PhD Proposal on

qRTDB: QoS-Sensitive Real-Time Database

Kyoung-Don Kang
Department of Computer Science
University of Virginia

December 6, 2001

1 Introduction

Recently the demand for real-time database services is exploding. The applications requiring such services include sensor data fusion, decision support, web information service, e-commerce, online trading, and data-intensive smart space applications. Furthermore, the information system is being globalized due to the fast growth of the Internet. Despite the importance and wide applicability, the performance and predictability of a database system[1] – the core component of global information systems – are relatively limited compared to the other real-time systems such as real-time operating systems. It can not be easily replicated due to the consistency problem. In addition, the database system has relatively low predictability compared to other real-time systems due to data dependence of the transaction execution, data and resource conflicts, dynamic paging and I/O, and transaction aborts and the resulting rollbacks and restarts [36]. Because of the limited performance and predictability, the database system can be the main service bottleneck in the upcoming global information infrastructure.

Previous research work has shown that QoS-sensitive approaches can improve the system responsiveness and scalability. However, QoS-related research is relatively scarce in database systems, possibly due to the inherent complexity. To address the problem, we propose **qRTDB** (QoS-sensitive Real-Time Database) to increase the responsiveness and scalability of a real-time database system. We present a QoS model and performance metrics applicable to the real-time database system. Based on the QoS model, we propose to develop database QoS management techniques. More specifically, we focus on the fundamental requirements for the success of real-time database services: transaction timeliness and data freshness. For successful database services, users require timely transaction processing based on fresh data. Unfortunately, the two requirements above can be conflicting to each other. For frequent updates, transaction timeliness can be compromised due to the high update workload. Alternatively, transactions may have to read stale (temporally inconsistent) data, if transactions are scheduled in a preferred manner to the updates [8]. For the effective QoS management, we think that the database has to *dynamically* balance the transaction and update workload based on the current system behavior while meeting the timeliness and freshness requirements. It is a challenging research issue considering the unpredictability of the workload and the tight interaction/interference between timeliness and freshness requirements. In our approach, the sensor data update policy can be dynamically adapted considering the current transaction workload while guaranteeing a certain degree of data freshness for timely transactions. Management of derived data, which can be derived from multiple sensor data possibly changing at high rates, is also an important problem considering the computational cost. To maintain derived data freshness in a cost-effective way, we propose to differentiate the the derived data freshness considering the popularity or the importance of the derived data. To further improve the the real-time database system performance, the transaction timeliness is proposed for differentiation. In our approach, transaction timeliness shall be differentiated among service classes under overload.

We apply feedback control theory to provide a systematic QoS management and predictable real-time database service. In a feedback control model, target performance can be converged by dynamically adjusting the system behavior based on the performance error measured in the feedback loop. This approach is especially effective when the workload is difficult to model in a predictable way. By the systematic QoS management, the proposed qRTDB is expected to improve the overall service availability of the database system due to the increased responsiveness and scalability. Service availability is a core issue for the feasibility of global information system, in which a large body of users may concurrently need database services. In our proposed approach, the key database service requirements – transaction timeliness and data freshness – can be met in a cost-effective manner. The expected contributions in the real-time database system can consist of (1) a novel QoS model, (2) new QoS adaptation techniques, (3) novel service differentiation techniques, (4) improved service predictability, (5) enhanced

[1]Note that fast service does not necessarily mean predictable service in a real-time database system [40].

responsiveness and scalability, and (6) the existence proof of the applicability of feedback control theory to the database QoS management. The practical impact of the research could be significant considering the importance of the performance and predictability of database services in the upcoming information era.

The rest of this proposal is organized as follows. In section 2, we discuss the real-time database and QoS model for our purpose. Database transaction semantics, data types and QoS performance metrics are described. Section 3 discusses the adaptive update policy to manage the transient overload. By dynamically adapting the update policy, transaction timeliness can be improved without affecting the user perceived data freshness. In section 4, we consider the real-time database service differentiation in terms of derived data freshness or transaction timeliness. In section 5, we describe performance evaluation issues. Section 6 describes related work. Section 7 concludes the proposal by giving the expected contributions and the research agenda.

2 Real-Time Database and QoS Model

In this section, real-time database and QoS models are described to characterize the basic features of qRTDB. Data types, temporal consistency problem, and transaction types are introduced to apply to our real-time database model. To describe our QoS model, performance metrics are described, and our QoS specification scheme is expressed.

2.1 Data and Transaction Model

We consider a firm real-time database system, in which tardy transactions − transactions that have missed the deadline − add no value to the system, therefore, are aborted. Data can be classified into two classes: static and continuous. Static data does not become outdated, whereas continuous data object may change continuously to reflect the real world state, e.g., current temperature, aircraft position, stock prices, etc. In this proposal, we mainly focus on freshness issues of continuous data. Each continuous data object has a timestamp of the latest update [2]. To address the consistency issue between the actual state and the reflected value in the database, temporal consistency was defined using validity intervals and subdivided into two components in a real-time database [36, 44, 23]:

- *Absolute validity interval* is defined between the environmental state and the value reflected in the database. A 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 a data object X. Therefore, absolute validity interval is the length of the time a continuous data object remains fresh or temporally consistent.
- *Relative validity interval* is defined among the data used to derive other data. Consider a data object Y is derived from a data set $R = \{X_1, X_2, ..., X_k\}$. Y is temporally consistent if the arbitrary data objects $X_i, X_j \in R$ meets the absolute consistency requirements, and $|timestamp(X_i) - timestamp(X_j)| < rvi(Y)$, where $rvi(Y)$ is the relative validity interval of Y.

Continuous data can be further classified into *base data* and *derived data*. A derived data object can be derived from possibly multiple base data. For example, the current aircraft position can be derived from multiple sensor readings. As another example, a composite index can be derived from various stock prices. Base data imports the view of the outside environment and directly associated with the absolute validity interval. A derived data object is associated with the absolute and relative validity interval.

Continuous data can be updated *periodically* or *aperiodically*: a periodic update occurs at every periodic interval, while an aperiodic update is not predictable by the database system and occurs only if the data value

[2] For simplicity, we assume the availability of a global clock.

is changed. For the sake of simplicity, we focus on the periodic updates of continuous data in this proposal. Periodic updates are common in real-time applications, e.g., sensor readings. Furthermore, aperiodic updates can be modeled as sporadic. For example, stock price updates can be modeled as sporadic based on the highest update frequency in the peak time, therefore, a validity interval can be associated with a stock item. A stock price can be marked stale if the update actually arrives in the validity interval. Otherwise, the validity interval can be extended for another period by renewing the timestamp as the current time.

In our real-time database model, transactions consist of three kinds as follows:

- *Update Transactions*: In a real-time database, base data objects should be updated periodically to reflect the current status of the real-world environment. Update transactions are write-only transactions specially designed for the purpose. For brevity, we call them *updates*.
- *Derived Data Recomputation Transactions*: To remain temporally consistent, a derived data has to be recomputed as the corresponding base data changes. The recomputation transactions may include multiple base data readings and arithmetic/logical operations before a write operation. For conciseness, we call them *recomputation*.
- *User Transactions*: A user-level transaction can be submitted to the real-time database with a deadline. User transactions include read-only transactions (queries) and read/write transactions. Note that user transactions do *not* write any continuous data object. User transactions are called *transactions* for short in the remainder of this proposal.

In our approach, each user transaction is associated with a certain deadline. The deadline of an update (a recomputation) is set to the update (recomputation) period. In the proposed approach, the outdated update or recomputation is not installed to the real-time database but aborted. Instead, the next update stream shall be reflected to the database. For scheduling, we initially consider EDF. A QoS-related scheduling issue is described in Section 5 in terms of the differentiated timeliness of transactions.

For admission control, we propose a simple but effective approach, in which a newly arriving transaction will be admitted to the system, if the requested server utilization estimate is currently available. The availability can be examined by aggregating the utilization estimates of the previously admitted transactions. We initially focus on the main memory database model to reduce the unpredictability due to I/O. In our main memory database model, CPU is the system resource of main consideration. To provide the durability, a disk resident back-end database can be employed. For concurrency control, we employ 2PL-HP [3]. 2PL-HP is selected, since it is free of priority inversion. Upon a conflict, a low priority transaction will be aborted and restarted.

2.2 QoS Model

In general, a simple and expressive QoS model is essential for effective QoS management. To express our QoS model, we first describe our performance metrics for QoS management, then introduce our QoS specification scheme.

2.2.1 Performance Metrics

Our QoS model includes four performance metrics:

- *Transaction Timeliness:* In a QoS specification, a user can specify the target deadline miss ratio, which can be tolerated. A QoS-sensitive real-time database should meet the miss ratio specified at the QoS contract for admitted transactions.

- *Data Freshness:* In real-time applications, time semantics of data is crucial for correct decision making. An incorrect decision can lead to a catastrophic result or a loss of profit. Examples include aircraft navigation system and online stock trading. We categorize data freshness into two classes: *database freshness* and *conditional freshness*. Database freshness implies the freshness of a database system, i.e., the ratio of fresh data to the entire data in a database. Conditional freshness measures the freshness of accessed data given that accessing transaction deadlines are met. It focuses on timely transactions, since only timely transactions can add value to our firm real-time database model. In contrast to database freshness, conditional freshness is measured only for a segment of database accessed by timely transactions. Note that only conditional freshness can be specified at the QoS contract. qRTDB supports the specified conditional freshness while managing the database freshness internally (hidden to the users). In this way, database freshness can be traded off to improve the timeliness, as long as conditional freshness can be supported as specified.

- *Differentiated Timeliness:* In a QoS negotiation, 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. In the performance evaluation, we will study if the relative service differentiation can be actually achieved given unpredictable workload.

- *Differentiated 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 can be updated frequently. Therefore, it can be computationally very expensive if a derived data object is recomputed on each base data change. We will study QoS adaptation techniques and the differentiated service scheme for derived data management.

2.2.2 QoS Specification

To provide general QoS support, qRTDB offers a simple yet powerful QoS specification interface, in which applications can specify the related QoS parameters and required performance based on the application-specific semantics. qRTDB can provide soft performance guarantees and enforce performance isolation between service classes, if the differentiated service is specified.

To specify the required QoS, a specific application should first specify whether or not the differentiated service is required. If the differentiated service is not required, the desired QoS can be expressed by a *fair service QoS-spec* (MR, FR_BASE, FR_DERIVED) such that:

- MR is the desired transaction timeliness in terms of the overall deadline miss ratio.

- FR_BASE is the desired conditional freshness of base data.

- FR_DERIVED is the desired conditional freshness of derived data.

For example, a fair QoS-spec (5%, 90%, 90%) specifies that the desired overall miss ratio is 5%, and the desired conditional freshness of base/derived data is 90%, respectively.

If the differentiated service is required, the desired QoS can be specified by a *differentiated service QoS-spec* (PERF_METRIC, NUM_CLASSES, DIFF_DEGREE) such that:

- PERF_METRIC is the performance metric to differentiate in a relative fashion, i.e., transaction timeliness or derived data freshness.

- NUM_CLASSES specifies the number of service classes.
- DIFF_DEGREE specifies the degree of service differentiation.

For example, a differentiated QoS-spec (Timeliness, 2, {1:2}) requires the timeliness to be differentiated between two classes in terms of relative miss ratios.

