Specifying and Managing Quality of Real-Time Data Services [*†]

Kyoung-Don Kang Sang H. Son John A. Stankovic

Department of Computer Science

University of Virginia

{kk7v, son, stankovic}@cs.virginia.edu

September 18, 2002

Abstract

The demand for real-time data services is increasing in many applications including e-commerce, agile manufacturing, and telecommunication 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, it is challenging to meet these fundamental requirements due to dynamic workloads and data access patterns in these applications and potential conflicts between transaction timeliness and data freshness requirements. In this paper, we define average/transient deadline miss ratio and new data freshness metrics to specify the required quality of real-time data services. We also present a novel QoS management architecture to support the required QoS even in the presence of unpredictable workloads and access patterns. To prevent overload and support the required QoS, the presented architecture applies feedback control, admission control, and flexible freshness management schemes. A simulation study shows that our QoS-aware approach can support the required QoS, whereas baseline approaches fail to support the required miss ratio and/or freshness. Further, our approach shows a comparable performance to the theoretical oracle that is privileged by a complete future knowledge of data accesses.

Keywords: QoS metrics for real-time data services, QoS management architecture, flexible freshness management, feedback/admission control

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

[†]An early version of this paper was presented in the 14th Euromicro Conference on Real-Time Systems [10].

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, 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] which 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. For example, Lockheed Martin 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 the UK, NEC in Japan, and ClusterRa in Norway have also implemented real-time databases for various application areas, but for similar reasons. Even though the need for real-time data services has been demonstrated, these and other real-time database systems are only initial attempts with many remaining issues.

Real-time databases need to execute transactions within their deadlines using fresh data, but meeting these fundamental requirements is very 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 QoS management architecture for Miss ratio and Freshness guarantees). A key advantage of QMF is that QMF can provide QoS guarantees (in terms miss ratio and data freshness) for admitted transactions even in the presence of unpredictable workloads and data access patterns. To support the required QoS, QMF dynamically adjusts the system behavior in the feedback control loop using admission control and flexible freshness management, if necessary.

Appropriate QoS metrics are essential to directly reflect (and support) the user required QoS. In QMF, we define average/transient miss ratio and new data freshness metrics to specify the required quality of real-time

[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.

data services. We introduce a notion of *perceived freshness* to measure the freshness of data accessed by timely transactions – transactions that commit within their deadlines. Using the notion of perceived freshness, we present an *adaptive update policy* to maintain the freshness in a cost-effective manner. Initially, all data are updated immediately when their new sensor readings arrive. Under overload, some sensor data can be updated on demand, if necessary, to improve the miss ratio (as long as the target perceived freshness is supported). This flexible approach contrasts to the existing database update policy, commonly accepted in the real-time database research such as [3, 11, 26, 37], which is fixed and not adaptable regardless of the current system status.

The adaptive update policy is effective in balancing possibly conflicting timeliness and freshness requirements [10]. However, it has a potential disadvantage. Some stale data accesses could be allowed to meet transaction deadlines even though the chances are small.[2] When a data object is updated on demand, a real-time transaction accessing that data may have to use an old version of the data to meet its deadline [3]. 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 introduce novel notions of *QoD (Quality of Data)* and *flexible validity intervals* to manage the freshness. Several QoD parameters are also provided to let users specify the acceptable range of QoD. When overloaded, update periods of some sensor data can be relaxed within the acceptable range of QoD to reduce the update workload, if necessary. However, sensor data are always maintained fresh in terms of flexible validity intervals. (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) [26]. 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. Further, we surveyed the update rates for various physical sensor products. 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 [3, 11, 26, 37] 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 large range of workloads and access patterns. In contrast, several baseline approaches, which model widely accepted transaction processing mechanisms, fail to support the specified miss ratio and/or data freshness. No-

[2] In Section 6 we show that we can support 98% target perceived freshness and 1% miss ratio threshold using the adaptive update policy (with the feedback-based miss ratio control and admission control schemes). This approach is acceptable for some applications in which a stale data access is undesirable, but not fatal, e.g., when a transaction can extrapolate current values from the stale data. However, it is still more desirable to maintain data as fresh as possible.

tably, QMF shows a comparable performance to a clairvoyant oracle that can support the optimal miss ratio and freshness by exploiting a complete future knowledge of data accesses.

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 is described. Section 6 presents the performance evaluation results. 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, real-time transaction types, deadline semantics, and average/transient miss ratio considered in this paper.

2.1 Database Model, Transaction Types, and Deadline Semantics

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, 8, 24, 32].

In this paper, we classify transactions as either user transactions or sensor updates. Periodic updates, commonly adopted in real-time databases such as [11, 26, 37], capture the continuously changing real-world state. User transactions 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 periodically 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 quality, resulting in wasted system resources, due to possible changes in the market or manufacturing status. As discussed before, database workloads and data access patterns might vary dynamically. For this reason, 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

The deadline miss ratio is one of the most important performance metrics in real-time applications. For admitted transactions, the deadline miss ratio is:

$$MR = 100 \times \frac{\texttt{\#Tardy}}{\texttt{\#Tardy} + \texttt{\#Timely}} \,(\%)$$

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.

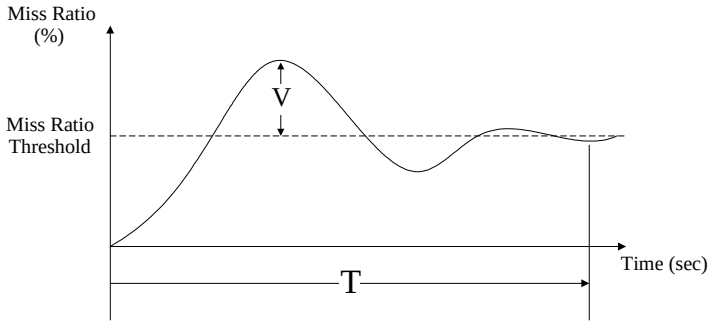


Figure 1: Definition of Overshoot (V) and Settling Time (T) in Real-Time Databases

Long-term performance metrics, e.g., average miss ratio, are not sufficient to specify the required performance of dynamic systems whose performance could change significantly in a relatively short time interval [18]. For this reason, transient performance metrics such as overshoot and settling time, shown in Figure 1, are adopted from control theory to specify the required performance of real-time systems:

- *Overshoot (V)* 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 (T)* 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 required 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 freshness management scheme selected between the two.

3 Flexible Freshness Management: QMF-1

In this section, we describe the notion of validity intervals, discuss data freshness metrics, and introduce 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 [26]. 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.[3]

For O_i, the update period $P_i = 0.5 \times avi(O_i)$ is necessary to support the sensor data freshness [26]. Data temporal consistency can be violated if we set $P_i = avi(O_i)$. Consider Figure 2 for example. Ideally, a real-time database starts updating O_i at time t as soon as a new sensor reading for O_i arrives. Even though the next sensor reading arrives at $t + P_i$, O_i could not be updated for some time interval due to the high data/resource contention. As a result, the second update may actually 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 the update period P_i. This situation can be avoided by setting $P_i = 0.5 \times avi(O_i)$.

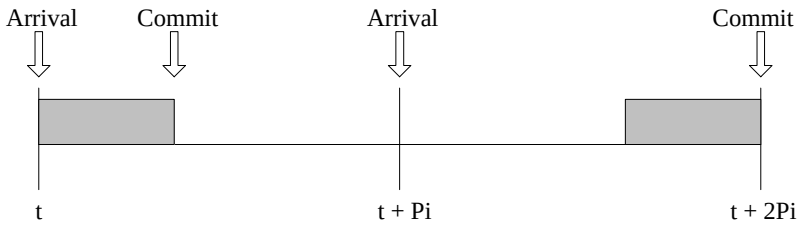


Figure 2: Periodic Updates for a Sensor Data Object

To manage data freshness in an adaptive manner, we consider two key freshness metrics, namely *database freshness* and *perceived freshness* [10]:

[3]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.

- *Database Freshness*: Database freshness, also called QoD (Quality of Data) in this paper, is the ratio of fresh (sensor) data to the entire data in a real-time database.

- *Perceived Freshness (PF)*: In contrast to database freshness, perceived freshness is defined for the data accessed by timely transactions. Let $N_{accessed}$ represent the number of data accessed by timely transactions. Let us call the number of fresh data accessed by timely transactions N_{fresh}.

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

Since we apply firm deadline semantics, the data freshness guarantee is provided in terms of perceived freshness. 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, which may cause many deadline misses.

3.2 Cost-Benefit Model for Updates

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. To consider the benefit, access frequency is measured for each data object. If a data object is accessed frequently, e.g., a popular stock price, an update of that data object 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:

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

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:

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

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 *Update Frequency*$[i]$ of O_i in a sampling period is known, we can compute AUR for O_i:

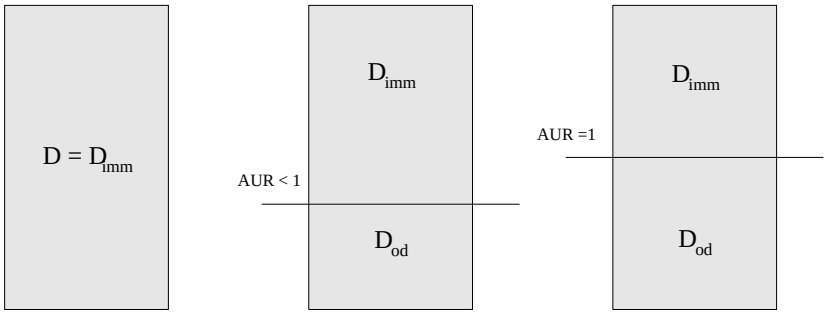
$$AUR[i] = \frac{SAF[i]}{Update\ Frequency[i]} \tag{3}$$

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. Otherwise, it is not cost-effective. For simplicity, let us call a data item *hot* if its AUR ≥ 1. Otherwise, we call it *cold*.

Note that 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.

3.3 Adaptive Update Policy

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. 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. (Cold data could always be updated on demand regardless of the current system load. However, that approach may increase the response time of transactions accessing cold data.)



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

Figure 3: Update Policy Adaptations

For example, consider Figure 3 in which D represents the set of all sensor data in the database. D_{imm} is the set of data updated immediately, while D_{od} stands for the set of data updated on demand. Since a data object is updated either immediately or on demand in our approach, $D = D_{imm} \bigcup D_{od}$ and $D_{imm} \bigcap D_{od} = \emptyset$. Initially, every data is updated immediately. As the load increases, a larger fraction of cold data objects is updated on demand. We call this *QoD degradation*, since lazy updates reduce the QoD (i.e., database freshness). In contrast, 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*.

3.3.1 QoD Degradation: Miss Ratio Adjustment

By degrading the QoD, we can reduce the update workload. A remaining question is how to estimate the saved CPU utilization from the QoD degradation. To handle this problem, we first consider the number of saved updates due to the degradation for O_i:

$$N = Update\ Frequency[i] - SAF[i] \tag{4}$$

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

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

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

In our approach, feedback controllers compute the CPU utilization adjustment, called ΔU, required to support the specified miss ratio. When the current miss ratio is over the specified threshold, e.g., 1%, ΔU becomes negative to request the reduction of the CPU utilization. (In Section 5, feedback control and its interactions with the QoD management are discussed in detail.) Under overload, the update policy can be degraded for a cold data item. After the update policy degradation for a single data object, the new CPU utilization adjustment is:

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

The degradation continues for the next data object in D_{imm} with the least AUR until $\Delta U_{new} \geq 0$ or the degradation bound, i.e., $AUR = 1$, is reached as shown in Figure 3(c). Note that further degradation past the degradation bound is meaningless. If a hot data object whose $AUR \geq 1$ is updated on demand, the number of updates may not be reduced, but many transactions may have to miss their deadlines waiting for the on-demand update. Hence, it is necessary to update hot data immediately.

3.3.2 QoD Upgrade: Freshness Adjustment

To ensure that timely transactions access fresh data, the update policy is switched back to the immediate policy for certain data objects upon a violation of the target perceived freshness as long as the specified miss ratio is not violated (i.e., $\Delta U > 0$). A key issue in QoD upgrade is to avoid a potential miss ratio overshoot in the next sampling period, while improving the perceived freshness as needed at the same time. For this purpose, we define the perceived freshness error and derive the upgrade bound as follows.

Given a target perceived freshness PF_t, the current perceived freshness PF can be measured in a sampling period. The perceived freshness error in a sampling period is:

$$PF_{error} = \begin{cases} 0 & \text{if} \quad PF \geq PF_t; \\ PF_t - PF & \text{if} \quad PF < PF_t. \end{cases} \tag{7}$$

Given a non-zero PF_{error}, it is clear that on-demand updates have failed to provide the target perceived freshness in the current sampling period. Therefore, some of these data, especially ones having relatively high AUR values, should be updated immediately in the next sampling period. The freshness can be improved approximately in proportion to the number of data moved from D_{od} to D_{imm}.

An *upgrade bound* K is derived in terms of the number of data whose update policy will be upgraded. Let D'_{imm} represent the set of data to be updated immediately in the next sampling period after the upgrade. The upgrade bound is determined in proportion to the current freshness error and the cardinality of the set D_{od}:

$$K = |D'_{imm}| - |D_{imm}| \approx PF_{error} \times |D_{od}| \tag{8}$$

By moving one data object O_i from D_{od} to D_{imm}, the number of updates and the corresponding CPU utilization may increase. Using Eq. 4, we estimate the potential increase in the number of updates. We also estimate the required extra CPU utilization for the increase using Eq. 5. The QoD upgrade is repeated until the upgrade bound

in Eq. 8 is reached, or the available CPU utilization becomes not enough, i.e., ΔU_{new} (= $\Delta U -$ the required CPU utilization for an upgrade) ≤ 0.

In QMF-1, the QoD is dynamically adjusted based on the current miss ratio or perceived freshness. Due to the approximation and unpredictable workloads/access patterns, our QoD adaptation may not be precise. However, the target performance can be achieved by continuously adjusting the QoD based on the performance error measured in the feedback loop. By balancing update and transaction workload efficiently, the specified deadline miss ratio and perceived freshness can be achieved at the same time.

4 Flexible Freshness Management: QMF-2

In this section, we consider an alternative approach for freshness management, called QMF-2, to avoid the stale data accesses possible in QMF-1. Novel notions of QoD and flexible validity intervals are introduced for this new approach. We also give a QoS specification that will be used to demonstrate the applicability of QMF against unpredictable workloads and access patterns.

4.1 Overview

In QMF-2, all sensor data are updated immediately. This is to avoid possible deadline misses due to the delay for on-demand updates as discussed before. When overloaded, the update period (instead of the corresponding update policy) of a relatively less critical data item can be increased to improve the miss ratio.[4] Since the update period can be adapted, the traditional notion of absolute validity intervals is not applicable to maintain the freshness in QMF-2. Instead, data freshness is maintained in terms of flexible validity intervals in QMF-2 (discussed in Section 4.2).

In this approach, we assume that the importance of data, e.g., popularity of stock items or relative importance of manufacturing steps regarding the final product quality, is available to a corporate user, e.g., a financial trading or factory automation company. This is a reasonable assumption, since these corporate users (rather than real-time database developers) usually have better understanding about application specific data semantics. In QMF-2, we let a (corporate) user (re)specify the required range of QoD to directly reflect application specific data semantics. Under overload, the QoD can be degraded (i.e., some sensor data can be updated less frequently) within the

[4] Task period adjustment is previously studied to improve the miss ratio in real-time (non-database) systems such as [14, 15]. However, database issues such as data freshness are not considered in these work.

specified range to improve the miss ratio, if necessary. For system operation and maintenance purposes, the DBA sets several QoD parameters (discussed in Section 4.3) to meet the user specified QoD requirements.

4.2 Quality of Data and Flexible Validity Intervals

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 [12]. However, their work neither considers miss ratio/freshness guarantee issues nor relaxes the sensor update frequency to reduce the miss ratio. Therefore, their definition of QoD is not directly applicable to QMF-2. When there are N sensor data objects in a real-time database, we define the current QoD:

$$QoD = 100 \times \sum_{i=1}^{N} \frac{P_{i_{min}}}{P_{i_{new}}} \ (\%) \tag{9}$$

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. 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.

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 $fvi_{new}(O_i) = 2 \times P_{i_{new}}$ 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. Therefore, the QoD defined in Eq. 9 is the only freshness metric in QMF-2.

4.3 QoD Management

In our approach, the DBA can set three QoD parameters considering the corporate user's QoD requirements in terms of acceptable range of QoD, which can vary from a specific real-time database application to another, as follows:

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

other set of data in the database, called D_{degr}, the QoD can be degraded under overload. When *Fixed-QoD* = 1, no QoD degradation is allowed at all, i.e., $D_{degr} = \emptyset$. In contrast, when *Fixed-QoD* = 0 the QoD can be degraded for all data, if necessary.

- **Max-Degr**: When *Fixed-QoD* < 1, users can also specify *Max-Degr* to avoid an indefinite QoD degradation. For a sensor data object $O_i \in D_{degr}$, $P_{i_{new}} \leq$ *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 graceful QoD degradation if any degradation is allowed. For example, when *Step-Size* = 10%, $P_{i_{new}} = 1.1 \times P_i$ for $O_i (\in D_{degr})$ after a QoD degradation. Upon a QoD degradation, the update period P_i is increased by 10% between two consecutive sampling periods 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 [31] allow users to specify acceptable periods (e.g., ranging between 1 minute and 60 minutes in Telerate Plus) to monitor stock prices. Further, our approach can adjust the QoD within the specified QoD range, if necessary, to improve the miss ratio.

4.4 QoS Specification

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

- **Miss Ratio**: The average miss ratio should be below 1%. An overshoot should be below 30% to support the consistent real-time performance. Therefore, the transient miss ratio should not exceed $1.3\% = 1 \times (1 + 0.3)\%$. The settling time should be shorter than 40sec, e.g., think time between trades.

- **Freshness Requirements**: For QMF-1, we set the target perceived freshness $PF_t = 98\%$, therefore, the perceived freshness should be at least 98%. For QMF-2, we set *Max-Degr* = 4 and *Step-Size*= 10%. We do not fix *Fixed-QoD*, but apply increasing *Fixed-QoD* for performance evaluation purposes (Section 6). As *Fixed-QoD* increases, *QoS-Spec* becomes more stringent because the QoD degradation is allowed for a less fraction of the entire sensor data.

5 Architecture for QoS Management of Real-Time Data Services

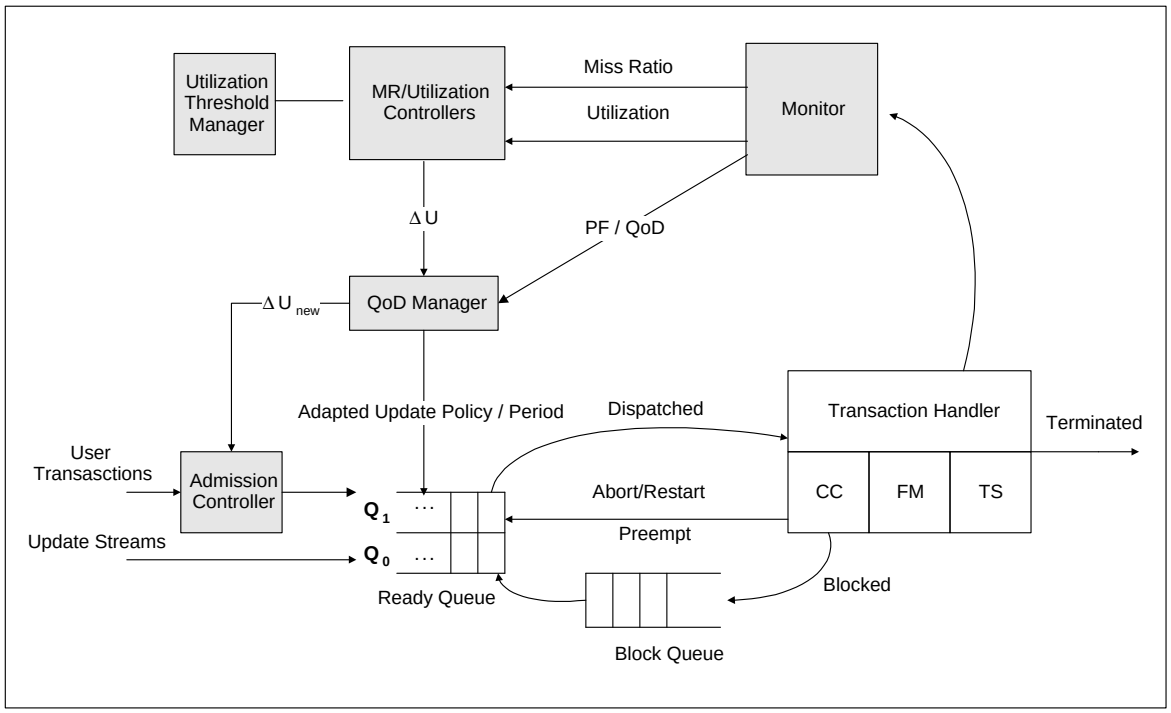


Figure 4: Real-Time Data Management Architecture for QoS Guarantees

Figure 4 shows our QoS management architecture. 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, CPU utilization, and PF/QoD are monitored. The miss ratio and utilization controller compute the required CPU utilization adjustment (ΔU) considering the current performance error such as the miss ratio overshoot or CPU underutilization. Based on ΔU, the QoD manager adapts the update policy or period (according to the selected freshness management scheme) to reduce the update workload, if necessary. The admission controller enforces the remaining utilization adjustment (ΔU_{new}) after potential update workload adaptation. A detailed discussion of the system components is given in the next subsections.

5.1 Transaction Handler

The transaction handler provides an infrastructure for real-time data services, which consists of a concurrency controller (CC), a freshness manager (FM), and a transaction scheduler (TS). 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. We selected 2PL-HP, since it is well studied in the real-time database community and is free of priority

inversion.

The FM checks the freshness before accessing a data item using the corresponding *avi* (or fvi if there is 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 (Q_0 and Q_1 as shown in Figure 4). 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 at Q_0. In each queue, transactions are scheduled in an EDF (Earliest Deadline First) manner. To guarantee the freshness, all immediate updates are scheduled in Q_0. On-demand updates (possible in QMF-1) will be scheduled in Q_1 if any transaction is blocked for the update to access the fresh data. Admitted user transactions are also scheduled in Q_1. To guarantee the required miss ratio for user transactions, we apply feedback control, flexible QoD management, and admission control. In this way, we can meet both miss ratio and data freshness requirements.

5.2 Interactions of Key System Components

The overall behavior of QMF, especially regarding the interactions among the feedback control, QoD management, and admission control, is described in Figure 5. If $\Delta U \geq 0$, i.e., the current miss ratio is below the required threshold (MR_t), and the freshness requirement is also met, more transactions are admitted to avoid potential underutilization. When the system is overloaded, i.e., $\Delta U < 0$, the CPU utilization should be reduced. To reduce the utilization, the QoD can be reduced if the degradation bound, i.e., AUR = 1 (or *Max-Degr*), is not reached yet in QMF-1 (or QMF-2).

When $\Delta U < 0$ and the QoD can not be degraded further to support the required freshness, we apply admission control to prevent overload. It is known that admission control is effective to prevent the database *thrashing*, in which the transaction throughput and response time could be degraded drastically due to overload [35]. Admission control can also improve the miss ratio of real-time transactions significantly, especially under overload [10, 11].

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.

1. Monitor the deadline miss ratio (MR), freshness, and CPU utilization. In QMF-1, also collect access statistics and compute AUR.
2. At each sampling period, compute the miss ratio and utilization control signals. Set ΔU to the minimum of the two control signals for a smooth transition from one system state to another. Based on ΔU and the current system behavior, perform one of the following alternative actions in either QMF-1 or QMF-2 as selected by the DBA.

QMF-1:

- If $\Delta U \geq 0$ (i.e., $MR < MR_t$) and $PF \geq PF_t$, admit more transactions to prevent potential underutilization.
- If $\Delta U < 0$ and $PF \geq PF_t$, degrade the QoD. Adjust $\Delta U_{new} = \Delta U +$ the utilization saved from a QoD degradation. Repeat until $\Delta U_{new} \geq 0$ or the degradation bound is reached. Inform the admission controller of ΔU_{new}.
- If $\Delta U > 0$ and $PF < PF_t$, upgrade the QoD. Set $\Delta U_{new} = \Delta U -$ the required CPU utilization for an upgrade. Repeat until $\Delta U_{new} \leq 0$ or the upgrade bound is reached. Inform the admission controller of ΔU_{new}.
- If $\Delta U < 0$ and $PF < PF_t$, do not admit any incoming transaction until ΔU becomes positive after some of the currently running transactions terminate.

QMF-2:

- If $\Delta U \geq 0$, admit more user transactions to avoid potential underutilization.
- If $\Delta U < 0$, *Fixed-QoD* < 1, and *Max-Degr* is not reached yet, increase the update periods by *Step-Size* for the sensor data objects in D_{degr}. Adjust $\Delta U_{new} = \Delta U +$ the CPU utilization saved from the QoD degradation. Stop the degradation when $\Delta U_{new} \geq 0$ or *Max-Degr* is reached. If $\Delta U_{new} < 0$ after all possible QoD degradations, apply admission control to newly incoming transactions.

Figure 5: QoS Management for Miss Ratio and Freshness Guarantees

5.3 Feedback Control

Feedback control is very effective to support the required performance when the system model includes uncertainties [23]. In this paper, we apply a feedback control real-time scheduling policy called FC-UM [19]. We selected FC-UM, since it can provide a certain miss ratio guarantee without underutilizing the CPU even when the precise workload model is not known *a priori*. As shown in Figure 6, FC-UM uses miss ratio and utilization controllers. When the system is underutilized (overloaded), the utilization (miss ratio) controller takes control to handle underutilization (overload). Since these two controllers work exclusively (depending on the current system load), FC-UM can support the stability of the feedback control system. Note that we do not consider designing a separate feedback controller for freshness management. This is because the miss ratio and freshness requirements can compete for CPU cycles and sensor data potentially leading to an unstable feedback control system (oscillating between many deadline misses and freshness violations). For these reasons, we use the QoD manager (an actuator from the control theory perspective) to manage the freshness while balancing potentially

conflicting miss ratio and freshness requirements as discussed before.

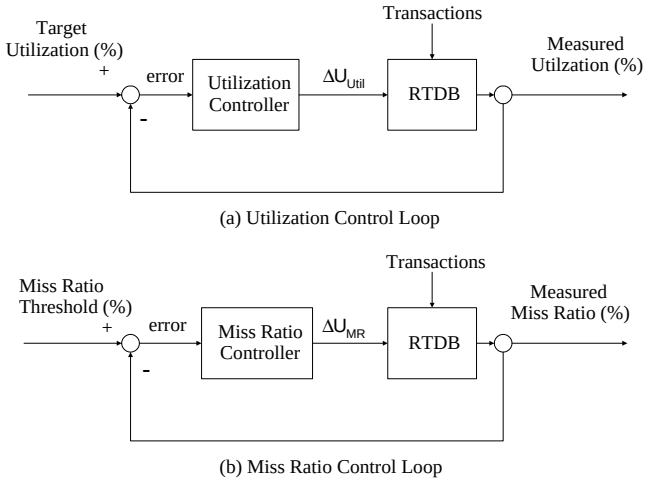


Figure 6: Miss Ratio/Utilization Controllers

5.3.1 Miss Ratio Controller

To support the required miss ratio, the miss ratio controller computes the required CPU utilization adjustment, called ΔU_{MR}, based on the miss ratio error as shown in Figure 6. More specifically, at the k^{th} sampling instant the miss ratio control signal ΔU_{MR} is computed in a digital PI (proportional and integral) controller:[5]

$$\Delta U_{MR} = KP \times Error_k + KI \times \sum_{i=1}^{k} Error_i \tag{10}$$

where $Error_k$ = miss ratio threshold $-$ current miss ratio. KP and KI are constants, called control gains [23], which directly affect the performance of a PI controller. (A detailed description of controller tuning, which determines KP and KI to support the required average/transient miss ratio, is given in Section 5.3.4).

5.3.2 Utilization Controller

A utilization control loop is employed to prevent 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 underutilization. At each sampling instant, the utilization controller computes the utilization control signal ΔU_{util} based on the utilization error as shown in Figure 6. The utilization control loop uses a separate digital PI controller

[5]We use a PI controller, since a P controller alone can not cancel the steady state error [23]. 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 because our PI controllers meet *QoS-Spec*.

to compute ΔU_{util}, similar to Eq. 10, where $Error_k$ = target utilization − current utilization at the k^{th} sampling instant. 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 [19].

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 [23]. For this purpose, the integrator antiwindup technique [23] 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 utilization controller's integrator. We further extend the utilization controller using the utilization threshold manager discussed as follows.

5.3.3 Utilization Threshold Manager

For many complex real-time systems, the schedulable utilization bound is unknown or can be very pessimistic [19]. 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 pessimistic utilization threshold can cause an unnecessary underutilization. In contrast, an excessively optimistic utilization threshold may incur a large miss ratio overshoot. It is a hard problem to decide a proper utilization threshold in complex real-time systems such as real-time databases.

In QMF, we use an online approach, in which the utilization threshold (the target utilization in Figure 6 (a)) is dynamically adjusted considering the current real-time system behavior as follows. Initially, the utilization threshold is set to a relatively low value, i.e., $U_{init} = 80\%$.[6] 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, this approach can effectively prevent a potential miss ratio overshoot due to a relatively slow back-off. We have also considered an alternative approach, in which the utilization threshold is incrementally decreased when the miss ratio controller takes control. That approach improved the average utilization slightly, but increased the miss ratio overshoot. Because we aim to support consistent QoS guarantees in terms of both average and transient metrics, we use the relatively conservative back-off scheme in this paper.

[6] This U_{init} is selected since no deadline miss is observed up to 80% workload in the profiling for controller tuning.

Note that our approach is computationally lightweight and self-adaptive requiring no *a priori* knowledge about a specific workload model. It can closely approximate the potentially time-varying utilization threshold.

5.3.4 Profiling and Controller Tuning

To support the average/transient miss ratio as specified in *QoS-Spec*, we tuned the feedback controllers. To tune the miss ratio controller, the miss ratio gain, $G_M = Max\{\frac{Miss\ Ratio\ Increase}{Unit\ Load\ Increase}\}$, should be derived under the worst case set-up [19]. To derive G_M, we profiled the performance of the controlled system, i.e., a real-time database. We measured the average miss ratio for loads increasing from 60% to 200% by 10%. 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 this profiling, we derived $G_M = 1.395$ when the load increases from 110% to 120%.

Frequent sampling could improve the transient performance such as overshoot and settling time [23]. However, too frequent sampling could cause a sudden QoD degradation in our approach, especially when overloaded. To support graceful QoD degradation, if necessary, we selected the longest sampling period among several tested periods, which can support the required overshoot and settling time. (To do this, we measured the overshoot and settling time in Matlab while increasing the sampling period by 1sec starting from 1sec.)

Using G_M and the 5sec sampling period (for both miss ratio and utilization controllers), we applied the Root Locus design method in Matlab [23] to tune KP and KI to satisfy the average/transient miss ratio specified in *QoS-Spec*.[7] By using the mathematically well established Root Locus method, we can avoid ad hoc and computationally expensive testing/tuning iterations. We have selected the closed loop poles at $p_0, p_1 = 0.773, 0.56$. The feedback control system is stable, since the closed loop poles are inside the unit circle. In the Root Locus design method, KP and KI are determined according to the selected (closed loop) poles [23]. The corresponding KP = 0.284 and KI = 0.176.

6 Performance Evaluation

For performance evaluation, we have developed a real-time database simulator, which 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 required miss ratio and freshness (described in *QoS-Spec*) even in the presence of a

[7] The utilization controller has been tuned in a similar manner. However, we do not include the utilization controller tuning process due to space limitations.

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 1: 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, the current real-world status is usually monitored by periodic updates, e.g., sensor readings and stock price trends, in real-time databases. 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 1, most representative ones are presented. (The other stock items not presented in Table 1 showed similar inter-trade times.) As shown in Table 1, the shortest average inter-trade time observed is 189ms for $\texttt{Item}_1$, while the longest one observed is 26sec for $\texttt{Item}_6$.[8]

From this study, we determined the range of sensor update periods for our simulation as shown in Table 2. 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 1. 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. We also surveyed the sampling periods for physical sensors widely used in medical and industrial applications [17, 33]. We confirmed that our range of update periods can closely model their sampling periods.

As shown in Table 2, 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

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

Table 2: Simulation Settings for Data and 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	$\approx 50\%$

(EET_i) and an update period (P_i) where $1 \leq i \leq 1000$. EET_i is uniformly distributed in a range (1ms, 8ms). 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 3: Simulation 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} (#Average Data Accesses)	$EET_i \times Data\ Access\ Factor = (5, 20)$
#Actual Data Accesses	$Normal(N_{DATA_i}, \sqrt{N_{DATA_i}})$
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 3. We selected this range of execution time to model a high performance main memory database. For example, a main memory database system, called TimesTen [32], 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) can process up to 200 transactions per second, i.e., 5ms per transaction execution. Further, we considered potentially more complex transactions requiring longer execution time.

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 $AET_i = (1 + EstErr) \times EET_i$, 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* $\times EET_i = (5, 20)$. 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 purposes, we have also applied other settings for execution time, data access factor, and slack factor different from the settings given in Table 3. We have confirmed that for different workload settings QMF can also support *QoS-Spec* by dynamically adjusting the system behavior based on the current performance error measured in feedback control loops (whose controllers are tuned using the Root Locus method [23]). However, we do not include the results here due to space limitations.

6.2 Baselines

To our best knowledge, no previous research has applied feedback/admission control and QoD adaptation to provide guarantees on miss ratio and freshness against unpredictable workloads and access patterns. For this reason, we have developed several baseline approaches 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-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. Note that most of database systems take open-loop and non-adaptive approaches as Open-IMU and Open-ODU do.

- *Open-IMU-AC*: This is a variant of Open-IMU in which admission control is applied.
- *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.
- *Theoretical Oracle*: For performance evaluation, we assume the existence of a *theoretical oracle* that has the complete future knowledge of data accesses. The clairvoyant oracle can schedule updates only if necessary. An update is scheduled only if some transaction *will* access the corresponding data before the next update. In this way, the update workload can be minimized, while providing the perfect perceived freshness. To implement it, we divided an experiment into two passes. (Each pass is a complete simulation run.) In the first pass, which may include unnecessary updates, the data access history of user transactions is recorded in a form of *(time, accessed data)*. In the second pass, the same simulation set-up and seed number are applied to generate the same set of transactions and data accesses. The oracle utilizes the access history collected in the first pass to *predict* the future. (Admission control is not applied to reproduce the exactly same workload in the second pass. If admission control is applied, a user transaction might be rejected in the first pass, but could be admitted in the second pass due to the reduced update workload. As a result, the corresponding data accesses are unknown *a priori*).

6.3 Workload Variables

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 $\approx (1+EstErr)\times$ 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 data contention changes [1, 9]. For this reason, we apply different access patterns by using the $x-y$ access scheme [9], 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). Usually, a high data access skew, i.e., a small hot spot size, is considered to incur a high degree of data contention [1, 9].

Table 4: 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%

- *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. We applied *Fixed-QoD* ranging from 0.5 to 1 increased by 0.1 to observe whether or not the required 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%$\times$ (*Fixed-QoD* + (1 $-$ *Fixed-QoD*)/4). (According to *QoS-Spec*, $P_{i_{new}} = 4 \times P_{i_{min}}$ for every $O_i \in D_{degr}$ after the full QoD degradation.) In Table 4, 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 required 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.)

6.4 Experiments

Table 5: Presented Sets of Experiments

Exp. Set	Varied	Fixed
1	$AppLoad = 70\%, 100\%, 150\%, 200\%$	EstErr = 0 HSS = 50% (uniform access)
2	$EstErr = 0, 0.25, 0.5, 0.75, 1$	AppLoad = 200% HSS = 50%
3	*Fixed-QoD* = 0.5 – 1.0	AppLoad = 200% EstErr = 1 HSS = 50%
4	$HSS = 10\%, 20\%, 30\%, 40\%, 50\%$	AppLoad = 200% EstErr = 1 Fixed-QoD = 1 (QMF-2)

Even though we have performed a large number of experiments for varying values of the workload variables, we only present the four most representative sets of experiments as summarized in Table 5 due to space limitations. We have verified that all the experiments, including Experiment Sets $1-4$ presented here, show a consistent performance trend: our approach can support the required *QoS-Spec*. In contrast, the open-loop baseline approaches (i.e., all baselines except the clairvoyant oracle) fail to support the specified miss ratio and/or freshness in the presence of unpredictable workloads and access patterns. We applied QMF-1 to Experiment Sets 1 and 2, QMF-2 to Experiment Set 3, and both of them to Experiment Set 4.[9] In this paper, we present our performance results in a stepwise manner. We first compare the performance of QMF 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-1 and QMF-2 for increasing $EstErr$ and *Fixed-QoD* in Experiment Sets 2 and 3, respectively. Finally, in Experiment Set 4 we compare QMF-1 and QMF-2 to the theoretical oracle for various access patterns.

- **Experiment Set 1**: As described in Table 5, no error is considered in the execution time estimation, i.e.,

[9] We also applied QMF-2 to Experiment Sets 1 and 2, and verified QMF-2 can support *QoS-Spec*.

$EstErr = 0$. 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%.

- **Experiment Set 2**: In this set of experiments, we increase $EstErr$ from 0 to 1 by 0.25 as shown in Table 5. We set $AppLoad = 200\%$.

- **Experiment Set 3**: In this set of experiments, we set $AppLoad = 200\%$ and $EstErr = 1$, i.e., the highest $AppLoad$ and $EstErr$ values tested in our experiments. Further, we increase *Fixed-QoD* from 0.5 to 1 by 0.1 to stress the modeled real-time database.

- **Experiment Set 4**: As shown in Table 5, different data access skews are applied to this set of experiments. *HSS* is varied between $10\% - 50\%$ (uniform access pattern). We set $AppLoad = 200\%$, $EstErr = 1$, and *Fixed-QoD* = 1 (for QMF-2).

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.)

6.5 Experiment Set 1: Effects of Increasing Load

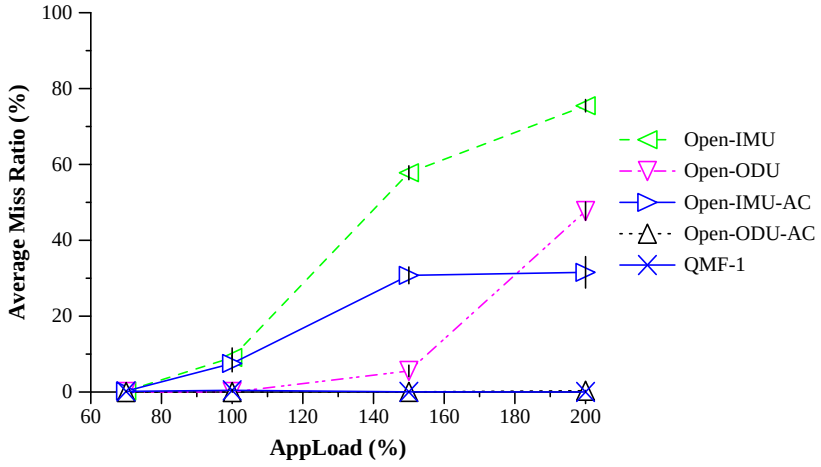


Figure 7: Average Miss Ratio

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

6.5.1 Average Miss Ratio

As shown in Figure 7, Open-IMU shows the highest average miss ratio exceeding 75% when $AppLoad = 200\%$. By applying admission control, Open-IMU-AC significantly improves the miss ratio, but the miss ratio reaches $31.55 \pm 4.14\%$ 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 reaches $47.7 \pm 2.39\%$ 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 $EstErr = 0$ leads to an effective admission control. However, in the following subsection we show that Open-ODU-AC fails to support the target freshness especially when *AppLoad* is high, whereas our approach can support both miss ratio and freshness requirements.

6.5.2 Perceived Freshness

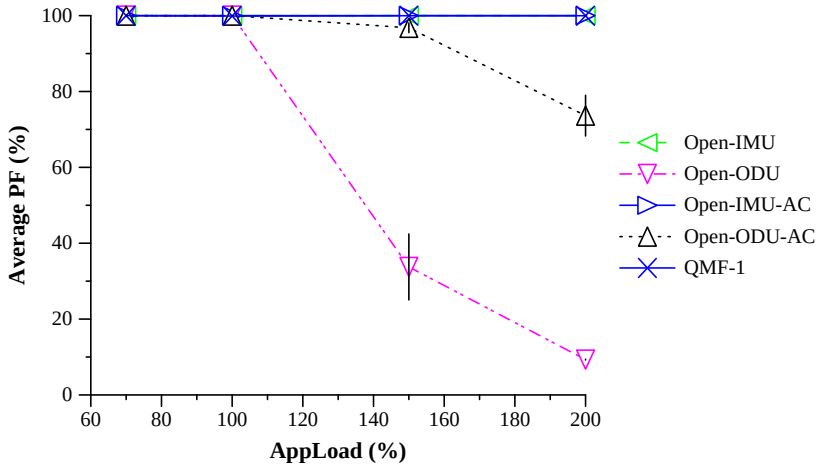


Figure 8: Average Perceived Freshness

In Figure 8, Open-IMU and Open-IMU-AC show a 100% perceived freshness. This is because all sensor data are updated immediately when their new sensor readings arrive. QMF-1 shows a near 100% perceived freshness.

The lowest perceived freshness measured was $99.97 \pm 0.03\%$ when $AppLoad = 200\%$.[10] (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 drops below 10% when $AppLoad = 200\%$. This is because all sensor data are updated on demand regardless of the current perceived freshness. To verify this, we observed the database freshness for Open-ODU, which was only $8.85 \pm 0.08\%$ when $AppLoad = 200\%$. As a result, many user transactions had to read stale data to meet their deadlines.

As shown in Figure 8, 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 less number of on-demand updates miss their deadlines. However, Open-ODU-AC also violates the required 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.[11]

6.5.3 Average Utilization

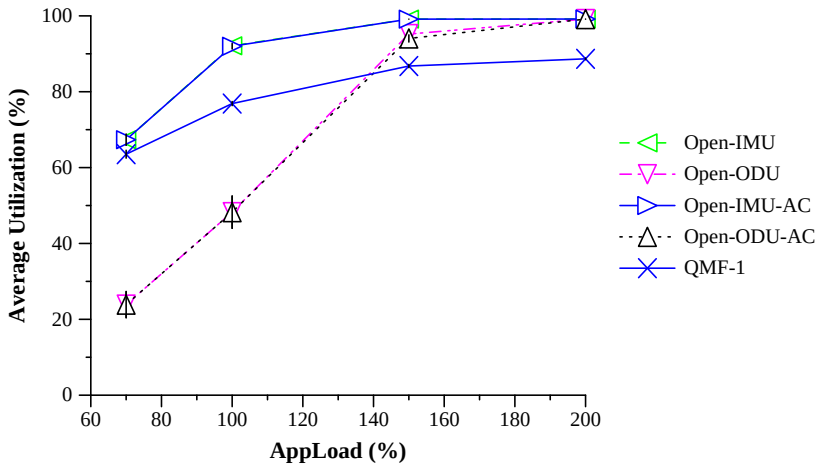


Figure 9: Average Utilization

In Figure 9, the measured average 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 for $AppLoad =$

[10] Recall in this set of experiments (for QMF-1) we assume it is acceptable to use stale data because the application can extrapolate.

[11] For Open-ODU and Open-ODU-AC, we also tested an alternative approach, in which a stale data access is not allowed even if the accessing user transaction misses its deadline. In this approach, Open-ODU and Open-ODU-AC showed a substantial miss ratio increase mainly due to the delay for on-demand updates.

70%, 100%. This is because updates in Open-ODU and Open-ODU-AC are scheduled purely on demand despite the possible CPU underutilization or freshness violation. In contrast, QMF-1 avoids both underutilization and overload as shown in Figure 9. The utilization ranges between 64% – 89% supporting the required miss ratio and freshness. This is because QMF-1 dynamically adapts the CPU utilization considering the current system status measured in the feedback control loops.

6.6 Experiment Set 2: Effects of Increasing Execution Time Estimation Error

In this section, we measure the performance of Open-IMU-AC, Open-ODU-AC, and QMF-1 for increasing $EstErr$. Open-IMU and Open-ODU are dropped due to their relatively poor performance as discussed in the previous section.

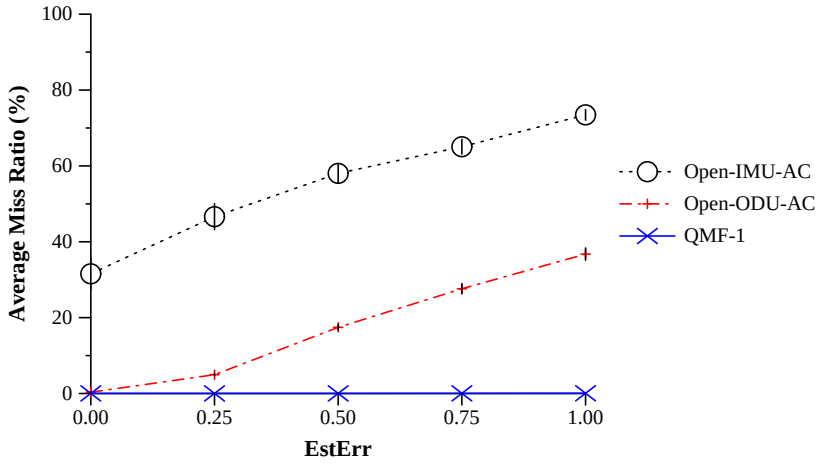


Figure 10: Average Miss Ratio

6.6.1 Average Performance

As shown in Figure 10, the average miss ratio of Open-IMU-AC increases as $EstErr$ increases. It exceeds 70% when $EstErr = 1$. Open-ODU-AC, which showed a near zero miss ratio in Experiment Set 1, shows the miss ratio over 27% when $EstErr = 1$. Hence, both Open-IMU-AC and Open-ODU-AC significantly violate the required average miss ratio 1%. As $EstErr$ increases, Open-IMU-AC and Open-ODU-AC may admit too many user transactions. Consequently, many of the admitted transactions miss their deadlines. In contrast, the miss ratio of QMF-1 does not increase despite the increasing $EstErr$. Even though Open-IMU-AC, Open-ODU-AC, and QMF apply the same admission control policy, QMF applies admission control (and QoD management) to

enforce the feedback control signal ΔU, which is computed considering the current system status. In this way, QMF achieves the required miss ratio unlike the open-loop baseline approaches.

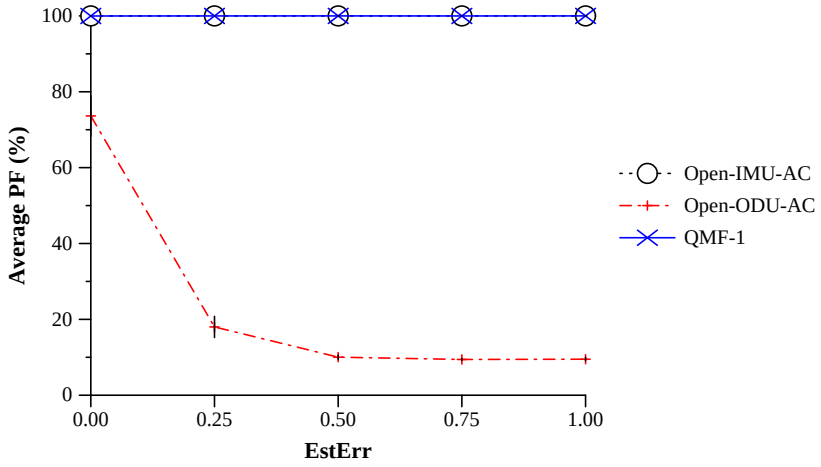


Figure 11: Average Perceived Freshness

In Figure 11, Open-IMU-AC shows a 100% perceived freshness due to immediate updates. QMF-1 shows a near 100% freshness; the lowest perceived freshness is $99.98 \pm 0.02\%$ when $EstErr = 0.75$. In contrast, Open-ODU-AC shows a $9.49 \pm 0.44\%$ perceived freshness when $EstErr = 1$ due to lazy (on-demand) updates. In Open-ODU-AC, the freshness drops as $EstErr$ increases, since admission control may include more errors resulting in many deadline misses of on-demand updates.

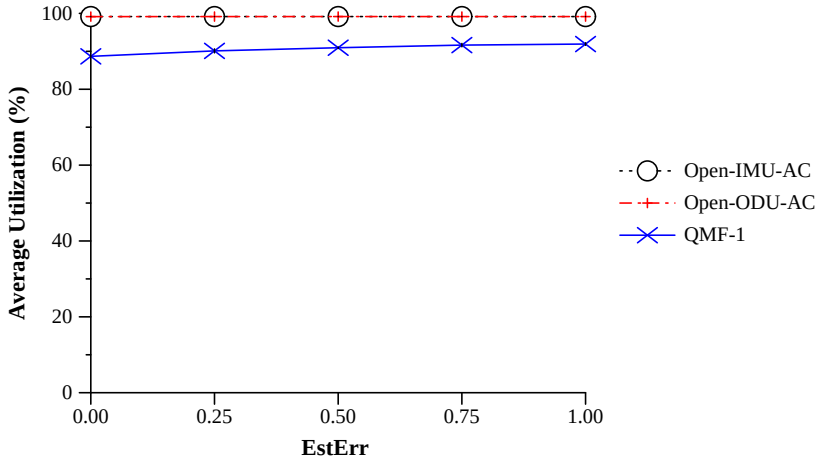


Figure 12: Average Utilization

Figure 12 plots the average CPU utilization. For Open-IMU-AC and Open-ODU-AC, the utilization is near 100% incurring many deadline misses. In QMF-1, the average utilization ranges between $88 - 92\%$. QMF-1 avoids potential overload, while exceeding the specified 80% utilization by dynamically adapting the utilization

threshold considering the current miss ratio (as discussed in Section 5.3.3).

6.6.2 Transient Performance of QMF-1

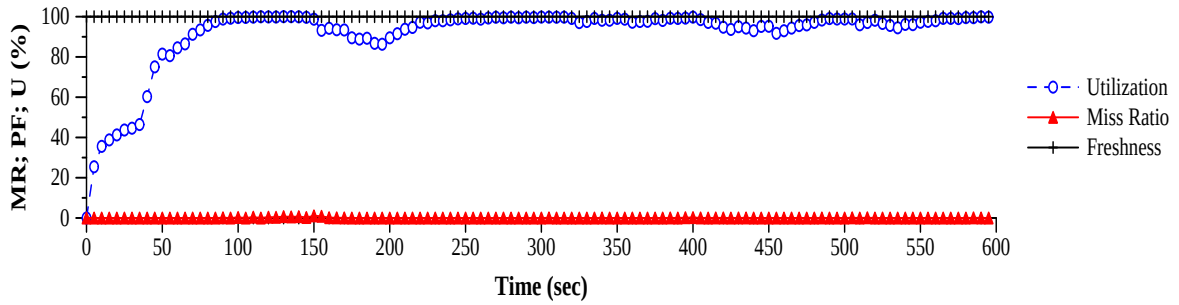


Figure 13: Transient Performance of QMF-1 (EstErr = 1, AppLoad = 200%)

In Figure 13, the transient miss ratio, perceived freshness, and utilization are measured for QMF-1 when $EstErr = 1$, i.e., the highest execution time estimation error.[12] As shown in Figure 13, the transient miss ratio is near zero without any miss ratio overshoot. Therefore, the required settling time is automatically satisfied. Also, the perceived freshness is 100% throughout the experiment.

From Experiment Set 2, observe that QMF-1 can support the required QoS even given a high *AppLoad* $(= 200\%)$ and large execution time estimation errors by applying the feedback/admission control and adaptive update policy. In contrast, the baseline approaches violated even the average miss ratio and/or freshness as discussed before.

6.7 Experiment Set 3: 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 not support the required freshness as discussed before.)

6.7.1 Average Performance

As shown in Figure 14, 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

[12] We also verified that QMF-1 meets the required QoS for lower $EstErr$ values. Due to space limitations, we only present the transient performance for the worst $EstErr$ (= 1) among the tested values.

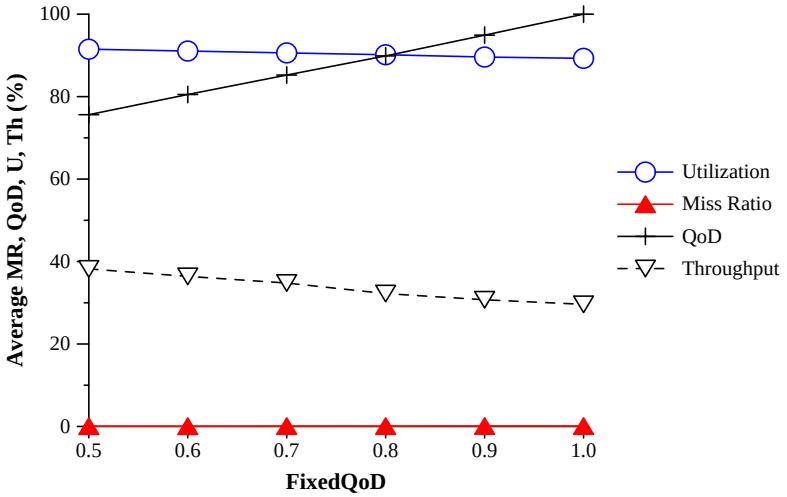


Figure 14: Average Performance of QMF-2

miss ratio is near zero.

As *Fixed-QoD* increases, a relatively less number of user transactions can be admitted/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:

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

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 perfect 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 14, the throughput of QMF-2 decreases from $38.21 \pm 0.84\%$ to $29.62 \pm 1.38\%$ as *Fixed-QoD* increases from 0.5 to 1. 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. The throughput of QMF-2 exceeds the ideal 33% when *Fixed-QoD* ≤ 0.7 due to the reduced but bounded QoD as required in

QoS-Spec.[13]

The utilization drops from approximately 91% to 89% when *Fixed-QoD* increases from 0.5 to 1. Notice that the overall utilization dropping 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 22.04 ± 0.87% throughput given $AppLoad = 200$% and $EstErr = 1$ (applied to Experiment Set 3). When *Fixed-QoD* = 1, QMF-2's throughput is approximately 7% higher than Open-IMU-AC's ($\approx 29.62 - 22.04$%), while providing the 100% QoD. When *Fixed-QoD* = 0.5, the corresponding improvement of throughput is more than 16% ($\approx 38.21\% - 22.04\%$). Given $AppLoad = 200$% and $EstErr = 1$, 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. (As presented in Section 6.6, Open-IMU and Open-IMU-AC missed over 70% and 27% of transaction deadlines when $AppLoad = 200$% and $EstErr = 1$.)

From these results, we observe it is sensible to prevent potential overload 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 required.

6.7.2 Transient Performance

Figures 15 and 16 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 15 and 16, QMF-2 does not exceed the 1% miss ratio threshold without any miss ratio overshoot throughout the experiments. In Figure 15, the QoD is decreasing to avoid a potential miss ratio overshoot given $AppLoad = 200$%.

As shown in Figure 16, 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 perfect QoD as discussed before.) Therefore, using QMF-2 a corporate user, e.g., a financial trading/factory automation company, can select an

[13] In our experiments, QMF-1's throughput was higher than QMF-2's by approximately $10 - 15$%. By updating a subset of cold data on demand instead of incrementally increasing their update periods, QMF-1 can further reduce the update workload at the cost of potential stale data accesses.

appropriate QoD considering the application specific data semantics without a miss ratio overshoot or severe utilization/throughput decrease.

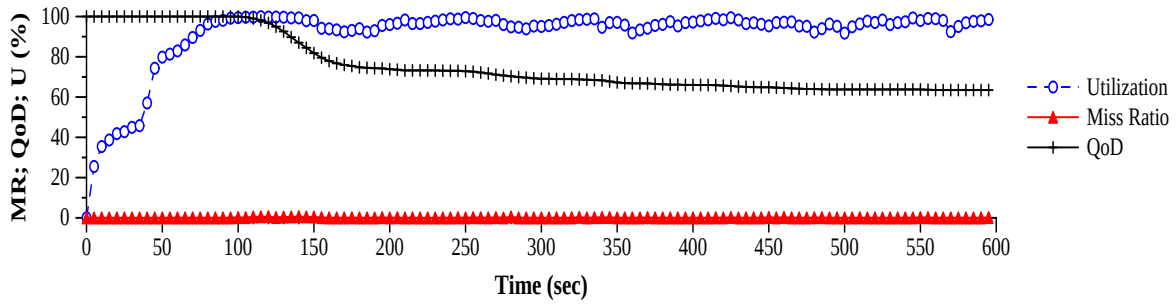


Figure 15: Transient Performance of QMF-2 (Fixed-QoD = 0.5)

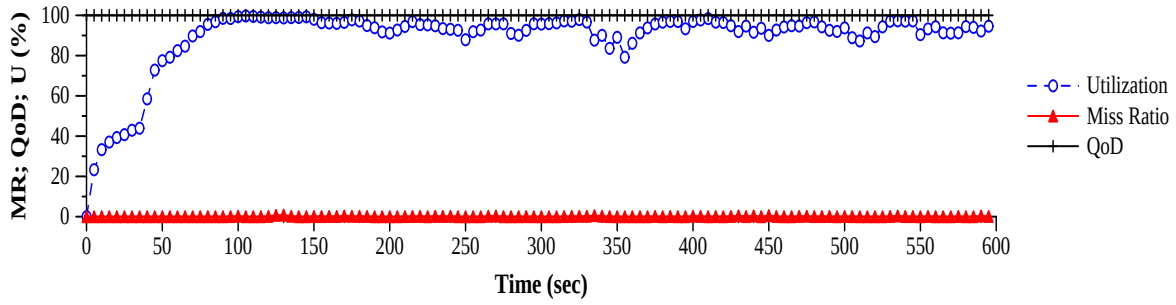


Figure 16: Transient Performance of QMF-2 (Fixed-QoD = 1)

In Experiment Sets 1 – 3, both QMF-1 and QMF-2 were able to support the required average/transient miss ratio, while providing the required *PF* or QoD. 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 required database QoS.

6.8 Experiment Set 4: Effects of Various Access Patterns

In this section, we compare the performance of QMF-1, QMF-2, and clairvoyant oracle for different access patterns. The performance results of other baselines are not included due to their relatively poor performance.

6.8.1 Average Performance

As shown in Figures 17 – 19, QMF-1, QMF-2, and the oracle achieve the near zero average miss ratio. As shown in Figure 17, QMF-1 achieves a near 100% perceived freshness. The lowest perceived freshness is $99.95 \pm 0.07\%$ when $HSS = 30\%$. In QMF-2, we set *Fixed-QoD* = 1 for Experiment Set 4. Therefore, the QoD = 100% as shown in Figure 18. Using the complete future knowledge of data accesses, the oracle can support the 100% freshness

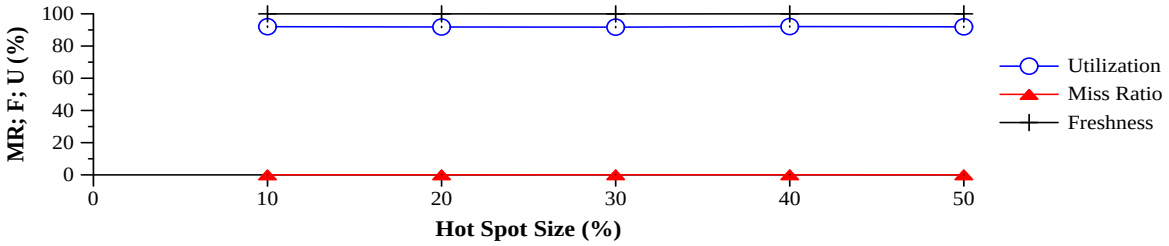


Figure 17: Average Performance of QMF-1

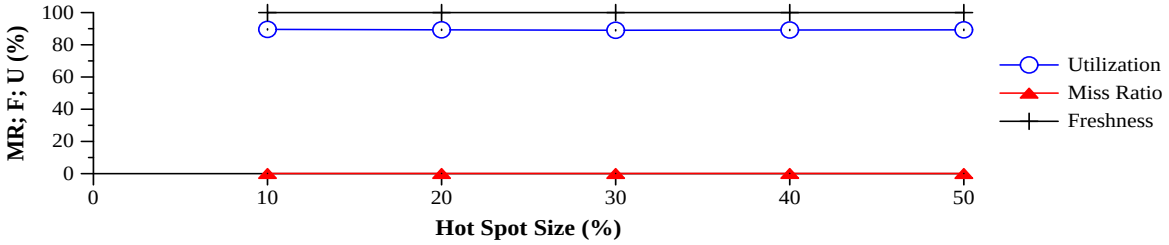


Figure 18: Average Performance of QMF-2

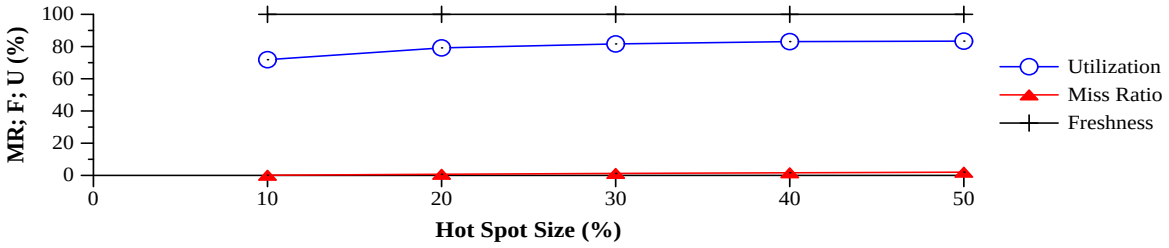


Figure 19: Average Performance of Oracle

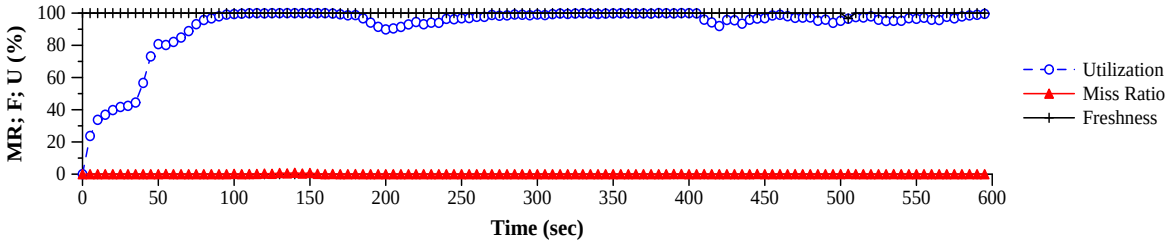


Figure 20: Transient Performance of QMF-1 (EstErr = 1, AppLoad = 200%, HSS = 10%)

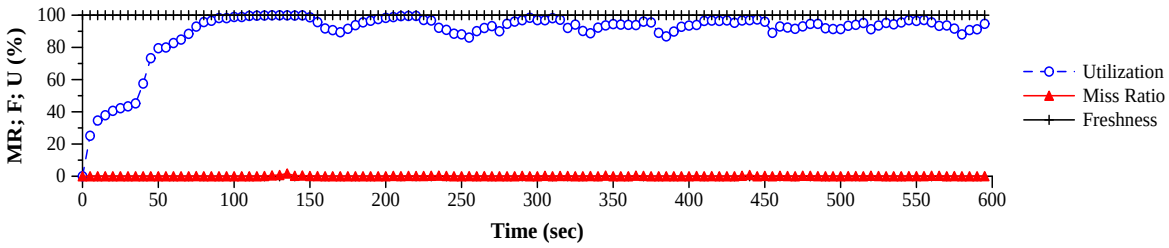


Figure 21: Transient Performance of QMF-2 (EstErr = 1, AppLoad = 200%, HSS = 10%)

as shown in Figure 19.

As shown in Figures 17 and 18, QMF-1 and QMF-2 achieve an approximately 90% utilization. As shown in Figure 19, the oracle shows the relatively low utilization compared to QMF-1 and QMF-2, since it only has to update a particular set of data which will be accessed before their next updates.[14] For increasing *HSS*, the oracle's utilization increases from approximately 71% to 83%. As *HSS* increases, i.e., sensor data are accessed in an increasingly uniform manner, the oracle has to update more data. As a result, the utilization increases.

6.8.2 Transient Performance

In Figures 20 and 21, the transient performance of QMF-1 and QMF-2 are plotted for $HSS = 10\%$, i.e., the smallest hot spot size among the tested *HSS* values, which may incur the highest degree of data contention. Both QMF-1 and QMF-2 show the near zero miss ratio, without any miss ratio overshoot, meeting *QoS-Spec*. The PF, QoD = 100% throughout the experiments. (We have also verified that QMF-1 and QMF-2 support *QoS-Spec* for other hot spot sizes.)

In summary, the performance of QMF (in terms of miss ratio and freshness) is similar to the clairvoyant oracle, which can support the optimal performance. These performance results substantiate the applicability of QMF to manage quality of real-time data services. Using QMF, users can specify various QoS requirements considering specific application semantics. Our approach supports the required QoS guarantees even in the presence of a wide range of unpredictable workloads and data access patterns.

7 Related Work

Previous research work has shown that QoS-aware approaches can improve the system performance in a cost-effective manner [2, 5, 7, 13, 20]. Despite the abundance of QoS related work and database research separately, database QoS guarantee issues have rarely been studied, and this can be a serious problem. For example, approximately $420 million revenue loss was reported due to late, i.e., non-real-time, e-commerce transaction processing in 1999 [27]. For these reasons, database self-tuning or auto-configuration is beginning to receive more attention [6, 28, 36]. A database self-tuning project, called AutoAdmin [6, 36], is going on at Microsoft Research to reduce the cost of manual database tuning, which is needed to support the required database performance for specific applications. Their work currently focuses on physical database design, e.g., identifying indexes and materialized

[14] In terms of throughput, the oracle is not directly comparable to QMF-1 and QMF-2, since we applied $AppLoad = 100\%$ to regenerate the same workload in pass 2 without admission control.

views appropriate for an application specific workload to optimize the performance of database systems. IBM DB2 Universal Database [28] aims to support database auto-configuration focusing on physical database design. However, these approaches [6, 28, 36] do not consider database QoS guarantee issues in terms of both timeliness and freshness as we do.

Trade-off issues between timeliness and data freshness have been studied in [3, 4, 12]. Stanford Real-Time Information Processor (STRIP) addressed the problem of balancing between the possibly conflicting freshness and timing constraints in real-time databases [3]. To study the trade-off issues between freshness and timeliness, several scheduling algorithms were introduced to schedule updates and transactions. In their later work, a similar trade-off problem was studied for derived data [4]. In [12], trade-off issues between response time and data freshness are considered in the context of the web server. Dynamically generated data are materialized at the web server and continuously refreshed by the back-end database. Response time can be improved if more views are materialized; however, data freshness can be reduced, and vice versa. Given a certain number of views to materialize, they presented an adaptive view selection algorithm for materialization to achieve the near optimal response time and data freshness. None of the work presented in [3, 4, 12] provided the performance guarantee in terms of either miss ratio or data freshness.

Various aspects of the real-time database performance other than data freshness can be traded off to improve the miss ratio. In [21] and [34], the correctness of answers to database queries can be traded off to enhance timeliness by using the database sampling and milestone approach [16], respectively. In these approaches, the accuracy of the result can be improved as the sampling/computation progresses, while returning approximate answers to the queries, if necessary, to meet their deadlines. In replicated databases, consistency can be traded off to reduce the response time. Epsilon serializability [25] allows a query processing despite the concurrent updates, while bounding the deviation between the query and answer. An adaptable security manager is proposed in [29], in which the database security level can be temporarily degraded to enhance timeliness. Priority Adaptation Query Resource Scheduling (PAQRS) provided timeliness differentiation of query processing in a memory-constrained environment [22]. By an intelligent memory management, they can differentiate the miss ratio between service classes. Note that none of the work presented in [21, 22, 25, 29, 34] support guarantees for both miss ratio and data freshness.

Feedback control has been applied to QoS management and real-time scheduling [7, 15, 19, 30] due to its robustness against unpredictable operating environments [23]. However, none of them considered performance guarantee issues regarding both timing and data freshness constraints in real-time databases. Feedback control

has also been recognized as a viable approach to manage the database performance [6, 35, 36]. Even though these work borrow the high level notion of feedback control (i.e., performance observation and dynamic adaptation of the database system behavior in a conceptual feedback loop), they do not apply mathematically well established control theory to manage the database performance. Also, they do not consider timing and freshness guarantee issues in real-time databases.

8 Conclusions

The demand for real-time data services is increasing due to many important applications such as e-commerce, agile manufacturing, and voice/data network management. It is essential but challenging to process real-time transactions within their deadlines using fresh data, which reflect the current real-world status. The key contributions of this paper are (1) defining QoS metrics (in terms of miss ratio and data freshness including novel notions of perceived freshness, QoD, and flexible validity intervals) to specify the required quality of real-time data services, (2) introducing flexible QoD management schemes, (3) presenting a new QoS management architecture that can support the required QoS even in the presence of unpredictable workloads and access patterns, and (4) performing extensive experiments, designed to consider freshness semantics of real-time data, to compare the performance of our approach to the baseline approaches. Notably, QMF showed a near optimal performance, which is comparable to the clairvoyant oracle. Using QMF, users can specify the required miss ratio and data freshness considering specific application semantics without worrying about potential miss ratio overshoots or data freshness violations.

As one of the first approaches to provide guaranteed real-time data services, the significance of our work will increase as the demand for (and importance of) real-time data services increases. We will further investigate the timeliness and freshness issues in real-time databases. Currently, we are investigating a differentiated real-time data service architecture to provide preferred services to important service classes under overload. We also plan to research other important 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] T. F. Abdelzaher and K. G. Shin. QoS Provisioning with qContracts in Web and Multimedia Services. In *Real-Time Systems Symposium*, Phoenix, Arizona, December 1999.

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

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

[5] N. Bhatti and R. Friedrich. Web Server Support for Tiered Services. *IEEE Network*, 13(5):64–71, September 1999.

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

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

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

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

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

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

[12] A. Labrinidis and N. Roussopoulos. Adaptive WebView Materialization. In *the Fourth International Workshop on the Web and Databases, held in conjunction with ACM SIGMOD*, May 2001.

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

[14] 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.

[15] 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.

[16] 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.

[17] City Technology Ltd. http://www.citytech.com/.

[18] C. Lu, J. Stankovic, T. Abdelzaher, G. Tao, S. H. Son, and M. Marley. Performance Specifications and Metrics for Adaptive Real-Time Systems. In *Real-Time Systems Symposium*, Orlando, Florida, November 2000.

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

[20] K. Nahrstedt and S. Narayan H.-H. Chu. QoS-aware Resource Management for Distributed Multimedia Applications. *Journal on High-Speed Networking, Special Issue on Multimedia Networking, IOS Press*, 8(3-4):227–255, 1998.

[21] 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.

[22] H. Pang, M. Carey, and M. Livny. Multiclass Query Scheduling in Real-Time Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):533–551, August 1995.

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

[24] Polyhedra Plc. Polyhedra White Papers, 2002.

[25] C. Pu and A. Leff. Replica Control in Distributed Systems: As Asynchronous Approach. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1991.

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

[27] ActivMedia Research. Real Numbers behind 'Net Profits. http://www.activmediaresearch.com/.

[28] B. