FLUTE: A Flexible Real-Time Data Management Architecture for Performance Guarantees *

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

July 1, 2002

Abstract

Efficient real-time data management has become increasingly important as real-time applications become more sophisticated and data-intensive. In data-intensive real-time applications, e.g., online stock trading, agile manufacturing, sensor data fusion, and telecommunication network management, it is essential to execute transactions within their deadlines using fresh (temporally consistent) sensor data, which reflect the current real-world states. However, it is very challenging to meet this fundamental requirement due to potentially time-varying workloads and data access patterns in these applications. Also, user service requests and sensor updates can compete for system resources, thereby making it difficult to achieve guaranteed real-time database services in terms of both deadline miss ratio and data freshness. In this paper, we present a novel real-time data management architecture, which includes feedback control, admission control, and flexible update schemes, to enforce the deadline miss ratio of admitted user transactions to be below a certain threshold, e.g., 1%. At the same time, we maintain data freshness even in the presence of unpredictable workloads and access patterns. In a simulation study, we illustrate the applicability of our approach by showing that stringent performance specifications can be satisfied over a wide range of workloads and access patterns. We also show that our approach can significantly improve performance compared to baselines.

1 Introduction

As real-time applications become more sophisticated and data-intensive, efficient real-time data management has become increasingly important. Data intensive real-time applications, e.g., online stock trading, agile manufacturing, sensor data fusion, and telecommunication network management, can benefit from the database support such as the efficient data access via indexing and correctness of concurrent transaction executions [21]. Unlike conventional (non-real-time) databases, it is essential for real-time databases to execute transactions within their deadlines, i.e., before the current market, manufacturing, or network status changes, using fresh (temporally consistent) sensor data[1], which reflect the current real-world states such as the current stock prices, automated process state, or network status. Current databases are poor in supporting timing constraints and data temporal consistency. 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 for

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

[1]In this paper, we do not restrict the notion of sensor data to the data provided by physical sensors. Instead, we consider a broad meaning of sensor data. Any data item, whose value reflects the time-varying real-world status, is considered a sensor data item.

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.

It is essential but very hard to process real-time transactions within their deadlines using fresh (sensor) data. Generally, transaction execution time and data access patterns are time-varying. For example, transactions in stock trading may decide whether or not to sell (or buy) a stock item considering the current market state. During the decision process, transactions may read varying sets of stock prices or composite indexes, if necessary. Transactions can be rolled back and restarted due to data/resource conflicts. Also, transaction timeliness and data freshness can often pose conflicting requirements: by preferring user transactions to updates, the deadline miss ratio is improved; however, the data freshness is reduced. Alternatively, the freshness increases if updates receive a higher priority [2].

To address this problem, we present a novel real-time data management architecture, called **FLUTE** (FLexible Updates for Timely transaction Execution), to achieve both miss ratio and freshness guarantees even in the presence of unpredictable workloads and data access patterns. To this end, FLUTE applies feedback-based miss ratio control, flexible update, and admission control schemes. We present novel notions of real-time QoD (Quality of Data) and flexible validity intervals generally applicable to real-time database applications. In FLUTE, the freshness can be traded off within a range of the specified QoD to improve the miss ratio, if necessary. Even after a trade-off, each data item is always maintained fresh in terms of (flexible) validity intervals. To our best knowledge, this is the first approach to guarantee both miss ratio and data freshness without any arbitrarily outdated data access.

In a simulation study, we provide stringent performance specifications to illustrate the applicability of our approach. The most stringent one requires that the miss ratio is below 1% without any freshness trade-off. Based on performance evaluation results, we show that FLUTE can achieve the required miss ratio and data freshness guarantees for a wide range of workloads and access patterns. FLUTE can also provide a choice between data freshness and throughput considering a specific application semantics without violating the specified miss ratio. In contrast, baseline approaches fail to support the specified miss ratio.

The rest of the paper is organized as follows. Section 2 describes our real-time database model. A flexible update scheme is presented in Section 3. In Section 4, our real-time data management architecture is described. Section 5 presents the performance evaluation results. Related work is discussed in Section 6. Finally, Section 7 concludes the paper and discusses future work.

2 Real-Time Database Model

In this section, we discuss the basic database model, real-time transaction types, and deadline semantics considered in this paper. Deadline miss ratio is defined in terms of both average and transient metrics.

2.1 Database Model and Transaction Types

We consider a main memory database model, in which the CPU is considered the main system resource. Main memory databases have been increasingly applied for real-time data management such as stock trading, e-commerce, and voice/data networking due to decreasing main memory cost and their relatively high performance [2, 5, 19, 24].

In our approach, transactions are classified as either user transactions or sensor updates. Continuously changing real-world states, e.g., the current sensor values, are captured by periodic updates. User transactions execute arithmetic/logical operations based on the current real-world states reflected in the real-time database to take an action, if necessary. For example, process control transactions in agile manufacturing may issue control commands considering the current process state, which is periodically updated by sensor transactions.

In our approach, each user transaction has a certain deadline, e.g., to finish a process control within a deadline. For each sensor transaction, the deadline is set to the corresponding update period. We apply firm deadline semantics, in which transactions can add value to the system only if they finish within their deadlines. A transaction is aborted upon its deadline miss. Firm deadline semantics are common in real-time database applications. For example, a late commit of a process control/stock trading transaction might adversely affect the product quality/profit due to the possible changes in the process/market state. Since database workloads and data access patterns might be time-varying as discussed before, we assume that some deadline misses are inevitable and a single deadline miss does not incur a catastrophic consequence. A few deadline misses are considered tolerable unless they exceed the threshold specified by a database administrator (DBA).

2.2 Deadline Miss Ratio

For admitted transactions, the deadline miss ratio is defined as: $MR = 100 \times \frac{\#Tardy}{\#Tardy + \#Timely}\ (\%)$ where #Tardy and #Timely represent the number of transactions that have missed their deadlines and the number of transactions that have committed within their deadlines, respectively. The DBA can specify a tolerable miss ratio threshold, e.g., 1%, for a specific real-time database application.

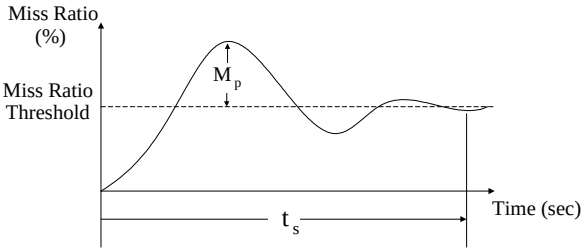

Figure 1: Definition of Overshoot and Settling Time in Real-Time Databases

Long-term performance metrics such as average miss ratio are not sufficient for the performance specification of dynamic systems, in which the system performance can be time-varying. For this reason, transient performance metrics such as overshoot and settling time are adopted from control theory for a real-time system performance specification [15] as shown in Figure 1:

- *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.
- *Settling time (t_s)* is the time for a transient miss ratio overshoot to decay and reach the steady state performance. In the steady state, the miss ratio should be below the miss ratio threshold.

In our approach, data freshness is also considered; however, we defer the discussion to Section 3 for the clarity of presentation.

3 Flexible Sensor Update Scheme

In this section, we discuss the motivation for our flexible sensor update scheme. The notion of data importance in real-time databases is discussed. We define novel notions of QoD (Quality of Data) and flexible validity intervals to measure and maintain the freshness of sensor data in the real-time database, respectively. Using these definitions, our QoD management scheme is discussed in detail. We also give a real-time database QoS specification describing the required miss ratio and QoD, which will be used to illustrate the applicability of FLUTE against unpredictable workloads and access patterns.

3.1 Motivation

In real-time databases, sensor data can be classified according to their relative importance, e.g., popularity of stock items. Under overload, it could be a reasonable approach to increase the update period of relatively unimportant (sensor) data to improve the user transaction miss ratio. For example, unpopular stock prices can be updated less frequently without affecting many user transactions. As another example, aircraft could be classified into hostile and friendly. Enemy aircraft positions can be updated frequently, while friendly aircraft positions far from the air base can be updated less frequently in a complex combat scenario. In such a case, timely defense actions, if necessary, can be taken at the expense of the potentially less frequent updates for some friendly aircraft. Our flexible sensor update scheme could be very effective in handling potential overloads. When overloaded, our approach can reduce the update workload, thereby reducing the possible conflicts between sensor updates and user requests. The QoD is degraded when the update period for a sensor data object is increased, since the corresponding sensor data item may represent a relatively old environmental state.