3 Adaptive Update Policy

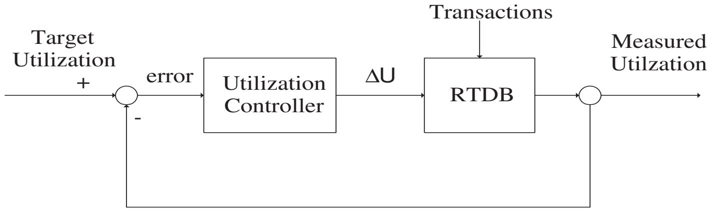
In this section, we discuss the feedback control real-time scheduling and our QoS adaptation scheme to enhance timeliness under overload. In this section, we mainly focus on the base data freshness issue. The freshness management issue of derived data is covered in section 4. We propose a novel approach to provide both timeliness and freshness guarantees in a real-time database. Despite the abundance of the research in real-time database, it is relatively scarce to explicitly provide both timeliness and freshness guarantees. First, we define a cost-effect model to quantify the utility of an update for a specific base data object. From the model, a novel database QoS adaptation technique is derived to provide graceful quality degradation under overload to enhance timeliness. Generally in a database system, update policy can not be adjusted based on the current workload or system behavior. However, we claim that update policy should be dynamically adaptable to efficiently balance the update and transaction workload.

3.1 Feedback Control Model

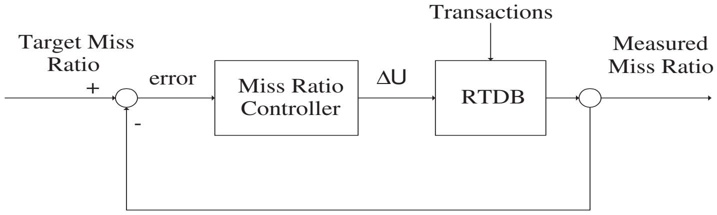
It is a challenging research problem to provide a certain timeliness guarantee for transactions that are *aperiodically* arriving in an unpredictable manner. Precise transaction execution time is unknown a priori in our workload model. We have to consider two performance metrics concurrently, namely transaction timeliness and data freshness for successful database services. Furthermore, the blocking due to data contention and wasted CPU utilization due to potential transaction aborts and restarts can complicate the controller design.

We propose to extend the general feedback control real-time scheduling framework FC-UM [29] to support QoS management in the real-time database. FC-UM employs two control loops for miss ratio and CPU utilization management respectively to avoid the control saturation problem, i.e., a utilization controller may saturate at utilization 100% and over while a miss ratio controller can saturate when the real-time system is underutilized (miss ratio is zero as a result). Each control loop generates a control signal to achieve the target utilization or miss ratio based on the current performance error, which is the difference between the target miss ratio (utilization) and the currently measured miss ratio (utilization). Each controller computes the control signal, called requested CPU utilization adjustment ΔU, to achieve the target miss ratio (or target utilization). A QoS manager is employed to enforce the utilization adjustment. If ΔU is negative (i.e., the system is overloaded), the QoS is degraded to reduce the requested CPU utilization and vice versa. We propose a novel QoS management scheme in the real-time database based on FC-UM to provide soft guarantees of timeliness and data freshness. The block diagram of the proposed feedback control system architecture is depicted in Figure 1. We will investigate to refine the control model and controller to consider the effects of blocking and wasted CPU utilization.

In fact, timeliness and freshness can compete for system resources, therefore, the control model design could be complex. If timeliness and freshness controllers are ignorant of each other, they can compete the CPU utilization to enhance their own performance metric in a selfish manner. As a result, it may end up with an unstable controller. Instead, we propose a simple but powerful approach, in which the freshness control logic collaborates to improve timeliness when overloaded. The freshness control logic helps to improve the miss ratio at the expense of the freshness of data with low update utility, which will be updated in a lazy manner under overload. The update utility is the reflective information derived from our cost-effect model described below. The key idea



(a) Utilization Control Loop



(b) Miss Ratio Control Loop

Figure 1: System Architecture

in our QoS adaptation is to reduce the number of overall updates, and utilize the remaining CPU utilization to improve timeliness when overloaded. This is achieved by adapting the update policy.

Note that a meaningful adaptation should not severely affect the user perceived freshness. Otherwise, the success of database services can be risked. To maintain the user perceived freshness, our adaptation scheme guarantees target *conditional freshness* as specified at the QoS contract, therefore, timely transactions can get a certain degree of freshness for accessed data. The QoS controller can deny QoS degradation, or even require more CPU utilization, if the target *conditional freshness* is violated. This mechanism can prevent the potential degradation of user perceived freshness for committed transactions.

3.2 Cost-Effect Model of Updates

In a real-time database, the update workload of continuous data might not be trivial. For example, a real-time database in a modern aircraft may include hundreds of sensors updated at high rates. In the stock quotes example, the update stream can reach up to 500 updates/second in peak time [17]. 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 [8]. To balance the update and transaction workload efficiently, an economic model is introduced to estimate the update utility. The cost is defined as the update frequency of a continuous data object. Intuitively, the more frequent is the update, the higher cost is attributed to the system. We assume that the frequency of periodic updates is known to the database system. To consider the effect, access frequency is measured for each data object. If a data object is accessed frequently, an update of the object can produce a relatively high effect. To quantify the cost-effect relationship, we define the Access Update Ratio (AUR) for an arbitrary 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 2, which is a snapshot of a real-time database. In the figure, the data objects in the database are ordered by non-increasing value of AUR. If a data object is on or above the horizontal line of AUR = 1, the effect of the corresponding data update is worth the cost, since it is accessed at least as frequently as 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.

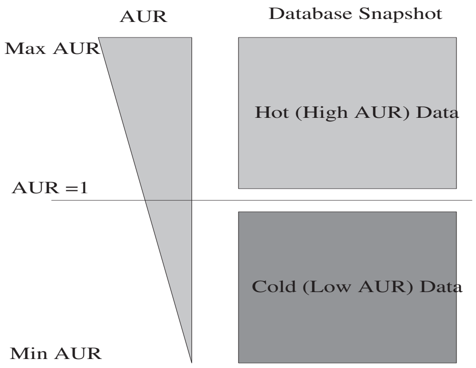


Figure 2: Database Snapshot Sorted by Access Update Ratio

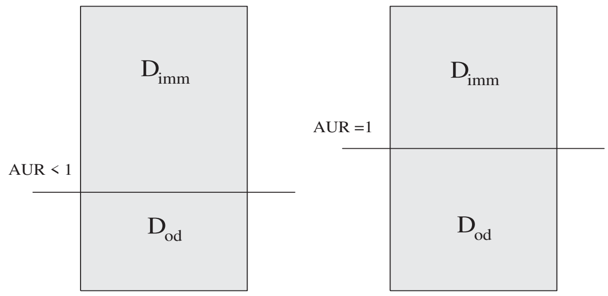
It is reasonable to update hot data in an aggressive manner. If a hot data object is out-of-date when accessed, potentially a multitude of transactions may miss the deadline waiting for the update, or can be forced to read stale data to meet the deadline; neither is desirable. Alternatively, it may not be necessary to update cold data aggressively when overloaded. Only a few transactions may miss deadlines waiting the update, or possibly read the stale version to finish in time. For cold data, we may be able to save the CPU utilization by applying a lazy update policy under overload. In fact, cold data could be always updated by a lazy update policy regardless of the current system load. However, that approach may increase the response time of the accessing transaction. We choose *immediate* and *on demand* policy as the aggressive and lazy update policy respectively. Let us assume that a new update arrives and installed into the database periodically without missing the update deadline, if updated immediately. Therefore, the corresponding data object can be regarded always fresh upon an access. In contrast, the freshness of cold data can be relatively low. A transaction may have to wait the next update or read the stale version to meet the transaction deadline.

Note that the notion of AUR does not depend on a specific access pattern or popularity model. It can be derived simply from the pre-known update frequency and monitoring the access frequency . Therefore, it greatly simplifies our QoS model and makes the model robust against potential unpredictability in data access pattern. Further refinement of the technique is endeavored to increase the robustness. The resulting QoS adaptation technique, namely dynamic update policy adaptation heuristic, is described as follows.

3.3 Dynamic Adaptation of Update Policy

Before a formal description, let us give a pictorial example of dynamic update policy adaptations based on the cost-effect model. In Figure 3, D represents the set of the all base data in the database. D_{imm} is the set of data updated immediately, and D_{od} stands for the set of data updated on demand. Since a data object is updated either immediately or on demand in our approach, $D = D_{imm} \bigcup D_{od}$ and $D_{imm} \bigcap D_{od} = \emptyset$. Initially, every data is updated immediately. As the load increases, a larger fraction of cold data objects are updated on demand. In Figure 3(c), the update policy degradation stops once it reaches a certain degradation bound, defined below. This figure will be revisited for detailed explanation of our adaptation policy later in this section.

The adaptation heuristic consists of three parts: (1) main procedure (2) degradation procedure, and (3) upgrade procedure. A brief description of the main procedure is in Table 1. In the main procedure, statistics are collected to compute $AUR[i]$ for each data object O_i in the database. On each access of O_i, the access counter $ACCESS[i]$ is incremented, which is initially 0 at a sampling period. Unfortunately, the access frequency may have a large deviation from a sampling period to another. To smooth the potentially large deviation, we take a moving average



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

Figure 3: Update Policy Adaptations

Table 1: Main Procedure for Update Policy Adaptation

1. Collect access statics and compute AUR.
2. Monitor deadline miss ratio, CPU utilization, and conditional freshness.
3. Call the degradation procedure, if the measured miss ratio is over the target miss ratio.
4. If CPU is underutilized or the measured conditional freshness is below the target conditional freshness, call the upgrade procedure. The upgrade is possible as long as $\Delta U \geq 0$.
5. After adaptation, inform the admission controller newly adjusted CPU utilization ΔU.

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 and vice versa.

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:

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

From the above, we can observe that our cost-effect model is not limited to periodic updates. In fact, the update frequency can be monitored and smoothed in the same way as the access frequency to handle aperiodic updates. 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 continuous data object can be considered stale upon the arrival of the corresponding update, which is not applied yet [9]. With the modifications, our cost-effect model can be directly applicable for aperiodic updates. Therefore, we mainly focus on periodic updates in this proposal for the sake of simplicity without losing generality.

If the current miss ratio is over the target miss ratio, the update policy degradation procedure is called. Based on the CPU load adjustment ΔU, which is negative when degradation procedure is called, the update policy is changed to reduce the number of updates and the corresponding CPU utilization for a certain data object O_i currently in D_{imm} with the least AUR in D_{imm}. Note that QoS adaptation is performed first before the admission control in our proposed approach. In this way, the variation to the incoming workload can be reduced, thereby, settling time (time to reach the steady state) can be reduced under the transient workload change.

An important question is how to estimate the CPU utilization saving from each degradation. The main interest is the difference between the required CPU utilization for purely immediate updates and for adaptive updates. The number of saved updates due to the degradation 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.

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 $\Delta U \approx 0$ or the degradation bound is reached. The horizontal line at AUR = 1 in Figure 2 and Figure 3(c) is the *update policy degradation bound*, which limits the degradation within a certain range of AUR. Further degradation past the degradation bound is meaningless due to the following reason. If a hot data object with $AUR \geq 1$ is updated on demand, the number of updates may not be reduced, but a transaction may have to suffer the potential delay for corresponding update or the reduced data freshness. An example of update policy degradations is shown in Figure 3. Initially, every update is performed immediately. Update policy is gradually degraded as the load increases, and the degradation stops if no more adaptation is allowed either by the violation of conditional freshness or reaching the degradation bound.

Sometimes update policy has to be upgraded for certain data objects to ensure that timely transactions access fresh data, i.e., to guarantee the specified conditional freshness. Different from the degradation, an upgrade can occur in two ways: if CPU is underutilized, i.e., $\Delta U > 0$, or the measured conditional freshness is below the target freshness. In the former, we can take a similar approach with the degradation. We can iteratively upgrade the update policy until ΔU reaches near zero. Here, the extra CPU requirement for immediate updates will be subtracted from the initial ΔU. In the latter case, we can upgrade the update policy until target freshness is reached, or until no more extra CPU utilization for the upgrade is available, i.e., $\Delta U \leq 0$.

A question is how to tell if the target freshness could be met in the next sampling period from certain upgrades. Since immediately updated data objects can be regarded always fresh as discussed before, the freshness control problem can be simplified. Furthermore, since it is clear that the on demand updates failed to provide the conditional freshness in the current sampling period, some of these data should be updated immediately in the next sampling period. Given a specified target conditional freshness F_{target}, the current conditional freshness $F_{current}$ can be measured in a sampling period. Then the performance error in terms of conditional freshness in a sampling period is:

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

The freshness can be improved approximately in proportion to the number of data moved from D_{od} to D_{imm}. We can achieve the required enhancement of conditional freshness by moving from D_{od} to D_{imm} a certain number of data with the highest AUR in D_{od}. To control the upgrades in a systematic way, we set a *update policy upgrade bound* in terms of the number of data for upgrades. Then the upgrade bound K is:

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

where D'_{imm} is the set of data to be updated immediately in the next sampling period after the upgrade.

By moving *one* data object O_i from D_{od} to D_{imm}, the number of updates and the corresponding CPU utilization may increase. The potential increase in the number of updates can be approximated from Eq. 4. We can estimate the required extra CPU utilization for the increase by Eq. 5. It continuously moves data until it has to stop the upgrade. It stops if the freshness improvement is expected to be achieved by moving the certain number of data computed from Eq. 8, or the available CPU utilization is not enough, i.e., $\Delta U \leq 0$.

In our approach, QoS level in terms of database freshness is dynamically adjusted based on the current miss ratio, utilization, or conditional freshness. Due to the approximation, unpredictable workload and access pattern, the QoS adaptation may not be precise in our approach. However, the target performance can be converged by continuously adjusting the QoS level based on the performance error in a feedback loop. By balancing update and transaction workload efficiently, target transaction timeliness can be achieved in a feedback control model while providing the specified conditional freshness.

4 Management of Derived Data Freshness

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 can include complex logical and/or arithmetic computations, and the recomputation can 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 propose a novel database QoS adaptation technique, which can batch the derived data recomputations considering the workload. Also, a differentiated service mechanism is proposed to further increase the performance improvement.

4.1 Adaptive Batching of Derived Data Recomputations

For database QoS adaptation, we propose to apply the notion of forced delay [9], which can batch the derived data recomputations during the delay 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 [9] 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 [10]. A possible disadvantage of the approach is that the accessing transaction may have to tolerate relatively long response time due to on demand recomputations.

We propose a novel QoS management scheme, in which 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 for every data in the database as the overload situation continues. When the system is underutilized, the derived data can be recomputed immediately. As the load increases, the recomputation is delayed. The amount of the delay can be increased, if necessary. 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:

$$AT[i] = \frac{Sampling\ Period}{SACCESS_{derived}[i]} \quad (9)$$

Table 2: Degradation Procedure for Derived Data Recomputation Policy

1. Increase the forced delay FD by Δt.
2. If $FD \geq AT[i]$ for a cold derived data object O_i, update it on demand. If O_i is hot, stop the degradation (do not increase FD).
3. Compute the estimated CPU utilization saving, δU.
4. The new CPU load adjustment is: $\Delta U = \Delta U_{old} + \delta U$.
5. If $\Delta U \geq 0$, stop the degradation. Else go to step 1.

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 in Section 3. 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 conditional freshness of the derived data. The derived data can be either pre-computed before the access or can be recomputed on demand.

The recomputation policy degradation procedure is briefly described in Table 2. The main procedure to initiate the upgrade/degradation and the upgrade procedure, which can be invoked when underutilized, are not included due to the limited space. They are conceptually similar to those described in Section 3.

4.2 Differentiated Service in Derived Data Recomputation

Considering the relatively high cost for derived data management, the real-time database performance can be further improved by a differentiated service. For example, aircraft position data (derived from various sensor data changing frequently) can be classified into two classes: friend or foe. Avionics applications require higher freshness on the enemy aircraft positions due to the criticalness in a complex air combat scenario. The freshness of aircraft position data can also 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-effect model described earlier can be extended. The extension can be straightforward considering the generality of the proposed cost-effect model. In practice, we propose to differentiate the length of the forced delay increase between derived data classes. The QoS adaptation algorithm in Table 2 can be extended to support differentiated service as follows. First, increase the forced delay for cold data class by $k\Delta t$, in which $k \geq 1$ specified at the QoS contract. If the saved CPU utilization is expected not enough, the hot data recomputation is delayed by Δt. The procedure iterates until the estimated CPU utilization save is enough to handle the current overload. By the freshness differentiation, the scalability of database services can be improved. The improvement could be non-trivial considering the cost of derived data recomputations. Finally, 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 Timeliness

In some applications, the timeliness of transactions can be differentiated between classes. For example, the query (read-only transaction) response time can be differentiated between classes in a frequently overloaded real-time database. The differentiated service can be provided in terms of relative transaction deadline miss ratios between classes, e.g., 1:2 between premium and basic classes. We propose to investigate a novel control model and controller for timeliness differentiation. A thorough investigation of the adverse effects of blocking and wasted utilization on the performance and predictability is needed for the control design. We propose to extend the high level semantics of proportional share CPU scheduling algorithms such as [41, 31, 43]. In a pure proportional share scheduling algorithm, no performance guarantee can be provided unless the admission is controlled, since the share of CPU can be arbitrarily low [34]. Furthermore, relative miss ratios can *not* be achieved by simply allocating CPU utilization in fixed proportions when the per class workload may fluctuate unpredictably. In our real-time database model, admission controller is employed to provide the performance guarantee. Without knowing future workload, a feedback control approach can achieve the relative service differentiation by dynamically adjusting per-class CPU share in a feedback loop.

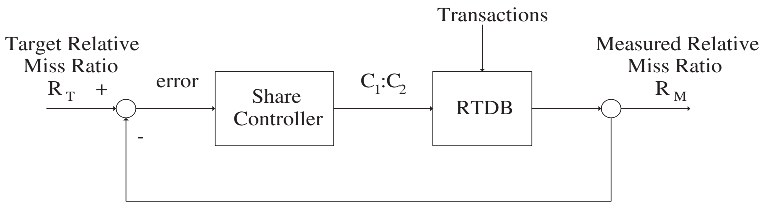


Figure 4: A Feedback Control Loop for Differentiated Timeliness

A feedback control real-time scheduler architecture consists of two control loops: utilization control loop and CPU share control loop to achieve the high utilization and relative miss ratios specified by the QoS contract. The architecture is shown in Figure 4. The utilization control loop is not depicted, since it is similar to Figure 1(a). The utilization control loop is activated when underutilized. Timeliness is differentiated under overload by the share control loop. In this way, system responsiveness and scalability can be improved while avoiding potential underutilization.

