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
$\{kk7v, son, stankovic, zaher\}@cs.virginia.edu$

December 2, 2001

Abstract

The demand for real-time database services has been increasing recently. Examples include sensor data fusion, decision support, web information services, e-commerce, online trading, and data-intensive smart spaces. 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, e-commerce, and data-intensive smart spaces. Current databases are not time-cognizant, are poor in supporting temporal consistency of real-time data, and therefore, they do not perform well in these applications.

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 [3, 24]. 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

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

and data access patterns[1]. 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 [3]. 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 [18] to 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. Feedback control is very effective to support the specified performance when the dynamics of the controlled system (a real-time database in this paper) includes uncertainties [22]. 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 − in terms of deadline miss ratio and data freshness − 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.

The rest of this paper is organized as follows. In Section 2, the data and transaction model for real-time databases is discussed. Section 3 introduces main performance metrics and discusses QoS specification issues. In Section 4, our QoS management scheme for miss ratio and freshness guarantees is presented. In Section 5, the performance evaluation is presented. Section 6 describes the related work. Section 7 concludes the paper and discusses the future work.

2 Data and Transaction Model

In this section, our real-time database model and temporal consistency requirements are described. Data and transaction types are given. We also describe the admission control, scheduling, and concurrency control mechanisms used in our approach.

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 the main memory database model. For their relatively high performance and the decreasing main memory cost, main memory

[1] In this paper, we assume that some deadline misses are inevitable due to the potential unpredictability in workloads and data access patterns. Also, a deadline miss does not incur a catastrophic result. A few deadline misses or temporal consistency violations are considered tolerable as long as they do not exceed certain thresholds.

databases have been increasingly applied to real-time data management such as online auction/stock trading, e-commerce and voice/data networking [3, 6, 25]. 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 [24].

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 [24]. We focus on the base data freshness issues in this paper. Derived data management is reserved for future work. Hence, in this paper only the absolute validity interval is used.

Temporal data can be updated *periodically* or *aperiodically*: a periodic update occurs at fixed intervals, while an aperiodic update is not predictable and occurs only if the data value is changed. Only periodic updates are considered in this paper. Periodic updates are common in real-time applications. Our QoS management scheme can be easily extended to handle aperiodic updates. A detailed discussion is given in Section 4.1.

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 such as a sensor data object, but they can read/write non-temporal data. For example, user transactions are not allowed to change the currently measured temperature or stock prices, whereas they can change PIN numbers or change the default temperature metric from Fahrenheit to Centigrade. A user transaction can perform arithmetic/logical operations based on a set of temporal/non-temporal data. User transaction execution time and data access pattern can be time-varying. For example, in a decision support system a transaction can read different sets of data and perform different operations according to the situation. 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 [16] 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, 10], 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 Performance Metrics and QoS Specification

In this section, we introduce the main performance metrics for QoS management in real-time databases. Using the performance metrics, QoS specification issues are discussed.

3.1 Main Performance Metrics

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

- *User Transaction Deadline Miss Ratio*: 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 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}.

$$Perceived\ Freshness = 100 \times \frac{N_{fresh}}{N_{accessed}}\ (\%)$$

  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).

For example, a DBA can specify 5% deadline miss ratio and 98% freshness. This QoS specification requires us to limit the miss ratio below 5% achieving at least 98% perceived freshness at the same time.

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 such as overshoot and settling time are adopted from control theory for real-time system performance specification [17]:

- *Overshoot* is the worst-case system performance in the transient system state, e.g., the highest miss ratio in the transient state.
- *Settling time* is the time for the transient overshoot to decay and reach the steady state performance. In the steady state, the miss ratio should be below the target and the perceived freshness should be above the specified value.

In Section 5, we compare our approach to several baselines in terms of both average and transient performance for different workloads and access patterns.

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 are 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 [14]. 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 [3]. 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:

$$AUR[i] = \frac{Access\ Frequency[i]}{Update\ Frequency[i]} \tag{1}$$

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 in the k_{th} sampling period:

$$SACCESS_k[i] = a \times SACCESS_{k-1}[i] + (1-a) \times ACCESS_k[i] \tag{2}$$

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 to compute the moving average 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:

$$AUR[i] = \frac{SACCESS_k[i]}{UPDATES[i]} \tag{3}$$

The concept of AUR is illustrated in Figure 1. In the figure, the data objects in the database are ordered by non-increasing value of AUR[2]. If a data object is on or above the horizontal line of AUR = 1, 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 whose AUR ≥ 1. Or, we call the data *cold*.

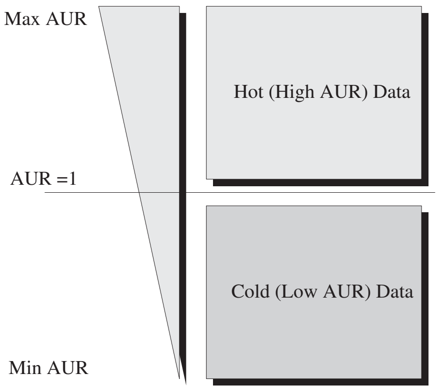


Figure 1: Database Snapshot Sorted by Access Update Ratio

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.

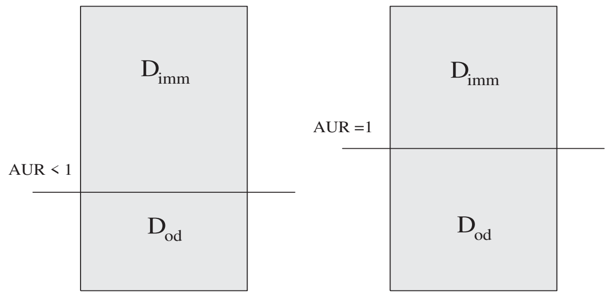
Our approach can also be extended to handle aperiodic updates. To handle aperiodic updates, the update frequency can be monitored and smoothed in the same way as the access frequency. For aperiodic updates the definition of data freshness may have to change, since there might be no explicit notion of validity interval related with aperiodic updates. In that case, a temporal data object can be considered stale upon the arrival of the corresponding update, which is not applied yet [4].

