Differentiated Real-Time Data Services for E-Commerce Applications *

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 e-commerce data services has been increasing recently. In many e-commerce applications, it is essential to process user requests within their deadlines, i.e., before the market status changes, using fresh data reflecting the current market status. However, current data services are poor at processing user requests in a timely manner using fresh data. To address this problem, we present a differentiated real-time data service framework for e-commerce applications. User requests are classified into several service classes according to their importance, and they receive differentiated real-time performance guarantees in terms of deadline miss ratio. At the same time, a certain data freshness is guaranteed for all transactions that commit within their deadlines. A feedback-based approach is applied to differentiate the deadline miss ratio among service classes. Admission control and adaptable update schemes are applied to manage potential overload. A simulation study, which reflects the e-commerce data semantics, shows that our approach can achieve a significant performance improvement compared to baseline approaches. Our approach can support the specified per-class deadline miss ratios maintaining the required data freshness even in the presence of unpredictable workloads and data access patterns, whereas baseline approaches fail.

*Supported, in part, by NSF grants EIA-9900895, CCR-0098269, and IIS-0208758.

1 Introduction

The demand for real-time data services has been increasing recently. Many e-commerce applications are becoming very sophisticated in their data needs. They span the spectrum from low level status data, e.g., current stock prices, to high level aggregated data, e.g., recommended selling/buying points. In e-commerce applications, it is desired to process transactions within their deadlines using fresh (temporally consistent) data reflecting the current market status. By meeting these timing and data freshness constraints, merchandise such as stock items can be traded in a timely manner before the current market conditions change, i.e., real-time data services are provided.

It is very challenging to provide real-time data services because of data-dependent resource requirements and highly uncertain workloads in e-commerce applications. For example, transactions in stock trading may read different sets of stock prices and perform different operations based on the current market status. Furthermore, the number of service requests can vary from time to time. Consequently, developing real-time data services should involve techniques for managing overload and unexpected failures to support the QoS requirements expressed in terms of deadline miss ratio and data freshness. However, current (non-real-time) databases are poor at supporting timing and data temporal constraints.

To address this problem, we present a novel real-time database QoS management framework called **Diff-Real** (Differentiated Real-time data services for e-commerce applications). Diff-Real provides performance guarantees in terms of differentiated deadline miss ratio and data freshness even in the presence of unpredictable workloads and data access patterns. Despite the abundance of real-time database research, Diff-Real is the first approach to provide guarantees on differentiated miss ratio and data freshness.

In our approach, user requests are classified into three classes according to their importance such as payments, namely, Classes 0, 1, and 2 where Class 0 is the highest priority class. This could model a typical e-commerce system, in which users can be classified into premium, business, and economy (or free-trial) classes. For Classes 0 and 1, we provide differentiated guarantees on their miss ratios to support real-time data services even under transient system status, while trying to meet as many Class 2 deadlines as possible (without any guarantee). Under overload, we can improve the system performance and provide a basic support to maximize the profit by providing preferred services to the high priority class(es).

Feedback control is the key technology we use to achieve differentiated miss ratio guarantees. An adaptable update scheme maintains the required data freshness for timely transactions – transactions that commit within their deadlines – in a cost-effective manner. Admission control is applied to prevent potential overload, which can lead to the loss of profit due to many deadline misses in e-commerce transaction processing.

For performance evaluation, we designed a simulation model inspired by a study of the online trading environment available at the Bridge Center for Financial Markets, University of Virginia (discussed in Section 4). In the simulation study, our approach showed a significant performance improvement compared to the baseline approaches. In our approach, the required per-class miss ratios and data freshness are supported, whereas the baseline approaches fail to support the specified QoS in the presence of unpredictable workloads and access patterns. Compared to the baseline approaches, our approach can also reduce the potential starvation for the least privileged Class 2.

The rest of this paper is organized as follows. In Section 2, real-time database model is discussed. Performance metrics and QoS specification issues are also discussed. Section 3 gives a detailed discussion of our differentiated service framework. Performance evaluation results are presented in Section 4. Related work is presented in Section 5 and Section 6 concludes the paper and discusses avenues for future work.

2 Real-Time Database Model and QoS Specification

In this paper, we assume that some deadline misses or freshness violations are inevitable due to unpredictable workloads and data access patterns in e-commerce applications as discussed before. A few deadline misses or freshness violations are considered tolerable as long as they do not exceed certain thresholds. Using our approach, a database administrator (DBA) can explicitly specify the required database QoS in terms of miss ratio and freshness, and the system supports the specified QoS. In this section, we describe the real-time database model and real-time transaction semantics adopted in Diff-Real. Average and transient performance metrics are defined to represent the miss ratio and data freshness perceived by users. QoS specification issues are discussed in terms of defined performance metrics.

2.1 Real-Time Database Model and Transaction Semantics

In this paper, we consider a main memory database model. Due to their relatively high performance and decreasing main memory cost, main memory databases have been increasingly applied to real-time data management such as online auction, stock trading, and voice/data networking [2, 7, 21, 27]. In our main memory database model, the CPU is the system resource of main consideration. Transactions are classified as either user transactions and data updates. Continuously changing real-world states are captured by periodic updates of temporal data, e.g., the current stock prices, which can become outdated according to the passage of time. User transactions execute arithmetic/logical operations, e.g., buy/sell operations, based on the current real-world states reflected in the real-time database. We assume that each user transaction has a deadline. The deadline of an update transaction is set to the corresponding update period. Our target applications are firm real-time data services, in which tardy transactions – transactions that have missed their deadlines – add no value to the system, and therefore, are aborted upon their deadline misses. This is because tardy transactions, e.g., late sell/buy operations after the market state has changed, can adversely affect the profit wasting system resources.

2.2 Average Performance Metrics

In Diff-Real, two main performance metrics are considered to specify the real-time database QoS: *per-class deadline miss ratio* and *freshness* of data accessed by timely transactions.

- **Miss Ratio**: Let $\texttt{\#Tardy}_\texttt{i}$ and $\texttt{\#Timely}_\texttt{i}$ represent the number of tardy and timely transactions for admitted transactions belonging to Class *i*. The Class *i* miss ratio is defined as:

$$MR_i = 100 \times \frac{\texttt{\#Tardy}_\texttt{i}}{\texttt{\#Tardy}_\texttt{i} + \texttt{\#Timely}_\texttt{i}} \ (\%)$$

- **Freshness**: Data in real-time databases can become outdated due to the passage of time, e.g., current stock prices. Thus, it is important for a real-time database to continuously update the (temporal) data to maintain the temporal consistency between the real-world states and the values reflected in the database. To measure the freshness of data in real-time databases, we use the notion of absolute validity intervals [23]. 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. Therefore, absolute validity interval is the length of the time a data object remains fresh. We further classify the notion of data freshness into: *database freshness* and *perceived freshness* [10]. 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 for the data accessed by timely transactions 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.

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

In a QoS specification only the perceived freshness is considered, since tardy transactions add no value to our firm real-time database model. In this way, we can leverage the inherent leeway in the QoD. Under overload, the QoD can be traded off for a certain subset of data to reduce the update workload as long as the perceived freshness requirement is not violated. As a result, the deadline miss ratio of user transactions can be improved without affecting the perceived freshness (Section 3.3).

2.3 Transient Performance Metrics

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 [15]:

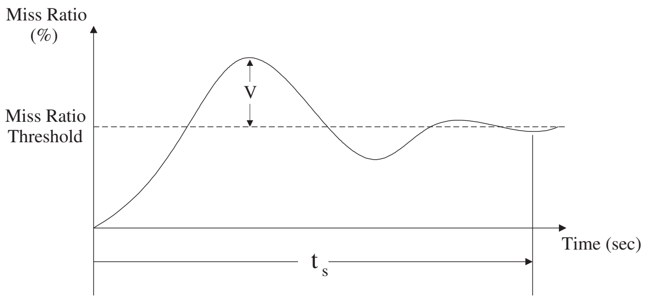

Figure 1: Definition of Overshoot and Settling Time in Real-Time Databases

- **Overshoot** (V) is the worst-case system performance in the transient system state. As shown in Figure 1, in this paper it is considered the highest miss ratio over the miss ratio threshold in the transient state.
- **Settling time** (t_s) is the time for the transient overshoot to decay and reach the steady state performance as shown in Figure 1. In the steady state, the miss ratio should be below the miss ratio threshold.

2.4 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 data service application. In this paper, we consider the following QoS specification as an example to illustrate the applicability of our approach for service differentiation:

"*QoS-Spec* = { [$MR_0 \leq 1\%$, $V_0 \leq 20\%$, $T_0 \leq 20sec$], [$MR_1 \leq 5\%$, $V_1 \leq 50\%$, $T_1 \leq 40sec$], $MR_2 =$ *best-effort*, $PF_{target} \geq 98\%$, $U \geq 80\%$ }"

This QoS specification requires us to control the average miss ratio below 1% (5%) for Class 0 (1). We also set $V_0 \leq 20\%$, therefore, a MR_0 overshoot (V_0) should not exceed $1.2\% = 1\% \times (1+0.2)$. Similarly, MR_1 overshoot (V_1) should be below $7.5\% = 5\% \times (1+0.5)$. An overshoot V_0 (V_1), if any, should decay within 20sec (40sec). Therefore, the system should be back in the steady state after 20sec (40sec), i.e., think time for trades, even if it is currently in the transient state. By enforcing *QoS-Spec*, the desired system performance can be provided in a consistent manner for Classes 0 and 1, while best-effort services are provided for Class 2. Regarding freshness, at least 98% perceived freshness is required for timely transactions (regardless of their service classes). We also require the CPU utilization to be at least 80% to prevent underutilization.

In our previous work [9], we presented an approach for service differentiation in real-time databases. However, in [9] the transient miss ratio guarantee is only provided for Class 0. In this paper, we extend the feedback-based miss ratio controllers to support the transient miss ratio guarantees for both Classes 0 and 1. It is challenging to provide average/transient miss ratio guarantees for both Classes 0 and 1. Class 1 performance can fluctuate as Class 0 workload increases, since Class 0 transactions receive preferred services in terms of scheduling and concurrency control for service differentiation purposes. (A detailed discussion of scheduling, concurrency control, feedback control is given in Section 3.) For this reason, as shown in *QoS-Spec* we set relatively flexible performance requirements for Class 1 compared to Class 0 in terms of average/transient miss ratio.

In general, *QoS-Spec* is a stringent QoS requirement considering the specified average/transient miss ratio requirements and the required data freshness. Further, the maximum allowed performance difference in terms of average miss ratio between Classes 0 and 1 is only 4%. Not only MR_0 but also MR_1 should meet the required overshoot and settling time. Note that these requirements may not leave a large leeway for temporary relaxations of MR_1, if necessary. 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.

3 Differentiated Real-Time Data Service Architecture

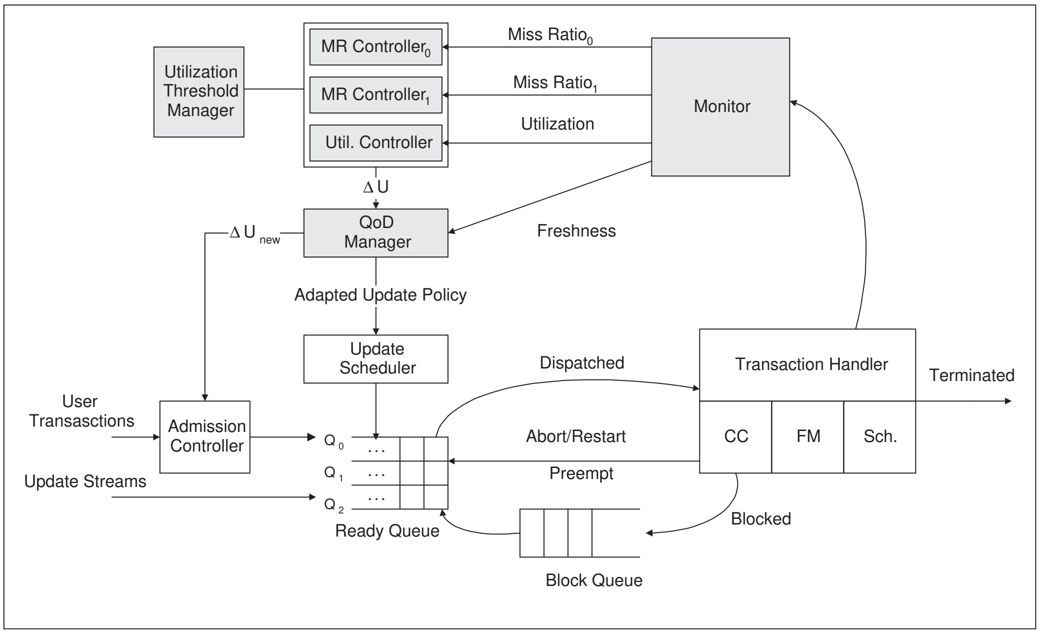

Figure 2: Real-Time Database Architecture for Service Differentiation

Our differentiated service architecture is shown in Figure 2. Transaction are scheduled in one of the multi-level ready queues according to their service classes, that is, Class i $(0 \leq i \leq 2)$ transactions are scheduled at Q_i. The transaction handler executes queued transactions. At each sampling instant, the current miss ratio, perceived freshness, and CPU utilization are monitored. The miss ratio and utilization controllers derive the required CPU utilization adjustment (ΔU in Figure 2) considering the current performance error such as the miss ratio overshoot or CPU underutilization. Based on ΔU, the QoD manager adapts the QoD (i.e., database freshness), 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}. A detailed discussion of each component is as follows.

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 extends the widely accepted 2PL protocol to eliminate priority inversions. Hence, it is appropriate to support service differentiation in real-time databases.

The FM checks the freshness before accessing a data item using the corresponding validity interval. 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 2) 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 an EDF manner. To provide the data freshness guarantee, all temporal data updates are scheduled at Q_0 in this paper. This is to minimize the possibility for user transactions to miss their deadlines waiting for updates of data, which are currently outdated.

By applying the fixed priority among the service classes, we provide a basic support for service differentiation in real-time databases. However, this is insufficient to provide guarantees on MR_0, MR_1, and perceived freshness in the presence of unpredictable workloads/access patterns. For this reason, we apply the feedback control and QoD management as follows.

3.2 Feedback Control

Feedback control is very effective in supporting a required performance specification when the system model includes uncertainties [20]. The target performance can be achieved by dynamically adapting the system behavior based on the current performance error measured in the feedback control loop. In this paper, feedback control is applied to provide average/transient guarantees on MR_0 and MR_1 despite unpredictable workloads and access patterns.

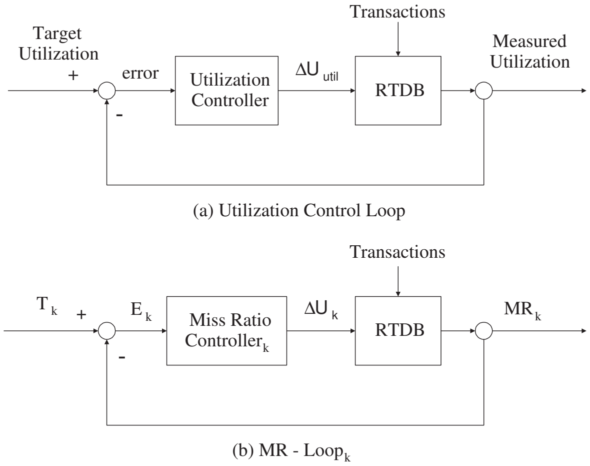

Figure 3: Miss Ratio/Utilization Controllers

3.2.1 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 3 (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 $E_k(n) = T_k - MR_k(n)$, i.e., the difference between the specified miss ratio threshold for Class k and the Class k miss ratio measured at the n^{th} sampling instant:

$$\Delta U_k = KP \times E_k(n) + KI \times \sum_{i=1}^{n} E_k(i) \tag{1}$$

When overloaded, ΔU_k can become negative to request the reduction of the CPU utilization.

3.2.2 Profiling and Controller Tuning

To support the specified average/transient MR_0 and MR_1, miss ratio controllers are tuned as follows.

