QoS Management in Web-based Real-Time Data Services *

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

January 4, 2002

Abstract

The demand for real-time data services has been increasing recently. Many e-commerce applications and web-based information services are becoming very sophisticated in their data needs that span the spectrum from low level status data, typically acquired from sensors, to high level aggregated data. A real-time database, a core component of many information services for such applications, can be a main service bottleneck. Current databases are not time-cognizant, are poor in supporting temporal consistency of real-time data and real-time access with guarantees, and therefore they do not perform well in these applications. Because of this, there has been significant increase in research in real-time data services in both academia and industry, and several commercial products are beginning to emerge in this area. To address this problem, we propose a QoS management scheme for real-time data services that provides guarantees on transaction timeliness and data freshness, which are considered two fundamental performance metrics for real-time data services. Using our approach, admitted user transactions can be processed in a timely manner and data freshness can be guaranteed even in the presence of unpredictable workloads and data access patterns. We present a framework for real-time data services in unpredictable environments, and an architecture for differentiated services based on feedback control and QoS adaptation. We also discuss how to extend our approach in distributed environments.

1 Introduction

The demand for real-time information services has been increasing recently. Many e-commerce applications and web-based information services are becoming very sophisticated in their data needs that span the spectrum from low level physical data, typically acquired from sensors, to high level aggregate data. Other examples include data fusion, decision support, and data-intensive smart spaces. A real-time database, a core component of many information services for such real-time applications, can be a main service bottleneck. Current databases are not time-cognizant, are poor in supporting temporal consistency of real-time data and real-time access with guarantees, and therefore they do not perform well in these applications. For example, Lockheed found that they could not use a commercial database system for military real-time applications and implemented a real-time database system called Eaglespeed. TimesTen, Probita, Polyhedra in UK, NEC in Japan, and ClusterRa in Norway are other companies that have also implemented real-time databases for various application areas, but similar reasons. While the need for real-time data services has been demonstrated, it is clear that these and other real-time database systems are initial attempts and have not yet solved all the problems. One main difficulty in meeting performance requirements by current real-time data services lies in their data-dependent resource requirements which cannot be predicted precisely a priori. Another major issue is that they may have highly uncertain workloads; it is hard to estimate how many users will request some resource in web-based

*Supported in part by NSF grant EIA-9900895 and CCR-0098269.

information services or how many people will walk into a smart space. As the applications that require real-time data services are being distributed, complex interactions among distributed sites add another set of issues. Consequently, developing real-time data services should involve techniques for managing unpredictability of the environment, handling imprecise or incomplete knowledge, reacting to overloads and unexpected failures (i.e., those not expressed by design-time failure assumptions), to achieve the performance requirements and temporal behavior necessary for accomplishing the specified tasks.

In this paper, we present a novel QoS management scheme for real-time data services to meet the requirements of transaction timeliness and data freshness, which are considered two fundamental performance metrics for real-time applications. The ultimate vision of this work is to provide a guarantee that admitted transactions are processed in a timely manner and required data freshness is satisfied even in the presence of unpredictable workloads and data access patterns. Unfortunately, timeliness and data freshness requirements can conflict with each other. The deadline miss ratio of user transactions can be decreased by giving a higher priority to user requests. In contrast, better freshness can be achieved by favoring sensor updates [4]. Effective balancing between the user transactions and sensor update workload is the key to provide a satisfactory service, which can meet both the specified deadline miss ratio and data freshness constraints. For this purpose, we propose a *dynamic balancing* scheme to balance the user transaction and sensor update workloads. We use a cost-benefit model for sensor updates and derive an adaptive update policy from the model.

Issues of performance control in the presence of unpredictability, incomplete knowledge, load disturbances and noise have been addressed in a different discipline; namely, automatic control. We propose to apply a feedback control theory [14, 34] to provide robustness against unpredictable workloads. In a feedback control system, target performance can be achieved by dynamically adjusting the system behavior based on the error measured in the feedback loop. A preliminary performance study indicates that our QoS-sensitive approach can achieve a significant performance improvement, in terms of transaction timeliness and data freshness, compared to several baseline and ad hoc approaches.

The rest of this paper is organized as follows. In Section 2, the semantics for real-time data services and their performance metrics are provided. Section 3 discusses a scientific methodology for modeling real-time data services in unpredictable environments. Section 4 presents a QoS management scheme for derived data, and Section 5 presents an architecture for differentiated services based on feedback control and QoS adaptation techniques. Section 6 presents some ideas on managing timeliness-freshness tradeoffs using replicated data objects in distributed environments. Related work is presented in Section 7, and finally, Section 8 concludes the paper.

2 Real-Time Data, Transactions, and QoS Metrics

In real-time databases, workloads and data access patterns can be time-varying. For example, in decision support systems users 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 metrics in database QoS. In this section we give an overview of real-time data and transactions, and discuss performance metrics.

2.1 Real-Time Data and Transactions

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, therefore, are aborted. In this paper, we consider the main

memory database model. Main memory databases have been increasingly applied to real-time data management such as online auction/stock trading, e-commerce and voice/data networking [4, 7, 38]. In our main memory database model, the CPU is the system resource of main consideration.

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

Temporal consistency was defined using validity intervals in a real-time database to address the consistency issue between the real world state and the reflected value in the database. A temporal data object X is considered temporally inconsistent or stale if $(current\ time - timestamp(X) > avi(X))$, where $avi(X)$ is the absolute validity interval of X. Therefore, absolute validity interval is the length of the time a temporal data object remains fresh or temporally consistent [36].

Temporal data can be further classified into *base data* and *derived data*. Base data objects import the view of the outside environment. A derived data object can be derived from possibly multiple base/derived data. For example, a composite index can be derived from various stock prices. A base data item is directly associated with an absolute validity interval. A derived data object is associated with absolute and relative validity intervals [36].

Temporal data can be updated *periodically* or *aperiodically*: a periodic update occurs at fixed intervals, while an aperiodic update is not predictable and occurs only if the data value is changed. Periodic updates will be considered first in this paper, since periodic updates are common in real-time applications.

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

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

- *User Transactions*: User transactions arrive aperiodically. User transactions do not write any temporal data object, but they can read/write non-temporal data. A user transaction can perform arithmetic/logical operations based on a set of temporal/non-temporal data. User transaction execution time and data access pattern can be time-varying. For example, in a decision support system a transaction can read different sets of data and perform different operations according to the situation.

