Managing Deadline Miss Ratio and Sensor Data Freshness in Real-Time Databases*

Kyoung-Don Kang
Department of Computer Science
State University of New York at Binghamton
kang@cs.binghamton.edu

Sang H. Son and John A. Stankovic
Department of Computer Science
University of Virginia
{*son, stankovic*}*@cs.virginia.edu*

September 3, 2003

Abstract

The demand for real-time data services is increasing in many applications including e-commerce, agile manufacturing, and telecommunications network management. In these applications, it is desirable to execute transactions within their deadlines, i.e., before the real-world status changes, using fresh (temporally consistent) data. However, meeting these fundamental requirements is challenging due to dynamic workloads and data access patterns in these applications. Further, transaction timeliness and data freshness requirements may conflict. In this paper, we define average/transient deadline miss ratio and new data freshness metrics to let a database administrator specify the desired quality of real-time data services for a specific application. We also present a novel QoS management architecture for real-time databases to support the desired QoS even in the presence of unpredictable workloads and access patterns. To prevent overload and support the desired QoS, the presented architecture applies feedback control, admission control, and flexible freshness management schemes. A simulation study shows that our QoS-aware approach can achieve a near zero miss ratio and perfect freshness, meeting basic requirements for real-time transaction processing. In contrast, baseline approaches fail to support the desired miss ratio and/or freshness in the presence of unpredictable workloads and data access patterns.

Keywords: Real-time database, deadline miss ratio, sensor data freshness, QoS management

*This work was supported, in part, by NSF grants EIA-990895 and IIS-0208758.

1 Introduction

The demand for real-time data services is increasing in many important applications including e-commerce, online stock trading, agile manufacturing, sensor data fusion, traffic control, target tracking, and telecommunications network management. In these applications, transactions should be processed within their deadlines, i.e., before the market, manufacturing, or network status changes, using fresh (temporally consistent) sensor data[1] that reflect the current real-world status. Existing (non-real-time) databases are poor at supporting timing constraints and temporal consistency of data. Therefore, they do not perform well in these applications.

Real-time databases need to execute transactions within their deadlines using fresh data, but meeting these fundamental requirements is challenging. Generally, transaction execution time and data access pattern are not known a priori, but could vary dynamically. For example, transactions in stock trading may read varying sets of stock prices, and perform different arithmetic/logical operations to maximize the profit considering the current market status. Transactions can be rolled back and restarted due to data/resource conflicts. Further, transaction timeliness and data freshness can often pose conflicting requirements. By preferring user requests to sensor updates, the deadline miss ratio is improved; however, the data freshness might be reduced. Alternatively, the freshness increases if updates receive a higher priority [3].

To address this problem, we present a novel real-time main memory database architecture called **QMF** (a <u>Q</u>oS management architecture for deadline <u>M</u>iss ratio and data <u>F</u>reshness). QMF provides several QoS parameters to let a database administrator (DBA) specify the desired miss ratio and data freshness for a specific application. QMF applies a feedback-based miss ratio control scheme since feedback control is very effective to support the desired performance when the system model includes uncertainties [13, 16, 23]. At each sampling instant, the feedback-based miss ratio controller measures the miss ratio and computes the miss ratio error, i.e., the difference between the desired miss ratio and the measured one. Based on the miss ratio error, the controller computes the control signal, i.e., the required workload adjustment to react to the error. QMF can achieve the desired miss ratio via series of reactions to reduce the error even if an individual reaction is not completely precise. When overloaded, the freshness manager, which manages data freshness in QMF, updates relatively less important sensor data on demand or increase their update periods according to the miss ratio control signal.[2] QMF also

[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.

[2] The freshness manager is an actuator from the control theory perspective. We do not consider a separate controller for freshness management, since miss ratio and data freshness requirements can conflict leading to an unstable feedback control system that may oscillate between many deadline misses and freshness violations.

applies admission control to incoming transactions under overload conditions. By adapting update workloads before applying admission control, QMF can admit (and process) more transactions.

In our earlier work [8], we presented an *adaptive update policy* to balance potentially conflicting miss ratio and freshness requirements. Initially, all data are updated immediately when their new sensor readings arrive. Under overload, some sensor data can be updated on demand to improve the miss ratio as long as the target freshness, perceived by the transactions that commit within their deadlines, is supported. The adaptive update policy can effectively balance miss ratio and data freshness requirements [8]. It contrasts to the existing database update policy commonly accepted in the real-time database research such as [3, 9, 17], which is fixed and not adaptable regardless of the current system status. However, the adaptive update policy has a potential disadvantage. Consider a case in which real-time transactions with tight deadlines need to access a sensor data object that is updated on demand. The transactions may have to miss their deadlines waiting for the on-demand update. Or, to meet their deadlines they may have to use a stale version of the data, outdated since the last on-demand update, even though the chances are small.[3] Neither is desirable.

To prevent potential deadline misses or stale data accesses due to the delay for on-demand updates, we present an alternative approach in which all data are updated immediately. In this approach, we present novel notions of *QoD (Quality of Data)* and *flexible validity intervals* to manage the freshness.[4] When overloaded, update periods of some sensor data can be relaxed within the specified range of QoD to reduce the update workload, if necessary. However, sensor data are always maintained fresh in terms of flexible validity intervals. Therefore, the age of sensor data is always bounded. (A detailed discussion is given in Section 4.)

Real-time databases should determine the frequency of sensor data updates considering the rate at which the real-world status changes (or may change) [17]. For this reason, we measured the average inter-trade time of popular stock items using the real-time NYSE trade information, which is streamed into an online trading laboratory at the University of Virginia. From these studies, we derived a range of sensor update periods used for our simulation study (discussed in detail in Section 6). Unlike our work presented in this paper, existing real-time database work such as [1, 3, 9] do not consider actual data freshness semantics, mainly determined by the update frequency, for performance evaluation.

Based on performance evaluation results, we show that QMF can support stringent QoS requirements for a

[3] In [8], we show that our approach can support the 98% perceived freshness. A detailed description of perceived freshness is given in Section 3.

[4] In [8], we introduced the notion of QoD. We adapt the definition of QoD for the new flexible freshness management scheme presented in this paper.

large range of workloads and access patterns. QMF achieves a near zero miss ratio and perfect freshness even given dynamic workloads and data access patterns.[5] In contrast, several baseline approaches, including *best existing* algorithms for timeliness and freshness trade-off in real-time databases [3], fail to support the specified miss ratio and/or data freshness. Further, our approach achieves a higher throughput than the baselines. This is because QMF can avoid overload conditions, which can cause many deadline misses, by carefully adapting the system behavior in the feedback loop as discussed before.

The rest of the paper is organized as follows. Section 2 describes our real-time database model. Flexible sensor update schemes are presented in Sections 3 and 4. In Section 5, our QoS management architecture including the feedback control scheme is described. The performance evaluation results are presented in Section 6. Related work is discussed in Section 7. Finally, Section 8 concludes the paper and discusses future work.

2 Real-Time Database Model

In this section, we describe the database model, transaction types, deadline semantics, and average/transient miss ratio considered in this paper. We consider a main memory database model, in which the CPU is considered the main system resource. Main memory databases have been increasingly applied to 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 [3, 19].

We classify transactions as either sensor updates or user transactions. Periodic sensor updates are write-only transactions that capture the continuously changing real-world state. User transactions can read sensor data and read/write non-sensor data such as PIN numbers that do not have temporal consistency constraints. User transactions can also execute arithmetic/logical operations based on the current real-world state 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 monitored by periodic sensor updates.

We apply firm deadline semantics, in which transactions add value to the system only if they finish within their deadlines. Hence, a transaction is aborted upon its deadline miss. Firm deadline semantics are common in many real-time database applications. A late commit of a real-time transaction may incur the loss of profit or product

[5] Since QMF is evaluated for a large range of workloads and access patterns, while handling a large number of transactions for statistical confidence, we consider the results are valid although our real-time database model is approximate based on aggregate system parameters such as the miss ratio. A finer grain modeling considering individual transaction characteristics, without requiring a priori knowledge about workloads, is reserved for future work.

quality, resulting in wasted system resources, due to possible changes in the market or manufacturing status.

The deadline miss ratio is one of the most important performance metrics in real-time applications. For admitted transactions, the deadline miss ratio is: $MR = 100 \times \frac{\texttt{\#Tardy}}{\texttt{\#Tardy + \#Timely}}$ $(\%)$ where #Tardy and #Timely represent the number of transactions that have missed and met their deadlines, respectively. The DBA can specify a tolerable miss ratio threshold, e.g., 1%, for a specific real-time database application. As discussed before, database workloads and data access patterns might vary dynamically. Therefore, 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 DBA.

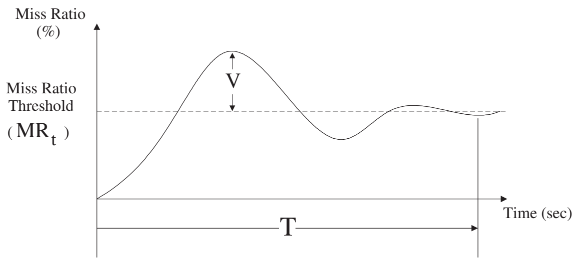

Figure 1: Definition of Overshoot (V) and Settling Time (T) in Real-Time Databases

Long-term performance metrics, e.g., average miss ratio, are not sufficient to specify the desired performance of dynamic systems whose performance could change significantly in a relatively short time interval [13]. For this reason, transient performance metrics such as overshoot and settling time shown in Figure 1 are adopted from control theory to specify the desired target performance of real-time systems:

• *Overshoot* 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 (MR_t) in the transient state.

• *Settling time* is the time for a transient miss ratio overshoot to decay. After T, the real-time database should enter the steady state, in which the miss ratio is within the range [0, $MR_t + 0.01 \times MR_t$].