- $MR-LOOP_0$: To tune a miss ratio controller, the miss ratio gain $G_{MR} = Max\{\frac{Miss\ Ratio\ Increase}{Unit\ Load\ Increase}\}$, should be derived under the worst case set-up to support a certain miss ratio guarantee [16]. To derive G_{MR}, the performance of the controlled system, i.e., a real-time database in this paper, should be profiled. In the profiling, to model the worst case all incoming user transactions are admitted and no QoD adaptation is performed regardless of the current system status. Average MR_0 was measured for loads increasing from 60% to 200% by 10%. From this, we derived the MR_0 gain $G_{MR_0} = 1.586$ when the load increases from 120% to 130%. Using G_{MR_0}, we applied the Root Locus design method in Matlab [20] to tune KP and KI to satisfy the average/transient MR_0 specified in *QoS-Spec*. By using the mathematically well established Root Locus method, we can avoid tuning/testing iterations. The sampling period for feedback control is set to 5sec. We have selected the closed loop poles at $p_0, p_1 = 0.778, 0.539$. The feedback control system is stable, since the closed loop poles are inside the unit circle. The corresponding values are KP = 0.183 and KI = 0.176.

- $MR - LOOP_1$: In this paper, it is also necessary to profile MR_1, since we aim to support not only the required average, but also the transient miss ratio for Class 1 unlike our previous approach [9]. For $MR - LOOP_1$, we measured MR_1 for the same workloads described above. For MR_1, the gain G_{MR_1} = 3.084 when the load increases from 80% to 90%. Note that G_{MR_1} is higher than G_{MR_0}, and the sharpest MR_1 increase is observed at lower loads (between $80\% - 90\%$) compared to Class 0. This is mainly because the CPU scheduling and concurrency control are performed in favor of Class 0 for service differentiation. To tune $MR - LOOP_1$, we applied the Root Locus method to support the average/transient MR_1 specified in *QoS-Spec*, similar to $MR - LOOP_0$ tuning. The closed loop poles are located inside the unit circle for the stability of $MR - LOOP_1$. The corresponding values are KP = 0.094 and KI = 0.176.

3.2.3 Utilization Controller

One utilization control loop is employed to prevent a potential underutilization, similar to [9, 16]. 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 3 (a). At the n_{th} sampling instant, the utilization control signal is:

$$\Delta U_{util} = KP \times E(n) + KI \times \sum_{i=1}^{n} E(i) \tag{2}$$

The utilization controller is also tuned using the Root Locus method, but we do not discuss the details due to space limitations. The utilization controller is further extended by employing the utilization threshold manager as follows.

3.2.4 Utilization Threshold Manager

For many complex real-time systems, the schedulable utilization bound is unknown or can be very pessimistic [16]. In real-time databases, the utilization bound is hard to derive, if it even exists. This is partly because database applications usually include unpredictable workloads/access patterns and 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 3 (a)) is dynamically adjusted considering the current real-time system behavior as follows. Initially, the utilization threshold is set to an initial 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, e.g., 2%, unless the resulting utilization threshold is over 100%. The utilization threshold will be continuously increased as long as no deadline miss is observed. The utilization threshold will be switched back to the initial utilization set point as soon as the miss ratio controller takes the control. This back-off scheme could be considered conservative, however, we take this approach to prevent a potential miss ratio overshoot due to an overly 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.

3.2.5 Derivation of a Single Control Signal

For the consistency of feedback control, we derive a single control signal ΔU from two miss ratio control signals (ΔU_k where $0 \leq k \leq 1$) and one utilization control signal (ΔU_{util}) as follows.

From ΔU_0 and ΔU_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 the 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 [9, 16]. After deriving ΔU_{MR}, we set the current control signal $\Delta U = Minimum(\Delta U_{util}, \Delta U_{MR})$ for a similar reason.

3.2.6 Integrator Antiwindup

All feedback controllers shown in Figure 3 are digital PI (proportional and integral) controllers. 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 [20]. For this purpose, the integrator antiwindup technique [20] is applied as follows.

- **Case 1** ($\Delta U_{MR} > \Delta U_{util}$): Turn on the integrator for the utilization controller, but turn off all integrators of $MR - LOOP_k$ $(0 \leq k \leq 1)$, 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, only turn on the integrator of $MR - LOOP_k$ ($k = 0$ or 1) whose miss ratio control signal is smaller than the other.

3.3 QoD Manager and Update Scheduler

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 data freshness. As a result, transactions may have to use stale data [2]. 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. Instead, we use the QoD manager, an actuator from the control theory perspective, to dynamically balance potentially conflicting miss ratio and freshness requirements.

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, as a result. In fact, it might not be necessary to schedule all incoming temporal data updates. Some data objects can 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.

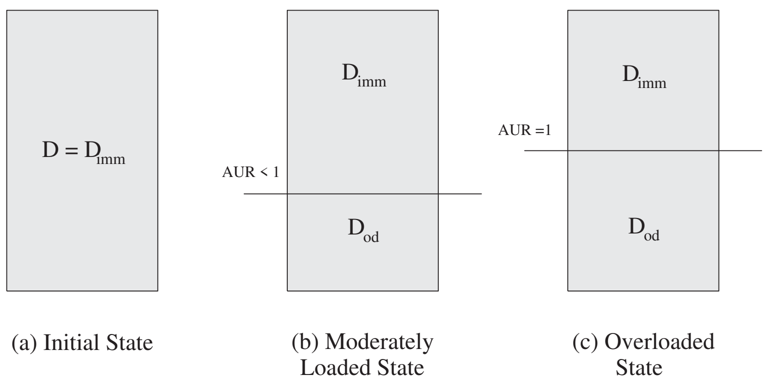

Figure 4: Update Policy Adaptations

For example, consider Figure 4. D is the set of the entire (temporal) data in a real-time database. D_{imm} is the set of data updated immediately when their new values arrive. In contrast, 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 the 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 [10].

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.

3.4 Admission Control

In addition to the QoD manager, the admission controller is another system component that enforces the control signal to prevent a miss ratio overshoot, which can lead to the loss of profit. 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.

In our approach, admission control is necessary, since under severe overload the QoD manager itself might not be able to enforce the required utilization reduction (i.e., $\Delta U < 0$) 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. Note that under overload it is impossible to meet the required per-class miss ratios and/or data freshness if all incoming transactions are simply admitted. Instead, it is reasonable to control the admission to improve the miss ratio/data freshness. Dropped user requests can be resubmitted later under appropriate market status rather than jeopardizing the database QoS for all user transactions currently in the system. Also, in Diff-Real the number of dropped transactions is reduced by applying QoD adaptation before admission control (Figure 3), if necessary. In contrast, more incoming transactions should be admitted when underutilized, i.e., $\Delta U > 0$. To enforce the remaining control signal after possible QoD adaptations, if any, the admission controller is informed of the adjusted control signal, i.e., ΔU_{new} as shown in Figure 2.

4 Performance Evaluation

In this section, we analyze the performance of our service differentiation approach. For performance evaluation, we have developed a real-time database simulator, which models the real-time database architecture depicted in Figure 2. Each system component in Figure 2 can be selectively turned on/off for performance evaluation purposes. The main objective of our simulation study is to show whether or not our approach can provide the performance guarantees as required in *QoS-Spec*, that is, per-class miss ratios are below the specified thresholds and the perceived freshness requirement is satisfied even in the presence of unpredictable loads and data access patterns. 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 workloads consisting of temporal data updates and user transactions described as follows.

Table 1: Simulation Settings for Data and Updates

Parameter	**Value**
#Data Objects	1000
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	$\approx 50\%$

4.1.1 Temporal Data and Updates

As shown in Table 1, there are 1000 temporal 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.

Table 2: Average Inter-Trade Times for S&P Stock Items

Stock Item	$\texttt{Item}_1$	$\texttt{Item}_2$	$\texttt{Item}_3$	$\texttt{Item}_4$	$\texttt{Item}_5$	$\texttt{Item}_6$
Average Time	189ms	4.41sec	8.84sec	16.89sec	22.03sec	25.99sec

Market status data such as stock prices can vary at arbitrary times when individual trades occur. However, financial trading tools such as Moneyline Telerate Plus [26], which is widely used in financial trading laboratories such as the Bridge Center for Financial Markets at University of Virginia and Sloan Trading Room at MIT, provides periodic price updates for periodic market monitoring. In Telerate Plus, users can select the update period in the range from 1 minute to 60 minutes for the real-time quote of an individual stock item. In this paper, we consider a more advanced system set-up, in which the update period ranges between 100ms $-$ 50sec. We have also studied the actual trace of real-time stock prices streamed to the Bridge Center for Financial Markets, and confirmed that our update periods can closely approximate NYSE trades for popular stock items. From 06/03/02 to 06/26/02, we measured the average time between two consecutive trades streamed into the Bridge Center for tens of S&P 500 stock items. In Table 2, most representative ones are presented. The other stock items not presented in Table 2 showed similar inter-trade times. As shown in Table 2, the shortest average inter-trade time observed is 189ms for $\texttt{Item}_1$, while the longest one observed is 26sec for $\texttt{Item}_6$[1].

[1] We have deleted the actual stock symbols for security purposes.

Table 3: Simulation Settings for User Transactions

Parameter	Value
EET_i (Estimated Execution Time)	$Uniform(5ms, 20ms)$
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)	$EET_i \times Data\ Access\ Factor = (5, 20)$
#Actual Data Accesses	$Normal(N_{DATA_i}, \sqrt{N_{DATA_i}})$

As shown in Table 1, the shortest update period selected for our experiments, i.e., 100ms, is approximately one half of the average inter-trade time of $\texttt{Item}_1$. In contrast, the longest update period of 50sec is approximately twice the average inter-trade time of $\texttt{Item}_6$ to model a wider range of update periods.

4.1.2 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 $EET_i = Uniform(5ms, 20ms)$ as shown in Table 3. By generating multiple sources, we can derive transaction groups with different average execution time and average number of data accesses in a statistical manner. Also, by increasing the number of sources we can increase the workload applied to the simulated database, since more user transactions will arrive in a certain time interval. We set $AET_i = (1 + EstErr) \times EET_i$, in which $EstErr$ is used to introduce the execution time estimation errors. Note that Diff-Real and all baseline approaches are 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 one group of user transactions generated by $Source_i$.

The number of data accesses for $Source_i$ is derived in proportion to the length of EET_i, i.e., $N_{DATA_i} = data\ access\ factor \times EET_i = (5, 20)$. 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 (10, 20). For an update, we set *deadline = next update period*.

For performance evaluations, we have also applied other settings for execution time, data access factor, and slack factor different from the settings given in Table 3. We have confirmed that for different workload settings Diff-Real can also support *QoS-Spec* by dynamically adjusting the system behavior based on the current performance error measured in feedback control loops. However, we do not include the results due to space limitations.

4.2 Baselines

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

- *Basic-IMU*: In this approach, a basic service differentiation is provided by using the fixed priority scheduling among the service classes and concurrency control policy without priority inversion, i.e., 2PL-HP, discussed in Section 3. The admission control scheme, described in Section 3, is applied to partially prevent overload. All temporal data are updated immediately when their values are newly observed, therefore, the highest possible QoD can be provided. The QoD adaptation and feedback-based closed loop scheduling are not applied. Therefore, all the shaded components in Figure 2 are turned off.

- *Basic-ODU*: This is similar to Basic-IMU except that all temporal data are updated on demand to reduce the update workload. The miss ratio could be improved at the expense of potentially reduced QoD compared to Basic-IMU because of lazy updates. In Diff-Real, the QoD can be dynamically adjusted, if necessary, between the highest value supported by Basic-IMU and the lowest value provided by Basic-ODU to meet both timing and freshness constraints.

In our experiments, we have also considered other baseline approaches. These approaches are similar to Basic-IMU and Basic-ODU except that admission control is not applied. Note that these baselines capture commonly accepted database system dynamics. In most of database systems, the feedback-based closed loop scheduling is not applied. The database update policy is usually fixed and not adaptable considering the system status. Further, admission control is rarely applied to manage potential overload. Compared to these baselines, Basic-IMU and Basic-ODU showed the lower miss ratio, i.e., smaller chances for profit loss, because of admission control. Due to the relatively poor performance of these baselines without admission control, we only compare the performance of Basic-IMU and Basic-ODU to Diff-Real.

4.3 Workload Variables and Experiments

To adjust the workload from various aspects for experimental purposes, we define the workload variables and describe the performed experiments as follows.

4.3.1 Workload Variables

- *AppLoad*: Computational systems generally 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 QoD management. 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. When $EstErr = 0$, there is no error in execution time estimates. When $EstErr$ = 1, the actual execution time is approximately twice the estimated execution time, since actual execution time $\approx (1 + EstErr) \times EET$. In general, a high execution time estimation error could induce a

Table 4: Presented Experiments

Experiment	Vary	Fix
1	$AppLoad = 70\%, 100\%, 150\%, 200\%$	EstErr = 0 HCR = 20% HSS = 50%
2	$EstErr = 0, 0.25, 0.5, 0.75, 1$	AppLoad = 200% HCR = 20% HSS = 50%
3	$HCR = 20\%, 40\%, 60\%, 80\%, 100\%$	AppLoad = 200% EstErr = 1 HSS = 50%

difficulty in real-time scheduling.

- *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 scheduling and concurrency control in favor of Class 0. To adjust the Class 0 workload, we define a workload variable:

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

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

- *HSS* (Hot Spot Size): Database performance can vary as the degree of data contention changes [1, 8]. For this reason, we apply different access patterns by using the $x - y$ access scheme [8], in which x% of data accesses are directed to y% of the entire data in the database and $x \geq y$. For example, under 90-10 access pattern 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).

4.3.2 Presented Experiments

Even though we have performed a large number of experiments for varying values of $AppLoad$, $EstErr$, HSS, and HCR, we present only three most representative sets of experiments as shown in Table 4 due to space limitations. We have verified that all the experiments show a consistent performance trend for varying workloads and access patterns: 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.

- **Experiment 1**: As described in Table 4, no error is considered in the execution time estimation, i.e., $EstErr = 0$. 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 the other sets of experiments, we fix $AppLoad$ = 200% to compare the adaptability of Basic-IMU, Basic-ODU, and Diff-Real under overload. In this set of experiments, we set HCR = 20%. Hence, the best case settings in our experiments are applied to Experiment 1.

- **Experiment 2**: In general, precise execution time estimates are not available. In this set of experiments, we apply increasing execution time estimation errors, i.e., $EstErr$ = 0, 0.25, 0.5, 0.75, and 1. We set $AppLoad$ = 200% and HCR = 20%.

- **Experiment 3**: In this set of experiments, the worst case settings in our experiments are applied. We set $AppLoad$ = 200% and $EstErr = 1$, i.e., the highest load and the largest execution time estimation error applied in our experiments. 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.)

4.4 Experiment 1: Effects of Increasing Load

In this section, we compare the performance of Basic-IMU, Basic-ODU, and Diff-Real for increasing $AppLoad$.

4.4.1 Miss Ratio

As shown in Figure 5, for Basic-IMU average MR_0 is near zero. However, average MR_1 for Basic-IMU increases as $AppLoad$ increases violating MR_1 threshold (5%) when $AppLoad = 150\%$ and 200% due to the relatively high update workloads. When $AppLoad = 200\%$, MR_1 exceeds 17% and MR_2 exceeds 90% as shown in Figure 5.

In contrast, both Basic-ODU and Diff-Real achieved near zero MR_0 and MR_1 as shown in Figures 6 and 7. Notably, Diff-Real also showed the lowest MR_2 among the tested approaches. When $AppLoad = 200\%$, average MR_2 for Diff-Real is $6.63 \pm 2.79\%$ as shown in Figure 7. MR_2 for Basic-IMU and Basic-ODU exceeds 90% and 50% as shown in Figures 5 and 6, respectively. From this, we can observe that our approach can support the specified average miss ratio for Classes 0 and 1, while significantly reducing the potential starvation for Class 2 compared to the baseline approaches. This is mainly because our approach dynamically adapts the system behavior considering the current system status. As a result, even the least privileged Class 2 can benefit.

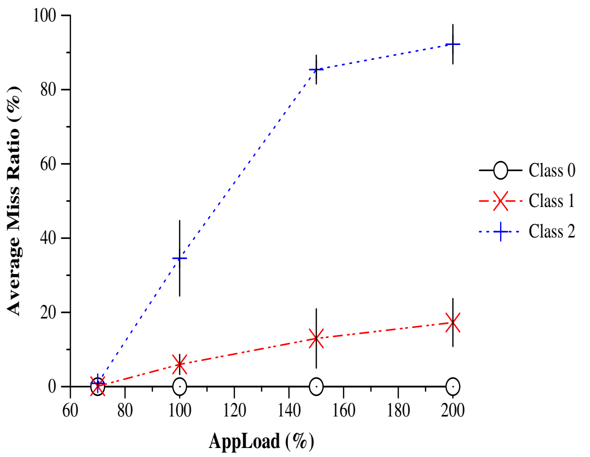

Figure 5: Average Miss Ratio for Basic-IMU

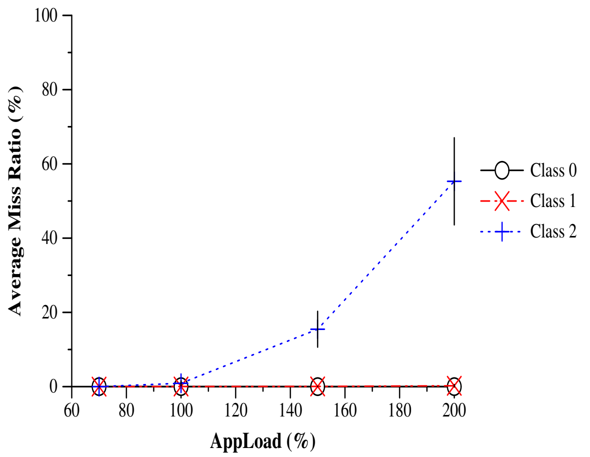

Figure 6: Average Miss Ratio for Basic-ODU

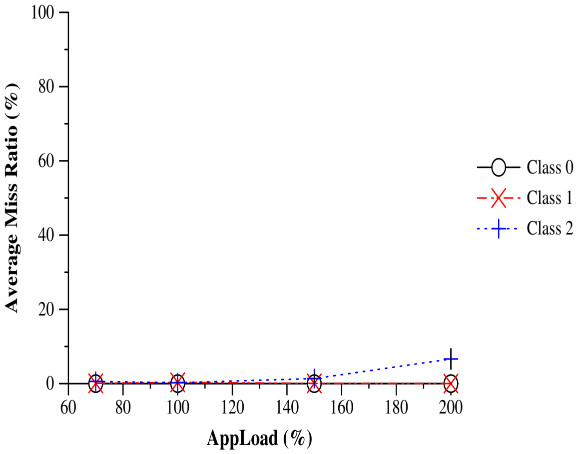

Figure 7: Average Miss Ratio for Diff-Real

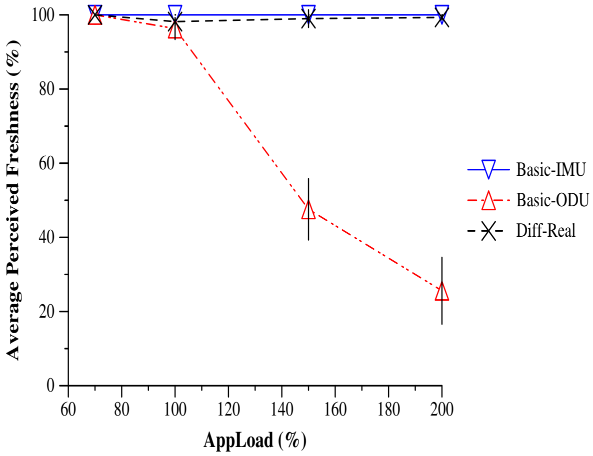

Figure 8: Average Perceived Freshness

Concerning the transient miss ratio, we have observed that Basic-ODU and Diff-Real satisfy the MR_0 and MR_1 overshoot as required by *QoS-Spec*. Basic-IMU satisfied the required performance in terms of MR_0. However, the required transient MR_1 is significantly violated. (Basic-IMU has violated even the specified average MR_1 (5%) for increasing loads as shown in Figure 5.)

Even though Basic-ODU showed a good MR_0 and MR_1, comparable to Diff-Real, we show that it can not support the perceived freshness requirement in the following subsection.

4.4.2 Perceived Freshness

Basic-IMU provides 100% perceived freshness as shown in Figure 8. This is because every update is immediately scheduled in Basic-IMU regardless of the current miss ratio. Diff-Real supports near 100% perceived freshness. The lowest freshness is $98.2 \pm 0.17\%$ when $AppLoad = 100\%$ achieving the target perceived freshness $PF_{target} = 98\%$. However, Basic-ODU has failed to support the PF_{target}. When $AppLoad = 200\%$, it showed $25.6 \pm 9\%$ perceived freshness as shown in Figure 8. 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 $9.1 \pm 0.08\%$.

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, for the clarity of presentation we take the latter approach, which allows stale data accesses to meet the deadline, if necessary. Currently, we are also investigating an alternative approach, which can support the perfect freshness and specified miss ratio for a single service class [11]. In the future, we shall extend this approach to provide the perfect freshness while differentiating the miss ratio among several service classes.

4.4.3 Utilization

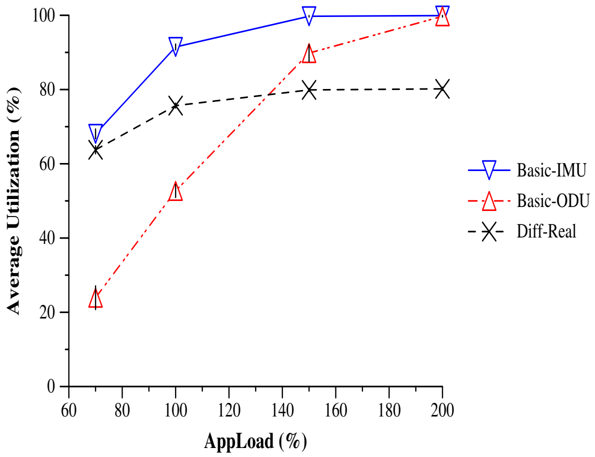

Figure 9: Average Utilization

In Figure 9, Basic-IMU shows the highest utilization among the tested approaches due to the high update workload. As shown in Figure 9, 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 as shown in Figures 5 and 8. In contrast, Diff-Real shows a relatively stable utilization ranging between $60\% - 80\%$ for increasing $AppLoad$ as shown in Figure 9. In Diff-Real, by avoiding the CPU saturation the remaining CPU utilization can be utilized to handle transient overloads, which can lead to the loss of profit due to large miss ratio overshoots.

Note that in Experiment 1 only Diff-Real can support both the PF_{target} and required 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 achieved the lowest MR_2 among the tested approaches as discussed before. The performance gap between Diff-Real and baseline approaches increases for Experiments 2 and 3, which apply more stringent simulation settings.

4.5 Experiment 2: Effects of Increasing Execution Time Estimation Error

In the remainder of this paper, we mainly focus on miss ratio comparisons among Basic-IMU, Basic-ODU, and Diff-Real. Perceived freshness has also been measured, but it is not plotted to avoid repetition. Basic-IMU and Diff-Real showed near 100% perceived freshness, while Basic-ODU failed to support the PF_{target}, similar to the results described in the previous section.

4.5.1 Average Performance

As shown in Figures 10, 11, and 12, MR_0 is near zero for all tested approaches. We have also confirmed that the transient MR_0 does not exceed the specified overshoot, i.e., 1.2%, for all tested approaches. This is mainly because $HCR = 20\%$; only 20% of the applied workload belongs to Class 0, which receive preferred services in terms of scheduling and concurrency control as discussed in Section 3.

As shown in Figure 10, Basic-IMU shows increasing MR_1 for increasing $EstErr$. When $EstErr = 1$, average MR_1 exceeds 60% significantly violating the required average MR_1 threshold (5%). As shown in Figure 11, average MR_1 for Basic-ODU is $5\% \pm 3.43\%$ when $EstErr = 1$. Diff-Real shows near zero MR_1 as shown in Figure 12. Even though Basic-ODU has closely met the required average MR_1, it violates the transient MR_1 requirement, while Diff-Real meets both average and transient requirements (discussed in Section 4.5.2).

As shown in Figures 10 and 11, MR_2 for Basic-IMU and Basic-ODU reach near 100% and 80% when $EstErr = 1$. In contrast, Diff-Real shows $3.79 \pm 5.51\%$ MR_2 when $EstErr = 1$ as shown in

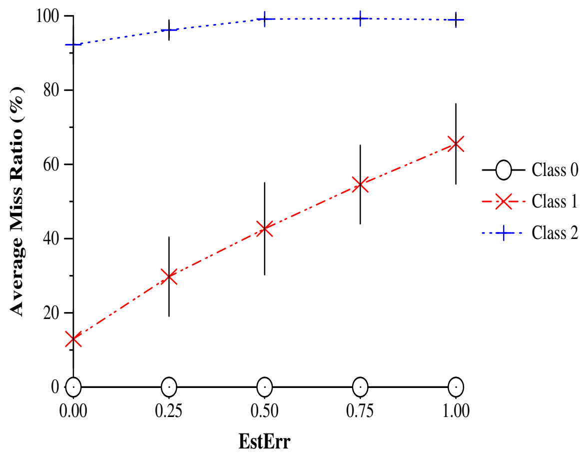

Figure 10: Average Miss Ratio for Basic-IMU

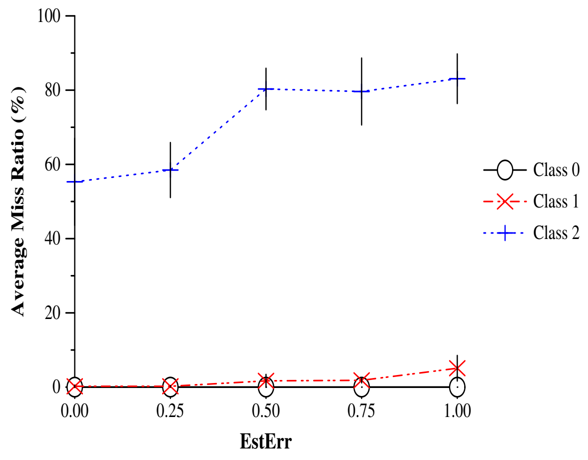

Figure 11: Average Miss Ratio for Basic-ODU

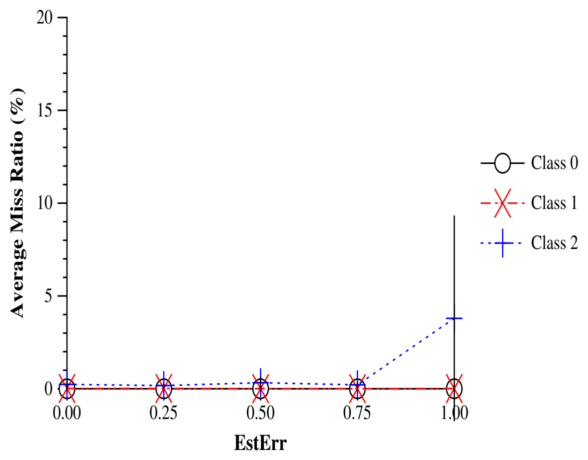

Figure 12: Average Miss Ratio for Diff-Real

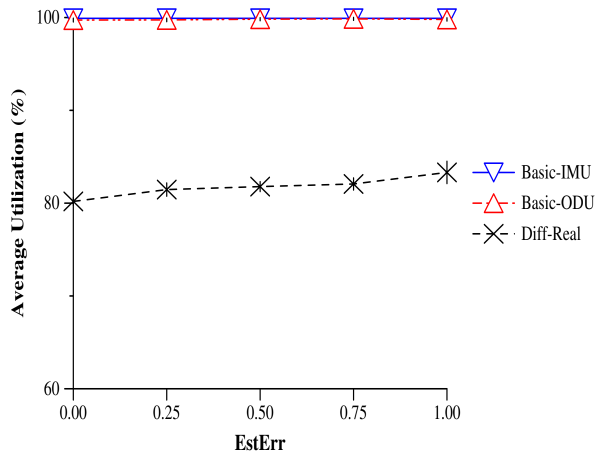

Figure 13: Average Utilization

Figure 12. This is mainly because Basic-IMU and Basic-ODU can admit too many transactions due to increasing execution estimation errors, whereas in our approach the system behavior is dynamically adapted according to the current system status as discussed before.

As shown in Figure 13, for Basic-IMU and Basic-ODU the utilization is near 100% leading to potential miss ratio overshoots and/or perceived freshness violations. In contrast, for our approach the utilization ranges between $80\% - 90\%$.

4.5.2 Transient Miss Ratio

In Figure 14, we plot the transient MR_1 of Basic-ODU and Diff-Real for $EstErr = 1$, i.e., the highest $EstErr$ tested in Experiment 2. (Transient MR_1 for Basic-IMU is not plotted, since it significantly

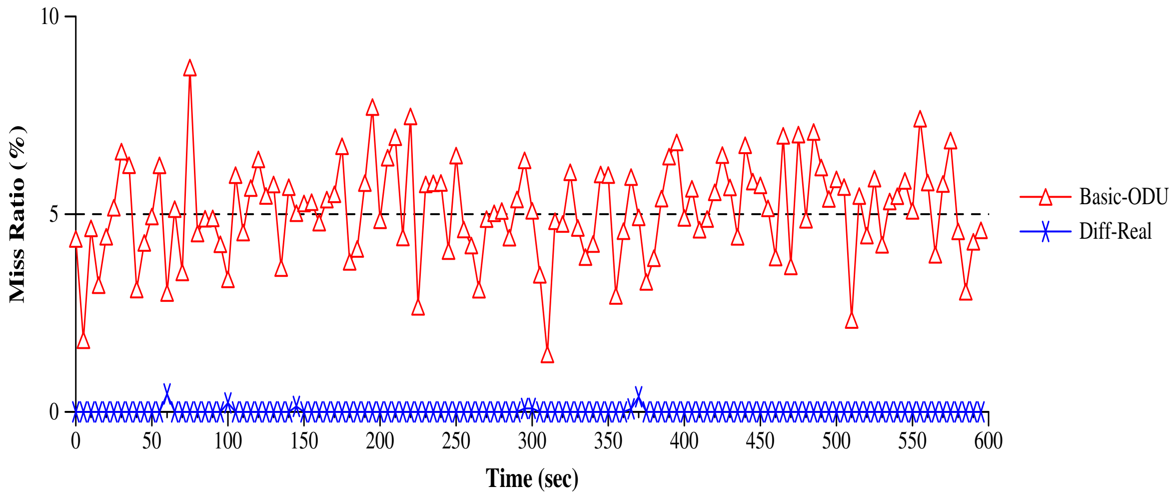

Figure 14: Transient MR_1 for Basic-ODU and Diff-Real

violated the average MR_1 threshold as shown in Figure 10.) We only compare MR_1 for the two approaches, since average/transient MR_0 is near zero for Basic-IMU, Basic-ODU, and Diff-Real in Experiment 2.

As shown in Figure 14, Basic-ODU, which has closely met the average MR_1 requirement as discussed in the previous section, significantly violates the transient MR_1. For Basic-ODU, the transient MR_1 overshoots, which exceed the required 5% MR_1 threshold (the dotted horizontal line in Figure 14), are prevalent through the experiment and do not decay. From this, observe that the average miss ratio metric is not enough to capture/control the potentially time-varying performance of dynamic systems such as real-time databases. In contrast, as shown in Figure 14 our approach shows no miss ratio overshoot throughout the experiment, since the miss ratio is consistently controlled in the feedback loop.

4.6 Experiment 3: Effects of Increasing the Highest Class Load

In real-time database applications, the *HCR* value might not be fixed but could be time-varying. This can affect the real-time database performance. Miss ratios, especially MR_1 and MR_2, may increase as *HCR* increases. To quantify this, we evaluate the 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.