We assume that each user transaction has a deadline. The deadline of a sensor transaction is set to the update period. A tardy transaction is aborted. For scheduling transactions, we apply earliest deadline first (EDF) algorithm [24] combined with our adaptive update scheduling policy.

Admission control can be applied to user transactions. A newly arriving user transaction is admitted to the system if the requested CPU utilization is currently available. The current utilization can be examined by aggregating the utilization estimates of the previously admitted transactions.

2.2 QoS Performance Metrics

Before we discuss QoS management and performance metrics for real-time data services, it is important to note that we do not aim to provide hard guarantees. Instead, we focus on soft/firm real-time applications, in which infrequent deadline misses or temporal consistency violations can be tolerated, if neither of them exceeds the specific limits specified in the QoS requirements. This type of requirement is typical of many complex real-time applications such as web-based information services and sensor networks in smart spaces.

Two major performance metrics are considered for QoS specification:

- *User Transaction Deadline Miss Ratio:* In a QoS specification, a database administrator can specify the target deadline miss ratio that can be tolerated for a specific real-time application.

- *Data Freshness:* We categorize data freshness into *database freshness* and *perceived freshness*. Database freshness is the ratio of fresh data to the entire temporal data in a database. Perceived freshness is the ratio of of fresh data accessed to the total data accessed by timely transactions. To measure the current perceived freshness, we exclusively consider timely transactions. This is because tardy transactions, which missed their deadlines, add no value to the system. Note that only the perceived freshness can be specified in the QoS requirement. Our QoS-sensitive approach provides the perceived freshness guarantee while managing the database freshness internally (hidden to the users).

In addition, we also consider two additional performance metrics:

- *Differentiated Timeliness:* In QoS requirements, relative response time between service classes can be specified. For example, relative response time can be specified as 1:2 between premium and basic classes. By allowing and enforcing the explicit specification, we can guarantee the distance between service levels of different classes. Therefore, the performance can be differentiated in proportion to the importance of service classes. It will be interesting to find out if relative service differentiation can be actually achieved, while satisfying both data freshness and transaction timeliness, given unpredictable workload.

- *Freshness of Derived Data:* To maintain the freshness, a derived data object has to be recomputed as the related base data changes. A recomputation of derived data can be relatively expensive compared to a base data update. It can include arithmetic and/or logical computations from multiple base data, which might be updated frequently. Therefore, it could be computationally very expensive if a derived data object is recomputed on each base data change.

We discuss these metrics further in Sections 4 and 5 on QoS adaptation techniques for derived data and on an architecture for differentiated services.

Long-term performance metrics such as ones listed above are not sufficient for performance specification of dynamic systems, in which the system performance can be widely time-varying. For this reason, we use transient performance metrics such as overshoot and settling time, adopted from control theory, as performance metrics for real-time data services [14, 28, 34]:

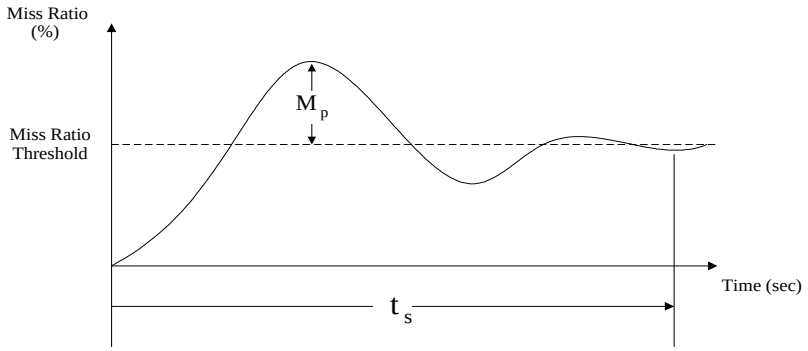


Figure 1: Definition of Overshoot and Settling Time in Real-Time Data Services

- *Overshoot* (M_p) is the worst-case system performance in the transient system state. In this paper, it is considered the highest miss ratio over the miss ratio threshold in the transient state as shown in Figure 1. Overshoot is an important metric because a high transient miss ratio can cause serious implications in several applications such as robots and e-commerce.

- *Settling time* (t_s) is the time for the transient overshoot to decay and reach the steady state performance, as shown in Figure 1. The settling time represents how fast the system can recover from overload. This metric has also been called reaction time or recovery time.

Similar transient performance metrics are proposed in [37] to capture the responsiveness of adaptive resource allocation in real-time systems. These transient performance metrics provide a much better understanding of highly dynamic systems than just average case metrics.

3 Feedback Control for QoS Management

In this section, we first present the feedback control approach for QoS management in real-time data services, and then introduce a cost-benefit model to quantify the utility of an update for a specific temporal data object. Using this model, we propose a database QoS adaptation technique for managing data freshness. In general, the update policy is not dynamically changed in databases. In contrast, our QoS adaptation technique dynamically adjusts the update policy based on the current workload and system behavior.

3.1 Feedback Control

It is a challenging research problem to provide a certain guarantee for transaction timeliness and data freshness, given aperiodical arrival of transactions and unpredictable workload. We have developed a methodology for feedback control scheduling for adaptive real-time systems [26, 27, 29, 40, 41]. An advantage of the feedback control scheduling is its use of feedback control theory, rather than ad hoc solutions, as a scientific underpinning. The control theory based design methodology allows to satisfy the steady state as well as transient performance specifications.

We have been working on developing a framework for QoS management of real-time data services, based on feedback control theory [20]. Using this framework, a designer can systematically design an adaptive QoS manager to satisfy the performance specifications with established analytical methods. This approach is in contrast with existing ad hoc methods that depends on design/testing/tuning iterations. The outline of our framework is as follows. It is similar to the process a control engineer uses to design a controller for a feedback control system to achieve the desired dynamic responses.