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. Also, note that our adaptive update policy is relatively coarse grained. Alternatively, the update periods can be dynamically adjusted for cold data under overload. However, this approach may incur more overhead, since the appropriate update periods and the corresponding temporal validity intervals should be dynamically managed per data object.

Consider Figure 2 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. Since a data object is updated either immediately or on demand in our approach, $D = D_{imm} \bigcup D_{od}$ and $D_{imm} \bigcap D_{od} = \emptyset$. Initially, every data is 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).

[2] In this paper, we assume the CPU utilization for sorting is relatively ignorable compared to the CPU utilization required for immediate updates of all temporal data. To reduce the sorting overhead, AUR can be measured for a group of data with a common update period. A thorough investigation is reserved for future work.



(a) Initial State (b) Moderately Loaded State (c) Overloaded State

Figure 2: Update Policy Adaptations

In Figure 2(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*. Note that we enforce a certain *upgrade bound* to avoid a miss ratio overshoot later due to the excessive update policy upgrades and the update workload increase, as a result. A detailed discussion of the upgrade bound is given in Section 4.4.4.

4.3 Update Scheduling

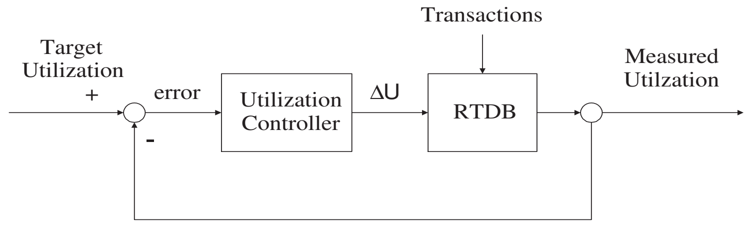
In our approach, an immediate update receives a higher priority than user transactions and on-demand updates. Conceptually, there are separate scheduling queues for immediate updates and for user transactions/on-demand updates, respectively. Immediate updates in the high priority queue are scheduled before user transactions and on-demand updates in the low priority queue. 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.

4.4 QoS Management Scheme

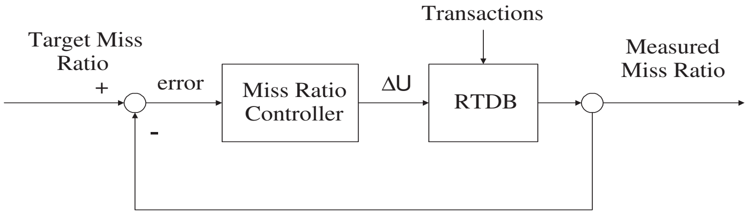
In this section, our QoS management scheme for miss ratio and freshness guarantees is described. The applied feedback control policy is discussed and the interactions between the feedback control and adaptive update policy, which manages the QoD, are described.

4.4.1 Feedback Control

We apply a feedback control real-time scheduling policy, called FC-UM [18], to control the miss ratio without underutilizing the CPU in the presence of unpredictable workloads. As shown in Figure 3, 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 ΔU, to achieve the target miss ratio (or target utilization).



(a) Utilization Control Loop



(b) Miss Ratio Control Loop

Figure 3: Miss Ratio/Utilization Controllers

We have selected FC-UM, since it can meet our basic 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 adapt and tune the feedback controllers as described in Appendix A. For detailed modeling and controller design, refer to [18].

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.

4.4.2 Feedback Control and QoD Management

Our QoS-sensitive approach for miss ratio and freshness guarantee (QMF) is described in Figure 4. We further discuss the interactions between the feedback control and QoD adaptations as follows.

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 ΔU 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. Without these conditions, a QoD degradation (upgrade) could adversely affect freshness (miss ratio) in the next sampling period.

In an extreme case, both miss ratio and and freshness constraints can be violated. In this case, incoming transactions will be rejected until a fraction of currently running transactions terminate and ΔU becomes positive, as a result. The chance of an extreme case is reduced by enforcing the QoD adaptation conditions as described

1. Collect access statistics and compute AUR.
2. Monitor the deadline miss ratio, CPU utilization, and perceived freshness.
3. At each sampling period, compute the miss ratio and utilization control signals based on the current miss ratio and utilization error, respectively. Get the CPU utilization adjustment ΔU = Minimum(miss ratio control signal, utilization control signal) for a smooth transition from a system state to another. Based on ΔU and the current system behavior, perform one of the following alternative actions.
4. If the measured miss ratio is below the target (i.e., $\Delta U \geq 0$) and the perceived freshness requirement is satisfied, no QoD adaptation is required. Inform the admission controller of ΔU to admit more transactions to prevent a potential underutilization.
5. If the measured miss ratio is over the target miss ratio (i.e., $\Delta U < 0$) and the current perceived freshness is above the target, degrade the QoD. Increase ΔU 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 ΔU.
6. If the current perceived freshness is below the target and $\Delta U > 0$, the QoD will be upgraded. Subtract the required utilization for an upgrade from ΔU. Repeat until $\Delta U \leq 0$ or a certain upgrade bound is reached. Inform the admission controller of the new ΔU.
7. If the specified miss ratio is violated (i.e., $\Delta U < 0$) and so is the perceived freshness requirement, do not admit any incoming transaction until a fraction of currently running transactions terminate, i.e., commit or abort upon their deadline misses, and ΔU becomes positive as a result.

Figure 4: QoS Management Scheme for Miss Ratio and Freshness Guarantees

before. Further, feedback control scheduling and admission control can prevent a severe overload by dynamically adjusting the CPU utilization based on the current system behavior and avoiding excessive admissions of user transactions to the system.