3.2 Data Importance

To apply the flexible sensor update scheme, the availability of data importance is required. The importance of data can be derived by a QoD management scheme itself or specified by a corporate user, e.g., a stock trading or automated manufacturing company, with particular knowledge about real-time data semantics for a specific application. In a limited scope, a QoD management scheme might be able to derive the importance of data. For example, in [8] access and update frequencies are monitored for sensor data to classify sensor data: a sensor data item is considered important if its access frequency (benefit) is higher than the update frequency (cost). However, for some real-time database applications this access update ratio may not be able to capture the importance of data, e.g., aircraft types and their potential threat.

In this paper, we consider an alternative approach to provide a more general QoD management scheme for real-time database applications: we assume that the importance of data, e.g., popularity of stock items, relative importance of manufacturing steps regarding the final product quality, and aircraft types and their potential threat, is available to corporate users, e.g., a financial trading/factory automation company and defense department. This is a reasonable assumption, since these corporate users (rather than real-time database developers) usually have better understanding about application specific data semantics, and our approach can support the required QoD. By allowing (corporate) users to (re)specify the required range of QoD, our QoD management scheme can directly reflect application specific QoD requirements. Within the specified range, the QoD can be degraded to improve the miss ratio, if necessary. For system operation and maintenance purposes, the DBA sets the QoD parameters (discussed in Section 3.5) to meet the QoD requirements specified by the user.

This sharply contrasts to current approaches for temporal consistency management in real-time databases such as [10, 21, 26], which neither provide users an interface to specify the required QoD nor are adaptive against potential overload. Existing real-time database work do not consider the adjustment of update periods regardless of the system behavior, and therefore, do not provide the notion of flexible validity intervals to maintain freshness after a possible QoD degradation. Task period adjustment is previously studied to improve the miss ratio in real-time (non-database) systems [12, 13]. However, data freshness is not considered in these work.

3.3 Quality of Data

In this section, we define a novel notion of QoD to measure the current freshness of sensor data in real-time databases. The notion of QoD was also introduced in other work [11], however, in these work the sensor update frequency is not relaxed to reduce the miss ratio. Therefore, these definitions of QoD are not directly applicable

to FLUTE. When there are N sensor data objects in a real-time database, we define the current QoD:

$$QoD = 100 \times \sum_{i=1}^{N} \frac{P_{i_{min}}}{P_{i_{new}}} \ (\%) \tag{1}$$

where $P_{i_{min}}$ is the minimum update period (before any QoD degradation) and $P_{i_{new}}$ is the new update period after a possible QoD degradation for a sensor data object O_i in the database. When there is no QoD degradation, the QoD = 100%, since $P_{i_{new}} = P_{i_{min}}$ for every sensor data object O_i in the database. The QoD decreases as $P_{i_{new}}$ increases. Using this metric, we can measure the current QoD for sensor data (compared to the minimum sensor update periods) in real-time databases.

3.4 Flexible Validity Intervals

In real-time databases, validity intervals are used to maintain the temporal consistency between the real-world states and sensor data in the database [21]. A sensor data object O_i is considered fresh (temporally consistent), if $(current\ time - timestamp(O_i) \leq avi(O_i))$ where $avi(O_i)$ is the absolute validity interval of O_i. The update period P_i for O_i is set to one-half of the $avi(O_i)$ to support the sensor data freshness [21][2]. Data temporal consistency can be violated if we set $P_i = avi(O_i)$. As shown in Figure 2, O_i can be updated at time t and the next update may commit at time $t + 2 \times P_i$. In this case, O_i is stale between $t + P_i$ and $t + 2 \times P_i$ even though the two updates commit within their deadlines, i.e., the corresponding update period P_i. This can be avoided by setting $P_i = 0.5 \times avi(O_i)$.

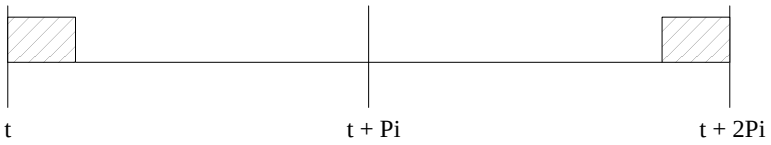

Figure 2: Periodic Updates for a Sensor Data Object

However, the notion of absolute validity intervals could be unnecessarily strict and inflexible: sensor data objects should always be updated at their minimum update periods regardless of their relative importance and the current system status. This is a common approach for freshness management in real-time databases, but it might not be cost-effective. As discussed before, under overload less important sensor data can be updated infrequently to reduce the number of deadline misses which can lead to the loss of profit, reduced product quality, or delayed defense actions. To maintain freshness even after a possible QoD degradation, we define a novel notion of *flexible validity intervals* (fvi). Initially, $fvi = avi$ for all data. Under overload, the update period P_i for a less important data object O_i can be relaxed. After the QoD degradation for O_i, we set $fvi_{new}(O_i) = 2 \times P_{i_{new}}$; O_i is updated at every one-half of $fvi_{new}(O_i)$ to maintain the freshness of O_i. Accordingly, after a QoD degradation O_i is considered fresh if $(current\ time - timestamp(O_i) \leq fvi_{new}(O_i))$.

3.5 QoD Management

In our approach, the DBA can set three QoD parameters to consider the corporate user's QoD requirements: Fixed-QoD, Max-Degr and Step-Size as follows.

- **Fixed-QoD**: For a certain fraction (ranging between $0-1$) of the entire sensor data in the database, the user can require the fixed QoD of 100%, i.e., no QoD degradation. For the other set of data in the database,

[2]Real-time databases may include derived data such as stock composite indexes. In this paper, we do not consider the derived data management and relative validity intervals.

called D_{degr}, the QoD can be degraded under overload. When *Fixed-QoD* = 1, no QoD degradation is allowed, i.e., $D_{degr} = \emptyset$. In contrast, when *Fixed-QoD* = 0 the QoD can be degraded for all data, if necessary. Users can select an appropriate value considering specific real-time database application semantics.

- **Max-Degr**: When *Fixed-QoD* < 1, users can also require to avoid an indefinite QoD degradation by specifying *Max-Degr*. For a sensor data object O_i, $P_{i_{new}} \leq$ *Max-Degr* $\times P_{i_{min}}$ after a QoD degradation. *Fixed-QoD* and *Max-Degr* can determine the worst possible QoD. For example, when *Fixed-QoD* = 0.7 and *Max-Degr* = 4 the lowest possible QoD is 77.5% = $100 \times$ (*Fixed-QoD* $+ (1 -$ *Fixed-QoD*$)/$*Max-Degr*$)\% = 100 \times (0.7 + (1 - 0.7)/4)\%$. In this case, for every sensor data object $O_i \in D_{degr}$, the current update period $P_{i_{new}} = 4 \times P_{i_{min}}$.

- **Step-Size**: Users can specify *Step-Size* to require the graceful QoD degradation, if any. For example, when *Step-Size* = 10%, $P_{i_{new}} = 1.1 \times P_i$ for $O_i \in D_{degr}$ after a QoD degradation. Thus, the update period P_i for O_i can be increased by 10% between two consecutive sampling periods, if necessary, to avoid a sudden QoD degradation.

By providing this interface to specify the required QoD, different QoD requirements for various real-time database applications, e.g., online stock trading and agile manufacturing, can be set as requested by users.