- The designer specifies the desired behavior using steady state as well as transient state performance metrics. This step maps the existing metrics of real-time data services to the dynamic responses of control systems.
- The designer establishes a dynamic model of the real-time data services for performance control. The model describes the relationship between the control input and the performance of the system with differential/difference equations. Modeling is important because it provides the basis for the analytical design of the QoS controller. To establish a dynamic model, system identification method [14] can be used to estimate the system behavior using profiling experiments. Our earlier work on system identification of web servers using profiling shows that such an approach could work well for software systems [26].
- Based on the system model and the performance specifications from previous steps, the designer applies existing mathematical techniques (e.g., the Root Locus method, frequency design, or state-based design) of feedback control theory to design the feedback controller with analytic guarantees on the desired transient as well as steady state behavior.

For real-time data services, we apply a feedback control real-time scheduling policy, called FC-UM [29], to manage the miss ratio without under-utilizing the CPU in the presence of unpredictable workloads. As shown in

Figure 2, FC-UM employs two control loops, one for miss ratio and one for CPU utilization management. FC-UM achieves the stability by integrating two control loops. The intuition behind this combination lies in the fact that the saturation conditions of the two control loops are mutually exclusive. A utilization controller is saturated at 100% utilization, while a miss ratio controller can be saturated when the real-time system is underutilized (0 miss ratio as a result). Each control loop generates a control signal to achieve the target utilization or miss ratio based on the current performance error, which is the difference between the target miss ratio (utilization) and the currently measured miss ratio (utilization). Each controller computes the control signal, called requested CPU utilization adjustment ΔU, to achieve the target miss ratio (or target utilization).

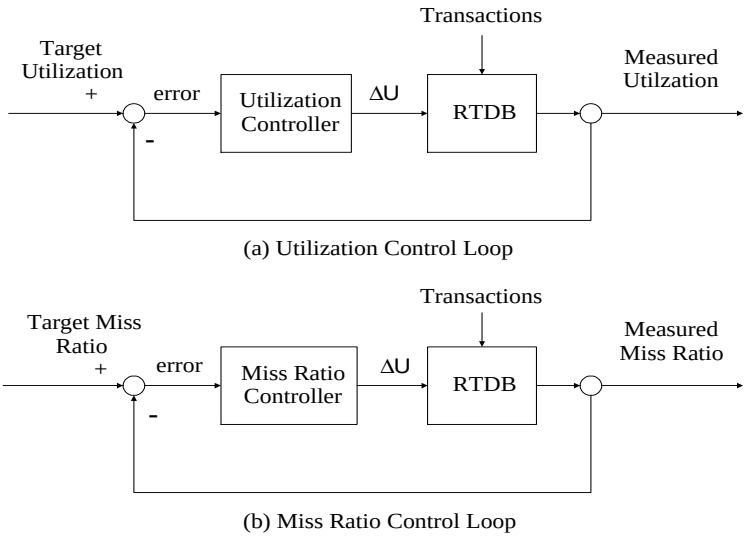


Figure 2: Miss Ratio/Utilization Controllers

We have selected FC-UM to manage QoS for real-time data services, since it can meet our basic requirements for real-time scheduling: miss ratio guarantees while not under-utilizing the CPU given a mix of periodic and aperiodic workloads. Another advantage of FC-UM is that it works well even without a precise workload model. The procedure to adapt and tune the feedback controllers is outlined in Appendix A. For detailed control modeling and controller design, refer to [29].

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

The interactions between FC-UM and our QoS management scheme are described in Figure 3. When ΔU is negative (indicating a violation of the specified miss ratio), QoS in terms of data freshness can be degraded to improve the miss ratio if the current perceived freshness is above the target and the degradation bound is not reached yet. (More discussions shall be given in the following sections.) When the perceived freshness is violated, QoS will be upgraded as long as $\Delta U > 0$ and a certain upgrade bound is not exceeded. The corresponding QoS upgrade/degradation stops when either of the conditions is not satisfied. Without these conditions, a QoS degradation (upgrade) could adversely affect freshness (miss ratio) in the next sampling period, leading to possible oscillations between many deadline misses and data freshness violations.

In an extreme case, not only miss ratio, but also freshness constraints can be violated. In this case, incoming transactions will be rejected until a fraction of currently running transactions terminate and ΔU becomes positive as a result. The chance of an extreme case is reduced by enforcing the QoS adaptation conditions as described before. Furthermore, feedback control scheduling and admission control can prevent a severe overload.

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

Figure 3: Database QoS Management Scheme

3.2 A Cost-Benefit Model of Updates

In a real-time database, the temporal data update workload might be high. For example, in the NYSE trace the update stream can reach up to 696 updates/sec [22]. If updates receive higher priority, there may not be enough time left to finish transactions in time. In contrast, if the transactions are scheduled in a preferred manner, they may have to read stale data [4]. To balance the update and transaction workload efficiently, we need to estimate the update utility which captures the cost-benefit relation of temporal data updates. The cost is defined as the update frequency of a temporal data object. To consider the benefit, access frequency is measured for each data object. If a data object is accessed frequently, e.g., a popular stock price, an update of the object can produce a relatively high benefit. To quantify the cost-benefit relationship, we define the update utility, called Access Update Ratio (AUR), for a data object O_i as follows:

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

The notion of AUR has several interesting features. It can be a guideline to decide a proper update policy for a certain data object. A pictorial description is given in Figure 4, which is a snapshot of a real-time database. In the figure, the temporal data objects in the database are ordered by non-increasing value of AUR. [1]. If a data

[1]To reduce the overhead of sorting, the granularity of data unit can be increased. AUR can be measured for a block of data with a common update period. Also, sorting can be avoided by classifying data into two classes, hot ($AUR \geq 1$) and cold ($AUR < 1$), and by randomly selecting a data object from a certain class to adapt the corresponding update policy. This can be considered a trade-off between QoS management overhead and accuracy of QoS adaptations.

object is on or above the horizontal line of AUR = 1, the benefit of the corresponding data update is worth the cost, since it is accessed at least as frequently as it is updated. Otherwise, it is not cost-effective. For simplicity, let us call the corresponding data *hot*, and the data below the horizontal line (AUR = 1) *cold*, respectively.

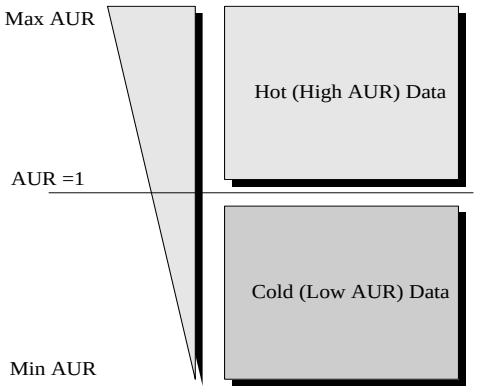


Figure 4: Database Snapshot Sorted by Access Update Ratio

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 the deadline waiting for the update. Alternatively, it may not be necessary to update cold data aggressively when overloaded. Only a few transactions may miss deadlines waiting for the update. For cold data, the CPU utilization can be reduced by applying a lazy update policy under overload. In fact, cold data could always be updated by a lazy update policy regardless of the current system load. However, that approach may increase the response time of the accessing transaction. There can be several policies to provide aggressive and lazy updates. In this proposed work, we will first consider *immediate* and *on-demand* policy as the aggressive and lazy update policy, respectively, and study their performance implications.

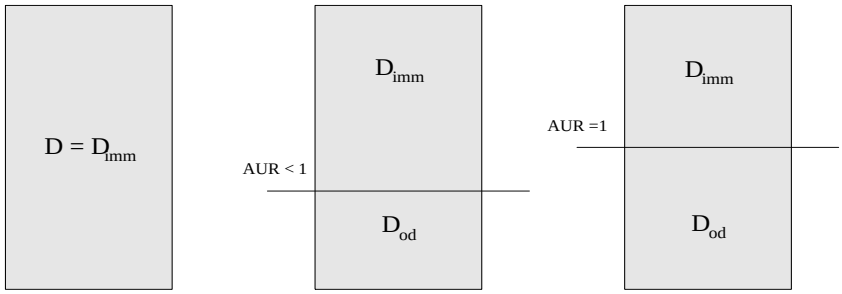
An immediate update receives a higher priority than user transactions and on-demand updates. Conceptually, there are separate scheduling queues for immediate updates and for user transactions/on-demand updates, respectively. Immediate updates in the high priority queue are scheduled before user transactions and on-demand updates in the low priority queue. In each queue, transactions are scheduled by EDF scheduling algorithm. As a result, the freshness of cold data can be relatively low compared to hot data, if a lazy update policy is applied for cold data.

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

3.3 Dynamic Adaptation of Updates

Figure 5 gives a pictorial example of our approach to dynamic update adaptation based on the cost-benefit model. In Figure 5, D represent the set of the all temporal base data in the database, in which D_{imm} is the set of data updated immediately, and D_{od} stands for the set of data updated on demand. Since a data object is updated either immediately or on demand, $D = D_{imm} \bigcup D_{od}$ and $D_{imm} \bigcap D_{od} = \emptyset$. Initially, every data is updated immediately. As the load increases, a larger fraction of cold data objects are updated on demand. This is called *degradation*, since it may potentially degrade data freshness. The update policy degradation stops once it reaches a certain degradation bound, namely $AUR = 1$.

Several issues need to be addressed in this approach. First, access statistics need to be collected to compute $AUR[i]$ for each temporal data object O_i. On each access of O_i, the access counter $ACCESS[i]$ is incremented.



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

Figure 5: Update Policy Adaptations

However, the access frequency may have a large deviation from one sampling period to another. To smooth the potentially large deviation, it might be necessary to take a moving average in the k_{th} sampling period:

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

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

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

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

To handle aperiodic updates, the update frequency can be monitored and smoothed in a similar way as the access frequency. For aperiodic updates the definition of data freshness may have to change, since there might be no explicit notion of validity interval related with aperiodic updates. In that case, a temporal data object can be considered stale upon the arrival of the corresponding update, which is not applied yet [5]. An important question is how to estimate the CPU utilization saved from each degradation. We need to find out the difference between the required CPU utilization for purely immediate updates and for adaptive updates. The number of saved updates due to the degradation is approximately:

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

Given the average CPU utilization per single update transaction, σU, the saved CPU utilization from the update policy degradation for O_i is approximately:

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

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

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

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

The degradation continues for the next data object in D_{imm} with the least AUR until $\Delta U \geq 0$ or the degradation bound is reached. The horizontal line at AUR = 1 in Figure 4 is the *update policy degradation bound*, which limits the degradation within a certain range of AUR. Further degradation past the degradation bound is meaningless because the number of updates may not be reduced if hot data object is updated on demand, while accessing transactions may suffer unnecessary delay for update. An example of update policy degradations is shown in Figure 5. Initially, every update is performed immediately. The update policy is gradually degraded as the load increases. The degradation stops if no more adaptation is allowed either by the violation of perceived freshness or reaching the degradation bound.

In QoS upgrades, the update policy is switched back to the immediate policy for certain data objects to ensure that timely transactions access fresh data. An important issue in QoS upgrades is to avoid a potential miss ratio overshoot in the next sampling period, while improving the perceived freshness as needed. Given a target perceived freshness F_{target}, the current perceived freshness $F_{current}$ can be measured in a sampling period. If $F_{current}$ is less than F_{target}, some of these data should be updated immediately in the next sampling period. The freshness can be improved approximately in proportion to the number of data moved from D_{od} to D_{imm}. The key question is how to estimate the number of data (and which ones) to be updated immediately. More details of QoS upgrade/degradation are discussed in [20].

Due to the approximation, unpredictable workloads and access patterns, our QoS adaptation may not be precise. However, the target performance can be achieved by continuously adjusting the QoS level based on the performance error measured in the feedback loop. By balancing update and transaction workload efficiently, a target deadline miss ratio and perceived freshness can be achieved at the same time. It will be interesting to investigate the accuracy of the approximation of CPU utilization and perceived freshness, and their impact on the feedback controller. That could provide a quantitative measure of the effectiveness of the proposed QoS management approaches in terms of guaranteeing the required timeliness and data freshness.

The QoS management approach proposed above is relatively coarse-grained. A more fine-grained approach is also possible which dynamically increases the update periods for cold data. We suspect that such a fine-grained approach may incur high overhead, since the appropriate update period and the corresponding temporal validity interval should be dynamically managed per data object. However, changing update period could be useful in applications such as web-based information services for smart spaces, because controlling the data generation rate of sensors could be very effective to control the volume of data to go through the network [19]. Our future work includes the details of such fine-grained approach and compare its performance to the coarse-grained QoS management scheme.

