume the existence of a *theoretical oracle* that has a complete future knowledge of data accesses. The clairvoyant oracle can schedule updates only when necessary. That is, an update is scheduled only if some transaction *will* access the corresponding data before the next update. In this way, the update workload can be minimized providing the perfect perceived freshness at the same time.

In this paper, we take a stepwise approach for performance evaluation. We first compare the performance of QMF to the open-loop approaches for increasing loads, i.e., 70%, 100%, 150%, and 200%. For this set of experiments, we set $EstErr = 0$, i.e., no execution time estimation error. From this, we selected the best performing open-loop baselines, i.e., Open-IMU-AC and Open-ODU-AC, for further performance comparisons with QMF while increasing the execution time estimation error. Finally, we compare the performance of our approach under overload with the theoretical oracle for different access patterns. Due to space limitations, we do not include the experimental results for the first set of experiments, which shows the performance effects of increasing loads when $EstErr = 0$. For more details, readers are referred to [3]. For all performed experiments, we observed that QMF supports the specified miss ratio and freshness, whereas other baseline approaches fail to support the specified miss ratio and/or freshness in the presence of unpredictable workloads and access patterns.

In our experiments, one simulation run lasts for 10 minutes of simulated time. For all performance data, we have taken the average of 10 simulation runs and derived 90% confidence intervals except that of the theoretical oracle's. In the figures showing the performance evaluation results, confidence intervals are plotted as vertical bars. (For some performance data, the vertical bars may not be noticeable due to the small confidence intervals.)

### 5.5 Effects of Increasing Execution Time Estimation Error

**Average Performance**: In general, precise execution time estimates are not available in complex computational systems such as real-time databases. For this reason, we compare the performance of various approaches for increasing estimation errors. We show that our approach provides performance guarantees both in terms of (long-term) average and transient performance metrics while other approaches fail in the presence of execution time estimation errors. For this set of experiments, 200% load is given. A uniform data access pattern is assumed. The average miss ratio, perceived freshness, and utilization are measured for different $EstErr$ values ranging from 0 to 1 increased by 0.25 as shown in Figure 5, 6 and 7, respectively. In this way, we can observe the effects of the increasing execution time estimation error to the performance when overloaded.

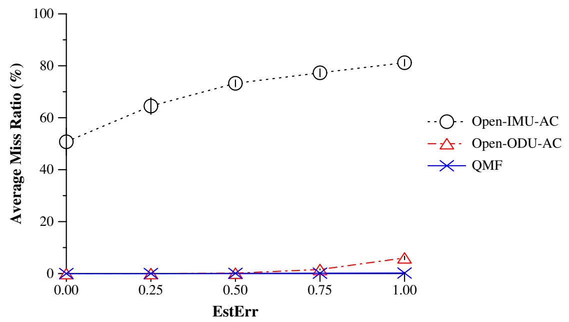


**Figure 5. Average Miss Ratio (Load = 200% , Uniform Access Pattern)**

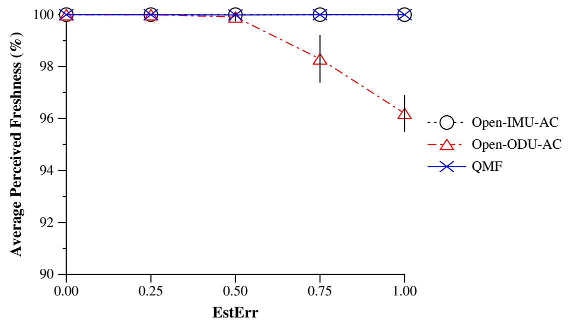


**Figure 6. Average Perceived Freshness (Load = 200% , Uniform Access Pattern)**

As shown in Figure 5, miss ratio increases sharply for Open-IMU-AC as $EstErr$ increases exceeding 80% when $EstErr = 1$. This is mainly due to the increased error in

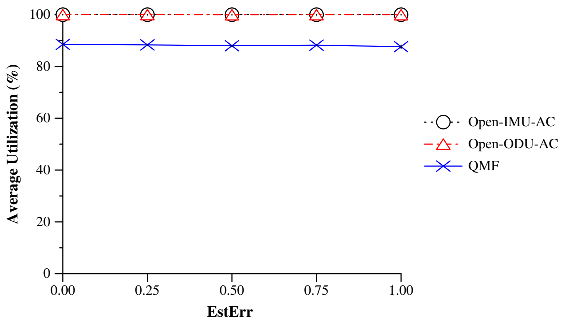


**Figure 7. Average Utilization (Load = 200% , Uniform Access Pattern)**

admission control and the corresponding overload. Open-ODU-AC shows $6.03 \pm 0.67\%$ miss ratio violating the target miss ratio 5% for $EstErr = 1$. In contrast, QMF shows near 0 deadline miss ratio.

