Service Differentiation in Real-Time Main Memory Databases *†

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

Abstract

The demand for real-time database services has been increasing recently. Examples include sensor data fusion, stock trading, decision support, web information services, and data-intensive smart spaces. In these systems, it is essential to execute transactions in time using fresh (temporally consistent) data. Due to the high service demand, many transactions may miss their deadlines regardless of their importance. To address the problem, we present a service differentiation architecture for real-time databases. Transactions are classified into several service classes based on their importance. Under overload, different degrees of deadline miss ratio guarantees are provided among the service classes according to their importance. A certain data freshness guarantee is also provided for the data accessed by timely transactions which finish within their deadlines. Feedback control is applied to support the miss ratio and freshness guarantees. In a simulation study, our service differentiation approach shows a significant performance improvement compared to the baseline approaches. The specified miss ratio and freshness are supported even in the presence of unpredictable workloads and data access patterns.

1. Introduction

In real-time databases, it is desirable to execute transactions within their deadlines using fresh (temporally consistent) data which can reflect the continuously changing external environments, e.g., current temperature or stock prices. Current databases are poor in supporting timing constraints and data temporal consistency.

Due to the high service demand and stringent timing/data freshness constraints, real-time databases can be overloaded. When overloaded, transactions miss their deadlines regardless of their importance. To address this problem, we present a service differentiation architecture for real-time databases, in which important transactions can be executed in a preferred manner under overload. We classify transactions into premium, basic, and best-effort classes according to their importance. The main objective of our approach is to limit the deadline miss ratios below the specified thresholds in the premium and basic classes while trying to meet as many best-effort transaction deadlines as possible. At the same time, we aim to support a certain freshness for the data accessed by timely transactions − transactions which finish within their deadlines − even in the presence of unpredictable workloads and data access patterns. Different degrees of miss ratio guarantees are provided among the service classes considering the application semantics. For example, consider agile manufacturing. For the key process control and state updates, a strong performance guarantee is required in terms of both average and transient miss ratios. In contrast, routine remote process monitoring transactions can be satisfied by guaranteeing the average miss ratio. The least important transactions not related to the process control, state updates, or remote monitoring can be handled in a best-effort manner. A similar strategy can be applied to other real-time database applications such as stock trading.

To provide differentiated services in terms of miss ratio, we extend a QoS-sensitive approach, called QMF [7], which can support a single class of miss ratio and freshness guarantees in real-time databases. We call the new approach **QMF−Diff** (QMF with Differentiated services). Feedback control is the key technology we use to achieve differentiated miss ratio guarantees.

In the simulation study, we apply a wide range of workloads and data access patterns to model potential unpredictability. The performance results show that our approach can provide the specified guarantees on miss ratio and data freshness, while other baseline approaches fail to support the miss ratio and/or freshness guarantees in the presence of unpredictable workloads and access patterns. Our ap-

* Supported, in part, by NSF grants EIA-9900895 and CCR-0098269.
† In the Proceedings of the 5th IEEE International Symposium on Object-oriented Real-time Distributed Computing, April 29 - May 1, 2002.

proach also achieves the lowest miss ratios in the basic and best-effort classes, respectively, compared to the baseline approaches.

2 Performance Metrics and QoS Specification

In real-time databases, workloads and data access patterns can be time-varying. For example, users of a decision support systems may read varying sets of data and perform different arithmetic/logical operations based on the current situation. In this paper, we assume that some deadline misses or freshness violations are inevitable due to unpredictable workloads and data access patterns. It is also assumed that a deadline miss or freshness violation does not incur a catastrophic result. A few deadline misses or freshness violations are considered tolerable as long as they do not exceed certain thresholds. For this reason, we consider the miss ratio and data freshness as key performance metrics in database QoS. A database administrator (DBA) can explicitly specify the required database QoS in terms of miss ratio and freshness.

2.1 Real-Time Database Model

We consider the firm real-time database model, in which tardy transactions – transactions that have missed their deadlines – are aborted upon their deadline misses. Transactions are classified as either user transactions or sensor updates. Continuously changing real-world states, e.g., the current sensor values, are captured by periodic updates. User transactions execute operations based on the current real-world states reflected in the real-time database.

In this paper, the main memory database model is considered. For their relatively high performance and the decreasing main memory cost, main memory databases have been increasingly used for real-time data management such as online auction/stock trading, e-commerce, and voice/data networking [2, 13]. In our main memory database model, the CPU is the main system resource for consideration.

2.2 Performance Metrics

In our real-time database model, two main performance metrics are considered: *per-class deadline miss ratio* and *freshness* of data accessed by timely transactions. In the remainder of this paper, we follow a convention in which the highest priority class is associated with the lowest class number. Thus, Classes 0, 1, and 2 represent the premium, basic, and best-effort classes, respectively.

- *Miss Ratio*: The miss ratio for Class i transactions is defined as:

$$MR_i = 100 \times \frac{\#\ tardy_i}{\#\ tardy_i \ + \ \#\ timely_i}\ (\%)$$

where $\#\ tardy_i$ and $\#\ timely_i$ represent the number of (Class i) transactions that have missed their deadlines and the number of transactions that have finished within their deadlines, respectively.