Our approach also provides data freshness metrics to specify the desired real-time database QoS. For the clarity of presentation, we defer the related discussion to Sections 3 and 4 in which we present two alternative approaches for flexible freshness management. In the remainder of this paper, we follow a convention that classifies QMF as QMF-1 or QMF-2 according to the selected freshness management scheme.

3 Flexible Freshness Management: QMF-1

In this section, we discuss data freshness metrics and describe a cost-benefit model for sensor data updates. Using this model, we present an adaptive update policy called QMF-1.

3.1 Freshness Metrics

In real-time databases, validity intervals are used to maintain the temporal consistency between the real-world state and sensor data in the database [9, 17]. A sensor data object O_i is considered fresh, i.e., temporally consistent, if $(current\ time - timestamp(O_i)\ \leq\ avi(O_i))$ where $avi(O_i)$ is the absolute validity interval of O_i.[6] For O_i, we set the update period $P_i = 0.5 \cdot avi(O_i)$ to support the sensor data freshness, similar to [9, 17]. To manage data freshness in an adaptive manner, we consider two key freshness metrics as follows:

• *Database Freshness*, also called QoD, is the ratio of fresh (sensor) data to the entire data in a real-time database.

• $Perceived freshness$ (PF) is defined for the data accessed by timely transactions. Let $N_{accessed}$ represent the number of data accessed by timely transactions. Let N_{fresh} stand for the number of fresh data accessed by timely transactions.

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

We support the desired data freshness in terms of perceived freshness since we apply firm deadline semantics. When overloaded, the QoD (database freshness) could be traded off to improve the miss ratio as long as the target perceived freshness is not violated. This approach could be effective considering potentially high update workloads, e.g., stock price updates during the peak trade time [3], which may cause many deadline misses.

3.2 Cost-Benefit Model for Updates and Adaptive Update Policy

To balance the update and transaction workload efficiently, we introduce a cost-benefit model for sensor data updates as follows. The cost is defined as the update frequency of a sensor data object. Intuitively, the more frequent is the update, the higher is the cost. We assume that the frequency of periodic updates is known to the database system.[7] To consider the benefit, access frequency is measured for each data object. Updating a

[6]Real-time databases may include derived data such as stock composite indexes. In this paper, we do not consider the derived data management. We reserve this for future work.

[7]We can also apply our cost-benefit model to aperiodic updates by monitoring the update frequency, similar to the access frequency monitoring (discussed in the remainder of this subsection). However, in this paper we only consider periodic updates that are commonly

frequently accessed data can produce a relatively high benefit. To quantify the cost-benefit relationship, we define Access Update Ratio (AUR) for a sensor data object O_i, which represents the importance of being fresh:

$$AUR[i] = \frac{AF[i]}{UF[i]} \tag{1}$$

where $AF[i]$ and $UF[i]$ are defined in Table 1, which summarizes the notations used in the remainder of this section.

Table 1: Notations for QMF-1

Notation	Description
$AUR[i]$	Access update ratio of O_i
$AF[i]$	Access frequency of O_i
$UF[i]$	Update frequency of O_i
$SAF[i]$	Smoothed access frequency of O_i
D	Set of the entire sensor data in a real-time database
D_{hot} (D_{cold})	Set of hot (cold) sensor data items
D_{imm} (D_{od})	Set of sensor data updated immediately (on demand)
D_{cold_imm}	$D_{cold} \cap D_{imm}$
ΔW	Required workload adjustment computed in the feedback control loop
ΔW_{new}	Required workload adjustment after (a) QoD degradation(s), if any
PF_t	Target perceived freshness
$Min\_AUR(X)$	Function returning the data item with the smallest AUR in a given set X
δW_i	CPU utilization saved by degrading the QoD of O_i
$AET(O_i)$	Average execution time needed to update O_i

Unfortunately, the access frequency may have a large deviation from one sampling period to another. To smooth the potentially large deviation, we take a moving average of the access frequency (AF) for O_i in the k^{th} sampling period:

$$SAF_k[i] = a \cdot SAF_{k-1}[i] + (1-a) \cdot AF_k[i] \tag{2}$$

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

Since the $UF[i]$ in a sampling period is known, we can compute AUR for O_i:

$$AUR[i] = \frac{SAF[i]}{UF[i]} \tag{3}$$

adopted in real-time databases to support the temporal consistency of sensor data [9, 17].

If AUR[i] ≥ 1, the benefit of updating O_i is worth the cost, since O_i is accessed at least as frequently as it is updated. We call O_i hot, i.e., $O_i \in D_{hot}$, if its AUR ≥ 1. Otherwise, we call O_i cold, i.e., $O_i \in D_{cold}$. Note that $D = D_{hot} \cup D_{cold}$ and $D_{hot} \cap D_{cold} = \emptyset$. The notion of AUR does not depend on a specific access pattern or popularity model. It can be derived simply from the update and access frequency for each data object. Therefore, it greatly simplifies our cost-benefit model, and makes the model robust against the potential unpredictability in data access patterns.

From the cost-benefit model, we observe that it is reasonable to update hot data immediately. If a hot data item is out-of-date when accessed, a multitude of transactions may miss their deadlines waiting for the update. Further, updating hot data on demand may not decrease the update workload because $SAF[i] \geq UF[i]$ for $\forall O_i \in D_{hot}$ as discussed before. Therefore, $D_{hot} \cap D_{od} = \emptyset$. Alternatively, it may not be necessary to immediately update cold data when overloaded. Only a few transactions may miss their deadlines waiting for the update. Under overload, we can save the CPU utilization by updating some cold data on demand.

Initially, every data is updated immediately, i.e., $D_{imm} = D$ and $D_{od} = \emptyset$. At each sampling instant, the feedback controllers compute the required workload adjustment, called ΔW, to support the desired miss ratio. (A detailed discussion of feedback control is given in Section 5.) When overloaded, ΔW becomes negative requiring the workload reduction. Accordingly, the freshness manager reduces the update workload, if the current $PF \geq PF_t$, as described in Figure 2.

$\Delta W_{new} = \Delta W$
$D_{cold\_imm} = D_{cold} \cap D_{imm}$
while ($\Delta W_{new} < 0$ **and** $PF \geq PF_t$ **and** $D_{cold\_imm} \neq \emptyset$)
{
    $O_i = Min\_AUR(D_{cold\_imm})$
    $D_{cold\_imm} = D_{cold\_imm} - \{O_i\}$
    $D_{imm} = D_{imm} - \{O_i\}$
    $D_{od} = D_{od} \cup \{O_i\}$
    $\Delta W_{new} = \Delta W + \delta W_i$ where $\delta W_i = AET(O_i) \cdot [UF[i] - SAF[i]]$
    $\Delta W = \Delta W_{new}$
}

Figure 2: QoD Degradation in QMF-1 under Overload Conditions

As shown in Figure 2, the CPU utilization saved due to a QoD degradation for a single data item O_i is approximately:

$$\delta W_i = AET(O_i) \cdot [UF[i] - SAF[i]] \tag{4}$$

where $AET(O_i)$ is the average execution time to update O_i. A DBA can measure $AET(O_i)$, preferably, offline to minimize the overhead. Note that we use the average execution time in Eq. 4, since the update time of sensor data, e.g., radar images, can be time-varying. If we consider the worst case update execution time, the miss ratio can be improved at the cost of potential underutilization. Or, a large miss ratio overshoot may occur when the update execution time is under-estimated. Generally, determining optimal values of real-time database model parameters such as AET_i or δW_i can be very hard, if ever possible, without *a priori* knowledge of workloads. As discussed before, this is the key motivation of applying feedback control and dynamic workload adaptation techniques to database QoS management in QMF.

We need to switch the update policy back to the immediate one for some cold data when the target perceived freshness is violated. We call this *QoD upgrade*. To upgrade the QoD for O_i, the extra CPU utilization of δW_i in Eq. 4 is required to switch the update policy of O_i back to the immediate policy. The QoD can be upgraded as long as δW_i is available and a certain upgrade bound is not reached yet to avoid a miss ratio overshoot in the next sampling period as a result of excessive QoD upgrades. Due to space limitations, we refer readers to [7, 8] for more details about QoD upgrade and degradation.

4 Flexible Freshness Management: QMF-2

In this section, we consider an alternative approach for freshness management called QMF-2. Novel notions of QoD and flexible validity intervals are discussed. A detailed description of QoD parameters and flexible freshness management is also given.

4.1 Quality of Data and Flexible Validity Intervals

In QMF-2, all sensor data are updated immediately to avoid the possible deadline misses or stale data accesses due to on-demand updates as discussed before. When overloaded, the update periods of relatively less critical sensor data, e.g., data with low AUR values in stock trading or slow moving friendly helicopters in target tracking, can be increased to improve the miss ratio.[8] We assume that the relative importance of data is available to a DBA working for a specific application, e.g., a financial trading or target tracking. This is a reasonable assumption: The AUR of stock prices, for example, can be available to (or measured by) the DBA at least to distinguish

[8]Task period adjustment is previously studied to improve the miss ratio in real-time (non-database) systems such as [11]. However, database issues such as data freshness are not considered in these work.

between hot and cold data. Also, the relative importance of aircraft must be available in target tracking.[9] Given the assumption, we define the current QoD when there are N sensor data objects in a real-time database:

$$QoD = \left[\frac{100}{N}\right]\left[\sum_{i=1}^{N}\frac{P_{i_{min}}}{P_{i_{new}}}\right](\%) \tag{5}$$

where $P_{i_{min}}$ and $P_{i_{new}}$ are defined in Table 2, which briefly describes the notations used in this section. 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 in real-time databases.

Table 2: Notations for QMF-2

Notation	Description
$P_{i_{min}}$	Minimum update period of O_i before any QoD degradation
$P_{i_{new}}$	New update period after a QoD degradation for O_i
P_i	Current update period of O_i (Initially, $P_i = P_{i_{new}} = P_{i_{min}}$)
$fvi_{new}(O_i)$	New flexible validity interval after a QoD degradation for O_i
D_{degr}	Set of sensor data whose QoD can be degraded
$Max\text{-}Degr$	Degree of maximum allowed QoD degradation
$Step\text{-}Size$	Degree of a unit QoD degradation
$\delta W_i{}'$	CPU utilization saved by degrading the QoD of O_i

To maintain the freshness of a sensor data item after a possible QoD degradation, we define a notion of *flexible validity intervals* (fvi). Initially, $fvi = avi$ for all data. Under overload, the update period P_i for a less critical data object O_i can be relaxed. After the QoD degradation for O_i, we set $fvi_{new}(O_i) = 2 \cdot P_{i_{new}}$ to maintain the freshness of O_i by updating it at every $P_{i_{new}}$. Accordingly, O_i is considered fresh if $(current\ time - timestamp(O_i) \leq fvi_{new}(O_i))$. Since all sensor data are maintained fresh in terms of (flexible) validity intervals, QMF-2 supports the 100% perceived freshness; that is, the age of each sensor data is always bounded by fvi. Therefore, the QoD defined in Eq. 5 is the only freshness metric in QMF-2.

4.2 QoD Parameters and QoD Management

A DBA, who is aware of application specific real-time data semantics, can specify the desired QoD using the following QoD parameters.

[9] Ideally, precise rankings of data importance can optimize the QoD by degrading the QoD of the least critical data item first, if necessary. However, our approach can still manage the QoD in a flexible manner even given an approximate information of data importance (at the cost of possibly suboptimal QoD).

• D_{degr}: A DBA can specify a certain set of sensor data D_{degr}, e.g., the set of data with AUR < 1, whose QoD can be degraded, if necessary, to support the target miss ratio.

• *Max-Degr*: When $D_{degr} \neq \emptyset$, a DBA can specify *Max-Degr* to avoid an indefinite QoD degradation. For a sensor data object $O_i \in D_{degr}$, $P_{i_{new}} \leq \textit{Max-Degr} \cdot P_{i_{min}}$ after a QoD degradation. D_{degr} and *Max-Degr* can determine the worst possible QoD. For the clarity of presentation, let *Fixed-QoD* = $1 - |D_{degr}| \,/\, |D|$ represent the fraction of D (the set of the entire sensor data in a real-time database) whose QoD can *not* be degraded. When *Fixed-QoD* = 0.7 and *Max-Degr* = 4, for example, the lowest possible QoD is 77.5% = $100 \cdot [\textit{Fixed-QoD} + (1 - \textit{Fixed-QoD}) \,/ \textit{Max-Degr}]\,\% = 100 \cdot (0.7 + 0.3/4)\%$ when the current update period $P_{i_{new}} = 4 \cdot P_{i_{min}}$ for every $O_i \in D_{degr}$.

• *Step-Size*: A DBA can also specify *Step-Size* for graceful QoD degradations, if any. For example, when *Step-Size* = 10%, $P_{i_{new}} = 1.1 \cdot P_i$ after a QoD degradation for $O_i (\in D_{degr})$ to avoid a sudden QoD degradation.

Our QoD parameters can effectively reflect QoD requirements in real-time database applications. For example, financial trading tools such as Moneyline Telerate Plus [14] usually allow users to specify acceptable periods ranging between 1 minute and 60 minutes to monitor stock prices. Further, our approach can adjust the QoD within the specified QoD range, if necessary, to improve the miss ratio.

```
ΔW_new = ΔW
while (ΔW_new < 0 and D_degr ≠ Ø)
{
    /* Increase P_i for O_i with the least importance. */
    P_i_new = (1 + Step-Size) · P_i
    if (P_i_new ≤ P_i_min · Max-Degr)
        ΔW_new = ΔW + δW_i' where δW_i' = AET(O_i)/P_i − AET(O_i)/P_i_new
        ΔW = ΔW_new
        P_i = P_i_new
        fvi_new(O_i) = 2 · P_i
    else
        D_degr = D_degr − {O_i}
        i++
}
```

Figure 3: QoD Degradation in QMF-2 under Overload Conditions

When overloaded, i.e., $\Delta W < 0$, the QoD can be degraded as described in Figure 3. The update period of the least critical data object O_i ($\in D_{degr}$) is increased by the *Step-Size*, if possible. The QoD degradation continues until the required workload adjustment is achieved, i.e., $\Delta W_{new} \geq 0$, or $D_{degr} = \emptyset$. (When O_i's QoD can not be degraded any further, O_i is removed from D_{degr} as shown in Figure 3.)

Note that the computation of $\delta W_i\,'$, the CPU utilization saved by degrading the QoD for O_i as shown in Figure 3, can include smaller errors compared to the computation of δW_i in QMF-1 (Eq. 4) using the smoothed access frequency ($SAF[i]$), which may vary from time to time.

5 Architecture for QoS Management of Real-Time Data Services

In this section, a QoS specification is given to illustrate the applicability of our approach. The QMF architecture is described. An overall behavior of QMF is described, followed by a detailed discussion about the system components consisting the architecture. The real-time database/feedback control models and controller tuning process needed to support the desired QoS are described in detail.

5.1 QoS Specification

We give a stringent QoS specification called *QoS-Spec* by mimicking a DBA who can specify the desired miss ratio and freshness as follows.

• **Miss Ratio**: The average miss ratio is desired to be below 1% to minimize the potential loss of profit or product quality even given dynamic workloads and access patterns. An overshoot needs to be below 30% to support the consistent real-time performance. Therefore, the transient miss ratio should not exceed $1.3\% = 1 \cdot (1+0.3)\%$. The settling time should be shorter than 40sec, e.g., a reasonable think time between trades.

• **Freshness Requirements**: For QMF-1, we set the target perceived freshness $PF_t = 98\%$. For QMF-2, we set *Max-Degr* = 4 and *Step-Size*= 10%. We do not fix D_{degr} (and *Fixed-QoD*), but decrease the cardinality of D_{degr} (i.e., increase *Fixed-QoD*) to make the *QoS-Spec* more stringent in Section 6.

5.2 QMF Architecture and Interactions among Key System Components

Figure 4 shows our QoS management architecture. The transaction handler supports the concurrency control, freshness check upon each sensor data access, and scheduling for real-time transaction processing. The monitor measures the current system status such as miss ratio, CPU utilization, and PF/QoD at each sampling instant. Based on the current system state, the miss ratio and utilization controller compute the required workload adjustment ΔW to support the specified miss ratio without severely underutilizing the CPU. The QoD manager and admission controller adapt the workload as required by the feedback controllers, if necessary, to support the

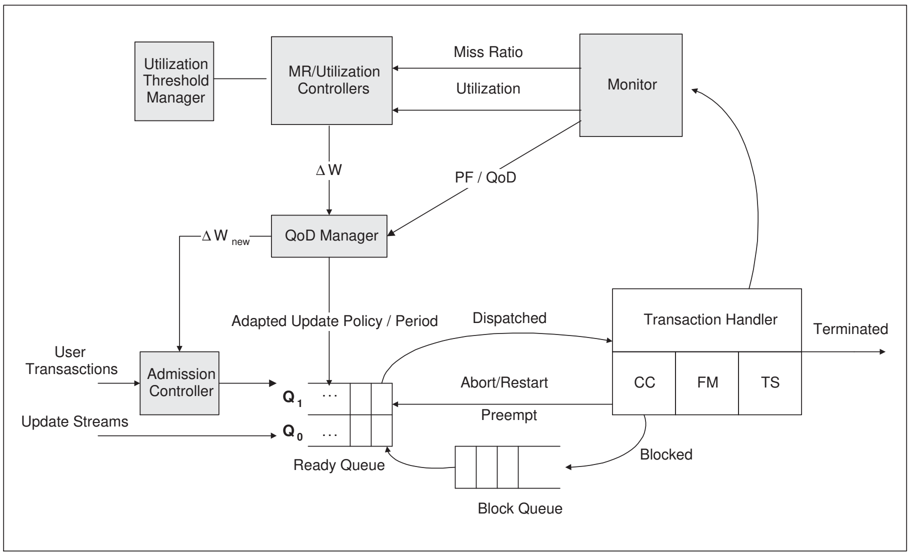

Figure 4: Real-Time Database Architecture for QoS Management

target miss ratio while meeting the freshness requirements. The utilization threshold manager applies a computationally lightweight method to closely approximate the potentially time-varying utilization bound for real-time transaction scheduling. Due to space limitations, we refer readers to [7] for more details.

The overall behavior of QMF is described in Figure 5. If $\Delta W \geq 0$, i.e., the current miss ratio is below the threshold (1% in *QoS-Spec*), and the freshness requirement is also met, more transactions are admitted to avoid potential underutilization. When the system is overloaded, i.e., $\Delta W < 0$, the workload should be reduced. To do this, the QoD can be reduced if $D_{cold\_imm} \neq \emptyset$ in QMF-1, or $D_{degr} \neq \emptyset$ in QMF-2.

When $\Delta W < 0$ and the QoD can not be degraded further to support the target freshness, we apply admission control to prevent overload. Admission control is known to be effective to prevent database thrashing due to severe data/resource contentions [21]. It can also improve the miss ratio of real-time transactions significantly, especially when overloaded [8, 9]. In QMF, 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.

5.3 Transaction Handler

The transaction handler consists of a concurrency controller (CC), a freshness manager (FM), and a transaction scheduler (TS). The **CC** uses two phase locking high priority (2PL-HP) [1] in which a low priority transaction is aborted and restarted upon a conflict to avoid priority inversions. 2PL-HP is also deadlock free when transaction

1. At each sampling period, measure the miss ratio (MR), freshness, and CPU utilization. In QMF-1, also collect access statistics and compute AUR.
2. The miss ratio and utilization controllers compute their control signals called ΔW_{MR} and ΔW_U, respectively. Set $\Delta W = Min(\Delta W_{MR}, \Delta W_U)$ for a smooth transition from a system state to another. Based on ΔW, perform one of the following alternatives in either QMF-1 or QMF-2.