In Figure 6, the average freshness is 100% for Open-IMU-AC and QMF for all $EstErr$ values. However, Open-ODU-AC violates the target perceived freshness. When $EstErr = 1$, perceived freshness drops to $96.19 \pm 0.69\%$ violating the target freshness of 98%. Observe that only our approach, among the tested algorithms, can meet both miss ratio and freshness requirements when only a rough estimation is available.

Average utilization is plotted in Figure 7. For other approaches, the CPU is overloaded leading to the performance violations in terms of miss ratio and/or freshness. In contrast, QMF can avoid potential overload even in the presence of large execution time estimation errors.

One can argue that the average performance improvement of our approach is marginal compared to Open-ODU-AC. However, Open-ODU-AC actually suffers relatively big variations of transient miss ratio and freshness. A detailed transient performance comparison is given next.

**Transient Performance**: It is desirable for real-time databases to provide guarantees on miss ratio and data freshness in a continuous manner even in the presence of unpredictable workloads. To this end, we measured instantaneous miss ratio and freshness for Open-ODU-AC and QMF at each sampling point. 200% load is applied for the experiment and $EstErr$ is set to 1. Data access pattern is assumed uniform. The average miss ratio, freshness and utilization of 10 runs are plotted at each sampling period. The target miss ratio 5% is drawn as a horizontal dashed line in Figures 8 and 9.

As shown in Figure 8, Open-ODU-AC violates miss ratio and freshness requirements significantly. For example, perceived freshness is $80 \pm 24.44\%$ at 170sec. The miss ratio overshoot is over 13% at 10sec. More importantly, the miss ratio and freshness violations do *not* decay as time in-

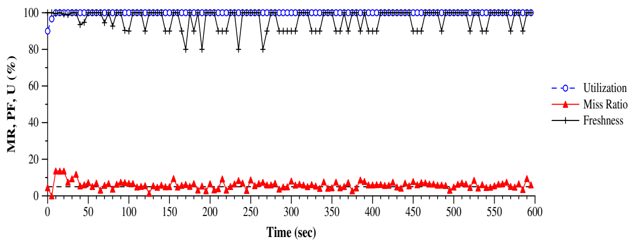


**Figure 8. Transient Performance of Open-ODU-AC (EstErr = 1, Load = 200%, Uniform Access Pattern)**

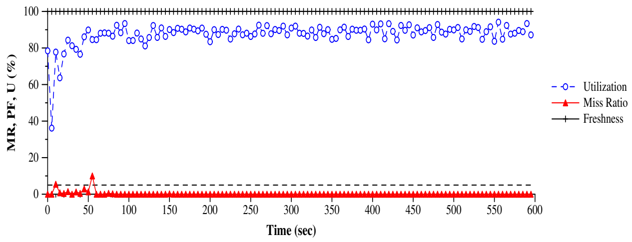


**Figure 9. Transient Performance of QMF (EstErr = 1, Load = 200%, Uniform Access Pattern)**

creases.

In contrast, QMF shows 100% freshness throughout the experiment and shows a relatively low miss ratio overshoot (10% at 55sec) compared to Open-ODU-AC (Figure 9). Furthermore, the initial miss ratio overshoot at 55sec decays in one sampling period achieving near 0 deadline miss ratio from 60sec through 600sec. Hence, it meets the theoretical settling time, 45sec. Unfortunately, the miss ratio overshoot (10%) has slightly exceeded the theoretical overshoot (6.35%) derived from the Root Locus design. This is mainly because the unit step load of Matlab is a deterministic load, whereas the exponential arrival pattern used to model the database workload is stochastic. Concerning the perceived freshness, no freshness error is observed for QMF as shown in Figure 9.

## 5.6 Effects of Varying Access Patterns

Database performance can vary significantly as the degree of data contention changes [1, 8]. Even if the typical database hot spot and the corresponding performance impacts are known, the degree of access skew can be time-varying. For this reason, we apply different access patterns by using the $x - y$ access scheme [8], in which $x\%$ of data accesses are directed to $y\%$ of the entire data in the database and $x \geq y$. For example, 90-10 access pattern means that

90% of data accesses are directed to the 10% of a database (i.e., hot spot). When $x = y = 50\%$, data are accessed in a uniform manner. We call a certain $y$ a hot spot size. The performance is evaluated for $y = 10\%$, 20%, 30%, 40%, and 50% (uniform access pattern).