- *Freshness*: Data in real-time databases can become out of date due to the passage of time. To measure the freshness of data in real-time databases, we use the notion of absolute validity interval [11]. A data object X is related to a timestamp indicating the latest observation of the real-world. X is considered temporally consistent or fresh if (*current time* $-$ *timestamp(X)* $\leq avi(X)$) where $avi(X)$ is the absolute validity interval of X. We further classify the data freshness into: *database freshness* and *perceived freshness* [7]. Database freshness, also called QoD (Quality of Data) in this paper, is the ratio of fresh data to the entire data in a database. In contrast, perceived freshness is defined as follows. Let us call the number of data accessed by timely transactions $N_{accessed}$. Let N_{fresh} represent the number of fresh data accessed by timely transactions.

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

We consider the miss ratio and perceived freshness as two main aspects of QoS in real-time databases. In a QoS specification only the perceived freshness is considered since tardy transactions add no value to our firm real-time database model.

Long-term performance metrics such as average miss ratio are not sufficient for the performance specification of dynamic systems, in which the system performance can be time-varying. For this reason, transient performance metrics such as overshoot and settling time are adopted from control theory for a real-time system performance specification [8]:

- *Overshoot* (M_p) is the worst-case 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.

2.3 QoS Specification

In a QoS specification, a DBA can specify a threshold of the deadline miss ratio for each service class and the target perceived freshness (PF_{target}). In this way, a DBA can explicitly specify the tolerable per-class miss ratios and the perceived freshness desired for a specific real-time database application. In this paper, we consider the following QoS specification as an example to illustrate our approach for service differentiation: "*QoS-Spec* = $\{(MR_0 \leq 1\%,\ M_p \leq 30\%,\ t_s \leq 100sec),\ MR_1 \leq 5\%,\ MR_2 =$ *best-effort*, $PF_{target} \geq 98\%\}$". Note that this specification requires that the average miss ratio is below 1% for Class 0. We also set M_p = 30%, therefore, a MR_0 overshoot should not exceed $1.3\% = 1\% \times (1 + 0.3)$, and the overshoot should decay within 100sec. The average miss ratio should be below 5% for Class 1, while the best-effort service is specified for Class 2. At least 98% perceived freshness is required for timely transactions. This is a stringent QoS requirement considering the specified average/transient miss ratio requirements and the required data freshness. In Section 4, various experiments are performed to determine whether or not this QoS specification can be supported even in the presence of unpredictable workloads and data access patterns.

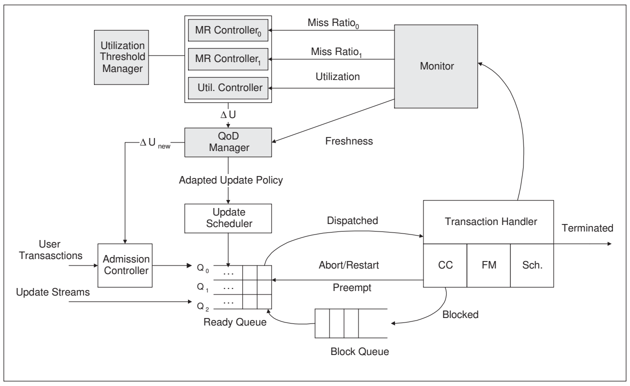

Figure 1. Real-Time Database Architecture for Service Differentiation

3 Differentiated Service Architecture

Our differentiated service architecture is shown in Figure 1. A transaction is scheduled in one of the multi-level ready queues according to its service class. The transaction handler executes queued transactions. At each sampling instant, the current miss ratio, perceived freshness, and the CPU utilization are monitored. The miss ratio and utilization controllers derive the required CPU utilization adjustment (ΔU in Figure 1) considering the current performance error such as the miss ratio overshoot or CPU underutilization. Based on ΔU, the QoD manager adapts the database QoD, if necessary. The update scheduler schedules an incoming update according to the update policy currently associated with the corresponding data object. The admission controller enforces the remaining utilization adjustment after potential QoD adaptations, i.e., ΔU_{new}. Each component is discussed in detail below.

3.1 Transaction Handler

The transaction handler provides an infrastructure for real-time database services, which consists of a concurrency controller (CC), a freshness manager (FM) and a basic scheduler. For concurrency control, we use 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 a priority inversion [1].

The FM checks the freshness before accessing a data item using the corresponding avi. It blocks a user transaction if an accessing data item 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.

By the basic scheduler, user transactions are scheduled in one of multi-level queues (Q_0, Q_1, and Q_2 as shown in Figure 1) according to their service classes. A fixed priority is applied among the multi-level ready queues. A transaction in a low priority queue can be scheduled if there is no ready transaction at the higher priority queue(s). A low priority transaction is preempted upon the arrival of a high priority transaction. In each queue, transactions are scheduled in EDF manner. To provide the data freshness guarantee, all sensor updates are scheduled at Q_0 in this paper.

By applying the fixed priority among the service classes, we provide a basic support for the service differentiation in real-time databases. However, this is insufficient to provide guarantees on the miss ratios (in premium and basic classes) and perceived freshness in the presence of unpredictable workloads/access patterns. For this reason, we apply the feedback control as follows.

3.2 Feedback Control

It is well known that feedback control is very effective in supporting a required performance specification when the system model includes uncertainties [10]. The target performance can be achieved by dynamically adapting the system behavior based on the current performance error measured in the feedback control loop. Due to its robustness, feedback control has recently been applied to various types of computational systems to provide performance guarantees [8, 9, 12]. We extend a feedback control scheduling policy, called FC-UM [8], to provide the differentiated service among the service classes in terms of miss ratio. FC-UM is selected since it can provide a certain miss ratio guarantee without underutilizing the CPU given unpredictable workloads.