QMF-1:

- If $\Delta W \geq 0$ and $PF \geq PF_t$, admit more transactions to prevent potential underutilization.
- If $\Delta W < 0$ and $PF \geq PF_t$, degrade the QoD and adjust ΔW_{new} as described in Figure 2. Apply admission control if $\Delta W_{new} < 0$ after all possible QoD degradations, if any.
- If $\Delta W > 0$ and $PF < PF_t$, upgrade the QoD as described in Section 3.
- If $\Delta W < 0$ and $PF < PF_t$, drop incoming transactions until ΔW becomes positive at a later sampling period due to the termination of some transactions currently in the system.

QMF-2:

- If $\Delta W \geq 0$, admit more transactions to avoid potential underutilization.
- If $\Delta W < 0$, degrade the QoD and adjust ΔW_{new} as described in Figure 3. Apply admission control if $\Delta W_{new} < 0$ after all possible QoD degradations, if any.

Figure 5: QoS Management for Miss Ratio and Freshness Support

priorities are unique. If several transactions have the same priority, we let the transaction that has first been admitted to the system get the required lock(s).[10]

The **FM** checks the freshness before accessing a data item using the corresponding avi (or fvi if there has been a QoD degradation in QMF-2). It blocks a user transaction if an accessed data item is currently stale. The blocked transaction(s) will be transferred from the block queue to the ready queue as soon as the update of the stale data item commits.

The **TS** schedules transactions in one of two ready queues , i.e., Q_0 and Q_1 as shown in Figure 4. A transaction in Q_1 can be executed if there is no ready transaction in Q_0, and can be preempted when a new transaction arrives at Q_0. In each queue, transactions are scheduled in an EDF (Earliest Deadline First) manner. To support the target freshness, all immediate updates are scheduled in Q_0. User transactions and on-demand updates, if any, are scheduled in Q_1.

[10] In this paper, we do not consider optimistic concurrency control policies. Several early studies about real-time databases such as [4, 6] showed contradicting results concerning the performance of optimistic and pessimistic concurrency control schemes. Later, Lee et al. [10] found that the performance of real-time concurrency control mechanisms vary depending on several factors such as the resource availability and deadline semantics. Especially, they showed that lock based approaches achieve a better performance in a resource constrained environment.

5.4 Feedback Control

In this section, we model real-time main memory databases in terms of the CPU utilization and miss ratio. We need both models because the miss ratio saturates at 0% when the system is underutilized. In contrast, the utilization saturates at 100% when the system is overloaded. Based on these models, we apply control theoretic approaches to support the desired miss ratio, while avoiding the CPU underutilization, similar to [13]. The utilization and miss ratio models, closed loop models applying feedback control, and controller tuning using the Root Locus method [16] to support the desired average/transient miss ratio (*QoS-Spec*) are discussed as follows.

5.4.1 Utilization Model

To apply control theoretic approaches, one should model the controlled system such as real-time databases by transfer functions that describe the relation between the control input, e.g., workload adjustment, and system output, e.g., resulting utilization or miss ratio, via differential or difference equations [16]. In this subsection, we model the utilization of real-time databases using difference equations in the discrete time domain where most computational systems operate. The notations used to describe the utilization model are summarized in Table 3.

Table 3: Notations for the Utilization Model

Notation	Description
$U(k)$	Utilization measured at the k^{th} sampling period
G_u	Utilization gain
G_U	$Max\{G_u\}$
$\Delta W_U(k-1)$	Workload adjustment performed at the $(k-1)^{th}$ sampling period to avoid underutilization
$T_U(z)$	Transfer function of the utilization model

When the workload is lower than the utilization threshold, all transactions can finish within their deadlines. Thus, the miss ratio is zero. The utilization at the k^{th} sampling period is:

$$U(k) = U(k-1) + G_u \cdot \Delta W_U(k-1) \tag{6}$$

where $\Delta W_U(k-1)$ is the workload adjustment at the previous sampling period due to admitting more incoming transactions (or upgrading the QoD, if necessary, to support the target perceived freshness in QMF-1) to avoid severe underutilization. Hence, Eq. 6 is the utilization model showing the relation between the input $\Delta W_U(k-1)$ and output $U(k)$ in the discrete time domain.

Since the workload adjustment may not be precise due to potential errors in execution time estimates, used for admission control, the actual workload adjustment is $G_u \cdot \Delta W_U(k-1)$ in Eq. 6 where G_u is called the utilization gain in this paper. In fact, the relation between $U(k)$ and $\Delta W_U(k-1)$ may not be linear since G_u can be time-varying. To linearize the utilization model, we replace G_u with $G_U = Max\{G_u\} = 2$ by assuming that the actual execution time is twice the estimated one in the worst case.

Next, we take the z-transform of Eq. 6 and derive the transfer function for the utilization model that describes the relation between the input, i.e., $\Delta W_U(k-1)$, and output, i.e., $U(k)$. The z-transform is a commonly used technique in digital control developed in the discrete time domain. It transforms difference equations into equivalent algebraic equations that are easier to manipulate. Although a complete review of z-transform is beyond the scope of this paper, we present one key property useful to manipulate the difference equations for our utilization and miss ratio models. Consider a sampled variable x, whose samples are represented by $x[1], x[2], x[3], \ldots$, etc.; that is, $x[k]$ denotes the k^{th} sample. Let the z-transform of $x[k]$ be $X(z)$, then the z-transform of $x[k-n]$ is $z^{-n}X(z)$. By applying this property to Eq. 6, we get the utilization model after some algebraic manipulation:

$$U(z) = \frac{G_U}{z-1} \Delta W_U(z) \tag{7}$$

Therefore, the transfer function of the utilization model is:

$$T_U(z) = \frac{G_U}{z-1} \tag{8}$$

5.4.2 Miss Ratio Model

When the workload is higher than the utilization threshold, i.e., the real-time database is overloaded, transactions begin to miss their deadlines, while the utilization saturates at 100% as discussed before. The utilization model can not consider the situation in which the workload increases causing deadline misses. Therefore, we also need to model real-time databases in terms of miss ratio. The notations used to describe the miss ratio model are summarized in Table 4.

When overloaded, the miss ratio at the k^{th} sampling period is:

$$M(k) = M(k-1) + G_m \cdot \Delta W_{MR}(k-1) \tag{9}$$

Table 4: Notations for the Miss Ratio Model

Notation	Description
$M(k)$	Miss ratio measured at the k^{th} sampling period
G_m	Miss ratio gain
G_M	$Max\{G_m\}$
$\Delta W_{MR}(k-1)$	Workload adjustment performed at the $(k-1)^{th}$ sampling period to support the target miss ratio
$T_M(z)$	Transfer function of the miss ratio model

where $\Delta W_{MR}(k-1)$ denotes the workload adjustment, via admission control and/or QoD degradation, performed at the previous sampling period to reduce the miss ratio. Thus, Eq. 9 shows the relation between the input $\Delta W_{MR}(k-1)$ and output $M(k)$ of the miss ratio model.

The miss ratio $M(k)$ may not increase linearly as the workload increases. Therefore, the miss ratio gain $G_m = \frac{dM(k)}{dW(k)}$, i.e., $\frac{Miss\ Ratio\ Increase}{Unit\ Load\ Increase}$, may vary from time to time. To linearize the miss ratio model, we replace G_m with $G_M = Max\{G_m\}$, which can be experimentally derived. More specifically, we have measured the average miss ratio for loads increasing from 60% to 200% by 10% to identify the (simulated) real-time database system in terms of miss ratio. (A detailed description of workloads is given in Section 6.) To model the worst case, all incoming transactions are admitted to the system and the QoD is not degraded regardless of the current system status. From these experiments, we have derived $G_M = 1.523$ when the load increases from 110% to 120%. To derive G_M, a DBA can apply application specific workload traces ranging between nominal and peak loads. QMF can support the desired miss ratio as long as the workload does not exceed the peak load used for the system identification. In summary, a DBA needs to (1) specify the target miss ratio and freshness, (2) measure the average time for sensor data updates (Section 3), and (3) identify the system in terms of G_M.

By taking the z-transform of Eq. 9 and doing some algebraic manipulation, we get:

$$M(z) = \frac{G_M}{z-1}\Delta W_{MR}(z) \tag{10}$$

Consequently, the transfer function of the miss ratio model is:

$$T_M(z) = \frac{G_M}{z-1} \tag{11}$$

5.4.3 Closed Loop Models and Controller Tuning

Figure 6 shows the closed loop (i.e., feedback-based) miss ratio and utilization controllers. Especially, we apply PI (proportional and integral) controllers to manage the miss ratio and utilization.[11]

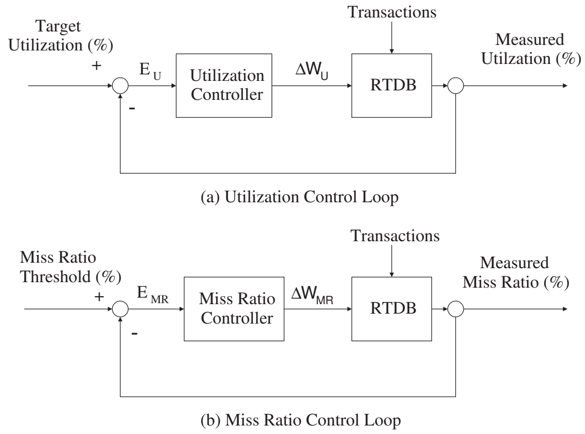

Figure 6: Miss Ratio/Utilization Controllers

Using the PI controller, the miss ratio control signal, i.e., the required workload adjustment to support the specified miss ratio, at the k^{th} sampling period is:

$$\Delta W_{MR}(k) = KP\left[E_{MR}(k) + KI\sum_{i=1}^{k} E_{MR}(i)\right] \tag{12}$$

where KP and KI, also described in Table 5, are proportional and integral control gains that need to be tuned to support the desired average/transient miss ratio. The performance error $E_{MR}(i)$, which is the input to the PI controller needed to compute the output $\Delta W_{MR}(k)$ at the k^{th} sampling period, is the difference between the miss ratio threshold and the miss ratio measured at the i^{th} sampling period where $1 \leq i \leq k$.[12]

From Eq. 12, we can derive a computationally more efficient form of a PI controller:

$$\Delta W_{MR}(k) = \Delta W_{MR}(k-1) + KP\left[(KI+1)E_{MR}(k) - E_{MR}(k-1)\right] \tag{13}$$

From this, we can derive the transfer function of a PI controller, which shows the relation between the input

[11]We use PI controllers, since a P controller alone can not cancel the steady state error [16]. Via simulation, we also verified that the CPU is underutilized when P controllers are used. In this paper, we do not consider more complex controllers such as a PID controller because our PI controllers meet *QoS-Spec*.