4.4.3 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:

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

Given the average CPU utilization per single update transaction, σU, 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, σU, 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 2(c).

4.4.4 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 as follows.

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. We first discuss our experimental goals and describe our simulation model. We also present the simulation settings, define the baselines for performance comparisons, and finally present detailed performance evaluation results.

5.1 Experimental Goals

The main objective of our performance evaluation is whether or not our approach can support the specified miss ratio and freshness even in the presence of unpredictable workloads and access patterns. To illustrate the applicability of our approach for miss ratio and freshness guarantees, we consider a QoS specification which requires us to limit the average miss ratio below 5%. A miss ratio overshoot shall be lower than 6.35% and the overshoot shall decay within 45sec. The specified miss ratio overshoot and settling time are derived from the feedback controller tuning described in Appendix A. At the same time, the perceived freshness of 98% is required. We evaluate the performance for increasing loads, increasing execution time estimation errors, and different access patterns to consider unpredictable workloads and access patterns. 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.

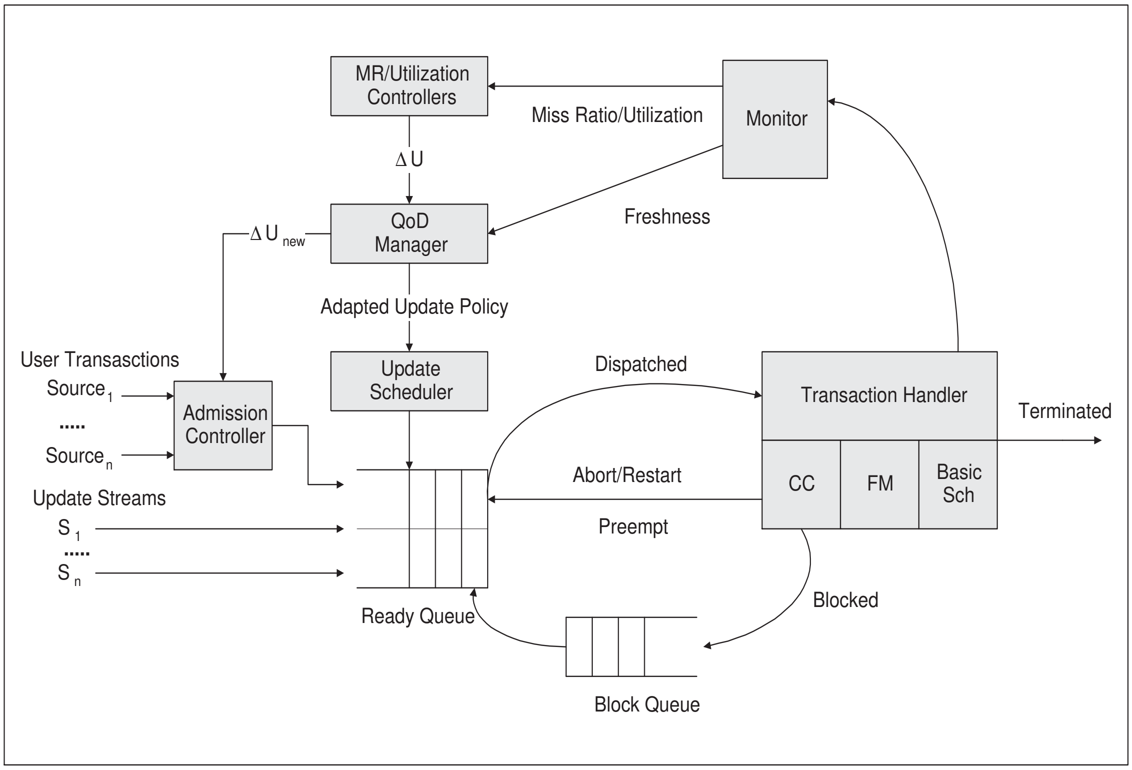


Figure 5: Simulator Architecture

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 5. 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.

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. A frequent sampling and the corresponding adaptations can improve transient performance of feedback controllers such as settling time [22]. However, 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 ΔU_{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 be always 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

The simulation settings for updates and user transactions are described as follows.

5.3.1 Set-Up for Data and Updates

The data and update transaction parameters are summarized in Table 1. 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 by adjusting the probability $P(temporal\ data\ access)$ in Table 1, 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) is uniformly distributed in the range (100ms, 50s). 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. 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.

5.3.2 Set-Up for User Transactions

Simulation settings for user transactions are summarized in Table 2. It is assumed that user transactions generally execute longer than temporal data updates, since it may perform arithmetic/logical operations based on a set of temporal data. For user transactions, accessing data sets and and execution times may vary as discussed in Section 2. 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

Table 1: Simulation Settings for Data and Updates

Parameter	Value
#Temporal Data Objects	1000
P(temporal data access)	1
Update Period	$Uniform(100ms, 50s)$
EET_i (Estimated Execution Time)	$Uniform(1ms, 8ms)$
Actual Execution Time	$Normal(EET_i, \sqrt{EET_i})$
Total Update Load	$\approx 50\%$

Table 2: Simulation Settings for User Transactions

Parameter	Value
EET_i (Estimated Execution Time)	$Uniform(20ms, 80ms)$
AET_i (Average Execution Time)	$EET_i \times (1 + EstErr_i)$
Actual Execution Time	$Normal(AET_i, \sqrt{AET_i})$
N_{DATA_i} (#Average Data Accesses)	$EET_i \times Data\ Access\ Factor = (10, 40)$
#Actual Data Accesses	$Normal(N_{DATA_i}, \sqrt{N_{DATA_i}})$

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) 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).