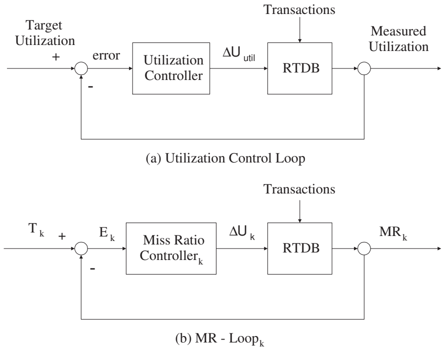

Figure 2. Miss Ratio/Utilization Controllers

3.2.1 Utilization Controller

One utilization control loop is employed to prevent a potential underutilization, similar to FC-UM. This is to avoid a trivial solution, in which all the miss ratio requirements are satisfied due to the underutilization. At each sampling instant, the utilization controller computes the utilization control signal ΔU_{util} based on the utilization error which is the difference between the target utilization and the current utilization measured by the Monitor at the current sampling instant as shown in Figure 2 (a). We further extend the utilization controller by employing the utilization threshold manager as follows.

3.2.2 Utilization Threshold Manager

For many complex real-time systems, the schedulable utilization bound is unknown or can be very pessimistic [8]. In real-time databases, the utilization bound is hard to derive, if it even exists. This is partly because database applications usually include unpredictable aborts/restarts due to data/resource conflicts. A relatively simple way to handle this problem is to set/enforce a pessimistic utilization threshold. However, this can lead to an unnecessary underutilization. In contrast, an excessively optimistic utilization threshold can lead to a large miss ratio overshoot. It is a hard problem to decide a proper utilization threshold in a complex real-time system such as a real-time database.

To address this problem, we propose a novel online approach in which the utilization threshold (the target utilization in Figure 2 (a)) is dynamically adjusted considering the current real-time system behavior as follows. Initially, the utilization threshold is set to a lower utilization set point, e.g., 80%. If no deadline miss is observed at the current sampling instant in Classes 0 and 1, the utilization threshold is incremented by a certain step size unless the resulting utilization threshold is over 100%. The utilization threshold will be continuously increased as long as no deadline miss is observed in Classes 0 and 1. The utilization threshold will be switched back to the lower utilization set point as soon as the miss ratio controller takes the control. This is to prevent a potential miss ratio overshoot due to an over-optimistic utilization threshold. Note that our approach is self-adaptive requiring no a priori knowledge about a specific workload model and is computationally light-weight. Using our approach, the potentially time-varying utilization threshold can be closely approximated.

3.2.3 Miss Ratio Controllers

In our approach, miss ratio controllers are employed for Classes 0 and 1, respectively, to provide guarantees on their miss ratios. Each class k $(0 \leq k \leq 1)$ is associated with a certain miss ratio threshold T_k as shown in Figure 2 (b). In Class k, a miss ratio control loop $MR - LOOP_k$ computes a miss ratio control signal ΔU_k based on the current performance error E_k, which is the difference between the class-specific threshold T_k and the current miss ratio of the user transactions in Class k, MR_k, measured by the Monitor at the current sampling instant. When overloaded, ΔU_k can become negative to request the reduction of the CPU utilization.

3.2.4 Derivation of a Single Control Signal

From the two miss ratio control signals (ΔU_k where $0 \leq k \leq 1$), a single miss ratio control signal, ΔU_{MR}, is derived. To derive ΔU_{MR}, we need to consider two cases: both ΔU_0 and ΔU_1 are currently negative, or at least one of the two control signals is non-negative. In the first case, both MR_0 and MR_1 are violated. Hence, we set $\Delta U_{MR} = \sum_{k=0}^{1} \Delta U_k$ to require enough CPU utilization adjustment (i.e., reduction) to avoid a significant miss ratio increase in the consecutive sampling instants. Otherwise, we set $\Delta U_{MR} = Minimum(\Delta U_0, \Delta U_1)$ to support a smooth transition from one system state to another, similar to [8]. After deriving ΔU_{MR}, we set the current control signal $\Delta U = Minimum(\Delta U_{util}, \Delta U_{MR})$ for a similar reason.

3.2.5 Integrator Antiwindup

Each feedback controller in Figure 2 is a digital PI (proportional and integral) controller. Combined with a proportional controller, an integral controller can improve the performance of the feedback control system. However, care should be taken to avoid erroneous accumulations of control signals by the integrator which can lead to a substantial overshoot later [10]. For this purpose, the integrator antiwindup technique [10] is applied as follows.

- **Case 1** ($\Delta U_{MR} > \Delta U_{util}$): All integrators of $MR - LOOP_k$ $(0 \leq k \leq 1)$ are turned off, since the current $\Delta U = \Delta U_{util}$.

- **Case 2** ($\Delta U_{MR} \leq \Delta U_{util}$): In this case, the integrator of the utilization controller is turned off, since currently $\Delta U = \Delta U_{MR}$. For the miss ratio controllers, if both ΔU_0 and ΔU_1 are negative, turn on the integrators for both $MR-LOOP_0$ and $MR-LOOP_1$. This is because the current $\Delta U = \sum_{k=0}^{1} \Delta U_k$. Otherwise, turn off the integrator of $MR-LOOP_k$ $(k = 0 \; or \; 1)$ whose miss ratio control signal is larger than the other.