[12]The PI controller for utilization control is similar to this equation except that $E_U(k)$, i.e., the difference between the target utilization and the current utilization, is used to compute the required workload adjustment $\Delta W_U(k)$ to achieve the target utilization. Due to space limitations, we only present the miss ratio controller.

Table 5: Notations for the Closed Loop Models

Notation	Description
KP	Proportional control gain
KI	Integral control gain
$E_{MR}(k)$	Miss ratio error at the k^{th} sampling period
$\Phi_U(z)$	Transfer function of the feedback-based utilization controller
$\Phi_M(z)$	Transfer function of the feedback-based miss ratio controller

$E_{MR}(z)$ and output $\Delta W_{MR}(z)$, via the z-transform and some algebraic manipulation:

$$C(z) = \frac{\alpha(z-\beta)}{z-1} \text{ where } \alpha = K_p(K_I+1) \text{ and } \beta = \frac{1}{K_I+1} \tag{14}$$

At each sampling instant, we set the current control signal $\Delta W = Minimum(\Delta W_{MR}, \Delta W_U)$ to support a smooth transition from one system state to another, similar to [13]. 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 may incur a substantial overshoot later. To address this problem, we apply the integrator antiwindup technique [16]. At each sampling period, turn off the miss ratio controller's integrator if $\Delta W_U < \Delta W_{MR}$, since the current $\Delta W = \Delta W_U$. Otherwise, turn off the utilization controller's integrator.

When the transfer functions of a controller and a controlled system such as a real-time database are $X(z)$ and $Y(z)$, the closed loop transfer function is $\frac{X(z)Y(z)}{1+X(z)Y(z)}$ [16]. By substituting the transfer functions for the utilization model of Eq. 8 (miss ratio model of Eq. 11) and PI controller of Eq. 14 into this equation, we get the closed loop transfer functions for utilization and miss ratio feedback control, respectively:

$$\Phi_U(z) = \frac{C(z)T_U(z)}{1+C(z)T_U(z)} \tag{15}$$

$$\Phi_M(z) = \frac{C(z)T_M(z)}{1+C(z)T_M(z)} \tag{16}$$

Given a closed loop transfer function, one (with basic knowledge about control theory) can determine the performance of the feedback control system by selecting the sampling period and locating poles, i.e., real/imaginary roots of the denominator of the corresponding closed loop transfer function [16]. Frequent sampling could improve the transient performance such as overshoot and settling time. However, too frequent sampling could cause a sudden QoD degradation in our approach, especially when overloaded. For these reasons, we selected a relatively long sampling period, i.e., 5sec.

When the sampling period and closed loop transfer function are given, one can apply the Root Locus method [16] to locate the poles graphically.[13] In this way, one can tune the corresponding feedback controller. Using the Root Locus method, we locate poles at $0.75 \pm 0.112i$ to support the desired overshoot and settling time specified in *QoS-Spec*. Since these poles are inside the unit circle, the feedback control system is stable according to control theory [16]. The corresponding KP = 0.139 and KI = 0.176 for the miss ratio controller. KP = 0.212 and KI = 0.176 for the utilization controller.

6 Performance Evaluation

For performance evaluation, we have developed a real-time database simulator that models the real-time database architecture depicted in Figure 4. Each system component in Figure 4 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 target miss ratio and freshness (described in *QoS-Spec*) even in the presence of a wide range of unpredictable loads and access patterns. In this section, we discuss the simulation model, describe baseline approaches for performance comparison purposes, and present the performance evaluation results.

6.1 Simulation Model

In our simulation, we apply workloads consisting of sensor data updates and user transactions described as follows.

6.1.1 Sensor Data and Updates

Table 6: Average Inter-Trade Times for S&P Stock Items

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

As discussed before, real-time databases usually monitor the current real-world state using periodic updates,

[13] Note that modeling and deriving transfer functions are independent from the Root Locus method. The Root Locus method is only one of several ways to tune a feedback controller by locating poles given the closed loop transfer function and sampling period.

e.g., periodic sensor readings and stock price tracking. Therefore, the range of data update periods is a main factor to determine data freshness semantics. To derive an appropriate range of update periods, we have studied the real-time trace of NYSE stock trades streamed into the Bridge Center for Financial Markets at the University of Virginia. From 06/03/02 to 06/26/02, we measured the average time between two consecutive trades for tens of S&P 500 stock items. In Table 6, most representative ones are presented. (The other stock items not presented in Table 6 showed similar inter-trade times.) As shown in Table 6, the shortest average inter-trade time observed is 189ms for $\texttt{Item}_1$, while the longest one observed is 26sec for $\texttt{Item}_6$. (We have deleted the actual stock symbols for privacy purposes.)

From this study, we determined the range of sensor update periods for our simulation as shown in Table 7. For each sensor data object O_i, its update period (P_i) is uniformly selected in a range (100ms, 50sec). The shortest update period selected for our experiments, i.e., 100ms, is approximately one half of the average inter-trade time of $\texttt{Item}_1$ shown in Table 6. In contrast, the longest update period, i.e., 50sec, is approximately twice the average inter-trade time of $\texttt{Item}_6$ to model a wider range of update periods.

Table 7: Simulation Settings for Data and Updates

Parameter	**Value**
#Data Objects	2000 (1000 sensor data)
Update Period	$Uniform(100ms, 50sec)$
EET_i	$Uniform(3ms, 6ms)$
Actual Exec. Time	$Normal(EET_i, \sqrt{EET_i})$
Total Update Load	$\approx 50\%$

As shown in Table 7, there are 2000 data objects (1000 sensor data objects) in our simulated real-time database. We intentionally model a small database usually incurring a high degree of data contention [1]. As a result, supporting the desired miss ratio and freshness could be relatively hard. Each sensor 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 $1 \le i \le 1000$. EET_i is uniformly distributed in a range (3ms, 6ms). Note that EET_i includes not only memory access time but also raw sensor data processing time, e.g., radar image processing time. Upon the generation of an update, the actual update execution time is varied by applying a normal distribution $Normal(EET_i, \sqrt{EET_i})$ to $Stream_i$ to introduce errors in execution time estimates. The total update workload is manipulated to require approximately 50% of the total CPU utilization when the perfect QoD is provided.

6.1.2 User Transactions

Table 8: Simulation Settings for User Transactions

Parameter	Value
EET_i	$Uniform(5ms, 20ms)$
AET_i	$EET_i \cdot (1 + EstErr_i)$
Actual Exec. Time	$Normal(AET_i, \sqrt{AET_i})$
N_{DATA_i} (#Average Data Accesses)	$EET_i \cdot Data\ Access\ Factor = (5, 20)$
#Actual Data Accesses	$Normal(N_{DATA_i}, \sqrt{N_{DATA_i}})$
P(sensor data access)	0.5
P(write | non-sensor data access)	0.5
Slack Factor	(10, 20)

A source, $Source_i$, generates a group of user transactions whose inter-arrival time is exponentially distributed. $Source_i$ is associated with an estimated execution time (EET_i) and an average execution time (AET_i). We set $EET_i = Uniform(5ms, 20ms)$ as shown in Table 8. We selected this range of execution time to model a high performance main memory database. For example, TimesTen [19], a commercial main memory database system, can handle approximately 1000 transactions per second when each transaction includes two or three ODBC (Open Database Connectivity) calls in a 4 CPU machine. In our model, the simulated real-time database (assumed to run on a single CPU machine for the clarity of presentation) can process up to 200 transactions per second, i.e., 5ms per transaction execution, if there is no abort, restart or deadline miss due to data or resource conflicts. We also considered more complex transactions which might require longer execution times.

By generating multiple sources, we can derive transaction groups with different average execution time and average number of data accesses in a statistical manner. By increasing the number of sources we can also increase the workload applied to the simulated database, since more user transactions will arrive in a certain time interval. We set $AET_i = (1 + EstErr) \cdot EET_i$, in which $EstErr$ is used to introduce the execution time estimation errors. Note that QMF and all baseline approaches are only aware of the estimated execution time. Upon the generation of a user transaction, the actual execution time is generated by applying the normal distribution $Normal(AET_i, \sqrt{AET_i})$ to introduce the execution time variance in one group of user transactions generated by $Source_i$.

The average number of data accesses for $Source_i$ is derived in proportion to the length of EET_i, i.e., $N_{DATA_i} = data\ access\ factor \cdot EET_i = (5, 20)$. 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. A user transaction can access either sensor or non-senor data with the same probability (0.5). Given that a user transaction accesses a non-sensor data item, the transaction writes the corresponding (non-sensor) data item with the probability of 0.5 to model a relatively high degree of data contention.

For a user transaction, we set *deadline = arrival time + average execution time · slack factor*. A slack factor is uniformly distributed in a range (10, 20). For an update, we set *deadline = next update period*. For performance evaluation purposes, we have also applied other settings for execution time, data access factor, and slack factor different from the settings given in Table 8. We have confirmed that for different workload settings QMF can also support *QoS-Spec* by dynamically adjusting the system behavior based on the current performance error measured in feedback control loops. However, we do not include the results here due to space limitations. Interested readers are referred to [7, 8].

6.2 Baselines

In general, the trade-off issues between timeliness and freshness have hardly been studied in real-time databases except the work by Adelberg et al. [3] and our previous work [8]. For performance comparisons, we have developed several baseline approaches based on the best existing algorithms presented in [3] as follows.

- *Open-IMU*: In this approach, all incoming transactions are admitted, and all sensor data are immediately updated at their minimum update periods regardless of the current system status. Hence, the PF, QoD = 100% as long as sensor updates commit within their deadlines. Admission control and QoD adaptation schemes are not applied. Neither the closed loop scheduling based on feedback control is applied. Therefore, all the shaded components in Figure 4 are turned off. Open-IMU is similar to the "Update First" algorithm [3] which showed the best data freshness among the algorithms presented in [3].

- *Open-ODU*: In this approach, all incoming transactions are admitted regardless of the current miss ratio, similar to Open-IMU. However, in this approach all sensor data are updated on demand. An incoming sensor update is scheduled only if any user transaction is currently blocked to use the fresh version of the corresponding data item. Open-ODU is modeled after the "On Demand" algorithm that generally showed the best miss ratio among the algorithms presented in [3]. Therefore, Open-IMU and Open-ODU represent the best existing algorithms to support potentially conflicting miss ratio and freshness requirements except our previous work [8].

- *Open-IMU-AC*: This is a variant of Open-IMU in which admission control is applied to manage potential overloads.

• *Open-ODU-AC*: This is a variant of Open-ODU in which admission control is applied. Note that we apply the same admission control policy (described in Section 5.2) to Open-IMU-AC, Open-ODU-AC, and QMF for the fairness of performance comparisons.

6.3 Workload Variables and Experiments

To adjust the workload for experimental purposes, we define workload variables as follows.

• $AppLoad$: Computational systems usually show different performance for increasing loads, especially when overloaded. We use a variable, called $AppLoad$ = update 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 applied to the simulated real-time database when all incoming transactions are admitted and no QoD degradation is allowed. The actual load can be reduced in a tested approach by applying the admission control and QoD degradation, if applicable.

• *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 $\approx (1+EstErr)\cdot$ 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 data contention changes [1, 5]. For this reason, we apply different access patterns by using the $x-y$ access scheme [5], in which x% of data accesses are directed to y% of the entire data in the database and $x \geq y$. For example, 90-10 access pattern means that 90% of data accesses are directed to the 10% of a database, i.e., 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 only consider the uniform access pattern due to space limitations. We have verified that QMF can support the desired QoS under various access patterns. For more details, refer to [7, 8].

• *Fixed-QoD*: Unlike other variables described before, this variable is only applicable to QMF-2. For increasing *Fixed-QoD*, the overall QoD will increase, but less flexibility can be provided for overload management.[14] We applied *Fixed-QoD* ranging from 0.5 to 1 increased by 0.1 to observe whether or not the

[14] For performance evaluation purposes, we assume that D_{degr} is given for a specific *Fixed-QoD* value.

Table 9: 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%

desired average/transient miss ratio 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 $50\% \cdot [\textit{Fixed-QoD} + (1 - \textit{Fixed-QoD})/4]$. (According to *QoS-Spec*, $P_{i_{new}} = 4 \cdot P_{i_{min}}$ for every $O_i \in D_{degr}$ after the full QoD degradation.) In Table 9, 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 $Appload$ = 150% and *Fixed-QoD* = 0.5 the actual load can be reduced to approximately 130% after the full QoD degradation. The admission controller should handle the remaining potential overload, if necessary, to support the desired miss ratio and QoD. (Generally, it is hard to compute the relieved load when QMF-1 is applied, since it depends on the fraction of cold data in the database. In our experiments, the load was relieved up to 30% using the adaptive update policy.)

Table 10: Presented Sets of Experiments

Exp. Set	**Varied**	**Fixed**
1	$AppLoad = 70\%, 100\%, 150\%, 200\%$	EstErr = 0 HSS = 50% (uniform access)
2	*Fixed-QoD* = 0.5 − 1.0	AppLoad = 200% EstErr = 1 HSS = 50%
3	$AppLoad = 70\% \Rightarrow 170\%, 220\%$	EstErr = 1 *Fixed-QoD* = 0.5 HSS = 50%

Even though we have performed a large number of experiments for varying values of the workload variables, we only present the three most representative sets of experiments as summarized in Table 10 due to space limitations. We have verified that in the other sets of experiments not presented in this paper our approach can support the desired *QoS-Spec*, whereas the open-loop baseline approaches fail to support the specified miss ratio and/or freshness in the presence of unpredictable workloads and access patterns. In this paper, we present our performance results in a stepwise manner. We first compare the performance of QMF-1 to the open-loop approaches for increasing $AppLoad$. From this set of experiments, we select the best performing open-loop baselines. We compare their performance to QMF-2 for increasing *Fixed-QoD* in Experiment Set 2. Finally, in Experiment Set 3 we show the performance of QMF-2 under the harshest experimental settings among the tested ones.

• **Experiment Set 1**: As described in Table 10, 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 generally not available in database applications, which may include unpredictable aborts/restarts due to data/resource conflicts. Performance is evaluated for $AppLoad = 70\%$, 100%, 150%, and 200%. (We have also performed experiments for increasing $EstErr$ when $AppLoad = 200\%$, and observed that our approach can support *QoS-Spec*. For more details, refer to [7, 8].)

• **Experiment Set 2**: In this experimental set, we set $AppLoad = 200\%$ and $EstErr = 1$. Further, we increase *Fixed-QoD* from 0.5 to 1 by 0.1 to stress the modeled real-time database.

• **Experiment Set 3**: In this set of experiments, we apply bursty workloads. Initially, $AppLoad = 70\%$ considering the periodic update workload and user requests generated via the exponential distribution as discussed before. At 200sec (simulated time), a group of user transactions arrive *simultaneously* generating an additional 100% (150%) workload to model bursty arrivals. Note that this additional workload is instantly applied to the real-time database at 200sec. We do not apply a statistical approach to generate bursty traffic patterns since it is unknown whether or not real-time database workloads are bursty following a specific traffic pattern such as self-similar traffic.

In summary, this is a large set of tests performed over a wide range of parameter settings. This represents a robust performance study. 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 always be noticeable due to the small confidence intervals.)

6.4 Experiment Set 1: Effects of Increasing Load

In this section, we present the performance results for increasing loads.

Average Miss Ratio: As shown in Figure 7, Open-IMU shows the highest average miss ratio exceeding 70% when $AppLoad = 200\%$. By applying admission control, Open-IMU-AC significantly improves the miss ratio, but the miss ratio reaches $14.25 \pm 0.77\%$ when $AppLoad = 200\%$ violating the 1% threshold. Due to the relatively low update workload, Open-ODU shows a lower miss ratio than Open-IMU. However, its miss ratio

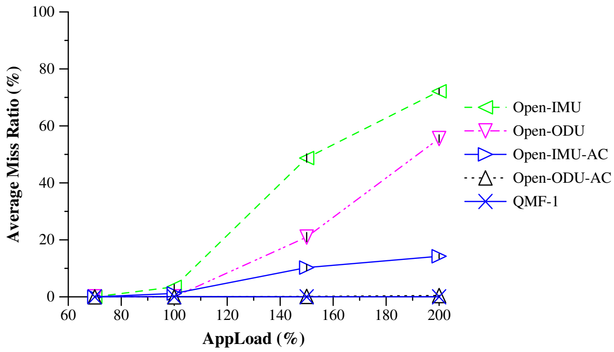

Figure 7: Average Miss Ratio (Exp. Set 1)

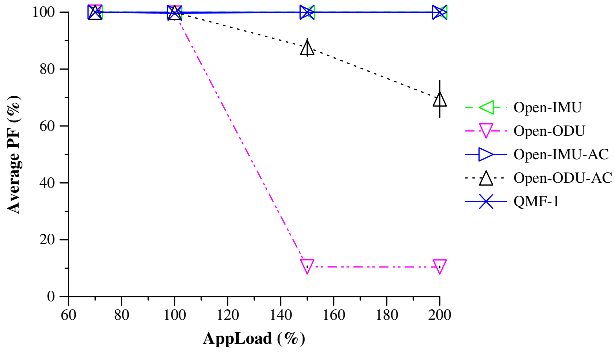

Figure 8: Average Perceived Freshness (Exp. Set 1)

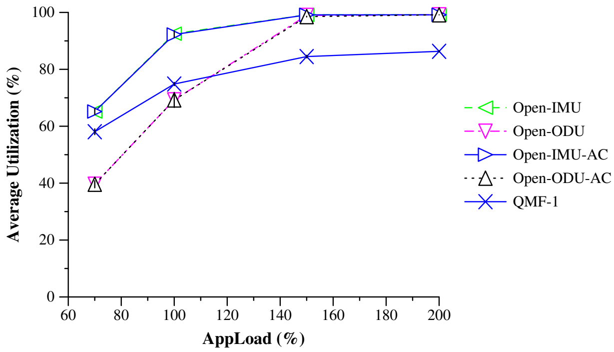

Figure 9: Average Utilization (Exp. Set 1)