4.6.1 Average Performance

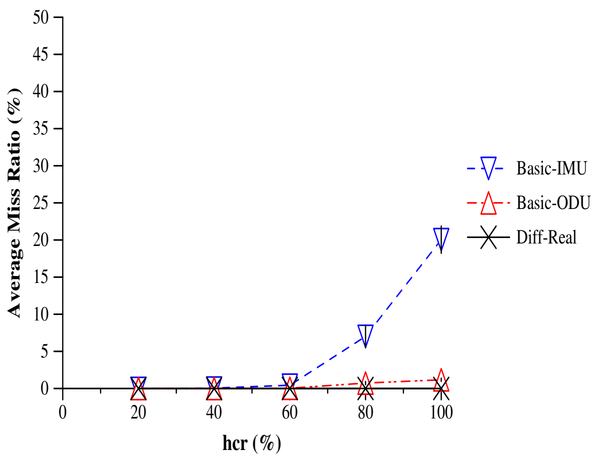

Figure 15: Average Miss Ratio for Class 0

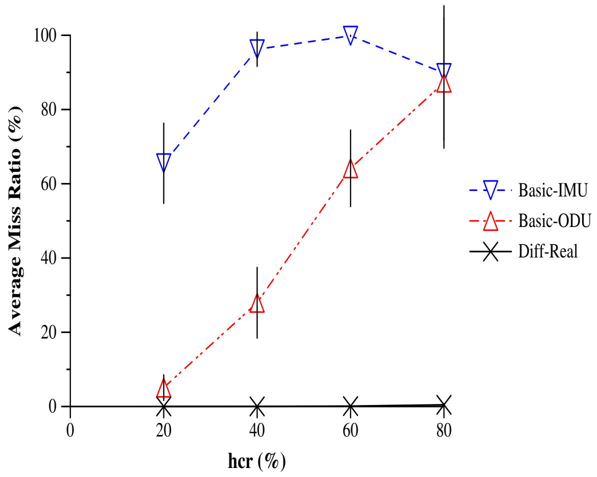

Figure 16: Average Miss Ratio for Class 1

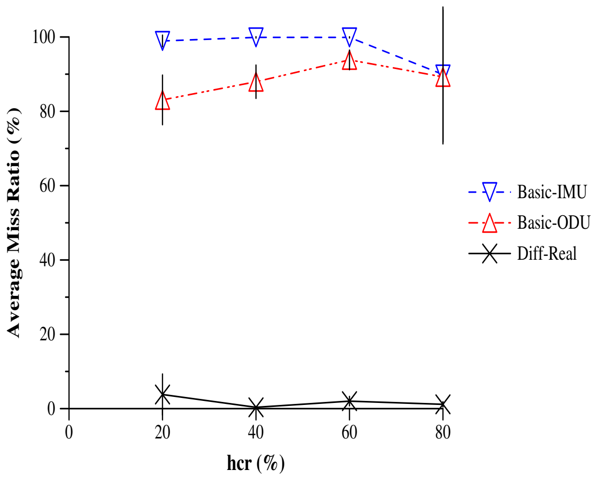

Figure 17: Average Miss Ratio for Class 2

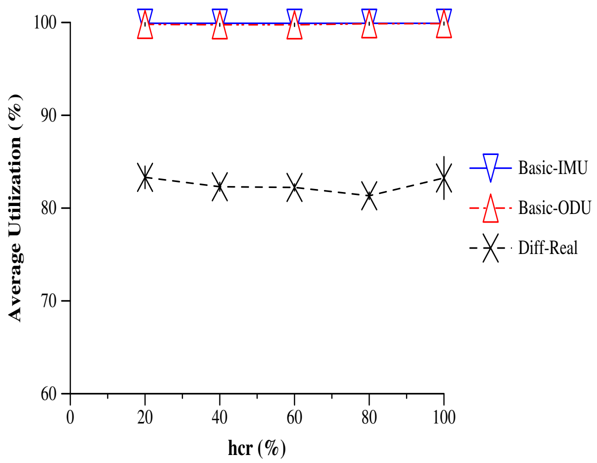

Figure 18: Average Utilization

As shown in Figure 15, for Basic-IMU average MR_0 continuously increases as HCR increases, exceeding 20% when $HCR = 100\%$. For Basic-ODU, the average MR_0 shows approximately 1% when $HCR = 100\%$ as shown in Figure 15. However, we found that the MR_0 overshoot for Basic-ODU reaches 3.56% violating the allowed overshoot of 1.2% (*QoS-Spec*). In contrast, average MR_0 for Diff-Real is maintained near zero despite increasing HCR, even when $HCR = 100\%$. In our approach, transient MR_0 is also near zero (discussed in Section 4.6.2). Further, the perceived freshness requirement is supported in Diff-Real, but violated in Basic-ODU, similar to the results presented in Section 4.4.

In Figure 16, 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 specified MR_1 threshold (5%) as HCR increases. In contrast, Diff-Real supports the required MR_0 and MR_1 by achieving near zero average MR_0 and MR_1 as shown in Figures 15 and 16. As shown in Figure 17, our approach also shows the lowest MR_2 among the tested approaches, similar to the results presented in Sections 4.4 and 4.5. In Figures 16 and 17, for a few cases MR_1 and MR_2 decrease when HCR increases from 60% to 80%. As HCR increases, more Class 0 transactions arrive and could be already in the system. As a result, lower priority transactions have relatively small chances to get admitted and the deadline miss ratio for those admitted ones may decrease.

As shown in Figure 18, for Basic-IMU and Basic-ODU the utilization is near 100% leading to many deadline misses. In contrast, the utilization for Diff-Real is between $80\% - 90\%$ satisfying the required utilization, similar to the results discussed in the previous section. In Diff-Real, the utilization decreases slightly as HCR increases as shown in Figure 18. This is because $MR - LOOP_1$ may request relatively large utilization reductions due to the increasing Class 0 workload, which can incur the transient MR_1 increase. 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.

4.6.2 Transient MR_0 and MR_1 for Diff-Real

Regarding transient performance, we only have to consider Diff-Real. For Basic-IMU and Basic-ODU, the specified average MR_0 and/or MR_1 are already violated as shown in Figures 15 and 16. Consequently, transient MR_0 and MR_1 for Basic-IMU and Basic-ODU also violate the required overshoot and settling time (*QoS-Spec*).

In Figure 19, we compare MR_0 and MR_1 of Diff-Real for $HCR = 80\%$, which shows the largest gap between MR_0 and MR_1 among the tested HCR values. As shown in Figure 19, transient MR_0 and MR_1 are differentiated due to the scheduling and concurrency control in favor of Class 0. MR_0 is near zero

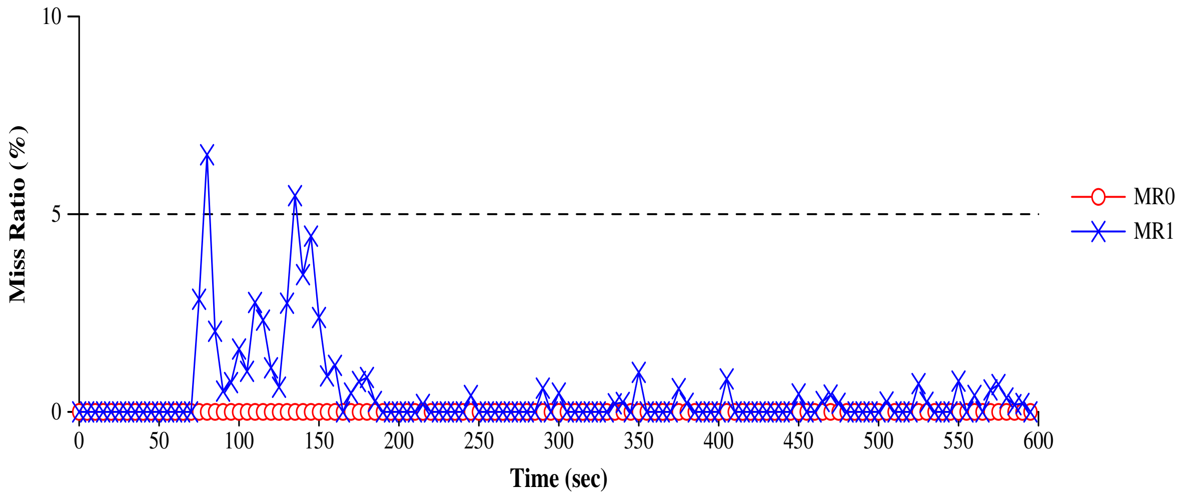

Figure 19: Transient MR_0 and MR_1 for Diff-Real

through the experiment, i.e., no MR_0 overshoot. As a result, the specified overshoot and settling time requirements for MR_0 are automatically satisfied.

As shown in Figure 19, MR_1 showed an overshoot of 6.5% at 80sec. However, it is below the allowed MR_1 overshoot, i.e., 7.5% as required in *QoS-Spec*, and it decayed within one sampling period, i.e., 5sec, meeting the required settling time of 40sec. Another MR_1 overshoot of 5.46% is observed at 135sec, but it also decayed in one sampling period. Since MR_2 is not managed by feedback control, the corresponding overshoot is much higher, reaching up to 14%, and recurrent[2]. To summarize, Diff-Real can guarantee the required per-class miss ratios and perceived freshness by applying feedback control, adaptable update, and admission control schemes even in the presence of unpredictable workloads and access patterns.

5 Related Work

Service differentiation techniques have been studied in various computational systems such as web servers, network routers, and proxy caches [3, 4, 5, 6, 14, 17]. By providing preferred services to the high priority class(es), limited system resources can be effectively utilized, especially under overload conditions. Despite the increasing demand for real-time data services, the related research for service

[2]Transient MR_2 is measured, but not plotted for the clarity of presentation.

differentiation is relatively scarce in real-time databases possibly due to the inherent complexity such as unpredictable data/resource conflicts and potentially conflicting timing/freshness constraints.

Existing service differentiation models can be categorized as: basic model [3, 6], proportional differentiation model [5, 19], absolute guarantee model [14], or hybrid model [4, 14]. In the simplest service differentiation model, called basic model in this paper, a high priority class receives better services. However, no QoS guarantee is provided and the performance difference among the service classes is usually unknown.

In the proportional differentiation model such as [5], the ratio of service delays between service classes can be maintained roughly as a constant, however, no upper bound is specified on the service delay. By an intelligent memory management, a proportional service differentiation is provided in real-time databases [19]. Given enough memory, queries can be processed in time. Otherwise, temporary files should be used during the query processing to save the intermediate results. As a result, the query processing may slow down. In this way, query response time was differentiated among the service classes. Admission control and real-time scheduling schemes are applied to manage potential overload. However, in their approach no upper bound is provided on the average or transient miss ratio. Neither data freshness issues are considered.

In the absolute guarantee model, some subsets of all service classes can receive certain delay guarantees. In [14], limited system resources, i.e., web server processes, are allocated according to the priority of the class to guarantee the web server connection delay. Conceptually, our approach can be considered to provide absolute guarantees for Classes 0 and 1 in terms of average/transient miss ratio. However, Diff-Real considers not only timing constrains but also database performance issues such as data freshness.

Hybrid models such as [4, 14] can provide both the absolute and proportional service differentiation. In [14], initially a proportional differentiation is provided. As the load increases, it is switched to an absolute guarantee model. However, the performance can fluctuate during the switching from one service differentiation model to another due to the delayed switching to provide a smooth transition. A hybrid model is also provided in the context of network routers [4]. These work [4, 14] consider web server connection scheduling and network routing, and therefore, are not directly applicable to real-time databases.

Trade-off issues between response time and data freshness are considered in [2, 12]. Stanford Real-Time Information Processor (STRIP) [2] introduced several algorithms to schedule temporal data updates and user requests in a balanced manner. In [12], trade-off issues between response time and data freshness are considered in the context of the web server. Dynamically generated data are materialized at the web server and continuously refreshed by the back-end database. Response time can be improved if more views are materialized, however, data freshness can be reduced, and vice versa. They presented an adaptive view selection algorithm to improve the response time and data freshness. However, in [2, 12] no performance guarantee is provided. In contrast, our approach provides guaranteed real-time data services in terms of data freshness and (differentiated) miss ratio.

Various aspects of the real-time database performance other than data freshness can be traded off to improve the miss ratio. In [18] and [28], the correctness of answers to database queries can be traded off to enhance timeliness by using the database sampling and milestone approach [13], respectively. In these approaches, the accuracy of the result can be improved as the sampling/computation progresses, while returning an approximate answers to the queries, if necessary to meet their deadlines. In replicated databases, consistency can be traded off to reduce the response time. Epsilon serializability [22] allows a query processing despite the concurrent updates, while the deviation of the answer to the query can be bounded. An adaptable security manager is proposed in [24], in which the database security can be temporarily traded off to enhance timeliness. Note that none of the work presented in [18, 22, 24, 28] provide performance guarantees in terms of both data freshness and miss ratio. Neither, service differentiation is considered.

Recently, feedback control has increasingly been applied to QoS management and real-time scheduling because of its robustness [4, 16, 17, 25]. However, none of them considered service differentiation issues in real-time databases regarding both timing and data freshness constraints.

6 Conclusions

The demand for real-time data services is increasing in e-commerce applications. In many e-commerce applications, it is desirable to process user service requests within their deadlines using fresh data. However, it is very challenging to satisfy this fundamental requirement due to possibly time-varying workloads and data access patterns. Further, timeliness and freshness requirements could conflict with each other. In Diff-Real, a database administrator can explicitly specify the required database QoS including the desired data freshness and differentiated miss ratios. To support the required real-time database QoS, we apply the feedback-based miss ratio differentiation, admission control, and adaptable update policy. According to the experimental results, our approach can achieve the required QoS, while the baseline approaches fail to support the specified miss ratio and/or freshness requirements in the presence of unpredictable workloads and access patterns. Our approach also showed the relatively low miss ratio in the less privileged class(es) compared to the baseline approaches, thereby reducing potential starvation. The significance of our work will become more evident as the demand for (and importance of) real-time data services increases in e-commerce applications.

In the future, we shall further investigate service differentiation/performance guarantee issues for real-time data services. Currently, secure real-time transaction processing with timing guarantees is under investigation. We shall also explore QoS management issues in distributed real-time databases.

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. *IEEE Network*, 13(5), Sept. - Oct. 1999.

[4] N. Christin, J. Liebeherr, and T. F. Abdelzaher. A Quantitative Assured Forwarding Service. In *IEEE INFOCOM*, June 2002.

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

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

[7] J. Baulier et al. DataBlitz Storage Manager: Main Memory Database Performance for Critical Applications . In *ACM SIGMOD - Industrial Session: Database Storage Management*, 2000.

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

[9] K. D. Kang, S. H. Son, and J. A. Stankovic. Service Differentiation in Real-Time Main Memory Databases. In *the 5th IEEE International Symposium on Object-oriented Real-time Distributed Computing*, April 2002.

[10] 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. In *the 14th Euromicro Conference on Real-Time Systems*, June 2002.

[11] K.D. Kang, S. H. Son, J. A. Stankovic, and T.F. Abdelzaher. FLUTE: A Flexible Real-Time Data Management Architecture for Peformance Guarantees. Technical Report CS-2002-17, Computer Science Department at University of Virginia, 2002.

[12] 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.

[13] 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.

[14] C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son. A Feedback Control and Design Methodology for Service Delay Guarantees in Web Servers. Technical Report CS2001-6, Computer Science Department at University of Virginia, 2001.

[15] 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.

[16] 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*, 23(1/2), 2002.

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

[18] G. Ozsoyoglu, S. Guruswamy, K. Du, and W-C. Hou. Time-Constrained Query Processing in CASE-DB. *IEEE Transactions on Knowledge and Data Engineering*, pages 865–884, Dec 1995.

[19] 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.

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

[21] Polyhedra Plc. http://www.polyhedra.com.

[22] 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.

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

[24] S. H. Son, R. Mukkamala, and R. David. Integrating Security and Real-Time Requirements using Covert Channel Capacity. *IEEE Transaction on Knowledge and Data Engineering*, 12(6):865–879, 2000.

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

[26] Moneyline Telerate. Telerate plus. http://www.futuresource.com/.

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

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