We have profiled the real-time main memory database (modeled in this paper) and tuned the feedback controllers based on the profiling results. Interested readers are referred to [6].

3.3 QoD Manager

In general, it is hard to meet both the timing and freshness constraints at the same time. High update workloads may increase the deadline miss ratio of user transactions; infrequent updates, however, reduce the QoD (database freshness) [2]. As a result, transactions may have to use stale data. (For this reason, we do not consider designing a separate feedback controller to manage the freshness. The specified miss ratio and freshness can pose conflicting requirements leading to a potentially unstable feedback control system.) To address the problem, we use the QoD manager (an actuator from the control theory perspective). When overloaded (i.e., $\Delta U < 0$), the QoD manager can degrade the current QoD to reduce the update workload and improve the deadline miss ratio. In fact, it might not be necessary to schedule all incoming sensor updates. Some data objects might be updated very frequently, but accessed infrequently. In contrast, other data objects can be accessed frequently within consecutive updates. Based on the access update ratio (AUR), we classify data as hot or cold: a data object is considered hot if the corresponding accesses are more frequent than the updates, i.e., AUR ≥ 1. Otherwise, it is considered cold. It is reasonable to update hot data in an aggressive manner. If a hot data object is out of date when accessed, potentially a multitude of transactions may miss their deadlines waiting for the update. For cold data objects, we can save the CPU utilization by applying a lazy update policy when overloaded. Only a few transactions might be affected by the update delay. We select *immediate* and *on-demand* policies as the aggressive and lazy update policies, respectively.

For example, assume that D is the set of the entire (sensor) data in a real-time database, D_{imm} is the set of data updated immediately when their new sensor readings arrive, and D_{od} is the set of data updated on demand. Initially, every data is updated immediately (i.e., $D = D_{imm}$). A subset of cold data can be updated on demand as the workload increases. The QoD is degraded, as a result. The QoD degradation is stopped once the required CPU utilization adjustment (ΔU) is achieved or the degradation bound (i.e., AUR = 1) is reached. The update policy is switched back to the immediate policy for a certain subset of data when the perceived freshness is violated. Using dynamic change of update policy, the QoD manager differentiates the QoD among hot and cold data classes when overloaded. For more details about the QoD management, refer to [7].

3.4 Update Scheduler

The update scheduler decides whether to schedule or drop an incoming update based on the update policy selected for a data object. Immediate updates will always be scheduled, whereas on-demand updates will be scheduled only if any transaction is blocked to access a fresh version of the corresponding data.

3.5 Admission Control

The admission controller is another system component that enforces the control signal in addition to the QoD Manager. After possible QoD adaptations, the admission controller is informed of the adjusted control signal, called ΔU_{new}, as shown in Figure 1. The admission control is necessary since the QoD Manager itself might not be able to enforce the required utilization adjustment (i.e., ΔU) entirely. The QoD degradation bound (AUR = 1) can be reached before achieving ΔU or the perceived freshness requirement is currently violated, therefore, no QoD degradation is possible. In contrast, more incoming transactions should be admitted when underutilized.

An incoming user transaction can be admitted if its estimated CPU utilization requirement is currently available. The current utilization is examined by aggregating the utilization estimates of previously admitted transactions which are in the same or higher service class(es). This is a *priority-aware* approach. A high priority class transaction is not rejected due to the low priority class transactions already admitted.

4 Performance Evaluation

In this section, we analyze the performance of our service differentiation approach. After the workloads used for our experiments are discussed, baseline approaches are introduced for performance comparisons and the performance evaluation results are presented.

4.1 Simulation Model

In our simulation, we apply a certain workload consisting of sensor updates and user transactions which are sum-

Table 1. Settings for Sensor Data/Updates

Parameter	Value
#Data Objects	1000
Update Period	$Uniform(100ms, 50s)$
EET_i	$Uniform(1ms, 8ms)$
Actual Exec. Time	$Normal(EET_i, \sqrt{EET_i})$
Total Update Load	$\approx 50\%$

Table 2. Settings for User Transactions

Parameter	Value
EET_i	$Uniform(20ms, 80ms)$
AET_i	$EET_i \times (1 + EstErr_i)$
Actual Exec. Time	$Normal(AET_i, \sqrt{AET_i})$
N_{DATA_i}	$0.5 \times EET_i = (10, 40)$
#Actual Data Accesses	$Normal(N_{DATA_i}, \sqrt{N_{DATA_i}})$

marized in Tables 1 and 2, respectively, and described as follows.

Sensor Data and Updates: There are 1000 sensor data objects in our simulated real-time database. Each data object O_i is periodically updated by an update stream, $Stream_i$, which is associated with an estimated execution time (EET_i) and an update period (P_i) where $1 \leq i \leq 1000$. EET_i and P_i are uniformly distributed in a range (1ms, 8ms) and in a range (100ms, 50sec), respectively. Upon the generation of an update, the actual update execution time is varied by applying a normal distribution $Normal(EET_i, \sqrt{EET_i})$ for $Stream_i$ to introduce errors in execution time estimates. The total update workload is manipulated to require approximately 50% of the total CPU utilization if every update is scheduled by the immediate update policy.