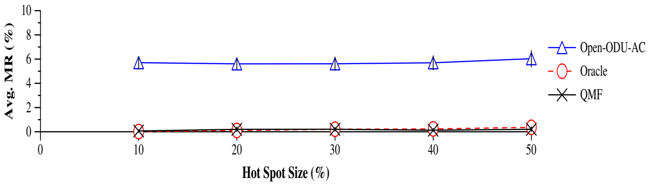


**Figure 10. Average Miss Ratio (EstErr = 1, Load = 200%)**

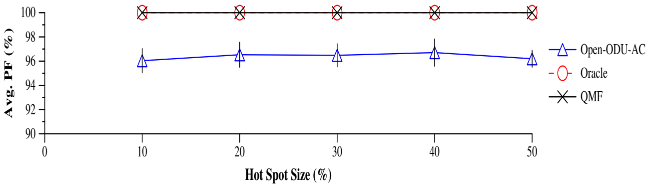


**Figure 11. Average Perceived Freshness (EstErr = 1, Load = 200%)**

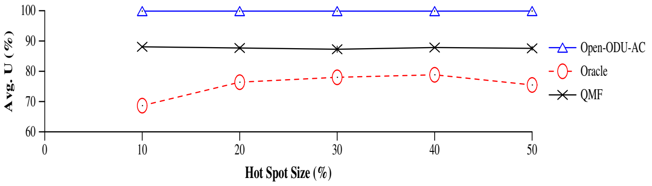


**Figure 12. Average Utilization (EstErr = 1, Load = 200%)**

In this section, Open-IMU-AC is dropped due to its relatively poor performance as shown in the previous section. Instead, we include the theoretical oracle for performance comparison. The theoretical oracle is given the future knowledge of data accesses, therefore, it is least affected by different hot spot sizes. Hence, the performance of QMF is compared to that of the Open-ODU-AC and the oracle.

In Figures 10 and 11, we observe that Open-ODU-AC violates the required miss ratio (5%) and perceived freshness (98%) for every hot spot size. In contrast, our QoS-sensitive approach satisfied the requirements. We have also collected the transient performance statistics in this experiment. It has shown similar trends in transient performance for Open-ODU-AC and QMF, respectively, as presented in the previous section. Open-ODU-AC has shown wide performance fluctuations in terms of miss ratio and freshness, whereas QMF has shown near 0 miss ratio and 100% freshness at each sampling point. We do not include the results here to avoid repetition.

Observe that our approach shows a comparable performance to the theoretical oracle in terms of miss ratio and perceived freshness while achieving the reasonable CPU utilization as shown in Figure 12. To summarize, our approach can provide guarantees on miss ratio and perceived freshness by dynamic QoD adaptations supported by feedback/admission control even in the presence of unpredictable workloads and access patterns. This is a crucial result for real-time databases, in which the freshness and miss ratio guarantees are essential for the success of the service.

## 6 Conclusions and Future Work

In this paper, we presented a QoS-sensitive approach, in which the specified miss ratio and freshness can be supported by applying the novel adaptive update policy with the feedback and admission control. We have shown that our approach can achieve a performance similar to that of the clairvoyant oracle, which is privileged by a complete future knowledge of data accesses.

## References


[1] R. Abbott and H. Garcia-Molina. Scheduling Real-Time Transactions: A Performance Evaluation. *ACM Transactions on Database System*, 17:513–560, 1992.

[2] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying Update Streams in a Soft Real-Time Database System. In *ACM SIGMOD*, 1995.

[3] K. D. Kang, S. H. Son, J. A. Stankovic, and T. F. Abdelzaher. A QoS-Sensitive Approach for Timeliness and Freshness Guarantees in Real-Time Databases. Technical Report CS-2001-22, Computer Science Department at University of Virginia, Oct. 2001.

[4] A. Labrinidis and N. Roussopoulos. Update Propagation Strategies for Improving the Quality of Data on the Web. In *Proc. of VLDB Conf.*, 2001.

[5] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback Control Real-Time Scheduling: Framework, Modeling and Algorithms. *Journal of Real-Time Systems, Special Issue on Control-Theoretical Approaches to Real-Time Computing*, 2002. To appear.

[6] K. Ramamritham. Real-Time Databases. *International Journal of Distributed and Parallel Databases*, 1(2), 1993.

[7] TimesTen Performance Software. *TimesTen White Paper*. http://www.timesten.com/library/index.html, 2001.

[8] B. Zhang and M. Hsu. Modeling Performance Impact of Hot Spots. In V. Kumar, editor, *Performance of Concurrency Control Mechanism in Centralized Database Systems*. Prentice Hall, 1996.