3.6 QoS Specification

To illustrate the applicability of our approach, we give a stringent QoS specification, called *QoS-Spec*, as follows:

- **Miss Ratio**: The average miss ratio should be below 1%. Overshoot (M_p) should be below 20%, therefore, the transient miss ratio should not exceed $1.2\% = 1 \times (1 + 0.2)\%$. The settling time should be shorter than 80sec. The sampling period for feedback control is set to 5sec. According to control theory [18], the overshoot and settling time usually have a trade-off relation. Hence, it is very hard, if possible, to optimize both overshoot and settling time. In this paper, we consider a marginal settling time increase tolerable as long as the corresponding potential miss ratio overshoot, which can lead to the loss of profit or reduced product quality, is low.

- **QoD Requirements**: We set *Max-Degr* = 4, therefore, $P_{i_{new}}$ for a sensor data object $O_i \in D_{degr}$ should not be longer than $4 \times P_{i_{min}}$ after a potential QoD degradation. We also set *Step-Size*= 10%. Hence, $P_{i_{new}} = 1.1 \times P_i$ after a QoD degradation, if necessary.

- **CPU Utilization**: To avoid underutilization, we aim to achieve at least 80% CPU utilization. Maximizing the CPU utilization is not the main objective of our approach; however, we can actually achieve a higher utilization than 80% due to the dynamic adjustment of the utilization threshold (discussed in Section 4.3.3).

For QoD management, we do not fix *Fixed-QoD* but consider it a workload variable, since it can directly affect the system adaptability by flexible sensor updates, if necessary. To measure the possible performance effects, we apply increasing *Fixed-QoD* for performance evaluation. In this way, we vary *QoS-Spec* posing more stringent performance requirements in terms of *Fixed-QoD*. For example, when *Fixed-QoD* = 1, no QoD degradation is allowed. (In this case, $D_{degr} = \emptyset$, and therefore, *Max-Degr* and *Step-Size* are irrelevant.) A detailed discussion is given in Section 5.

4 Real-Time Data Management Architecture

Figure 3 shows our architecture for real-time data management. A transaction is scheduled in one of the two ready queues according to its scheduling priority. The transaction handler executes queued transactions. At each

sampling instant, the current miss ratio and the CPU utilization are monitored. The miss ratio and utilization controllers derive the required CPU utilization adjustment, called ΔU as shown in Figure 3, considering the current performance error such as the miss ratio overshoot or CPU underutilization. Based on ΔU, the QoD Manager adapts sensor update periods to reduce the update workload, if necessary. The admission controller enforces the remaining utilization adjustment after potential update period adaptation, i.e., ΔU_{new}. A detailed discussion for the system components is given in the next subsections.

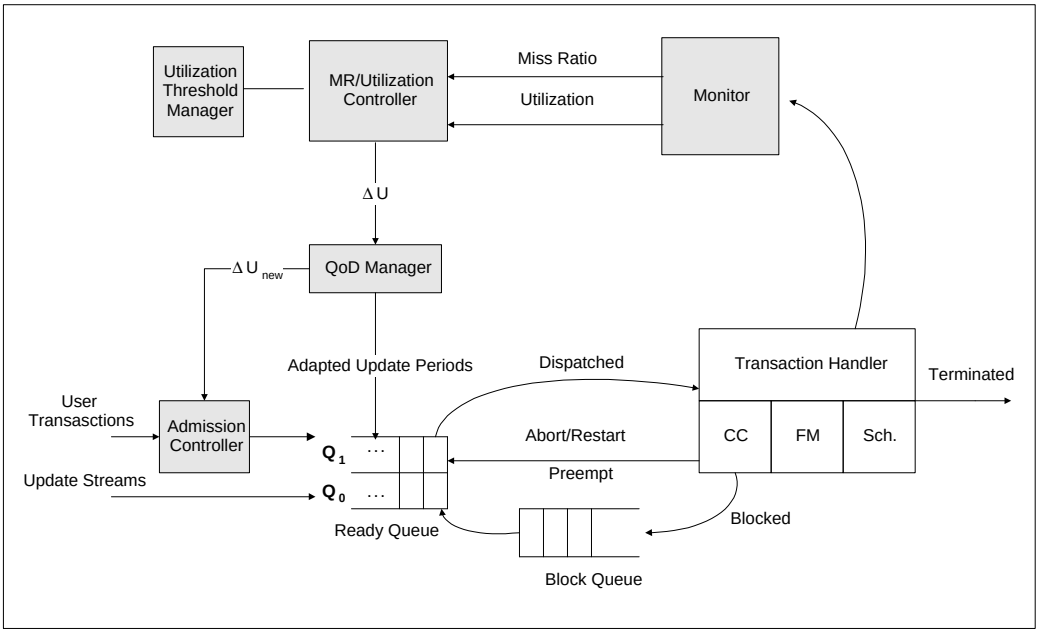

Figure 3: Real-Time Data Management Architecture for Performance Guarantees

4.1 Transaction Handler

The transaction handler provides an infrastructure for real-time database services, which consists of a concurrency controller (CC), a freshness manager (FM) and a real-time 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.

The FM checks the freshness before accessing a data item using the corresponding *avi* or fvi, if there was a QoD degradation. 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.

Transactions are scheduled in one of two ready queues (Q_0 and Q_1 as shown in Figure 3). A transaction in Q_1 can be scheduled if there is no ready transaction in Q_0, and can be preempted when a new transaction arrives to Q_0. In each queue, transactions are scheduled in an EDF (Earliest Deadline First) manner. To provide the data freshness guarantee, all sensor updates are scheduled in Q_0 in this paper. User transactions are scheduled in Q_1. One may argue that this could be relatively unfair for user transactions compared to sensor updates. However, user transactions will be blocked anyway if their accessing data items are currently stale. To consider the fairness, we apply a feedback-based approach to control the user transaction miss ratio below a certain threshold. Also, the sensor update frequency can be reduced to improve the user transaction miss ratio, if necessary. Hence, user transactions are not treated in a completely unfair manner.

To maintain data freshness, all sensor data are updated immediately when their new sensor readings arrive. This to avoid possible deadline misses due to the delay for lazy updates such as on-demand updates. When sensor

data are updated on demand, it is possible for user transactions accessing the corresponding data to miss their deadlines waiting for the on-demand updates. Or, they may have to use stale data, which can be arbitrarily old since the last on-demand update, to meet their deadlines [2].

1. Monitor the deadline miss ratio and CPU utilization.
2. At each sampling period, compute the miss ratio and utilization control signals, called ΔU_{MR} and ΔU_{util}, based on the current miss ratio and utilization, respectively. Get the required CPU utilization adjustment $\Delta U = Minimum(\Delta U_{MR}, \Delta U_{util})$ for a smooth transition from one system state to another. Based on ΔU, perform one of the following alternative actions.
3. If $\Delta U \geq 0$, admit more user transactions to avoid potential underutilization.
4. If $\Delta U < 0$, i.e., the specified miss ratio is violated, *Fixed-QoD* < 1, and *Max-Degr* is not reached yet, increase the sensor update periods by the *Step-Size* for sensor data objects in D_{degr}. Adjust $\Delta U_{new} = \Delta U$ + the CPU utilization saved from the QoD degradation. If $\Delta U_{new} < 0$ after the possible QoD degradation, apply admission control to newly incoming transactions.

Figure 4: Interactions between the Feedback Control and QoD Management/Admission Control

4.2 QoD Manager and Admission Controller

The interactions between the feedback control and QoD management/admission control are described in Figure 4. The deadline miss ratio and CPU utilization are measured at each sampling period, i.e., 5sec in this paper. The required CPU utilization adjustment ΔU is derived from the miss ratio/utilization feedback controllers shown in Figure 5.

If $\Delta U \geq 0$, i.e., the CPU utilization should be increased to achieve the target utilization, admit more transactions to prevent a potential underutilization. When the system is overloaded, i.e., the required CPU utilization adjustment $\Delta U < 0$, the CPU utilization should be reduced according to the current ΔU. Under overload, the QoD can be degraded if *Fixed-QoD* < 1 and the specified *Max-Degr* is not reached yet[3].

When the QoD manager can not entirely enforce ΔU, possibly due to the severe overload, admission control is applied to handle the potential overload, which can lead to the loss of profit or reduced product quality. Note that under overload it is impossible to support the specified miss ratio threshold if all incoming transactions are simply admitted. Instead, it is reasonable to control the admission to improve the miss ratio. Trade requests can be resubmitted under appropriate market status later, or a product can come back to a manufacturing unit through a loop conveyer belt.