Intuitively, if more CPU utilization is allocated to a service class, the average deadline miss ratio can improve for this class. If per class workload were known in advance, we could tell the required CPU utilization to achieve the target performance. Without a priori knowledge, a feedback control is needed to dynamically adjust the allocation of CPU utilization based on the difference between the target performance and measured performance in terms of relative miss ratios between classes. When the specified relative miss ratio is $K_1 : K_2$ between two classes, the target relative miss ratio is:

$$R_T = \frac{K_1}{K_2} \tag{10}$$

Given the measured miss ratio M_i in each service class, the measured relative miss ratio is:

$$R_M = \frac{M_1}{M_2} \tag{11}$$

From Eq. 10 and Eq. 11, the performance error at a sampling period is defined:

$$error = R_T - R_M \tag{12}$$

In the feedback loop, CPU utilization shares between the premium and basic classes, $C_1 : C_2$ where $C_1 + C_2 = 1$, are dynamically adjusted based on the current performance error to achieve the desired service differentiation in terms of relative miss ratio.

6 Performance Evaluation and Testbed

A thorough evaluation will be performed as a part of this research. qRTDB will be tested mainly in two ways: via simulation and realistic experiments in a testbed. For the evaluation, we assume a centralized database model running on a single processor.

6.1 Simulation

We hypothesize that the proposed approach could improve the database performance and the service availability in terms of miss ratio and throughput under overload while providing the specified freshness. To evaluate our QoS architecture, a comprehensive simulation study will be performed first to substantiate the feasibility of our cost-effect hypothesis and control theoretical approaches for database QoS management. By simulation we can get insights into the performance of our QoS-sensitive real-time database model compared to other real-time database models unaware of database QoS.

6.1.1 Experiments

For performance evaluation, we propose several simulation studies as follows.

- *Experiment 1:* A simplistic baseline can be a real-time database model with EDF. The proposed admission control scheme, described in section 2.1, can be turned on and off to compare the relative performance gain provided by the admission control scheme. Deadline miss ratio, freshness, and throughput can be compared to measure the performance improvement.

- *Experiment 2:* The second baseline can be a feedback control real-time scheduling framework such as FC-UM [29] without database QoS management. Therefore, the data is always updated immediately. The performance can be compared to the proposed fair service model of qRTDB, which does not differentiate the service but employs the QoS adaptation scheme.

- *Experiment 3:* The differentiated service model of qRTDB can be compared to the fair service model. This experiment consists of two subcomponents: timeliness and freshness differentiation models. Furthermore, the performance of the two differentiated service models can be compared to each other.

The performance of the feedback control real-time system can be further studied by the transient metrics such as overshoot and settling time[27].

6.2 Realistic Application and Testbed Evaluation

From the simulation study, we can shed light on the run-time semantics of QoS-sensitive RTDB, performance improvement, implementation issues, and potential improvement of the QoS model itself. However, the simulation study is limited to the synthetic workload, therefore, the applicability to real world applications could not be sufficiently evaluated. For this reason, we plan to implement qRTDB model on an actual database system for a realistic application. An interesting test can be the emulation of a web server backed by a database system, e.g., a financial news service such as stock quotes. Through a QoS specification interface, it can specify required QoS to the database system, which can provide predictable services in a QoS-sensitive manner. The database system performance will be evaluated to determine if the specified QoS can be provided given a realistic workload. The trace of the stock price changes can be monitored on the web for a day or two to model the actual updates to the continuous data, while the client requests (queries) can be emulated by a web workload generator such as Surge [12].

There are three candidates for the testbed: BeeHive [39], MySQL [1], and PostgreSQL [2]. BeeHive is a real-time database being developed at University of Virginia. We have access to the source code and development experience. MySQL and PostgreSQL are open source databases popular in web applications. Furthermore, they support most of SQL (Structured Query Language) features.

Note that the implementation of a realistic experiment may include an enormous implementation effort due to the inherent complexity of database systems. To address the implementation issue, we also propose an alternative approach, in which the web server will be extended to facilitate simple database functionalities for our experimental purpose. Continuous data, e.g., stock prices, can be cached as materialized views at the web server, possibly in a shared memory between the back-end database and the front-end web server. The front-end web server can measure the update utility and inform the back-end database of the appropriate update policy for a continuous data object. In the stock quote example, the extended web server can generate an answer for a query from the materialized stock prices based on the client portfolio. In this approach, every performance-critical tasks can be handled in the front-end, whereas the back-end database can provide durability and update the materialized view. We expect this approach could reduce the implementation effort while the proposed performance study is feasible.

7 Related Work

Recently, the notion of QoS has been well studied and widely applied to computational systems. Numerous QoS architectures were proposed; for a few examples refer to [4, 5, 6, 15, 21, 22]. In an adaptive QoS model, it is generally assumed that the users can tolerate quality fluctuations from switching among multiple service levels. System responsiveness can increase by gracefully degrading the quality level under transient overload. Furthermore, resource allocation can be optimized through the QoS-level adaptation. For example, QoS-sensitive near-optimal resource allocation algorithms were proposed in [25]. In multimedia applications, QoS adaptation techniques are studied in depth [14, 16, 20, 5, 24]. In [7], admission control and scheduling algorithm are described to provide better service to the premium class in web or multimedia servers. Several service differentiation architectures for web servers can be found in [7, 11, 13, 19].

Despite the abundance of the QoS research, QoS-related work is relatively scarce in the database system. Priority Adaptation Query Resource Scheduling (PAQRS) provided timeliness differentiation of query processing in a memory-constraint 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. Also, data freshness issues were not considered.