User Transactions: A source, $Source_i$, generates a group of user transactions whose inter-arrival time is exponentially distributed. $Source_i$ is associated with an estimated execution time (EET_i) and an average execution time (AET_i). We set $EET_i = Uniform(20ms, 80ms)$ [1]. By generating multiple sources, we can derive transaction groups with different average execution time and average number of data accesses in a statistical manner. We set $AET_i = (1 + EstErr) \times EET_i$, in which $EstErr$ is used to introduce the execution time estimation errors. Note that the simulator is only aware of the estimated execution time. Upon the generation of a user transaction, the actual execution time is generated by applying the normal distribution $Normal(AET_i, \sqrt{AET_i})$ to introduce the execution time variance in the user transaction group.

[1]We assume that user transactions execute longer than updates since they access data and perform arithmetic/logical operations based on the accessed data.

The number of data accesses for $Source_i$ is derived in proportion to the length of EET_i, i.e., N_{DATA_i} = *data access factor* $\times EET_i = (10, 40)$. As a result, longer transactions access more data in general. Upon the generation of a user transaction, $Source_i$ associates the actual number of data accesses with the transaction by applying $Normal(N_{DATA_i}, \sqrt{N_{DATA_i}})$ to introduce the variance in the user transaction group.

We set *deadline* = *arrival time* + *average execution time* $\times$ *slack factor* for a user transaction. A slack factor is uniformly distributed in a range (20, 40). For an update, we set *deadline* = *next update period*.

4.2 Baselines

To our best knowledge, no previous research has applied feedback control and QoD adaptations to provide the service differentiation in real-time databases. For this reason, we have developed two baselines as follows to compare the miss ratio and perceived freshness with *QMF-Diff*:

- *Basic-IMU*: In this approach, a basic service differentiation is provided by using the fixed priority scheduling among the service classes and the priority-aware admission control as described in Sections 3.1 and 3.5, respectively. The immediate update policy is used for all updates, therefore, it provides the highest possible QoD. All the shaded components in Figure 1 are turned off. Thus, the feedback control and QoD adaptations are not applied.

- *Basic-ODU*: This is similar to Basic-IMU except that every data is updated on demand. An update is scheduled if at least one user transaction is blocking to access the fresh version of the corresponding data. Because of lazy updates, the QoD (database freshness) is generally low compared to Basic-IMU approach. However, the user transaction miss ratio can be improved due to the relatively low update workload.

When the on-demand update policy is applied, it is possible that a blocked transaction can not finish in time to wait for an on-demand update. This problem can be handled in one of the two alternative ways: either aborting the corresponding user transaction, or allowing a stale data access to meet the transaction deadline [2]. The selection between the two alternatives is application dependent, that is, it depends on the criticalness of the stale data access in a specific real-time database application. In this paper, we take the latter approach. An in-depth comparison between the two alternatives is reserved for future work.

Table 3. Presented Experiments

Exp.	Varied	Fixed
1	$AppLoad =$ 70%, 100%, 150%, 200%	EstErr = 0 HSS = 50% $HCR = 20\%$
2	$HCR =$ 20%, 40%, 60%, 80%, 100%	AppLoad = 200% EstErr = 1 HSS = 50%

4.3 Experimental Goals

The main objective of our simulation study is to show whether or not our approach can provide the performance guarantees as specified in Section 2.3 (*QoS-Spec*). For this purpose, we vary the applied workloads from various aspects. The workload variables and the presented experiments are discussed below. The presented experiments are also summarized in Table 3.

4.3.1 Workload Variables

- *AppLoad*: Computational systems may show different performance for increasing loads, especially when overloaded. We use a variable, called $AppLoad$, to apply different workloads in the simulation. Note that this variable indicates the load assuming that all incoming transactions are admitted and all updates are immediately scheduled. The actual load can be reduced in a tested approach by applying the admission control and scheduling the updates according to the selected update policy. For performance evaluation, we applied $AppLoad = 70\%, 100\%, 150\%$, and 200%.

- *EstErr* (Execution Time Estimation Error): $EstErr$ is used to introduce errors in execution time estimates as described before. We have evaluated the performance for $EstErr$ = 0, 0.25, 0.5, 0.75, and 1. In general, a high execution time estimation error could induce a difficulty in real-time scheduling.

- *HSS* (Hot Spot Size): Database performance can vary as the degree of the data contention changes [1, 5]. For this reason, we apply different access patterns by using the $x - y$ access scheme [5], 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 (HSS). The performance is evaluated for $HSS = 10\%, 20\%, 30\%, 40\%$, and 50% (uniform access pattern).

- *HCR* (Highest Class Ratio): In general, the performance of a service differentiation scheme may change according to the high priority class workload. In our approach, MR_1 and MR_2 could increase as the Class 0 load increases due to the fixed priority scheduling applied among the service classes. To adjust the Class 0 workload, we define a workload variable:

$$HCR = 100 \times \frac{\textit{\#Class 0 User Transactions}}{\sum_{k=0}^{2} \textit{\#Class k User Transactions}} (\%).$$

We evaluate the performance for $HCR = 20\%$, 40%, 60%, 80%, and 100%.