An incoming transaction is admitted to the system if the requested CPU utilization is currently available. The current CPU utilization can be estimated by adding the CPU utilization estimates of the previously admitted transactions.

[3]The QoD is managed by the QoD manager (an actuator from the control theory perspective). We do not consider designing a separate feedback controller for freshness management due to the potential conflicts between the miss ratio and freshness requirements, which can lead to an unstable feedback control system.

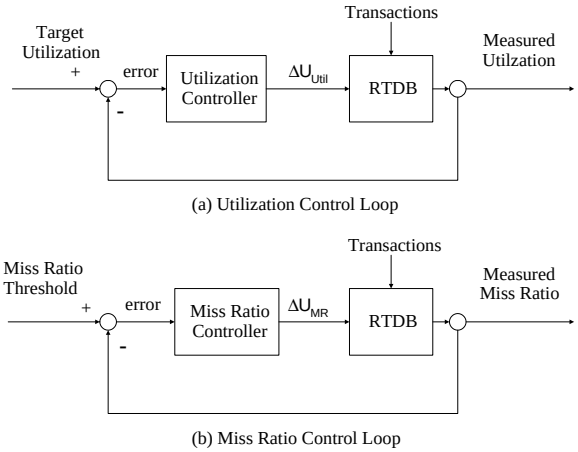

Figure 5: Miss Ratio/Utilization Controllers

4.3 Feedback Control

Feedback control is very effective in supporting a required performance specification when the system model includes uncertainties [18]. The target performance can be achieved by dynamically adapting the system behavior based on the current performance error measured in the feedback control loop. In this paper, we apply an extended version of a feedback control scheduling policy, called FC-UM [15]. FC-UM is selected since it can provide a certain miss ratio guarantee without underutilizing the CPU against unpredictable workloads. As shown in Figure 5, FC-UM employs two controllers. This is because 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). By using both miss ratio and utilization controllers, the required miss ratio can be achieved without underutilizing the CPU, since the saturation conditions for the two controllers are mutually exclusive.

4.3.1 Miss Ratio Controller

Based on the current miss ratio error, i.e., the difference between the miss ratio threshold and the current miss ratio measured by the Monitor as shown in Figure 5, the miss ratio controller computes the required CPU utilization adjustment, ΔU_{MR}, to support the specified miss ratio threshold. More specifically, at a sampling instant k the miss ratio control signal ΔU_{MR} is computed in a digital PI (proportional and integral) controller:

$$\Delta U_{MR} = KP \times Error_k + KI \times \sum_{i=1}^{k} Error_i \tag{2}$$

where $Error_k$ = miss ratio threshold − current miss ratio. KP and KI stand for the control gains for proportional and integral controllers, respectively. To guarantee the miss ratio required by *QoS-Spec*, the miss ratio controller should be tuned. For controller tuning, we profiled the miss ratio for workloads increasing from 60% to 200% by 10% under the worst case set-up, in which all incoming transactions are admitted and sensor data are updated at their minimum periods regardless of the current system behavior. Based on the profiling results, the control gains are chosen by applying the mathematically well established Root Locus method in Matlab [18], which can avoid ad hoc testing/tuning iterations. The utilization controller is also tuned using the Root Locus method. For more details of profiling and tuning, refer to [9].

4.3.2 Utilization Controller

A utilization control loop is employed to prevent a potential underutilization. This is to avoid a trivial solution, in which all the miss ratio requirements (in terms of both average and transient metrics) are satisfied due to the underutilization. At each sampling instant, the utilization controller computes the utilization control signal ΔU_{util} based on the utilization error, i.e., the difference between the target utilization and the utilization measured at the current sampling instant as shown in Figure 5. The utilization control loop uses a separate digital PI controller to compute ΔU_{util}, similar to Eq. 2, where $Error_k$ = target utilization − current utilization. At each sampling instant, we set the current control signal $\Delta U = Minimum(\Delta U_{util}, \Delta U_{MR})$ to support a smooth transition from one system state to another, similar to [15].

When an integral controller is used together with a proportional controller, the performance of the feedback control system can be improved. However, care should be taken to avoid erroneous accumulations of control signals by the integrator, which can lead to a substantial overshoot later [18]. For this purpose, the integrator antiwindup technique [18] is applied: turn off the miss ratio controller's integrator if $\Delta U_{util} < \Delta U_{MR}$, since the current $\Delta U = \Delta U_{util}$. Otherwise, turn off the integrator for the utilization controller. We further extend the utilization controller by employing the utilization threshold manager as follows.

4.3.3 Utilization Threshold Manager

For many complex real-time systems, the schedulable utilization bound is unknown or can be very pessimistic [15]. In real-time databases, the utilization bound is hard to derive, if it even exists. This is partly because database applications usually include unpredictable aborts/restarts due to data/resource conflicts. A relatively simple way to handle this problem is to set/enforce a pessimistic utilization threshold. However, this can lead to an unnecessary underutilization. In contrast, an excessively optimistic utilization threshold can lead to a large miss ratio overshoot. It is a hard problem to decide a proper utilization threshold in a complex real-time system such as a real-time database.

In this paper, we use an online approach for the dynamic adjustment of the utilization threshold (the target utilization in Figure 5 (a)). The utilization threshold is dynamically adjusted considering the current real-time system behavior as follows. Initially, the utilization threshold is set to a relatively low value, e.g., $U_{init} = 80\%$. If no deadline miss is observed at the current sampling instant, the utilization threshold is incremented by a certain step size, e.g., 2%, unless the resulting utilization threshold is over 100%. The utilization threshold will be continuously increased as long as no deadline miss is observed. The utilization threshold will be switched back to the initial utilization set point (i.e., U_{init}) as soon as the miss ratio controller takes control. This back-off policy might be somewhat conservative, however, we take this approach to prevent a potential miss ratio overshoot due to a relatively slow back-off. Also, the utilization threshold will be increased again, if no deadline miss is observed at later sampling instants. Using this self-adaptive and computationally light-weight approach, the potentially time-varying utilization threshold can be closely approximated.

5 Performance Evaluation

For performance evaluation, we have developed a real-time database simulator, which models the real-time database architecture depicted in Figure 3. Each system component in Figure 3 can be selectively turned on/off for performance evaluation purposes. The main objective of our performance evaluation is to show whether or not our approach can support the required miss ratio and QoD (described in *QoS-Spec*) even in the presence of a wide range of unpredictable loads and access patterns. In this section, we discuss the the simulation model, describe baseline approaches for performance comparison purposes, and present the performance evaluation results.

5.1 Simulation Model

In our simulation, we apply a workload consisting of sensor updates and user transactions which are summarized in Tables 1 and 2, respectively, and are discussed as follows.

5.1.1 Sensor Data and Updates

There are 1000 sensor data objects in our simulated real-time database. Each data object O_i is periodically updated by an update stream, $Stream_i$, which is associated with an estimated execution time (EET_i) and an update period (P_i) where $1 \leq i \leq 1000$. EET_i and P_i are uniformly distributed in a range (1ms, 8ms) and in a range (100ms, 50sec), respectively. Upon the generation of an update, the actual update execution time is varied by applying a normal distribution $Normal(EET_i, \sqrt{EET_i})$ for $Stream_i$ to introduce errors in execution time estimates. The total update workload is manipulated to require approximately 50% of the total CPU utilization when no QoD degradation is applied. This leaves the remaining 50% of the CPU utilization for user transaction processing.

In real-time database applications, the current real-world status is usually monitored by periodic updates, e.g., sensor readings and stock price trends. For example, financial trading tools such as Moneyline Telerate Plus [23], which is widely used in financial trading laboratories such as the Bridge Center for Financial Markets at University of Virginia and Sloan Trading Room at MIT, provide periodic stock price updates to observe the market trends. In Telerate Plus, users can select the update period in the range from 1 minute to 60 minutes for the real-time quote of an individual stock item. In this paper, we consider a more advanced system set-up, in which the update period for a sensor data item uniformly ranges between 100ms $-$ 50sec. Furthermore, the selected range of update periods can approximate the data requirements for other real-time database applications such as agile manufacturing, in which electro/mechanical robots are controlled using various sensor data.