It is assumed that the number of data accesses (N_{DATA_i}) for $Source_i$ is proportional to the length of 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* (our QoS-sensitive approach that employs feedback control scheduling, adaptive update policy and admission control for miss ratio and freshness guarantees) with the several baselines as follows:

- *Open-IMU:* Admission control and the closed loop scheduling by feedback control are turned off. All data are updated immediately. Update policy is not adapted in this open-loop scheduling framework.
- *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 the 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. To implement it, we divided an experiment into two passes. (Each pass is a complete simulation run.) In the first pass, which may include unnecessary updates, the data access history of user transactions is recorded in a form of *(time, accessed data)*. In the second pass, the same simulation set-up and seed number are applied to generate the same set of transactions and data accesses. The update scheduler can utilize the access history collected in the first pass to *predict* the future [3].

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. From the first experiment, we select the best performing open-loop baselines for the further performance comparison 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. Our approach is considered successful, if it can consistently provide guarantees on miss ratio and perceived freshness against increasing loads/execution time estimation errors and different degrees of data contentions.

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 Experiment 1: Effects of Increasing Load

Computational systems may show different performance for different loads, especially when the system is overloaded. Therefore, it is important to compare the relative performance of our approach to the baselines as the load increases. For this purpose, we give the simulated real-time database different loads: 70%, 100%, 150% and 200% which represent the loads assuming that all incoming user transactions are admitted and every temporal data object is updated immediately. The tested approaches may actually reduce the given loads by applying admission control and/or on-demand update policy. Also, it is important to note that in this experiment no execution time estimation error is assumed, i.e., $EstErr = 0$. For data accesses, uniform access pattern is applied.

In Figure 6, Open-IMU (the top curve) shows the highest miss ratio reaching $83.6 \pm 1.65\%$ given 200% load. Open-IMU-AC reduced miss ratio by admission control, however, the miss ratio is $50.74 \pm 5.28\%$ for 200% load.

Open-ODU shows the reasonable miss ratio up to 150% load. However, its miss ratio exceeds 40% for 200% load possibly due to the increased waiting time for on-demand updates. As shown in Figure 7, the perceived freshness of Open-ODU drops significantly as the load increases. In contrast, all the other approaches provide 100% freshness. (For this reason, the top four curves of all the other approaches overlap in Figure 7. Open-ODU is the only exception.)

As shown in Figures 6 and 7, admission control has significantly improved the performance of ODU approach. Open-ODU-AC shows a similar performance (near 0% miss ratio and 100% freshness) to that of QMF. (See the two bottom miss ratio curves for Open-ODU-AC and QMF overlap in Figure 6.) This is mainly because $EstErr = 0$, which is ideal, for this experiment. In Section 5.6, we show that Open-ODU-AC fails to guarantee miss ratio and perceived freshness when only rough execution time estimates are available.

Figure 8 plots the average utilization measured for different workloads. We observe that our approach (QMF) shows relatively steady utilization ranging from $69.47 \pm 1.2\%$ to $88.46 \pm 0.12\%$ for the given loads (70%, 100%, 150%, 200%). Note that Open-ODU and Open-ODU-AC show severe underutilization before the load reaches

[3] Admission control is turned off to reproduce the exactly same workload. If admission control is turned on, a user transaction might be rejected in the first pass, but could be admitted in the second pass due to the reduced update workload. As a result, the corresponding data accesses are unknown a priori.

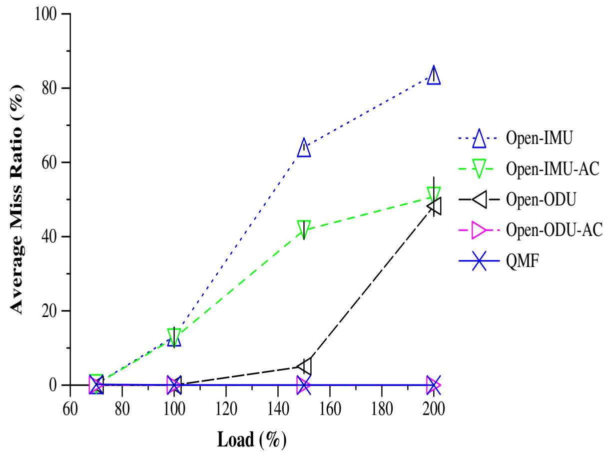


Figure 6: Average Miss Ratio (EstErr = 0, Uniform Access Pattern)

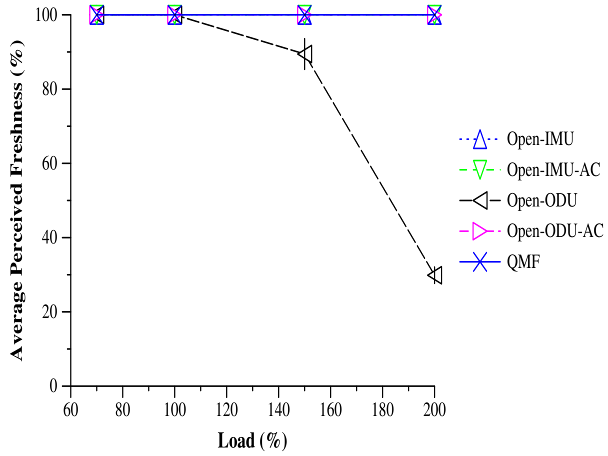


Figure 7: Average Perceived Freshness (EstErr = 0, Uniform Access Pattern)

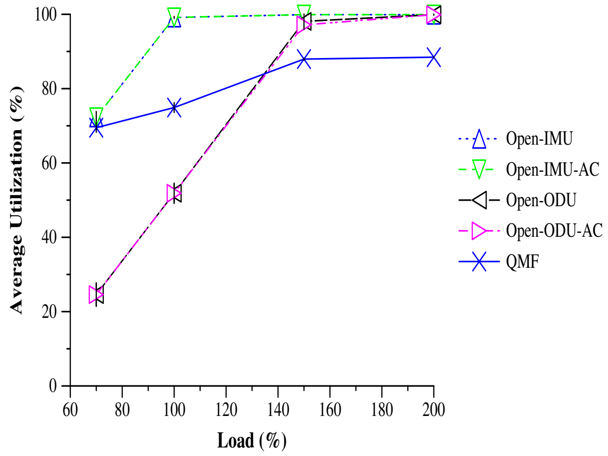


Figure 8: Average Utilization (EstErr = 0, Uniform Access Pattern)