4.3.2 Experiments

Even though we have performed a large number of experiments for varying values of $AppLoad$, $EstErr$, HSS and HCR, we present only two sets of experiments as shown in Table 3 due to the space limitation. We have verified that all the experiments including $Experiments$ 1 and 2 show a consistent performance trend: only our approach can provide guarantees on MR_0, MR_1, and perceived freshness, while the baselines fail to provide guarantees on miss ratios and/or perceived freshness in the presence of unpredictable workloads and access patterns. In that sense, we present $Experiments$ 1 and 2 that represent the two ends of the spectrum.

- *Experiment 1*: As described in Table 3, no error is considered in the execution time estimation. Note that this is an ideal assumption since precise execution time estimates are usually not available. Performance is evaluated for $AppLoad = 70\%$, 100%, 150%, and 200%. In all the other sets of experiments, we fix $AppLoad = 200\%$ to compare the adaptiveness of Basic-IMU, Basic-ODU, and QMF-Diff under overload. We also fix $HCR = 20\%$. Hence, the best case settings are applied for $Experiment$ 1.

- *Experiment 2*: In this set of experiments, the worst case settings are applied. We set $AppLoad = 200\%$ and $EstErr = 1$, i.e., the highest load and the largest execution time estimation error are applied. We also increase $HCR = 20\%$, 40%, 60%, 80%, and 100% to stress the modeled real-time database. As HCR increases, miss ratios, especially MR_1 and MR_2, may increase significantly.

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 the 90% confidence intervals. Confidence intervals are plotted as vertical bars in the graphs showing the performance evaluation results. (For some performance data, the vertical bars may not be noticeable due to the small confidence intervals.)

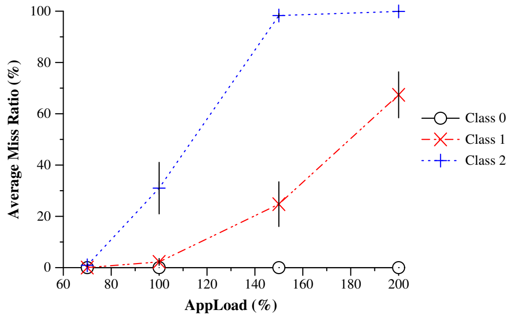

Figure 3. Average Miss Ratio for Basic-IMU

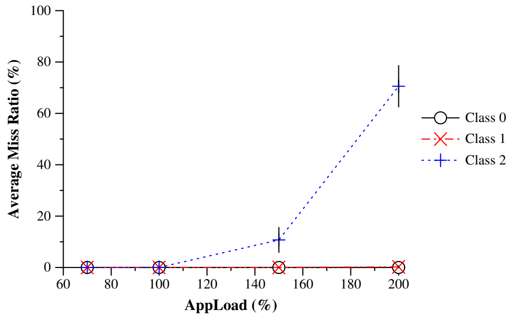

Figure 4. Average Miss Ratio for Basic-ODU

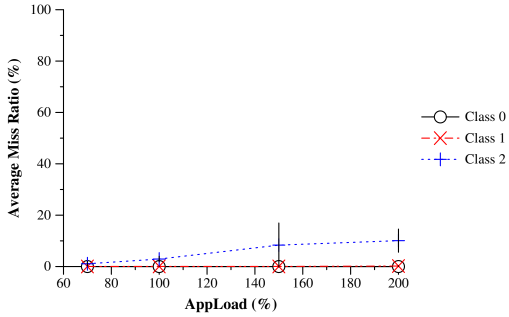

Figure 5. Average Miss Ratio for QMF-Diff

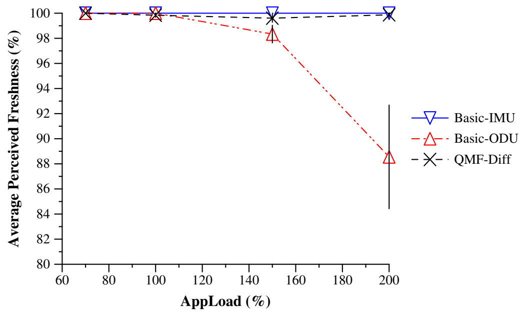

Figure 6. Average Perceived Freshness

4.4 Experiment 1: Effects of Increasing Load

In this section, we compare the performance of Basic-IMU, Basic-ODU, and QMF-Diff for increasing *AppLoad*. As shown in Figures 3, 4, and 5, all tested approaches have shown near zero MR_0 in terms of average. We have also observed that none of them has violated the specified MR_0 overshoot, 1.3% as specified in Section 2.3. This is mainly due to the best-case settings, especially because $EstErr = 0$ and $HCR = 20\%$. Therefore, concerning the miss ratio we only compare the average miss ratios among the service classes in the remainder of this section.

Average Miss Ratio: As shown in Figure 3, for Basic-IMU, MR_1 increases as the *AppLoad* increases violating MR_1 threshold (5%) when $AppLoad = 150\%$ and 200% due to the relatively high update workloads. The corresponding MR_1 values are $24.75 \pm 8.72\%$ and $67.38 \pm 8.96\%$, respectively, and MR_2 values are near 100%. In contrast, both Basic-ODU and QMF-Diff have limited MR_0 and MR_1 below the specified thresholds achieving near zero MR_0 and MR_1 as shown in Figures 4 and 5.