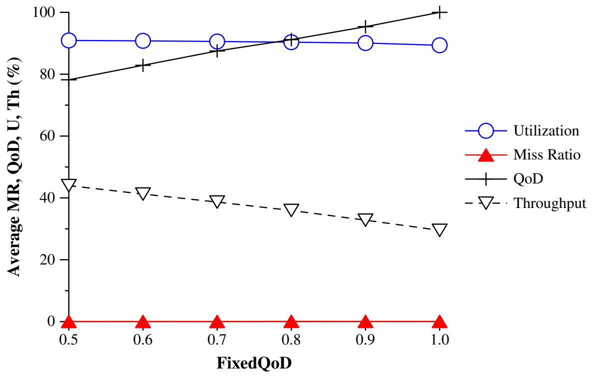

Figure 10: Average Performance of QMF-2 (Exp. Set 2)

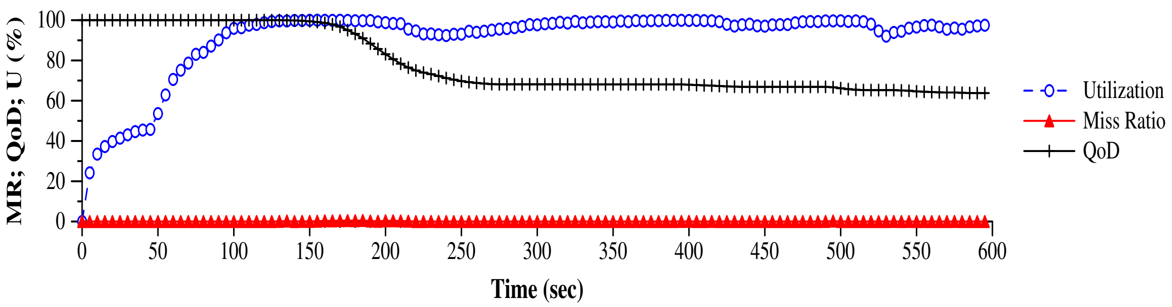

Figure 11: Transient Performance of QMF-2 in Exp. Set 2 (Fixed-QoD = 0.5)

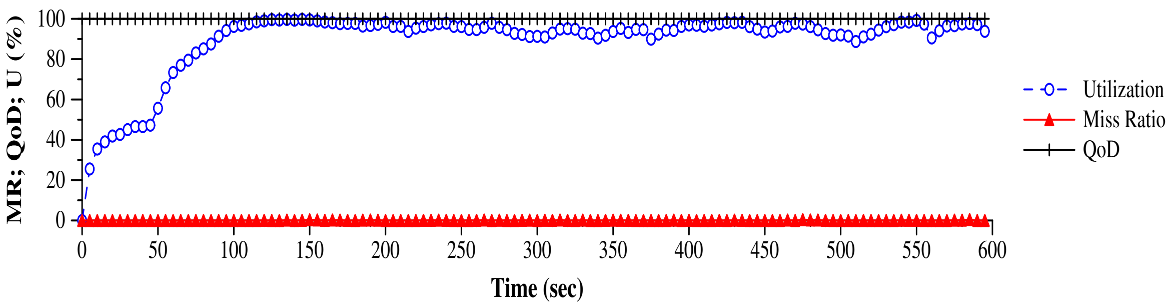

Figure 12: Transient Performance of QMF-2 in Exp. Set 2 (Fixed-QoD = 1)

reaches $55.56 \pm 1.30\%$ when $AppLoad = 200\%$.

In Figure 7, Open-ODU-AC shows a near zero miss ratio satisfying the 1% threshold for the tested $AppLoad$ values, similar to QMF-1. Both Open-ODU-AC and QMF-1 also showed a near zero transient miss ratio without any miss ratio overshoot in Experiment Set 1. (Due to space limitations, we do not plot the transient performance here.) Open-ODU-AC shows a good miss ratio in Experiment Set 1 because the zero $EstErr$ leads to an effective admission control. However, in the following subsection, we show that Open-ODU-AC fails to support the target freshness. When $AppLoad$ is high, many (backlogged) user transactions may have to use stale data instead of waiting for on-demand updates to meet their deadlines. Consequently, the perceived freshness drops. Further, more on-demand updates can miss their deadlines for increasing loads. Note that transactions can be forced to wait until on-demand updates finish without allowing stale data accesses. We have also considered this alternative approach for Open-ODU and Open-ODU-AC, and observed a significant miss ratio increase given a high $AppLoad$. In contrast, our approach can support both miss ratio and freshness requirements even given unpredictable workloads.

Perceived Freshness: In Figure 8, Open-IMU and Open-IMU-AC show the 100% perceived freshness by updating all sensor data immediately. QMF-1 shows a near 100% perceived freshness; the lowest perceived freshness was $99.95 \pm 0.08\%$ when $AppLoad = 200\%$.[15] As a result, the three curves representing the perceived freshness for Open-IMU, Open-IMU-AC, and QMF-1 overlap in Figure 8.

In contrast, Open-ODU significantly violates the target perceived freshness (98%). As shown in Figure 8, the freshness is approximately 10% when $AppLoad = 150\%$ and 200%. Open-ODU-AC achieves a relatively high freshness compared to Open-ODU. This is because Open-ODU-AC applies admission control to incoming user transactions. As a result, a fewer number of on-demand updates miss their deadlines. However, Open-ODU-AC also violates the target 98% freshness; its perceived freshness is below 80% when $AppLoad = 200\%$. This freshness violation can be a serious problem, since it may incur a large profit or product quality loss by processing many transactions using stale data.

Average Utilization: In Figure 9, the average CPU utilization is plotted for all tested approaches. The utilization of Open-IMU and Open-IMU-AC quickly reaches near 100%, since all sensor data are immediately updated without considering the current miss ratio. Open-ODU and Open-ODU-AC show a severe underutilization when $AppLoad = 70\%$ and 100%. This is because updates in Open-ODU and Open-ODU-AC are scheduled purely on

[15] Recall in this set of experiments (for QMF-1) we assume it is acceptable to use stale data because the application can extrapolate as long as the target freshness is not violated.

demand despite the possible underutilization (or freshness violation). In contrast, QMF-1 avoids both underutilization and overload as shown in Figure 9. The utilization ranges between $60\% - 85\%$ supporting the specified miss ratio and freshness. This is because QMF-1 dynamically adapts the workload considering the current system status measured in the feedback control loops.

6.5 Experiment Set 2: Effects of Increasing Fixed-QoD

In this section, we evaluate QMF-2's miss ratio, QoD, utilization, and real-time database throughput (defined in the following subsection) for increasing *Fixed-QoD*. We compare the performance of QMF-2 to Open-IMU and Open-IMU-AC. (In this experimental set, we do not consider Open-ODU and Open-ODU-AC, since these approaches can support neither the specified freshness nor miss ratio as discussed before.)

Average Performance: As shown in Figure 10, the QoD increases for increasing *Fixed-QoD* as expected. Also, QMF-2 achieves the near zero miss ratio at the cost of reduced utilization and throughput. When *Fixed-QoD* = 1, the QoD = 100% and the miss ratio is near zero.

As *Fixed-QoD* increases, a fewer number of user transactions can be admitted and processed due to the increasing update workload. Therefore, the user transaction throughput can be decreased. To further investigate this relation between *Fixed-QoD* and throughput, we define the throughput of real-time databases which apply firm deadline semantics:

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

where `#Timely` and `#Submitted` represent the number of user transaction committed within their deadlines and that submitted to the system (before admission control), respectively. Using this equation, we can theoretically compute the maximum possible throughput when $AppLoad = 200\%$ and the 100% QoD is required. The applied update and user transaction workloads are approximately 50% and 150%, respectively. Hence, the maximum possible throughput supporting the 100% QoD 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 QMF-2, Open-IMU, and Open-IMU-AC to this ideal 33% throughput.

As shown in Figure 10, the throughput of QMF-2 decreases from $43.97 \pm 0.93\%$ to $29.51 \pm 1.47\%$ as *Fixed-QoD* increases from 0.5 to 1. This means approximately 43% and 29% of the submitted user transactions are actually admitted and committed within their deadlines when *Fixed-QoD* = 0.5 and 1, respectively. The throughput of QMF-2 exceeds the ideal 33% when *Fixed-QoD* ≤ 0.8 due to the reduced but bounded QoD as described in *QoS-Spec*.

The utilization drops from approximately 91% to 89% when *Fixed-QoD* increases from 0.5 to 1. The overall utilization drop is smaller than the (user transaction) throughput decrease. This is because more updates are executed as *Fixed-QoD* increases.

Note that Open-IMU-AC showed a $22.35 \pm 1.29\%$ throughput given $AppLoad = 200\%$ and $EstErr = 1$ (applied to this set of experiments). When *Fixed-QoD* = 1, QMF-2's throughput is approximately 7% higher than Open-IMU-AC's ($\approx 29.51\% - 22.35\%$), while providing the 100% QoD. When *Fixed-QoD* = 0.5, the corresponding improvement of throughput is more than 21% ($\approx 43.97\% - 22.35\%$). For the same $AppLoad$ and $EstErr$, the throughput of Open-IMU was below 20%. In Open-IMU and Open-IMU-AC, the simulated real-time database is overloaded, since all incoming transactions are simply admitted or admission control is not effective enough due to large errors in execution time estimates. As a result, many deadlines are missed causing the throughput decrease. (Open-IMU and Open-IMU-AC missed approximately 79% and 73% of transaction deadlines when $AppLoad = 200\%$ and $EstErr = 1$.)

From these results, we observe it is sensible to prevent potential overloads using admission control (and QoD management) according to the feedback control signal. In this way, QMF-2 can improve the throughput compared to Open-IMU and Open-IMU-AC, while achieving a near zero miss ratio and 100% QoD, if desired.

Transient Performance: Figures 11 and 12 show the transient miss ratio, QoD, and utilization when *Fixed-QoD* is set to 0.5 and 1, respectively. (We have also measured the transient performance for other *Fixed-QoD* values. We observed the similar 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 11 and 12, QMF-2 does not exceed the 1% miss ratio threshold without any miss ratio overshoot throughout the experiments. In Figure 11, the QoD is decreasing to avoid a potential miss ratio overshoot given $AppLoad = 200\%$.