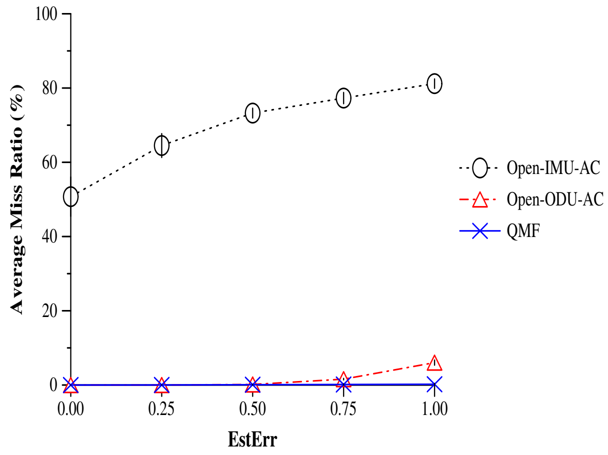


Figure 9: Average Miss Ratio (Load = 200% , Uniform Access Pattern)

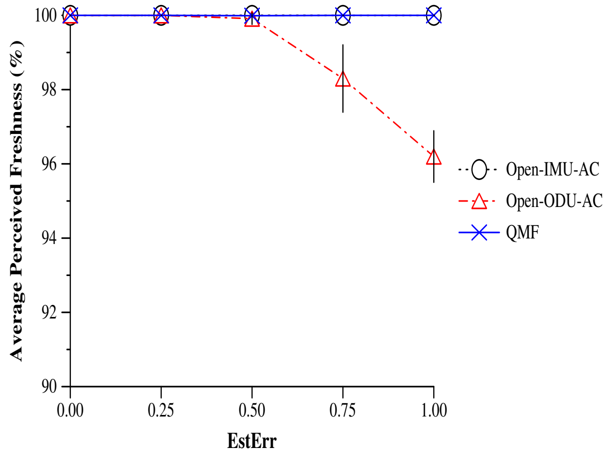


Figure 10: Average Perceived Freshness (Load = 200% , Uniform Access Pattern)

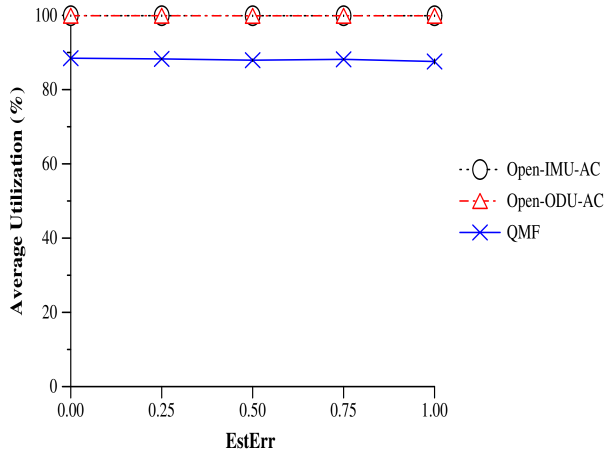


Figure 11: Average Utilization (Load = 200% , Uniform Access Pattern)

150% due to unscheduled updates. To verify this, we have measured the average *database* freshness for Open-ODU and Open-ODU-AC. The average database freshness is below 10% at every load for both Open-ODU and Open-ODU-AC as a result of lazy updates. Even though the system is underutilized, all updates are scheduled on demand in Open-ODU and Open-ODU-AC. In contrast, all updates are scheduled in a preferred manner to user transactions in Open-IMU and Open-IMU-AC despite the high deadline miss ratio of user transactions. QMF shows the relatively stable performance by dynamically adapting the update policy considering the system status, if necessary. It is further aided by feedback control.

We have dropped Open-IMU and Open-ODU from further performance evaluation due to their relatively poor performance. It is simply not feasible for these approaches to provide guarantees on miss ratio and freshness under overload.

5.6 Experiment 2: Effects of Increasing Execution Time Estimation Error

5.6.1 Average Performance

In the previous section, we assumed that $EstErr = 0$ (i.e., no execution time estimation error). However, precise execution time estimates are generally not available. To compare the performance of various approaches, we introduce estimation errors in this section. 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. 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 9, 10 and 11, respectively. In this way, we can observe the effects of the increasing execution time estimation error to the performance when overloaded.

As shown in Figure 9, 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 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 10, 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 from the algorithms tested only our approach can meet both miss ratio and freshness requirements when only a rough estimation is available.

Average utilization is plotted in Figure 11. 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.

5.6.2 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 12 and 13.

As shown in Figure 12, Open-ODU-AC violates miss ratio and freshness requirements significantly[4]. For

[4]Confidence intervals are derived, but not plotted in transient performance evaluations for the clarity of presentation.

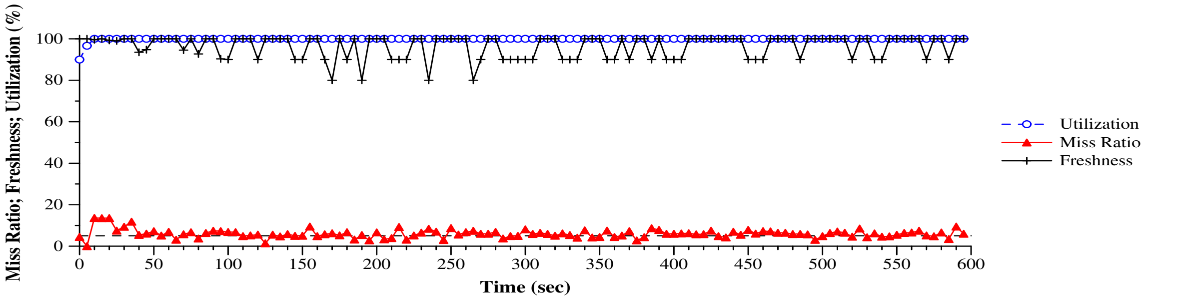


Figure 12: Transient Performance of Open-ODU-AC (EstErr = 1, Load = 200%, Uniform Access Pattern)