4 Derived Data Management

Management of derived data, which are derived from multiple sensor data possibly changing at high rates, is an important problem. In our QoS-sensitive database model, base data and derived data are treated in different ways according to the respective freshness maintenance cost. Derived data recomputation can be relatively expensive; it may include complex logical and/or arithmetic computations, and the recomputation might occur very frequently. It may not be acceptable to recompute the derived data upon each base data change unless the system is underutilized. To reduce the recomputation workload under overload, we need a QoS management technique that adapts to the workload. For QoS adaptation for real-time data services, we apply the notion of forced delay [5] to batch the derived data recomputations during the overload to reduce the number of recomputations. For example, the recomputation of a composite index can be delayed for a certain period of time. During the delay, a related stock price (base data) may change multiple times. The simulation study in [5] shows the performance improvement by applying the forced delay. However, they did not show how to determine the length of the delay. To reduce the computational cost, every derived data can be recomputed on demand [6]. A possible disadvantage of the approach is that the accessing transaction may have to tolerate relatively long response time due to on

demand recomputations.

Our QoS management scheme can be extended such that the freshness of the derived data is gradually degraded by increasing the forced delay. We provide two alternatives for derived data freshness management: QoS adaptation and differentiated service. In the first approach, the forced delay is gradually increased while the system is overloaded. The QoS degradation should stop, if a further increase of the forced delay is meaningless, i.e., the average inter-access time is shorter than the delay. The forced delay for a specific derived data object O_i should *not* be longer than the Average Time between two consecutive accesses of O_i, which is approximated:

$$AT[i] = \frac{Sampling\ Period}{SACCESS_{derived}[i]} \tag{7}$$

where $SACCESS_{derived}[i]$ is to measure the smoothed access frequency for a derived data object i similar to $SACCESS[i]$ defined for a base data item. If the further increase of the delay is not possible, the recomputation policy changes to on demand for a cold data object. If the data object is hot, the forced delay is simply not increased. A hot data object is never recomputed on demand for a similar reason discussed in the previous section. By limiting the forced delay below the average inter-access time, soft guarantees can be provided for the perceived freshness of the derived data.

Considering the relatively high cost for derived data management, the real-time database performance can be further improved by a differentiated service. For example, the freshness of aircraft position data can be differentiated according to the vicinity for collision detection, e.g., near or far. In the online stock trading, the trading server can specify differentiated freshness between composite index classes considering the popularity. To capture the popularity and the recomputation frequency (recomputation cost) of derived data objects, the cost-benefit model described earlier can be extended. The extension should be straightforward considering the generality of the proposed cost-benefit model. Further, the length of the forced delay increase between derived data classes can be differentiated. The improvement by the freshness differentiation could be non-trivial considering the cost of derived data recomputations. Note that we do not need a different control model to support differentiated service in terms of derived data freshness. It can be achieved by extending the QoS manager in the proposed real-time database model.

5 Differentiation of Transaction Classes

In several applications, the timeliness requirements of transactions of different classes need to be treated differently. For example, the deadlines of the transactions in the highest class should be satisfied with certain guarantees even if the system is overloaded. In general, service differentiation improves the system performance by effectively utilizing the system resources under overload [9, 10, 11, 18]. In real-time databases, differentiated services can be provided in terms of deadline miss ratios and data freshness constraints for different transaction classes. The importance of service differentiation is increasing as the demand for real-time data services has been increasing. In this section, we consider an architecture for differentiated services for real-time data services. Although there has been a number of differentiated service models proposed recently [3, 9, 11, 18], none of them have considered transaction timeliness for real-time data services. A thorough investigation of the control model and controller design alternatives for real-time transactions is needed for the differentiated services in databases.

The differentiated service architecture is shown in Figure 6. It consists of a transaction handler, a monitor, feedback control loops, a QoS manager, an update scheduler, and an admission controller.

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.

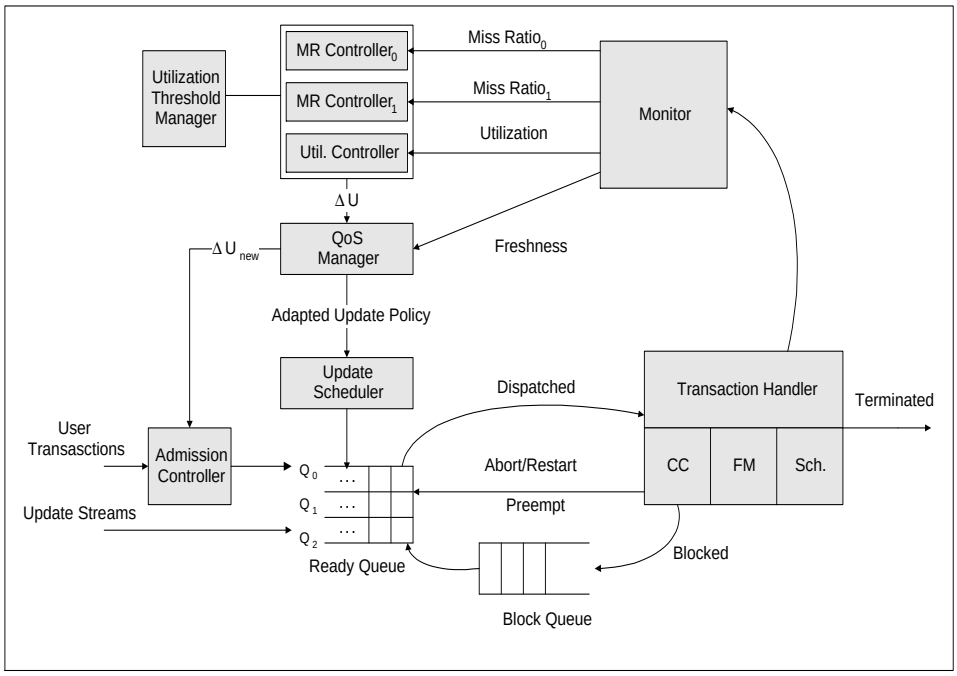


Figure 6: Real-Time Database Architecture for Differentiated Services

The FM checks the freshness before accessing a data item using the corresponding absolute 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 6) according to their service classes, i.e., Classes 0, 1, or 2. 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 updates are scheduled at Q_0.