As shown in Figure 12, QMF-2 achieves the 100% QoD without any miss ratio overshoot when *Fixed-QoD* = 1. (The average utilization and throughput are slightly reduced to support the 100% QoD as discussed before.) Therefore, for example, a DBA of a financial trading or factory automation company can select an appropriate QoD considering the application specific data semantics without a miss ratio overshoot at the cost of a possible throughput decrease.

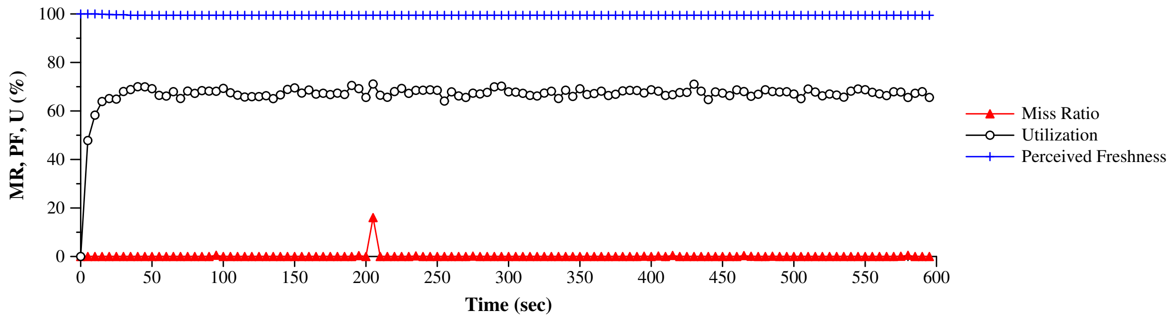

Figure 13: Transient Performance in the Presence of Bursty Arrivals (100% Workload Burst)

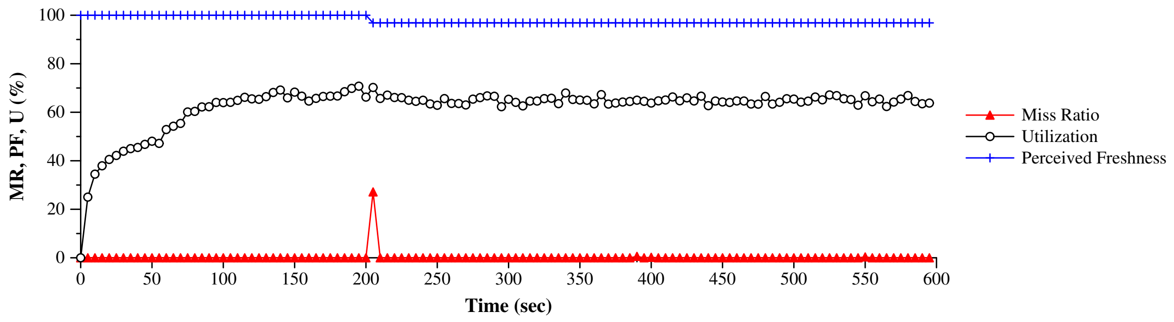

Figure 14: Transient Performance in the Presence of Bursty Arrivals (150% Workload Burst)

In Experiment Sets 1 and 2, both QMF-1 and QMF-2 were able to support the desired average/transient miss ratio, while providing the target PF or QoD. Note that QMF-1 and QMF-2 have achieved a near zero miss ratio and perfect freshness. Therefore, QMF can be considered to meet fundamental requirements for real-time transaction processing. At the same time, QMF achieved the higher throughput than the open-loop baselines did. These results show the effectiveness of QMF to support the specified real-time database QoS.

6.6 Experiment Set 3: Effects of Bursty Traffic

In this subsection, we consider bursty workloads unlike all the other sets of experiments presented previously. Despite the bursty arrivals, QMF-2 supports the specified *average* miss ratio and data freshness. When the 100% workload burst is given in addition to the initial 70% $AppLoad$, the average miss ratio is 0.16 ± 0.03% and QoD is 99.83 ± 0.03%. Given the 150% burst workload (in addition to the initial 70% $AppLoad$), the average miss ratio is 0.23 ± 0.03% and QoD is 97.91 ± 0.38%. (Recall *Fixed-QoD* = 0.5 in this set of experiments.) This is because the bursty workload is transitory compared to the 10min simulation length.

As shown in Figure 13, the transient miss ratio increases to approximately 16% at 205sec, i.e., the next sampling instant after the bursty arrivals, given the 100% workload burst at 200sec. Given the 150% workload burst, the transient miss ratio is approximately 27% at 205sec as shown in Figure 14. (Both in Figures 13 and 14, the miss ratio is back to near zero after 205sec.) These miss ratio overshoots are experienced mainly because our current feedback control model does not consider bursty workloads, and therefore, it is not responsive enough against bursty arrivals. (As a result, the QoD is not degraded enough to manage the bursty overload. Neither, enough admission control is applied to incoming bursty requests.)

A further research is necessary to characterize real-time database workloads in terms of user request arrival patterns. Given a specific traffic pattern, the feedback control model can be further refined. A traffic shaper can also be developed to smooth bursts, if any. These research issues are reserved for future work.

7 Related Work

Considering the abundance of QoS and database research work separately, relatively less work has been done on real-time database QoS management [3, 20]. This can be a serious problem; it has been reported that approximately $420 million revenue loss resulted due to late, i.e., non-real-time, e-commerce transaction processing in 1999 [2].

Adelberg et al. have found that timeliness and freshness requirements may conflict in real-time databases [3]. To balance the conflicting requirements, they presented several algorithms for scheduling sensor data updates and user transactions. Our flexible freshness management schemes extend their work by dynamically adapting the update policy or periods, if necessary, to manage overloads. Consequently, QMF achieves a better performance as discussed before.

Other aspects of the real-time database performance can be traded off to improve the miss ratio. In [15] and [20], the correctness of answers to queries can be traded off to enhance timeliness by using the database sampling and milestone approach [12], respectively. In these approaches, the accuracy of the result can be improved as the sampling or computation progresses, while returning approximate answers to the queries, if necessary, to meet their deadlines. An adaptable security manager is proposed in [18], in which the database security level can be temporarily degraded to enhance timeliness. These approaches carefully trade off conflicting performance metrics, and thereby significantly improve the miss ratio. Conceptually, our flexible freshness management scheme is analogous to these approaches in terms of performance trade-off. However, it is hard to directly compare our freshness metrics to their performance metrics, i.e., accuracy of query results and data security. Also, a naive combination of our scheme with theirs may cause unexpected side effects incurring intolerable inaccuracy of query results or security risks, which are beyond the scope of this paper.

Feedback control has been applied to QoS management and real-time scheduling [11, 13, 23] due to its robustness against unpredictable operating environments [16]. However, these work do not consider real-time transaction processing and freshness management issues; therefore, they are not directly comparable to our work. Feedback control has also been recognized as a viable approach to manage the (non-real-time) database performance [21, 22]. They substantially improved the database throughput/response time by applying high level notions of feedback control, i.e., performance observation and dynamic adaptation of the system behavior in a conceptual feedback loop. However, they intend to manage the database throughput, which is known to be very different from managing the miss ratio of real-time transactions [9, 17]. Generally, a lot of work remains to be done to manage the database QoS, since the related research is still in its infancy.

8 Conclusions

Processing real-time transactions within their deadlines using fresh data is essential but challenging. To address the problem, we have (1) defined QoS metrics in terms of miss ratio and novel data freshness metrics to let a DBA specify the desired QoS, (2) introduced flexible QoD management schemes, (3) presented a new QoS management architecture that can support the desired QoS even given dynamic workloads and access patterns, and (4) designed real-time database workloads considering the real-time data semantics observed in NYSE trade traces and transaction execution times modeling high performance main memory databases. In an extensive simulation study, QMF achieved a significant performance improvement compared to several baselines including best existing algorithms for miss ratio and freshness trade-off in real-time databases, while supporting the target miss ratio and freshness (except for bursty workloads). As one of the first work on QoS management in real-time databases, the significance of our work will increase as the demand for (and importance of) real-time data services increases. In the future, we will further investigate the timeliness and freshness issues in both centralized and 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] ActiveMedia Research, Real Numbers behind 'Net Profits. http://www.activmediaresearch.com/.

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

[4] J. R. Haritsa, M. J. Carey, and M. Livny. On Being Optimistic about Real-Time Constraints. In *ACM PODS*, 1990.

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

[6] J. Huang, J. A. Stankovic, K. Ramamritham, and D. Towsley. Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes. In *VLDB*, 1991.

[7] K. D. Kang. *QoS-Aware Real-Time Data Management*. PhD thesis, University of Virginia, May 2003.

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

[10] J. Lee and S. H. Son. Performance of Concurrency Control Algorithms for Real-Time Database Systems. In V. Kumar, editor, *Performance of Concurrency Control Mechanisms in Centralized Database Systems*. Prentice Hall, 1996.

[11] 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.

[12] 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.

[13] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback Control Real-Time Scheduling: Framework, Modeling and Algorithms. *Real-Time Systems, Special Issue on Control-Theoretical Approaches to Real-Time Computing*, 23(1/2), May 2002.

[14] Moneyline Telerate Plus. http://www.futuresource.com/.

[15] 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.

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

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

[18] 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.

[19] The TimesTen Team. In-Memory Data Management for Consumer Transactions The Times Ten Approach. In *ACM SIGMOD*, 1999.

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

[21] G. Weikum, C. Hasse, A. Mönkeberg, and P. Zabback. The COMFORT Automatic Tuning Project. *Information Systems*, 19(5):381–432, 1994.

[22] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback. Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. In *VLDB Conference*, 2002.

[23] R. Zhang, C. Lu, T. F. Abdelzaher, and J. A. Stankovic. ControlWare: A Middleware Architecture for Feedback Control of Software Performance. In *International Conference on Distributed Computing Systems*, 2002.