Table 1: Settings for Sensor Data/Updates

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

5.1.2 User Transactions

A source, $Source_i$, generates a group of user transactions whose inter-arrival time is exponentially distributed. $Source_i$ is associated with an estimated execution time (EET_i) and an average execution time (AET_i). We set $EET_i = Uniform(5ms, 20ms)$. By generating multiple sources, we can derive transaction groups with different average execution time and average number of data accesses in a statistical manner. Also, by increasing the number of sources we can increase the load applied to the simulated real-time database, since more transactions will arrive in a certain period of time. We set $AET_i = (1 + EstErr) \times EET_i$, in which $EstErr$ is used to introduce the execution time estimation errors. Note that all approaches tested in this paper (including FLUTE) are only aware of the estimated execution time. Upon the generation of a user transaction, the actual execution time is generated by applying the normal distribution $Normal(AET_i, \sqrt{AET_i})$ to introduce the execution time variance in the user transaction group.

The number of data accesses for $Source_i$ is derived in proportion to the length of EET_i, i.e., $N_{DATA_i} =$ *data access factor* $\times EET_i = (5, 20)$. As a result, longer transactions access more data in general. Upon the

generation of a user transaction, $Source_i$ associates the actual number of data accesses with the transaction by applying $Normal(N_{DATA_i}, \sqrt{N_{DATA_i}})$ to introduce the variance in the user transaction group.

We set *deadline = arrival time + average execution time* $\times$ *slack factor* for a user transaction. A slack factor is uniformly distributed in a range (10, 20). For an update, we set *deadline = next update period*. For performance evaluation, we have also applied other settings for execution time, data access factor, and slack factor different from the settings described in Table 2. We have confirmed that our approach can support *QoS-Spec* by dynamically adapting the system behavior based on the error measured in the feedback control loop for different workload settings, but we do not include the results due to space limitations.

Table 2: Settings for User Transactions

Parameter	**Value**
EET_i	$Uniform(5ms, 20ms)$
AET_i	$EET_i \times (1 + EstErr_i)$
Actual Exec. Time	$Normal(AET_i, \sqrt{AET_i})$
N_{DATA_i}	$1 \times EET_i = (5, 20)$
#Actual Data Accesses	$Normal(N_{DATA_i}, \sqrt{N_{DATA_i}})$
Slack Factor	$Uniform(10, 20)$

5.2 Baselines

To our best knowledge, no previous research has applied feedback control and QoD adaptation to provide guarantees for both potentially conflicting miss ratio and freshness requirements despite unpredictable workloads and access patterns in real-time databases. For this reason, we have developed two baseline approaches as follows.

- *Basic*: In this approach, all incoming transactions are admitted and all sensor data are always updated at their minimum update periods. Hence, the QoD = 100% as long as sensor updates commit within their deadlines. All the shaded components in Figure 3 are turned off. Therefore, the feedback-based closed loop scheduling, admission control, and QoD adaptation are not applied. Note that most of database systems take this open-loop and non-adaptive approach.

- *Basic-AC*: This is a variant of *Basic*, for which the admission control policy (described in Section 4.2) is applied. For the fairness of performance comparisons, we apply the same admission control policy to FLUTE and Basic-AC.

5.3 Workload Variables and Experiments

To adjust the workload for experimental purposes, we define workload variables and describe the performed experiments using the workload variables.

5.3.1 Workload Variables

- $AppLoad$: Computational systems usually show different performance for increasing loads, especially when overloaded. We use a variable, called $AppLoad$ = update load + user transaction load $\approx$ 50% + user transaction load, to apply different workloads to the simulated real-time database. For performance evaluation, we applied $AppLoad = 70\%, 100\%, 150\%,$ and 200%. Note that this variable indicates the load assuming that all incoming transactions are admitted and each sensor data item O_i is updated at every

$P_{i_{min}}$. The actual load can be reduced in a tested approach by applying the admission control and QoD degradation, if necessary. The actual load is also related to another workload variable, i.e., *Fixed-QoD*.

- *Fixed-QoD*: For increasing *Fixed-QoD*, the overall QoD will increase, but less flexibility can be provided for overload management. We applied *Fixed-QoD* ranging from 0.5 to 1 increased by 0.1 to observe if the average/transient miss ratio specified in *QoS-Spec* can be supported for increasing *Fixed-QoD*.

  Given a *Fixed-QoD*, the resulting CPU utilization requirement for sensor updates after the full QoD degradation is approximately 50%$\times$ (*Fixed-QoD* + (1 $-$ *Fixed-QoD*)/4), since according to *QoS-Spec* $P_{i_{new}} = 4 \times P_{i_{min}}$ for every $O_i \in D_{degr}$ after the full QoD degradation.

  From this, in Table 3 we show the tested *Fixed-QoD* values, approximate update workload after the full QoD degradation, and load relieved from the full degradation. For example, when *Fixed-QoD* = 0.5 and $Appload$ = 150% the actual load can be reduced to approximately 130% after the full QoD degradation. The admission controller can handle the remaining overload to support the specified miss ratio and QoD, if necessary.

Table 3: Fixed-QoD vs. Update Workload

Fixed-QoD	0.5	0.6	0.7	0.8	0.9	1.0
Update Load	31.25%	35%	38.75%	42.5%	46.25%	50%
Relieved Load	18.75%	15%	11.25%	7.5%	3.75%	0%

- *EstErr* (Execution Time Estimation Error): $EstErr$ is used to introduce errors in execution time estimates as described before. We have evaluated the performance for $EstErr$ = 0, 0.25, 0.5, 0.75, and 1. When $EstErr$ = 0, the actual execution time is approximately equal to the estimated execution time. The actual execution time is roughly twice the estimated execution time when $EstErr$ = 1, since actual execution time $\approx (1 + EstErr) \times$ estimated execution time. In general, a high execution time estimation error could induce a difficulty in real-time scheduling.

- *HSS* (Hot Spot Size): Database performance can vary as the degree of the data contention changes [1, 7]. For this reason, we apply different access patterns by using the $x-y$ access scheme [7], in which x% of data accesses are directed to y% of the entire data in the database and $x \geq y$. For example, 90-10 access pattern means that 90% of data accesses are directed to the 10% of a database, i.e., a hot spot. When $x = y = 50\%$, data are accessed in a uniform manner. We call a certain y a hot spot size (HSS). The performance is evaluated for HSS = 10%, 20%, 30%, 40%, and 50% (uniform access pattern).

In this paper, we set $U_{init} = 80\%$, i.e., the initial target utilization in the utilization control loop is set to 80%. At run time, the target utilization can be dynamically adjusted considering the current miss ratio as discussed in Section 4.3.3. We also performed experiments for increasing U_{init}, and found that for increasing U_{init} the utilization and throughput slightly increase at the expense of increasing miss ratio overshoot. The detailed results are not included due to space limitations.

5.3.2 Experiments

Even though we have performed several sets of experiments for varying values of the workload variables, we present only the three most representative sets of experiments as summarized in Table 4 for the clarity of presentation. We have verified that all the experiments including Experiment Sets 1, 2, and 3, presented in this paper, show a consistent performance trend: our approach can provide a guarantee on miss ratio, while enforcing the

Table 4: Presented Experiments

Exp.	Varied	Fixed
1	$AppLoad = 70\%, 100\%, 150\%, 200\%$	EstErr = 0 Fixed-QoD = 0.5 HSS = 50%
2	*Fixed-QoD* = $0.5 - 1.0$	AppLoad = 200% EstErr = 1 HSS = 50%
3	$HSS = 10\% - 50\%$ (uniform access)	AppLoad = 200% EstErr = 1 Fixed-QoD = 1

QoD as required by *QoS-Spec* at the same time. In contrast, the baseline approaches fail to support the specified miss ratio in the presence of unpredictable workloads and access patterns. The presented experiments are discussed as follows.

- **Experiment Set 1**: As described in Table 4, performance is evaluated for $AppLoad = 70\%$, 100%, 150%, and 200%. No error is considered in the execution time estimation, i.e., $EstErr = 0$. Note that this is an ideal assumption since precise execution time estimates are usually not available. We also fix *Fixed-QoD* = 0.5, which is increased from 0.5 to 1 in Experiment Set 2. Hence, the best case settings in our experiments are applied to Experiment Set 1.

- **Experiment Set 2**: In this set of experiments, we set $AppLoad = 200\%$ and $EstErr = 1$, i.e., the worst case *AppLoad* and *EstErr* values applied in our experiments. We also increase *Fixed-QoD* from 0.5 to 1 by 0.1 to stress the modeled real-time database. As *Fixed-QoD* increases, the miss ratio may also increase due to the less adaptability against potential overload as discussed before.

- **Experiment Set 3**: We vary the hot spot size ($10\% - 50\%$) to observe whether or not *QoS-Spec* can be supported for varying degrees of data contention. As shown in Table 4, we set $AppLoad = 200\%$, $EstErr = 1$, and *Fixed-QoD* = 1. In addition to the worst *AppLoad* and *EstErr* values tested, we do not allow any QoD degradation. Hence, this is the worst case set-up among the tested simulation settings.

In our experiments, one simulation run lasts for 10 minutes of simulated time. For all performance data, we have taken the average of 10 simulation runs and derived the 90% confidence intervals. Confidence intervals are plotted as vertical bars in the graphs showing the performance evaluation results. (For some performance data, the vertical bars may not be noticeable due to the small confidence intervals.)

5.4 Experiment Set 1: Effects of Increasing Load

In this section, we compare the performance of Basic, Basic-AC, and FLUTE for increasing *AppLoad*.

5.4.1 Average Miss Ratio

As shown in Figure 6, for increasing *AppLoad* FLUTE shows a near zero miss ratio, which meets the specified 1% miss ratio threshold, while Basic and Basic-AC show a significant miss ratio increase. For FLUTE, we also measured the transient miss ratio and found that the specified overshoot and settling time are satisfied for all *AppLoad* values. We observed that the average/transient miss ratio was near zero for FLUTE throughout

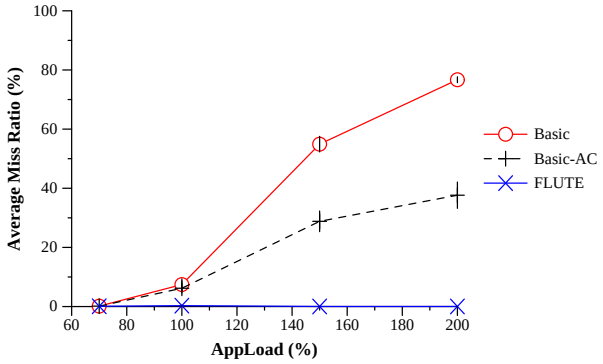

Figure 6: Average Miss Ratio for Exp. 1

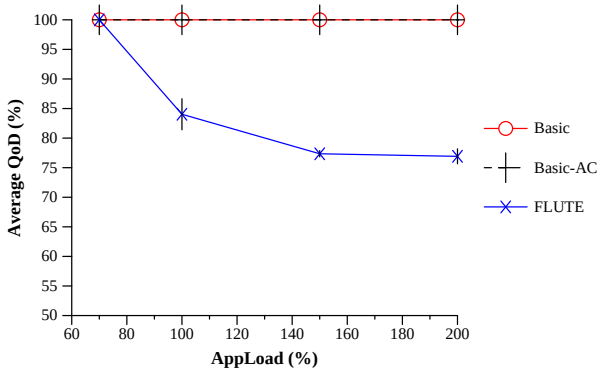

Figure 7: Average QoD for Exp. 1

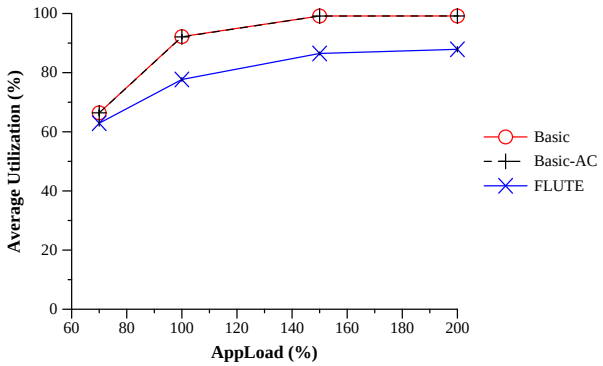

Figure 8: Average Utilization for Exp. 1

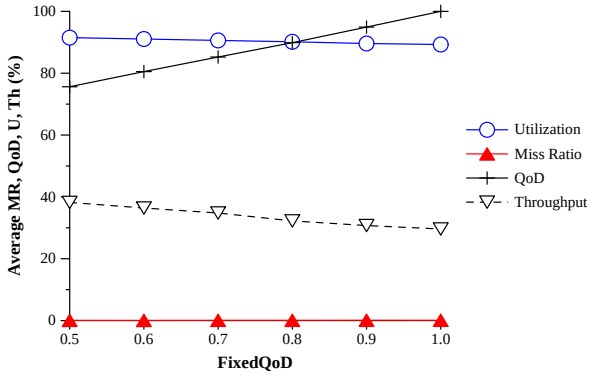

Figure 9: Average Performance of FLUTE for Exp. 2

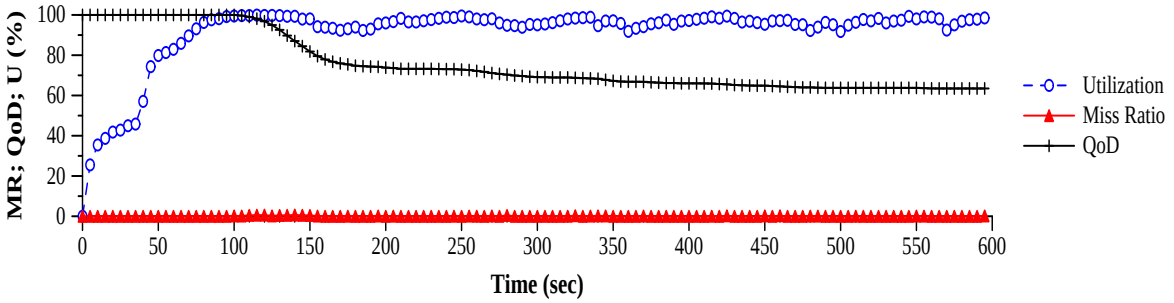

Figure 10: Transient Performance of FLUTE for Exp. 2 (Fixed-QoD = 0.5)

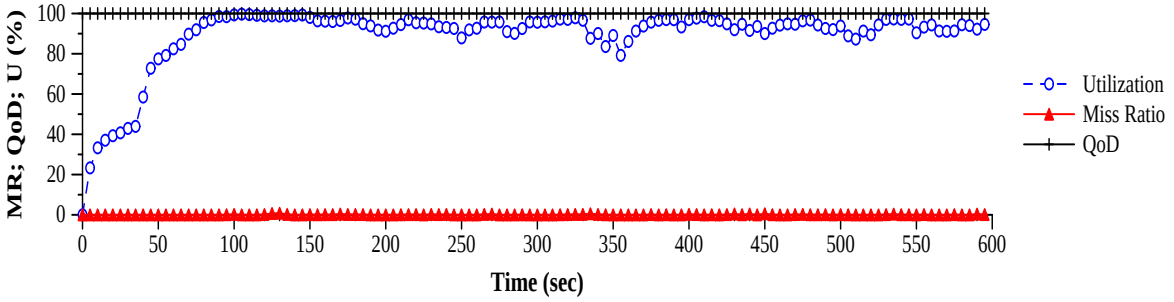

Figure 11: Transient Performance of FLUTE for Exp. 2 (Fixed-QoD = 1)

the experiments. For this reason, in this section we only compare the average performance among the tested approaches.

As shown in Figure 6, Basic-AC improved the miss ratio of Basic by applying admission control. However, Basic-AC also shows a high miss ratio reaching $37.65 \pm 4.45\%$ when $AppLoad$ = 200%. From this, we observe that Basic-AC is not adaptable enough against potential overloads. This is mainly because it is very hard, if at all possible, to predict potential rollbacks/restarts and the resulting waste of CPU cycles at the admission control stage. Therefore, for Basic-AC the admission control by itself may not be able to completely handle potential overloads. In contrast, FLUTE controls admission of newly arriving transactions and adapts the QoD according to the control signal computed in the feedback control loop considering the current miss ratio.

5.4.2 Average QoD

As shown in Figure 7, Basic and Basic-AC provide 100% QoD, since in these approaches the minimum update period is maintained for every data regardless of the current miss ratio. For FLUTE, the QoD decreases as $AppLoad$ increases achieving $76.92 \pm 1.27\%$ when $AppLoad = 200\%$ to improve the miss ratio as shown in Figure 7.

One may argue that our approach sacrificed the QoD to improve the miss ratio. However, our QoD degradation is bounded as required in the QoD specification: no more QoD degradation is allowed once the *Max-Degr* is reached. Also, every sensor data O_i is fresh in terms of the $fvi(O_i)$ even after a QoD degradation. More importantly, in Sections 5.5 and 5.6 we show that FLUTE can support the required miss ratio even if no QoD degradation is allowed, therefore, providing the perfect QoD = 100%!

5.4.3 Average Utilization

For Basic and Basic-AC, the utilization quickly increases to 100% leading to a large number of deadline misses as $AppLoad$ increases (Figure 8). In contrast, FLUTE shows the relatively stable utilization ranging between $63\% - 88\%$ for increasing $AppLoad$, while supporting the required miss ratio and freshness guarantees.

Due to the relatively poor performance (in terms of miss ratio) of the baseline approaches, in the remainder of this paper we mainly present the performance results of FLUTE for varying workloads and access patterns except some performance comparisons between FLUTE and the baseline approaches in Experiment Set 2.

5.5 Experiment Set 2: Effects of Increasing Fixed-QoD

For some real-time database applications, corporate users might require a high QoD and a low miss ratio at the same time, e.g., 100% QoD and 1% miss ratio. To consider this, in this section we evaluate the performance of FLUTE for increasing *Fixed-QoD* to observe whether or not the specified average/transient miss ratio can be supported. The utilization and throughput are also measured for performance comparisons among different *Fixed-QoD* values.

5.5.1 Average Miss Ratio, QoD, and Utilization

As shown in Figure 9, for increasing *Fixed-QoD* the average QoD increases up to 100% when *Fixed-QoD* = 1. For increasing *Fixed-QoD*, FLUTE provides a near zero average miss ratio at the expense of the slightly decreased CPU utilization, from 91% down to 89% as shown in Figure 9. This is mainly because of the relatively strict admission control to prevent a miss ratio overshoot for increasing *Fixed-QoD*: a larger portion of the requested CPU utilization reduction ($\Delta U < 0$), if any, should be handled by admission control as *Fixed-QoD* increases. However, the utilization decrease is small, since more sensor updates are executed for increasing *Fixed-QoD*.

5.5.2 Average Throughput

For increasing *Fixed-QoD*, a smaller number of user transactions might be processed in a timely manner due to increasing update workloads. When #Timely and #Submitted represent the number of user transactions committed within their deadlines and the number of user transactions submitted to the system (before admission control), we define the real-time database throughput:

$$\texttt{RTDB Throughput} = 100 \times \frac{\texttt{\#Timely}}{\texttt{\#Submitted}}(\%)$$

From this, the maximum possible user transaction throughput when $AppLoad = 200\%$ and *Fixed-QoD* = 1 can be theoretically derived as follows. The applied update and user transaction workloads are approximately 50% and 150%, respectively. The maximum possible throughput is approximately 33% = 50% / 150% = (Total CPU Capacity − Update Workload) / (Applied User Transaction Workload) assuming that there is no deadline miss when the CPU utilization is 100%. In the following, we compare the throughput of FLUTE and the baseline approaches to this ideal throughput of 33%.

As shown in Figure 9, in FLUTE the throughput decreases from $38.21 \pm 0.84\%$ to $29.62 \pm 1.38\%$ for increasing *Fixed-QoD*. This means approximately 38% and 29% of the submitted user transactions are actually admitted and committed within their deadlines when *Fixed-QoD* = 0.5 and 1, respectively ($AppLoad = 200\%$). As *Fixed-QoD* increases, a smaller number of user transactions can be admitted due to the increasing update workload. As a result, the user transaction throughput decreases. The throughput of FLUTE exceeds the ideal 33% when *Fixed-QoD* ≤ 0.7 at the expense of the reduced but bounded QoD as specified in *QoS-Spec*.

In contrast, Basic-AC showed the $22.04 \pm 0.87\%$ throughput, which is lower than that of FLUTE when *Fixed-QoD* = 1 ($29.62 \pm 1.38\%$), for the same experimental settings with Experiment Set 2 shown in Table 4 (except *Fixed-QoD*, since no QoD degradation is applied in Basic-AC). This is because Basic-AC admits too many transactions due to a relatively high execution time estimation errors ($EstErr = 1$ in Experiment Set 2). As a result, many admitted transactions may eventually miss their deadlines, and are aborted in our firm real-time database model. When *Fixed-QoD* = 0.5, our approach improves the user transaction throughput by more than 16% ($\approx 38.21\% - 22.04\%$) compared to Basic-AC. Further, for Basic the throughput was below 20% due to too many deadline misses.

From this, we can observe that it is a sensible approach to prevent potential overload by QoD management/admission control. In FLUTE, the number of timely transactions − transactions that commit within their deadlines − is actually increased compared to the baseline approaches, which model widely accepted real-time database frameworks.

5.5.3 Transient Performance

Figures 10 and 11 show the transient miss ratio, QoD, and utilization when *Fixed-QoD* is 0.5 and 1, respectively. We have also measured the transient performance for other *Fixed-QoD* values and observed similar performance results in terms of miss ratio and utilization, while the QoD increases for increasing *Fixed-QoD*. Due to space limitations, we only present the performance results for the two ends of the tested *Fixed-QoD* range.

As shown in Figures 10 and 11, for FLUTE the transient miss ratio is near zero without exceeding the 1% threshold, i.e., no miss ratio overshoot. In Figure 10, the QoD is decreasing to avoid potential miss ratio overshoot given $AppLoad = 200\%$. The QoD is not degraded in Figure 11, since *Fixed-QoD* = 1. At the same time, the miss ratio is below 1% (i.e., no miss ratio overshoot) at the expense of the relatively low throughput compared to the smaller *Fixed-QoD* values as shown in Figure 9. Observe that the transient performance is better than the required *QoS-Spec* in terms of overshoot and settling time. This shows the effectiveness of our QoD management and admission control schemes; the required CPU utilization adjustment computed in the feedback control loop is well enforced by degrading the QoD and/or controlling admissions, if necessary.

In Experiment Sets 1 and 2, for a wide range of workloads FLUTE can support the specified average/transient miss ratio, while providing the perfect QoD, if required. At the same time, FLUTE showed a higher throughput compared to the baseline approaches. Using our approach, a corporate user can select an appropriate QoD considering the application specific throughput and data semantics, free of a potential miss ratio overshoot or QoD violations.