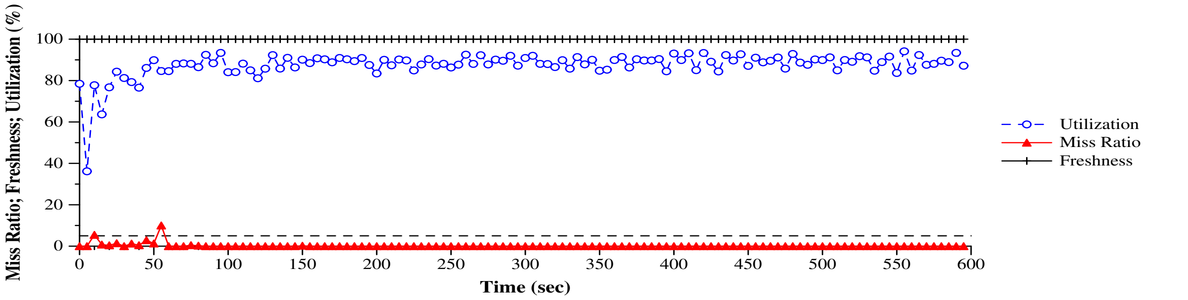


Figure 13: Transient Performance of QMF (EstErr = 1, Load = 200%, Uniform Access Pattern)

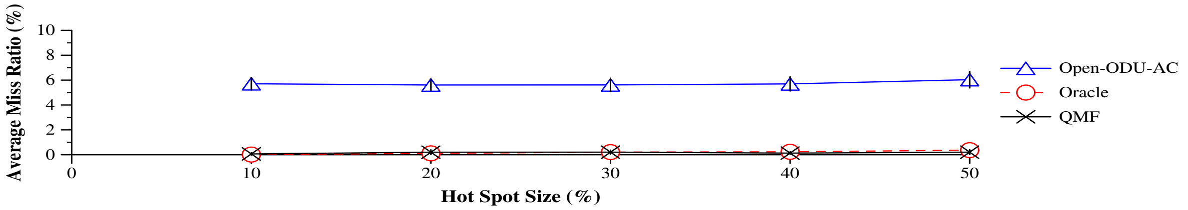


Figure 14: Average Miss Ratio (EstErr = 1, Load = 200%)

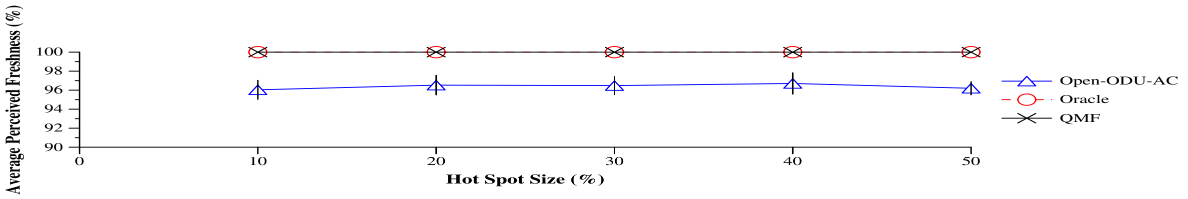


Figure 15: Average Perceived Freshness (EstErr = 1, Load = 200%)

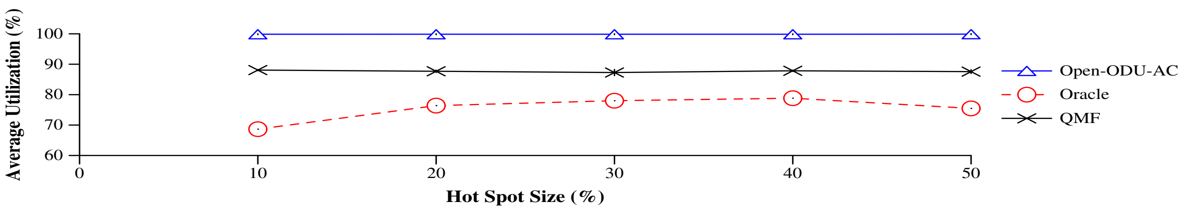


Figure 16: Average Utilization (EstErr = 1, Load = 200%)

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 increases.

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 13). 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, derived from the controller tuning (Appendix A) by Root Locus design in Matlab [22]. 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 13.

5.7 Experiment 3: Effects of Varying Access Patterns

Database performance can vary significantly as the degree of data contention changes [1, 29]. 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 [29], in which $x\%$ of data accesses are directed to $y\%$ of the entire data in the database and $x \leq y$. 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).

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 14 and 15, 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 16. 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 Related Work

Previous research work has shown that QoS-sensitive approaches can improve system performance in a cost-effective manner [2, 7, 9, 11, 12]. Despite the abundance of the QoS research, QoS-related work is relatively scarce in database systems. Priority Adaptation Query Resource Scheduling (PAQRS) provided timeliness differentiation of query processing in a memory-constrained environment [21]. From the observation that the performance of queries can vary significantly depending on the available memory, per-class query response time was differentiated by an appropriate memory management and scheduling. Given enough memory, queries can read the operand relations at once to produce the result immediately. If less memory is allocated, they have to use temporary files to save the intermediate results, therefore, the query processing may slow down. In this way, query deadline miss ratios were differentiated between the classes. However, the performance could easily fluctuate under the workload changes. Neither any data freshness issue was considered.