Stanford Real-Time Information Processor (STRIP) addressed the problem of balancing between the freshness and transaction timeliness [8]. In a real-time database, data should be maintained fresh to correctly reflect the status of the real-world environment, also transactions should be processed in a timely manner. To study the trade-off between freshness and timeliness, four scheduling algorithms were introduced to schedule updates and transactions, and the performance was compared. In their later work, a similar trade-off problem was studied for derived data [9]. Ahmed et al proposed a new approach to keep the temporal consistency of derived data [10]. Different from STRIP, an update of a derived data object is explicitly associated with a certain timing constraint, and triggered by the database system only if the time constraint can be met. By a simulation study, the relative performance improvement was shown compared to the forced delay scheme of STRIP. A main focus of our proposed approach is the trade-off problem between data freshness and transaction timeliness. However, different from their work, we also provide quality adaptation and differentiated service by the systematic QoS management.

The correctness of answers to database queries can be traded off to enhance the timeliness. A query processor, APPROXIMATE, can provide approximate answers depending on the availability of data or time [42]. An imprecise computation technique called milestone approach is applied by APPROXIMATE. In the milestone approach, the accuracy of the intermediate result increases monotonically as the computation progresses [26]. Therefore, the correctness of answers to the query could monotonically increase as the query processing progresses. A relational database system called CASE-DB [32] 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 a database system, timeliness can be enhanced by relaxing consistency. 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 [18]. An adaptable security manager is proposed in [37], in which the security level can be temporarily degraded to enhance timeliness. These performance trade-off schemes lack a systematic QoS management architecture and service differentiation.

A closed-loop scheduling algorithm, FC-EDF (Feedback Control Earliest Deadline First), is presented in [28]. Adjusting the quality level and admission control based on continuous feedback, FC-EDF can support soft timing guarantee when an accurate workload model is not available. In the feedback loop, miss ratio is measured and compared to the target miss ratio, and the real-time system behavior is adjusted based on the error. In FC-EDF^2, another control loop is added to monitor the current CPU utilization to further improve the performance [27]. A good introduction to feedback control scheduling is available in [38]. FC-EDF and FC-EDF^2 were designed for *periodic* tasks. However, transactions can arrive *aperiodically* in a database system. A general feedback control real-time scheduling framework, FC-UM was presented in [29]. It expanded their previous work to provide the soft performance guarantee in terms of miss ratio given the mix of periodic and aperiodic tasks. Our proposed approach extends FC-UM for QoS management in the real-time database system. Furthermore, we propose to consider the blocking and wasted CPU utilization problem, which were not addressed in [29].

In [30], hit ratio of a web proxy cache is differentiated in proportion to the importance of the web content classes. To achieve the relative hit ratio between content classes (HTML text and image), per class cache space allocation is dynamically adjusted in a feedback loop. Feedback control theory was applied to Apache web server to achieve the relative delay among service classes via dynamic connection scheduling and process reallocation [29]. Analogously, per class CPU utilization is dynamically adjusted to achieve the relative miss ratio in our approach. However, our proposed differentiated service architecture is specifically designed for QoS management in a real-time database system dealing with CPU utilization.

8 Expected Contributions and Research Agenda

8.1 Expected Contributions

The demand for real-time information service is rising for applications such as agile manufacturing, online trading, e-commerce, mobile applications, and smart spaces. Due to the fast growth of the Internet, information services are being globalized. Despite the exploding demand, the performance of database systems, the core system component of the global information system, is relatively limited due to the difficulty of replication and the inherent unpredictability. It is possible that database systems can be a performance bottleneck of the global information infrastructure in the upcoming information age. To reduce the potential problem, we propose a QoS-sensitive real-time database system called qRTDB. We propose to support (1) predictable services in terms of soft guarantees for miss ratio and conditional freshness, (2) QoS adaptation, and (3) differentiated services. The current state-of-the-art from the real-time and QoS research will be integrated and further enhanced to enable QoS-sensitive real-time database services. In this way, the predictability, responsiveness and scalability of the database system can be improved. The impact can be dramatic considering the importance and criticalness of the database performance in the upcoming global information era. Scalability is an important issue to determine the feasibility of the global information infrastructure, in which the high service demand is expected. Timeliness and data freshness are core issues for satisfactory database services; the users need fresh data processed in time. The expected contributions can be summarized as follows.

- *Novel notion of predictable service in a real-time database*: Predictability in a real-time database implies timeliness guarantees in general. However, predictability in terms of timeliness is insufficient to define satisfactory database services. To address the problem, we extend the notion to include data freshness guarantees to provide soft guarantees for both timeliness and freshness.

- *A novel control model for systematic QoS management in a real-time database*: The adverse effect of blocking and wasted utilization are handled effectively. A systematic real-time database QoS management framework is provided. Despite the unpredictable workload and blocking/wasted utilization problem, the database QoS can be guaranteed due to the robustness of the feedback control. The expected contribution is significant considering the relative scarceness of the related research.

- *Novel QoS adaptation and differentiated service techniques*: Database QoS adaptation and differentiated service techniques are provided. More specifically, an adaptive update policy is provided to manage the freshness of base data. QoS management schemes are provided to manage derived data freshness. Furthermore, transaction timeliness can be differentiated, if specified.

- *Improved Responsiveness and Scalability*: The responsiveness to the transient overload can be improved due to the QoS-sensitive approaches described above. Scalability of the real-time database system could also increase for a similar reason. As a result, the performance and predictability problem of database systems can be partially alleviated in a global information infrastructure.

8.2 Research Agenda

Due to the limited space, we briefly describe the key research issues associated with the proposed research as follows.

- *Database-specific QoS issues* have to be considered to find the fundamental factors directly affecting the database service quality. In the proposed research, transaction timeliness and data freshness are mainly considered as database QoS parameters. Miss ratio and conditional freshness are considered as the performance metrics to measure the success of the real-time database service.

- *Feedback control model for database QoS management* shall be studied. Specifically, the effect of blocking and wasted utilization can change the real-time system dynamics. The feedback control model could effectively consider the disturbance.