By applying the fixed priority among the service classes, we provide a basic support for the service differentiation in real-time applications. However, this is insufficient to provide guarantees on the miss ratio and perceived freshness in the presence of unpredictable workloads/access patterns.

A feedback control scheduler [29] is extended to differentiate per-class miss ratios in a guaranteed manner despite potential unpredictability in workloads. The QoS manager enforces the control signal computed in the feedback control loops to adjust the CPU utilization based on the current system behavior observed by the Monitor. More specifically, it dynamically adjusts the update workload considering the current miss ratio and perceived freshness by adapting the database update policy. As a result, some data is updated on demand while others are updated immediately. The utilization threshold is dynamically adjusted considering the current system behavior to increase the CPU utilization avoiding miss ratio overshoots. The QoS manager dynamically adapts database update policy to enforce the CPU utilization adjustment. The Update scheduler decides whether to schedule or drop an incoming update based on the current update policy. Admissions of incoming transactions are controlled to avoid a potential overload or an underutilization.

One of the key issues in providing different classes of services on the web is the type of performance guarantees, since there are multiple classes of transactions and multiple performance metrics used to specify the requirements. Most ad hoc approaches can be called the *best effort differentiation*, since they do not provide any guarantees, although they provide better service in general to high class clients. In particular, they do not provide

guarantees on the extent of the difference between different classes. There are three different types of performance guarantees which are stronger than the best effort differentiation: *absolute, relative,* and *hybrid* [26]. In the absolute guarantee model, a set of specified performance guarantees are provided for each class. Since the workload may increase arbitrarily high for web-based data services, it may not be possible to meet the absolute guarantees of all classes under a severe overload.

In the relative guarantee model, a fixed ratio of performance between classes are enforced. The system tries to support the desired "distance" between the performance levels of different classes. This model might be more precise in specifying the difference between classes, and also more flexible than the absolute guarantee model, since it does not require to achieve an absolute performance level for each class.

Depending on the semantics of applications and the nature of the overload situations, either model of performance guarantee could be desirable. The relative guarantee model may not be appropriate in severe overload situations, since even high class transactions may get unacceptable performance (although the requirement of differentiation between classes are satisfied). During normal operation conditions, however, the relative guarantee model may be more appropriate. The hybrid guarantee model might be the most desirable in some applications, since it can switch between the absolute and relative guarantees dynamically, depending on the requirement for differentiation and the system workload. For example, a hybrid guarantee can support a relative differentiation between the classes while the performance in each class is in the acceptable range. When the performance in the high class becomes unacceptable, it changes to the absolute guarantee model to enforce the required performance in that class, possibly at the cost of sacrificing the performance in the low class.

We plan to implement all three guarantee models in our differentiated services architecture. Other interesting issues that remain to be addressed for differentiated services include:

- complex profiling of database systems for feedback controller design, since transactions are scheduled in different service classes,
- control of the interactions of multiple control loops which could introduce an unexpected behavior (e.g., a surge in overshoot or longer settling time during transient states), and
- the effectiveness of our differentiated service architecture compared to alternative architectures.

6 Replicated Data in a Distributed Environment

While a number of problems must still be addressed in a centralized environment for managing QoS, distributed systems issues are also critical for improving the performance of real-time data services. Although there are many issues in a distributed environment in general, in this section, we focus on data replication. Data replication has been used to improve the performance and availability in distributed applications. Performance in terms of transaction timeliness can be improved since replicated copies are available at the site where they are frequently accessed. Availability can be improved since the sites at which they are stored have independent failure modes. However, theses benefits come at the cost of overhead associated with update transactions that can be committed only after all sites containing replicated copies agreed to commit to enforce mutual consistency. This approach, called synchronous replication, has serious performance problems in practice [32]. Some commercially available distributed database systems such as Sybase and Oracle 7 support synchronous replication as well as lazy (asynchronous) replication, in which the updates of replicated data objects can be deferred. With lazy replication, a transaction can commit without waiting for all the replicated copies to be updated, relaxing the mutual consistency property. For lazy replication, data freshness is a key concern.

For web-based information services, replication of data/information using proxies and web caching has been popular. Although web caching and proxies addressed the scalability problem in providing desired performance for static contents, it introduced another problem of data freshness for dynamic contents. As the number of web

pages that contain dynamic contents that are continuously changing is increasing rapidly, providing any guarantee for the freshness of data becomes important.

We consider the problem of maintaining acceptable freshness in lazy replication by extending our QoS management scheme. For each replicated temporal data object, there is a primary copy (denoted by capital letters) which is updated by a sensor transaction. We call the site which contains the primary copy of a data object a master node (of that data object). Other copies are called a secondary copy and the sites which contain a secondary copy is called a slave node. The set of replicated copies, consisting of the primary copy (e.g., R) and all secondary copies (r_1, r_2, ...), is referred to *replicated data*.

When an application starts to run at a site which does not have the temporal data object it needs, a secondary copy (r_i) is created and initialized by the current value of the primary copy (R). Secondary copies can be updated by either push or pull. We consider the pull model first, because it reduces the overhead at the master node and provides flexibility to each slave node to change its access pattern (i.e., a slave node does not need to inform the master node to make any change on its behalf). Each slave node pulls the value of the primary copy at the master node periodically, considering the local workload and data access statistics. The period of pull can be dynamically adapted by the QoS manager of the node using the current status information on timeliness, freshness, and utilization. Some of the cold data might be pulled on demand as discussed in the previous section.

We assume that when initially created, all the secondary copies are updated at the same rate of the primary copy. Each slave node keeps the track of the access pattern for its secondary copies. Using Access Update Ratio (AUR) as discussed in Section 3.3, the slave node computes the candidates of its secondary copies for delayed update. When an overload occurs at a node, it can now adjust CPU utilization in several ways. It can switch its update policy from immediate update to on-demand update for some of its primary copies. Alternatively, it can delay updates for some of its secondary copies. Another possibility is to consider both primary copies and secondary copies together. Instead of giving priority to one over the other, whichever copy that brings in higher access demand (or being accessed by high class/priority transactions) will be kept fresh as much as possible – either by updating immediately (primary copies) or pull the value of the primary copy at the initial rate.

To provide appropriate QoS over replicated data in distributed environment, there is another issue to deal with: *update scheduling problem*. Since secondary copies are being refreshed constantly, a separate update scheduler is responsible for scheduling the updates, using the portion of the CPU utilization allocated for refreshing secondary copies at the node. The primary objective of the update scheduler is to satisfy the specified QoS by achieving the maximum overall perceived freshness of replicated data. Since the cost of refreshing secondary copies could be different in general for each replicated data, the update scheduler should consider the frequency of access as well as the cost for each one to determine the right ordering.

We want to avoid any unnecessary refreshing from being scheduled, since it would waste CPU utilization without any benefit to the applications. One simple method is have the master node send a special reply (e.g., *nochange*), when a slave node pulled the value of R for r_i, if the current value of R is the same as r_i. When a slave node receives *nochange*, it discards it so that the CPU utilization is not wasted. This idea can be extended further by introducing the notion of imprecise computation [15, 25]. For each replicated data, we may specify the tolerance level of imprecision (R_p). If the change in the value of R is not more than R_p, a slave node will receive *nochange*. The proper value for R_p depends on the semantics of the data and applications that access it. We can customize the value of R_p, depending on how each copy is used by applications at each slave node. If r_i is used by an application whose R_p is different from that of applications that use r_j, the slave node that maintains r_i should specify its R_p when r_i is created.

Depending on the application semantics, real-time data services may need alert rules to be included for exception handling. The master node of a replicated data may have to push the data upon a certain condition, although the system is relying on the pull model. For example, consider the following situation. In a smart building, the temperature of a vacant room is pulled less frequently when the system has a transient overload. However, an alert can be triggered at the master node if the temperature exceeds a certain threshold (i.e., a possibility of fire). In that case, the master node should push its value to all the slave nodes with secondary copies.

7 Related Work

Previous research work has shown that QoS-sensitive approaches can improve system performance in a cost-effective manner [2, 8, 13, 16, 17]. Despite the abundance of the QoS research, QoS-related work is relatively scarce in database systems. Priority Adaptation Query Resource Scheduling (PAQRS) provided timeliness differentiation of query processing in a memory-constrained environment [33]. From the observation that the performance of queries can vary significantly depending on the available memory, per-class query response time was differentiated by an appropriate memory management and scheduling. Given enough memory, queries can read the operand relations at once to produce the result immediately. If less memory is allocated, they have to use temporary files to save the intermediate results, therefore, the query processing may slow down. In this way, query deadline miss ratios were differentiated between the classes. However, the performance could easily fluctuate under the workload changes. No data freshness issues were considered.

A novel on-line update scheduling policy has been proposed in the context of the web server [22]. The performance of a web server can be improved by caching dynamically generated data at the web server and the back-end database continuously updates them. Given the views to materialize, the proposed update scheduling policy can significantly improve the data freshness compared to FIFO scheduling. They discuss a complementary problem in [21], i.e., view selection problem to materialize. Trade-off issues between response time and data freshness are considered in their work. However, they provide neither miss ratio nor data freshness guarantees.

Stanford Real-Time Information Processor (STRIP) addressed the problem of balancing between the freshness and transaction timeliness [4]. In a real-time database, data should be maintained fresh to correctly reflect the status of the real-world environment, and transactions should be processed in a timely manner. To study the trade-off between freshness and timeliness, four scheduling algorithms were introduced to schedule updates and transactions, and the performance was compared. In their later work, a similar trade-off problem was studied for derived data [5]. Ahmed et al proposed a new approach to maintain the temporal consistency of derived data [6]. Different from STRIP, an update of a derived data object is explicitly associated with a certain timing constraint, and is triggered by the database system only if the timing constraint could be met. By a simulation study, the relative performance improvement was shown compared to the forced delay scheme of STRIP. None of the two approaches considers dynamic adaptations of update policy. Also, performance guarantee is not provided.

The correctness of answers to database queries can be traded off to enhance the timeliness. A query processor, called APPROXIMATE [43], can provide approximate answers depending on the availability of data or time. An imprecise computation technique (milestone approach [23]) is applied by APPROXIMATE. In the milestone approach, the accuracy of the intermediate result increases monotonically as the computation progresses. Therefore, the correctness of answers to the query could monotonically increase as the query processing progresses. A relational database system called CASE-DB [31] can produce approximate answers to queries within certain deadlines. Approximate answers are provided processing a segment of the database by sampling, and the correctness of answers can improve as more data are processed. Before beginning each data processing, CASE-DB determines if the segment processing can be finished in time. In replicated databases, consistency can be traded off for shorter response time. For example, epsilon serializability [35] allows a query processing despite the concurrent updates. Notably, the deviation of the answer to the query can be bounded, different from a similar approach called quasi serializability [12]. An adaptable security manager is proposed in [39], in which the database security level can be temporarily degraded to enhance timeliness. These performance trade-off schemes lack a systematic QoS management architecture and none of them consider providing guarantees for both miss ratio and data freshness.

An algorithm for update scheduling was proposed in a recent [22]. Their goal is to achieve the maximum overall QoD (quality of data), which is the weighted sum of the freshness probabilities of all views in the database. They have demonstrated the importance of the update scheduling problem using server log traces of trade and quote database from New York Stock Exchange. They defined the rank function for each view in the database as a ratio between the frequency of access to that view over the cost required to refresh the view. Then their update

scheduling algorithm considers the popularity weight from the dependency graph among the views and relations to determine the update with the biggest impact on QoD to schedule next. Our approach is different from their work in that we have the timeliness metric to support, and we use the feedback control based QoS management scheme, instead of open loop approach.

Recently, feedback control has been widely applied to QoS management and real-time scheduling [2, 29, 42, 30]. However, to our best knowledge none of them considered QoS management issues in real-time databases considering the timeliness and data freshness constraints.

8 Conclusions

The demand for real-time information services is rising in several new applications. Databases, the core components of many information systems, could be a service bottleneck in the upcoming information era due to their relatively low predictability. In this paper, we presented an approach for QoS management to meet the fundamental requirements for real-time data services, i.e., deadline miss ratio and data freshness guarantees, even in the presence of unpredictable workloads and data access patterns.

By adopting our QoS management approach in real-time data services, sensor transactions and user service requests can be dynamically balanced to guarantee potentially conflicting miss ratio and freshness requirements at the same time. A cost-benefit model is derived to measure the update utility. A novel QoS management scheme is developed based on the model. Combined with the feedback control scheduling and admission control, our QoS management approach can provide guarantees on miss ratio and perceived freshness while other non-adaptive approaches fail. A preliminary performance study indicates that our approach can achieve a significant performance improvement, compared to several ad hoc approaches. We have discussed the issues in applying our method for derived data, differentiated services, and replicated data in a distributed environment.

References

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

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

[3] T. F. Abdelzaher and K. G. Shin. QoS Provisioning with qContracts in Web and Multimedia Services. In *Real-Time Systems Symposium*, Phoenix, Arizona, December 1999.

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

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

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

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

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

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

[10] N. Christin, J. Liebeherr, and T. F. Abdelzaher. A quantitative assured forwarding service. Technical Report CS-2001-21, Computer Science Department at University of Virginia, 2001.

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

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

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

[14] G. F. Franklin, J. D. Powell, and A. Emami-Naeini. *Feedback Control of Dynamic Systems (3rd edition)*. Addision Wesley, 1994.

[15] J. Hansson, M. Thuresson, and S. H. Son. Imprecise Task Scheduling and Overload Management using OR-ULD. In *International Conference on Real-Time Computing Systems and Applications*, Cheju Island, Korea, December 2000.

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

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

[18] P. Hurley, J. Y. Le Boudec, P. Thiran, and M. Kara. ABE: Providing Low Delay Service within Best Effort. *IEEE Networks*, 15(3):60–69, May 2001.

[19] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Int. Conference on Mobile Computing and Networks (MobiCom)*, Boston, Massachusetts, August 2000.

[20] 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.

[21] 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.

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

[23] 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.

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

[25] J. W. Liu, K. Lin, W. Shih, A. Yu, J. Chung, and W. Zhao. Algorithms for Scheduling Imprecise Computation. *IEEE Computer*, 24(5), 1991.

[26] C. Lu, T. Abdelzaher, J. Stankovic, and S. H. Son. A Feedback Control Approach for Guaranteeing Relative Delays in Web Servers. In *IEEE Real-Time Technology and Applications Symposium*, Taipei, Taiwan, May 2001.

[27] 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.

[28] 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.

[29] 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,*, 2001. To appear.

[30] 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.

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

[32] E. Pacitti and E. Simon. Update Propagation Strategies to Improve Freshness in Lazy Master Replicated Databases. *VLDB Journal*, 8:305–318, 2000.

[33] 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.

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

[35] 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.

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

[37] D. Rosu, K. Schwan, S. Yalmanchili, and R. Jha. On Adaptive Resource Allocation for Complex Real-Time Applications. In *IEEE Real-Time Systems Symposium*, Dec 1997.

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

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

[40] J. Stankovic, T. He, T. Abdelzaher, M. Marley, G. Tao, and S. H. Son. Feedback Control Scheduling in Distributed Real-Time Systems. In *Real-Time Systems Symposium*, London, England, December 2001.

[41] J. Stankovic, C. Lu, S. Son, and G. Tao. The Case for Feedback Control Real-Time Scheduling. In *EuroMicro Conference on Real-Time Systems*, June 1999.

[42] 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.

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

Appendix A Feedback Controller Tuning

To tune the controllers of FC-UM, the performance of the controlled system should be profiled under the worst case set-up that can cause the highest miss ratio [29]. The worst case should be considered to provide a certain miss ratio guarantee. For the profiling under the worst case set-up, we turned off admission control and QoS management. All updates are applied immediately in a preferred manner to user transactions. As a result, the user transaction deadline miss ratio increases sharply as load increases (Figure 7). Average deadline miss ratio and utilization are measured for loads increasing from 60% to 200% by 10%. Execution time estimation error is set to 1, which indicates that the average actual execution time could be two times of the average estimated execution time. Update workload is designed to be about 50% of the total CPU utilization for each load. Uniform access pattern is assumed for data accesses. For each load, 10 simulation runs are performed and 90% confidence intervals are derived (vertical bars in Figure 7).

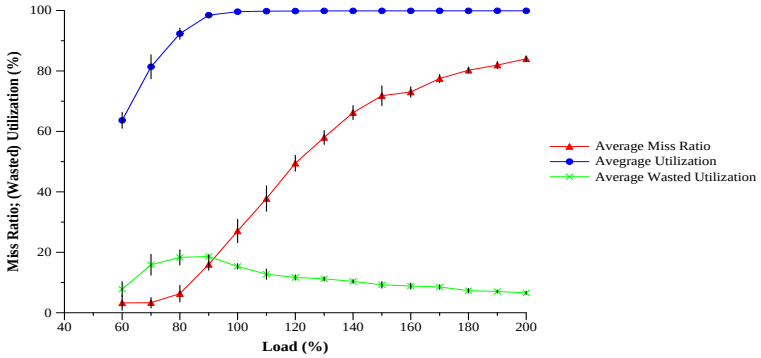


Figure 7: System Profiling Results

The miss ratio gain, $G_M = Max\{\frac{Miss\ Ratio\ Increase}{Unit\ Load\ Increase}\}$, should be derived to tune the controllers [29]. According to our profiling results shown in Figure 7, the miss ratio gain is approximately 1.1682 when the load increases from 110% to 120%. We set the sampling period to 5sec for feedback control. Given the sampling period and miss ratio gain G_M, Root Locus method [14, 34] of Matlab can be used to tune the controllers to support 0 steady state error. The closed loop poles are $p_0, p_1 = 0.552 \pm 0.153i$. The feedback control system is stable, since the closed loop poles are inside the unit circle. The tuned feedback control system can provide the following transient performance:

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

Careful readers may have noticed that the measured average utilization tends to be higher than the load applied to the system before it saturates. This is because the data/resource conflicts increase between updates and user transactions as load increases. (More transactions access temporal data updated by update transactions increasing potential read/write conflicts). From Figure 7, we can observe that the wasted utilization increases until the system saturates. (It decreases after the system is saturated, since tardy transactions can be aborted even before accessing temporal data). The increase of wasted utilization adversely affects the total utilization and miss ratio. This observation motivates the necessity of dynamic balancing between updates and user transactions considering the current system status.