Perceived Freshness: Basic-IMU provides 100% perceived freshness as shown in Figure 6. This is because every update is scheduled immediately in Basic-IMU. QMF-Diff supports near 100% perceived freshness. The lowest freshness is $99.6 \pm 0.32\%$ when $AppLoad = 150\%$ exceeding the $PF_{target} = 98\%$. However, Basic-ODU has failed to support the PF_{target}. When $AppLoad = 200\%$, it showed $88.55 \pm 4.13\%$ perceived freshness. This is because every data is updated on demand in Basic-ODU. As a result, many user transactions are forced to read stale data to meet their deadlines. To verify this, we have measured the average QoD for Basic-ODU in 10 simulation runs when $AppLoad = 200\%$. The average QoD provided by Basic-ODU was only $9.1 \pm 0.08\%$.

Average Utilization: In Figure 7, Basic-IMU shows the highest utilization among the tested approaches due to the high update workload. As shown in Figure 7, Basic-ODU shows the significant underutilization until $AppLoad = 100\%$ due to unscheduled updates. Both in Basic-IMU and Basic-ODU, the utilization increases sharply as the *AppLoad* increases leading to potential overload, in which MR_1 threshold or the PF_{target} is violated, respectively, as shown in Figures 3 and 6. In contrast, QMF-Diff shows a relatively stable utilization ranging from $66.43 \pm 0.83\%$ to $86.19 \pm 0.82\%$ as *AppLoad* increases from 70% to 200% as shown in Figure 7. Note that in $Experiment$ 1 only QMF-Diff can support both the PF_{target} and miss ratio guarantees on MR_0 and MR_1, while Basic-IMU and Basic-ODU fail to provide MR_1 and perceived freshness guarantees, respectively. Our approach also produces the lowest miss ratio for Class 2 among the tested approaches, e.g., when $AppLoad = 200\%$ QMF-Diff has $10.08 \pm 4.5\%$ MR_2 (Fig-

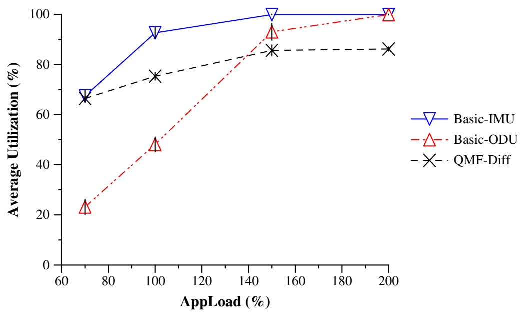

Figure 7. Average Utilization

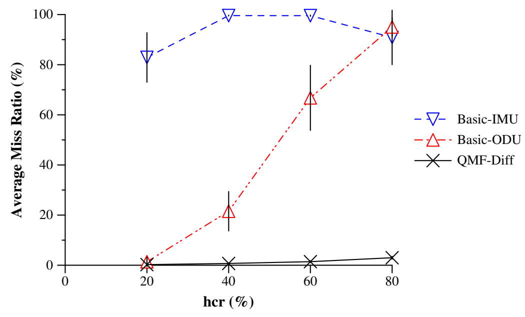

Figure 9. Average Miss Ratio for Class 1

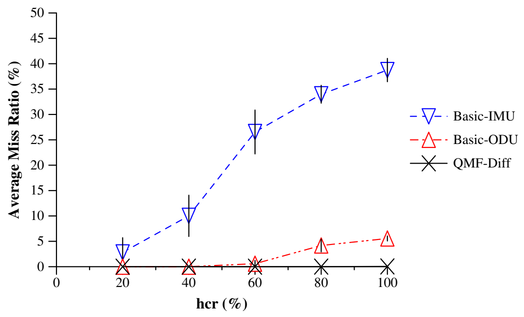

Figure 8. Average Miss Ratio for Class 0

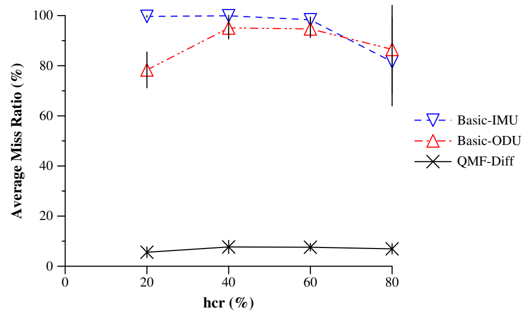

Figure 10. Average Miss Ratio for Class 2

ure 5), while Basic-IMU and Basic-ODU have near 100% and 70% miss ratios for Class 2, respectively (Figures 3 and 4).

4.5 Experiment 2: Effects of Increasing the Highest Class Load

The HCR value and the corresponding performance (especially in terms of MR_1 and MR_2) might be time-varying. To quantify this, we evaluate performance for HCR values increasing from 20% to 100%. When HCR becomes 100%, every transaction belongs to Class 0 and the service differentiation is not applicable, as a result. In this section, we mainly focus on miss ratio comparisons among Basic-IMU, Basic-ODU, and QMF-Diff. Perceived freshness has also been measured, but it is not plotted in this section to avoid repetition. Concerning the perceived freshness, Basic-IMU and QMF-Diff have shown near 100% perceived freshness for all tested HCR values, while Basic-ODU has failed to support the PF_{target}, similar to the results described in the previous section. QMF-Diff has shown 100% perceived freshness except when HCR = 80% in which the perceived freshness is $99.92 \pm 0.15\%$.