- *Database QoS adaptation techniques* need to be developed considering the database-specific QoS parameters and performance metrics. The techniques should consider the database-specific characteristics in an efficient manner to maximize the benefit. Our main focus is to improve the timeliness under overload while not severely affecting the user perceived data freshness.

- *Differentiated service schemes* are considered to further improve the scalability of the real-time database system. Especially, it could be an interesting research issue to study the performance improvement by the proposed freshness management scheme for derived data and the potential impact on the user perceived derived data freshness.

- *Real world applications* shall be considered to show the applicability of the QoS-sensitive real-time database. Related applications will be studied to discuss the fundamental research and implementation issues. Based on the study, the QoS model could be improved. A testbed shall be constructed for realistic experiments.

8.3 Research Time-line

Finally, we present an approximate time-line to complete the various parts of the proposed research. The time-line is based on the current plan for the future progress, which can be refined to consider the specific research issues of interest as they rise.

- 3/01 - 7/01: A case study is being performed to verify the feasibility of the proposed approach. The study includes a simulation study to analyze the performance improvement due to the proposed QoS adaptation techniques. The existing control model of FC-UM [29] is extended to support QoS adaptation in the real-time database. In parallel, a new control model will be investigated to consider the blocking and wasted utilization problem in a database system.
- 8/01 - 12/01: The feedback control model will be extended to support the differentiation of the transaction timeliness between classes. We will extend the simulator to support the differentiated service in terms of the derived data freshness and transaction timeliness, respectively. A detailed simulation study will be performed to evaluate the performance improvement due to the differentiated service. Further research and implementation issues will be investigated.
- 1/02 - 7/02: We will implement and perform a realistic experiment based on the database QoS management scheme. We will design and implement a realistic application on a testbed. The experiment will show the robustness of the proposed QoS-sensitive real-time database under unpredictable workload.
- 8/02 - 12/02: The thesis will be written while revisiting the research issues with the results from the study. The overall experience and insights from the proposed research will be described in a coherent manner including the experimental results.

References

[1] MySQL. Available on the web, http://www.mysql.com.

[2] PostgreSQL. Available on the web, http://www.pgsql.com.

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

[4] T. F. Abdelzaher, E. M. Atkins, and K. G. Shin. QoS Negotiation in Real-Time Systems and Its Application to Automated Flight Control. In *Real-Time Technology and Applications Symposium*, Montreal, Canada, June 1997.

[5] T. F. Abdelzaher and K. G. Shin. End-Host Architecture for QoS-Adaptive Communication. In *Real-Time Technology and Applications Symposium*, Montreal, Canada, June 1997.

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

[7] 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.

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

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

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

[11] J. Almedia, M. Dabu, A. Manikntty, and P. Cao. Providing Differentiated Levels of Service in Web Content Hosting. In *First Workshop on Internet Server Performance*, Madison, Wisconsin, June 1998.

[12] P. Barford and M. E. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of Performance '98/ACM SIGMETRICS '98*, 1998.

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

[14] A. Cambell, G. Coulson, and D. Hutchison. A Quality of Service Architecture. In *ACM Computer Communications Review*, April 1994.

[15] S. Chatterjee, J. Sydir, B.Sabata, and T. Lawrence. Modeling Applications for Adaptive QoS-based Resource Management. In *Proceedings of the 2nd IEEE High-Assurance System Engineering Workshop*, Bethsda, Maryland, August 1997.

[16] D. Chen, R. Colwell, H. Gelman, P. K. Chrysanthis, and D. Mosse. A Framework for Experimenting with QoS for Multimedia Services. In *International Conference on Multimedia Computing and Networking*, 1996.

[17] M. Cochinwala and J. Bradley. A Multidatabase System for Tracking and Retrieval of Financial Data. In *Proceedings of the 20th VLDB Conference*, 1994.

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

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

[20] B. Field, T. Znati, and D. Mosse. V-net: A Framework for A Versatile Network Architecture to Support Real-Time Communication Performance Guarantees. In *IEEE INFOCOM*, 1995.

[21] 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.

[22] 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.

[23] Y. Kim and S. H. Son. Supporting Predictability in Real-Time Database Systems. In *Real-Time Technology and Applications Symposium*, pages 38–48, Boston, Massachusetts, June 1996.

[24] L. Krshnamurthy. *AQUA: An Adaptive Quality of Service Architecture for Distributed Multimedia Applications*. PhD thesis, University of Kentucky, 1997.

[25] C. Lee. *On Quality of Service Management*. PhD thesis, Carnegie Mellon University, 1999.

[26] 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.

[27] 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.

[28] C. Lu, J. Stankovic, G. Tao, and S. Son. Design and Evaluation of a Feedback Control EDF Algorithm. In *Real-Time Systems Symposium*, December 1999.

[29] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback Control Real-Time Scheduling: Framework, Modeling and Algorithms. Technical Report CS2001-5, Computer Science Department at University of Virginia, 2001.

[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] J. Nieh and M. S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *ACM Symposium on Operating System Principles*, St. Amlo, France, October 1997.

[32] 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.

[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] T. Plagemann, V. Goebel, P. Halvorsen, and O. Anshus. Operating System Support for Multimedia Systems. *Computer Communications Journal, Special Issues on Iterative Distributed Multimedia Systems and Telecommunications Services*, 1998.

[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] 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.

[38] 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.

[39] J. Stankovic and S. H. Son. An Architecture and Object Model for Distributed Object-Oriented Real-Time Databases. *Journal on Computer Systems Science and Engineering*, 14(4):251–259, July 1999.

[40] J. Stankovic, S. H. Son, and J. Hansson. Misconceptions About Real-Time Databases. *IEEE Computer*, 32(6):29–36, June 1999.

[41] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *IEEE Real-Time System Symposium*, December 1996.

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

[43] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, 1995.

[44] M. Xiong, R. Sivasankaran, J. Stankovic, K. Ramamritham, and D. Towsley. Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics. In *Real-Time Systems Symposium*, December 1996.