5.6 Experiment Set 3: Effects of Varying Access Patterns

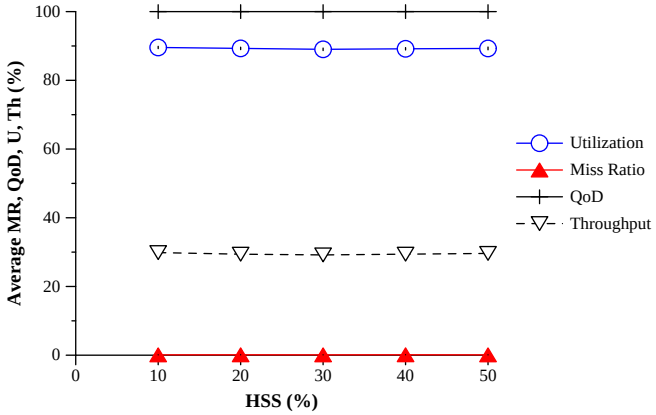

Figure 12: Average Performance of FLUTE for Exp. 3

As shown in Figure 12, for FLUTE the average performance is hardly affected for varying access patterns: the average miss ratio is near zero, utilization is approximately 90%, and throughput is approximately 30%, and QoD = 100% since *Fixed-QoD* = 1 in this set of experiments.

Concerning the transient performance, transient miss ratio overshoots are observed for $HSS = 10\%$ and $HSS = 20\%$ possibly due to the relatively high data contention compared to the other *HSS* values tested. For $HSS = 10\%$, the miss ratio overshoot of 1.7% − slightly violating the specified 1.2% overshoot (*QoS-Spec*) − was observed, however, it decayed within 5sec, i.e., one sampling period. For $HSS = 20\%$, the miss ratio overshoot was 1.26% and it also decayed within one sampling period. Considering the relatively high data contention for $HSS = 10\%, 20\%$ and the demanding experimental settings applied to Experiment Set 3 (Table 4), we can conclude that our approach has closely met *QoS-Spec* in terms of overshoot and settling time. For other hot spot values including the uniform access, no miss ratio overshoot was observed, similar to the results presented in Section 5.5. (We have also performed similar experiments for lower *Fixed-QoD* (< 1) values, and found no miss ratio overshoot while achieving the higher throughput/utilization compared to the results shown in Figure 12. We do not include the detailed results due to space limitations.)

Generally, concurrency improves the database performance unless there is a hot spot, in which many transactions access the same data object. Hot spots are usually eliminated by redesigning the corresponding database application, or they can be tolerated by using a special concurrency control techniques [6]. However, both of these approaches could be computationally expensive, and are not directly applicable to real-time database applications. For example, lots of transactions in stock trading may access time-varying sets of a few data items, causing data contention, depending on the market status. From Experiment Set 3, we claim that our approach has a considerable adaptability against potential data contention due to hot spots, while supporting the specified miss ratio and perfect QoD = 100%. This is mainly because FLUTE dynamically adapts the user/update workloads by applying the admission control/QoD adaptation according to the control signal computed in the feedback loop considering the current system behavior. As a result, the required real-time database QoS can be achieved even in the presence of a wide range of unpredictable workloads and data access patterns.

6 Related Work

Trade-off issues between timeliness and data freshness have been studied in [2, 3, 11]. Stanford Real-Time Information Processor (STRIP) addressed the problem of balancing between the possibly conflicting freshness and timing constraints in real-time databases [2]. To study the trade-off between freshness and timeliness, several scheduling algorithms were introduced to schedule updates and transactions, and their performance was compared. In their later work, a similar trade-off problem was studied for derived data [3]. In [11], trade-off issues between response time and data freshness are considered in the context of the web server. Dynamically generated data are materialized at the web server and continuously refreshed by the back-end database. Response time can be improved if more views are materialized, however, data freshness can be reduced, and vice versa. Given a certain number of views to materialize, they presented an adaptive view selection algorithm for materialization to improve the response time and data freshness. A database self-tuning project, called AutoAdmin [4], is going on at Microsoft Research to reduce the cost of database tuning for specific applications. Their work currently focuses on physical database design, i.e., identifying indexes and materialized views appropriate for an application specific workload to optimize the performance of database systems. None of the work presented in [2, 3, 4, 11] provided any performance guarantee in terms of either miss ratio or data freshness.

The QoD management scheme presented in this paper contrasts to our previous work presented in [8]. In [8], access/update frequencies are measured for each sensor data item. A data item is considered hot (important) if its access frequency is higher than the update frequency. Otherwise, it is considered cold. Under overload, some cold data can be updated on demand to reduce the update workload, if necessary. However, in this approach some cold data updated on demand can be arbitrarily old since the last on-demand update, even though the chances are small. In this paper, we presented novel notions of QoD and flexible validity intervals. Using these notions, our new QoD management scheme can bound the QoD degradation, and maintain the freshness even after a QoD degradation, if any. Also, in this approach there is no overhead to keep track of the access update ratio statistics and accordingly classify the data.

Various aspects of the real-time database performance other than data freshness can be traded off to improve the miss ratio. The correctness of answers to database queries can be traded off to enhance timeliness. A query processor, called APPROXIMATE [25], can provide approximate answers depending on the availability of data or time. An imprecise computation technique, called milestone approach [14], 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 [17], 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 to reduce the response time. Epsilon serializability [20] allows a query processing despite the concurrent updates. In their approach, the deviation of the answer to the query can be bounded. An adaptable security manager is proposed in [22], in which the database security can be temporarily traded off to enhance timeliness. Under overload, covert channels for illegal information flow between different security classes can be temporarily allowed in a controlled manner to improve the miss ratio. Note that none of the work presented in [17, 20, 22, 25] provide performance guarantees in terms of both miss ratio and data freshness.

Recently, feedback control has been applied to QoS management and real-time scheduling [13, 15, 16] due to its robustness against unpredictable operating environments [18]. However, to our best knowledge none of them considered performance guarantee issues regarding both timing and data freshness constraints in real-time databases.

7 Conclusions and Future Work

The demand for real-time database services is increasing. It is essential but very challenging to process transactions within their deadlines using fresh data reflecting the current real-world status. A key contribution of this paper is a new real-time data management framework, which can provide guarantees for both deadline miss ratio and sensor data freshness. Using FLUTE, users can explicitly specify the required miss ratio and QoD for specific real-time database applications. The specified miss ratio and QoD can be supported even in the presence of unpredictable workloads and data access patterns. We presented novel notions of QoD and flexible validity intervals. From this, a flexible QoD management scheme is derived to effectively manage the QoD considering the current system behavior, while maintaining freshness of sensor data even after a QoD degradation, if any. By applying the flexible QoD management scheme with feedback and admission control, FLUTE can support potentially conflicting miss ratio and freshness requirements at the same time, whereas the baseline approaches fail.

Our approach is important for many data-intensive real-time database applications. Due to the increasing complexity of real-time data needs, more research effort should be devoted to this area. As an initial work to provide guaranteed real-time database services, the importance of our work will increase as real-time applications become more sophisticated and data-intensive. Currently, we are investigating a differentiated real-time database service framework to provide preferred services to important service classes under overload. We are also considering other important research issues such as secure real-time transaction processing and timeliness/freshness issues in distributed real-time databases.

References

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

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

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

[4] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-tuning, RISC-style Database System. In *Very Large Databases*, 2002.

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

[6] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1994.

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

[8] K. D. Kang, S. H. Son, J. A. Stankovic, and T. F. Abdelzaher. A QoS-Sensitive Approach for Timeliness and Freshness Guarantees in Real-Time Databases. In *the 14th Euromicro Conference on Real-Time Systems*, June 2002.

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

[10] S. Kim, S. Son, and J. Stankovic. Performance Evaluation on a Real-Time Database. In *IEEE Real-Time Technology and Applications Symposium*, 2002. To appear.

[11] 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.

[12] C. G. Lee, C. S. Shih, and L. Sha. Service Class Based Online QoS Management in Surveillance Radar Systems. In *IEEE Real-Time Systems Symposium*, December 2001.

[13] B. Li and K. Nahrstedt. A Control-Based Middleware Framework for Quality of Service Adaptations. *IEEE Journal on Selected Areas in Communications*, 17(9), 1999.

[14] 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.

[15] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback Control Real-Time Scheduling: Framework, Modeling and Algorithms. *Journal of Real-Time Systems, Special Issue on Control-Theoretical Approaches to Real-Time Computing*, 2002. To appear.

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

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

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

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

[20] C. Pu and A. Leff. Replica Control in Distributed Systems: An Asynchronous Approach. In *ACM SIGMOD*, May 1991.

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

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

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

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

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

[26] 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.