Average Miss Ratio: As shown in Figure 8, for Basic-IMU and Basic-ODU the average MR_0 continuously increases as HCR increases, violating the specified MR_0 threshold (1%). In contrast, for QMF-Diff the average MR_0 is maintained near zero despite the increasing HCR as shown in Figure 8.

In Figure 9, the average MR_1 is plotted for increasing HCR values except when HCR = 100%, in which there is neither a Class 1 nor a Class 2 transaction. Both Basic-IMU and Basic-ODU show significant violations of the MR_1 threshold (5%) as HCR increases. Especially, Basic-ODU, which showed a good performance in terms of MR_0 and MR_1 in *Experiment* 1, fails to support both MR_0 and MR_1 guarantees as shown in Figures 8 and 9. In contrast, QMF-Diff supports the MR_0 and MR_1 guarantees. Our approach shows near zero MR_0 and the worst $MR_1 = 2.99 \pm 0.55\%$ when $HCR = 80\%$ as shown in Figures 8 and 9. As shown in Figure 10, our approach also shows the lowest miss ratio for Class 2, similar to the performance evaluation results presented in the previous section. In Figures 9 and 10, for a few cases MR_1 and MR_2 decrease when HCR increases from 60% to 80%. This is due to the priority-aware admission control described in Section 3.5. As HCR increases, more Class 0 transactions arrive. As a result, lower priority transactions have a less chance to get admitted and the deadline miss ratio for those admitted may decrease. As shown in Figures 8 and 9, for QMF-Diff the difference between MR_0 and MR_1 is below 3% satisfying the allowed maximum difference (4%) described in *QoS-Spec*. (The average MR_0 is near zero and the highest MR_1 is $2.99 \pm 0.55\%$ when $HCR = 80\%$.)

Average Utilization: As shown in Figure 11, for Basic-IMU and Basic-ODU the utilization is near 100%,

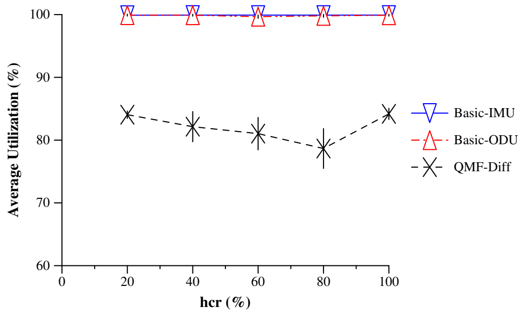

Figure 11. Average Utilization

whereas for QMF-Diff the utilization is approximately between $80\% - 90\%$. In the baseline approaches, the simulated real-time database is overloaded leading to the violations of the specified MR_0 and MR_1 guarantees as discussed before. QMF-Diff exceeds the lower utilization set point of 80% except when $HCR = 80\%$ as shown in Figure 11. In QMF-Diff, the utilization decreases slightly as HCR increases. This is because $MR - LOOP_1$ requests relatively large utilization reductions for the increasing MR_1. The utilization increases again when $HCR = 100\%$, in which $MR - LOOP_1$ does not request any utilization reduction since no Class 1 transactions are generated. QMF-Diff satisfies our main objective which is to provide guarantees on MR_0, MR_1, and perceived freshness without severely underutilizing the CPU instead of maximizing the CPU utilization. We have also measured the transient performance for the tested approaches. For the transient MR_0, QMF-Diff closely satisfy the specified overshoot and settling time for varying HCR values while the baseline approaches fail. For more details, refer to [6].

5 Related Work

Service differentiation techniques have been widely studied in various computational systems such as web servers, network routers, and proxy caches [3, 4, 9]. However, the related research is relatively scarce in real-time databases despite the increasing service demand. Our admission control and fixed priority scheduling applied among the service classes are similar to a service differentiation scheme developed in the context of a web server [3]. Feedback control has been increasingly applied to QoS management and real-time scheduling recently [8, 9, 12]. However, to our best knowledge none of the existing service differentiation models/control-theoretical approaches considered service differentiation issues in real-time databases regarding both timing and data freshness constraints.

6 Conclusions and Future Work

In many applications, it is desirable for users to receive guaranteed real-time database services. Using QMF-Diff, a DBA can explicitly specify the required database QoS including the miss ratio differentiation among the service classes. According to the experimental results, our approach can provide the specified QoS when the baseline approaches fail to support the miss ratio and/or freshness guarantees in the presence of unpredictable workloads and access patterns. Our approach also shows the relatively low miss ratio in the basic and best-effort classes compared to the baseline approaches, thereby reducing potential starvation. As the demand for guaranteed real-time database services increases, the benefits of our approach will be more significant.

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] N. Bhatti and R. Friedrich. Web Server Support for Tiered Services. In *IEEE Network*, September 1999.

[4] C. Dovrlois, D. Stiliadis, and P. Ramanathan. Proportional Differentiated Services: Delay Differentiation and Packet Scheduling. In *SIGCOMM*, Aug 1999.

[5] M. Hsu and B. Zhang. Performance Evaluation of Cautious Waiting. *ACM Transactions on Database Systems*, 17(3):477–512, 1992.

[6] K. D. Kang, S. H. Son, and J. A. Stankovic. Service Differentiation in Real-Time Main Memory Databases. Technical Report CS-2002-02, Computer Science Department at University of Virginia, 2002.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

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