The notion of QoD was introduced in [14]. In their work, a novel online update scheduling policy was proposed in the context of the web server. The performance of a web server is improved by caching dynamically generated data at the web server and the back-end database continuously updates them. Given the views to materialize, the proposed update scheduling policy can significantly improve the data freshness compared to FIFO scheduling. They discuss a complementary problem in [13], i.e., view selection problem to materialize. Trade-off issues between response time and data freshness are considered in their work. However, they provide neither miss ratio nor data freshness guarantee.

Stanford Real-Time Information Processor (STRIP) addressed the problem of balancing between the freshness and transaction timeliness [3]. In a real-time database, data should be maintained fresh to correctly reflect the status of the real-world environment, also transactions should be processed in a timely manner. To study the trade-off between freshness and timeliness, four scheduling algorithms were introduced to schedule updates and transactions, and the performance was compared. In their later work, a similar trade-off problem was studied for derived data [4]. Ahmed et al proposed a new approach to maintain the temporal consistency of derived data [5]. Different from STRIP, an update of a derived data object is explicitly associated with a certain timing constraint, and is triggered by the database system only if the timing constraint could be met. By a simulation study, the relative performance improvement was shown compared to the forced delay scheme of STRIP. None of the two approaches considers dynamic adaptations of update policy. Also, performance guarantee is not provided.

The correctness of answers to database queries can be traded off to enhance timeliness. A query processor, called APPROXIMATE [28], can provide approximate answers depending on the availability of data or time. An imprecise computation technique (milestone approach [15]) is applied by APPROXIMATE. In the milestone approach, the accuracy of the intermediate result increases monotonically as the computation progresses. Therefore, the correctness of answers to the query could monotonically increase as the query processing progresses. A relational database system, called CASE-DB [20], can produce approximate answers to queries within certain deadlines. Approximate answers are provided processing a segment of the database by sampling, and the correctness of answers can improve as more data are processed. Before beginning each data processing, CASE-DB determines if the segment processing can be finished in time. In replicated databases, consistency can be traded off for shorter response time. For example, epsilon serializability [23] allows a query processing despite the concurrent updates. Notably, the deviation of the answer to the query can be bounded, different from a similar approach called quasi serializability [8]. An adaptable security manager is proposed in [26], in which the database security level can be temporarily degraded to enhance timeliness. These performance trade-off schemes lack a systematic QoS management architecture and none of them consider providing guarantee for both miss ratio and freshness.

Recently, feedback control has been widely applied to QoS management and real-time scheduling [2, 18, 27, 19]. However, to our best knowledge none of them considered QoS management issues in real-time databases considering timing and data freshness constraints.

7 Conclusions and Future Work

The demand for real-time information services is rising in several new applications. In these applications, real-time databases can be overloaded due to the high service demand and unpredictable workloads/data access patterns. As a result, users may suffer poor services, e.g., many deadline misses or temporal consistency violations. In QMF, a database administrator can explicitly specify the required database QoS in terms of deadline miss ratio and perceived freshness. By adopting QMF in real-time databases, updates and user service requests can be dynamically balanced to guarantee potentially conflicting miss ratio and freshness requirements at the same time. Applying the novel adaptive update policy with the feedback and admission control, the specified miss ratio and freshness can be supported, while other non-adaptive approaches fail. In the performance evaluation, we have shown that our approach can achieve a performance, in terms of miss ratio and data freshness, similar to that of the clairvoyant oracle which is privileged by a complete future knowledge of data accesses.

In future, we shall further investigate the trade-off issues between timeliness and freshness. We shall also investigate an effective management technique of derived data freshness, a feedback control framework for disk-resident real-time databases, differentiation of transaction timeliness, and a real-time database middleware to provide the performance guarantee/differentiated service. We also plan to apply our approach to realistic workloads such as stock trading and agile manufacturing.

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] T. F. Abdelzaher and K. G. Shin. Adaptive Content Delivery for Web Server QoS. In *International Workshop on Quality of Service*, June 1999.

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

[4] B. Adelberg, B. Kao, and H. Garcia-Molina. Database Support for Efficiently Maintaining Derived Data. In *ETDB*, 1996.

[5] Q. N. Ahmed and S. V. Vrbsky. Triggered Updates for Temporal Consistency in Real-Time Databases. *Real-Time Systems Journal*, 19:209–243, 2000.

[6] J. Baulier, P. Bohannon, S. Gogate, C. Gupta, S. Haldar, S. Joshi, A. Khivesera, H. F. Korth, P. McIlroy, J. Miller, P. P. S. Narayan, M. Nemeth, R. Rastogi, S. Seshadri, A. Silberschatz, S. Sudarshan, M. Wilder, and C. Wei. DataBlitz Storage Manager: Main Memory Database Performance for Critical Applications . In *ACM SIGMOD - Industrial Session: Database Storage Management*, 2000.

[7] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating User-Perceived Quality into Web Server Design. In *9th International World Wide Web Conference*, 2000.

[8] W. Du and A. Elmagarmid. Quasi serializability: A Correctness Criterion for Global Concurrency Control in InterBase. In *Proceedings of the 15th International Conf. on Very Large Data Bases*, 1989.

[9] L. Eggert and J. Heidemann. Application-Level Differentiated Services for Web Services. *World Wide Web Journal*, 3(2), March 1999.

[10] J. Huang, J. A. Stankovic, D. F. Towsley, and K. Ramamritham. Experimental Evaluation of Real-Time Transaction Processing. In *IEEE Real-Time Systems Symposium*, pages 144–155, 1989.

[11] D. Hull, A. Shankar, K. Nahrstedt, and J. W. S. Liu. An End-to-End QoS Model and Management Architecture. In *Proceedings of IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, Sanfrancisco, California, December 1997.

[12] M. Humphrey, S. Brandt, G. Nutt, and T. Berk. The DQM Architecture: Middleware for Application-centered QoS Resource Management. In *Proceedings of IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, Sanfrancisco, California, December 1997.

[13] A. Labrinidis and N. Roussopoulos. Adaptive WebView Materialization. In *the Fourth International Workshop on the Web and Databases, held in conjunction with ACM SIGMOD*, May 2001.

[14] A. Labrinidis and N. Roussopoulos. Update Propagation Strategies for Improving the Quality of Data on the Web. In *the 27th International Conference on Very Large Data Bases (VLDB'01)*, Rome, Italy, September 2001.

[15] K. J. Lin, S. Natarajan, and J. W. S. Liu. Imprecise Results: Utilizing Partial Computations in Real-Time Systems. In *Real-Time System Symposium*, December 1987.

[16] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1979.

[17] C. Lu, J. Stankovic, T. Abdelzaher, G. Tao, S. H. Son, and M. Marley. Performance Specifications and Metrics for Adaptive Real-Time Systems. In *Real-Time Systems Symposium*, Orlando, Florida, November 2000.

[18] 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.

[19] Y. Lu, A. Saxena, and T. F. Abdelzaher. Differentiated Caching Services; A Control-Theoretical Approach. In *21st International Conference on Distributed Computing Systems*, Phoenix, Arizona, April 2001.

[20] G. Ozsoyoglu and W.-C. Hou. Research in Time- and Error-Constrained Database Query Processing. In *Workshop on Real-Time Operating Systems and Software*, May 1990.

[21] H. Pang, M. Carey, and M. Livny. Multiclass Query Scheduling in Real-Time Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):533–551, August 1995.

[22] C. L. Phillips and H. T. Nagle. *Digital Control System Analysis and Design (3rd edition)*. Prentice Hall, 1995.

[23] C. Pu and A. Leff. Replica Control in Distributed Systems: As Asynchronous Approach. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1991.

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

[25] TimesTen Performance Software. *TimesTen White Paper*. Available in the World Wide Web, http://www.timesten.com/library/index.html, 2001.

[26] S. H. Son, R. Zimmerman, and J. Hansson. An Adaptable Security Manager for Real-Time Transactions. In *Euromicro Conference on Real-Time Systems*, pages 63–70, Stockholm, Sweden, June 2000.

[27] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 1999.

[28] S. Vrbsky. *APPROXIMATE: A Query Processor that Produces Monotonically Improving Approximate Answers*. PhD thesis, University of Illinois at Urbana-Champaign, 1993.

[29] 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.

Appendix A Feedback Controller Tuning

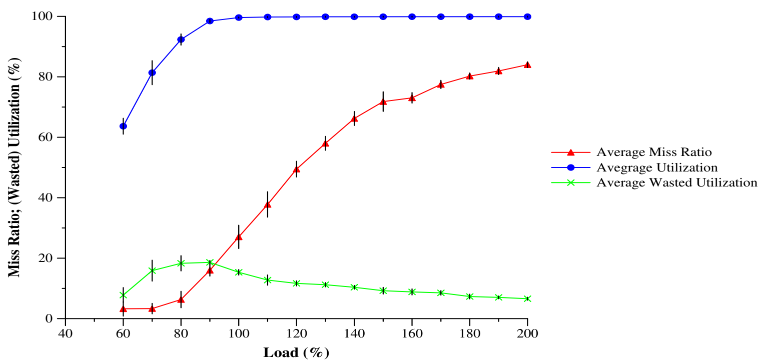


Figure 17: System Profiling Results

To tune the controllers of FC-UM, the performance of the controlled system, i.e., a simulated real-time database in this paper, should be profiled under the worst case set-up that can cause the highest miss ratio [18]. The worst case should be considered to provide a certain miss ratio guarantee. For the profiling under the worst case set-up, we turned off admission control and QoS management. All updates are applied immediately in a preferred manner to user transactions. As a result, the user transaction deadline miss ratio increases sharply as load increases (Figure 17). Average deadline miss ratio and utilization are measured for loads increasing from 60% to 200% by 10%. Execution time estimation error ($EstErr$) is set to 1. Each simulation run lasts for 10 min. Update workload is designed to be about 50% of the total CPU utilization for each load. Uniform access pattern is assumed for data accesses. For each load, 10 simulation runs are performed and 90% confidence intervals are derived (vertical bars in Figure 17).

The miss ratio gain, $G_M = Max\{\frac{Miss\ Ratio\ Increase}{Unit\ Load\ Increase}\}$, should be derived to tune the controllers [18]. According to our profiling results shown in Figure 17, the miss ratio gain is approximately 1.1682 when the load increases from 110% to 120%. We set the sampling period to 5sec for feedback control. Given the sampling period and miss ratio gain G_M, Root Locus method [22] of Matlab can be used to tune the controllers to support 0 steady state error. The closed loop poles are $p_0, p_1 = 0.552 \pm 0.153i$. The feedback control system is stable, since the closed loop poles are inside the unit circle. The tuned feedback control system can provide the following transient performance:

- The theoretical overshoot (the worst case performance, e.g., highest deadline miss ratio) is 27% for a unit step input. For example, if the target deadline miss ratio is 5%, the theoretical miss ratio overshoot is $5\% \times (1 + 0.27) = 6.35\%$.
- From the Root Locus design, the theoretical settling time (the time for system transients to decay) is 45sec (i.e., 9 sampling periods). In the previous example, the miss ratio overshoot should decay within 45sec for a unit step input.

Careful readers may have noticed that the measured average utilization tends to be higher than the load applied to the simulated real-time database before it saturates as shown in Figure 17. This is because the data/resource conflicts increase between updates and user transactions as load increases. (More transactions access temporal data updated by update transactions increasing potential read/write conflicts). From Figure 17, we can observe that the wasted utilization increases until the system saturates. (It decreases after the system is saturated, since tardy transactions can be aborted even before accessing temporal data). The increase of wasted utilization adversely affects the total utilization and miss ratio. This observation motivates the necessity of dynamic balancing between updates and user transactions considering the current system status.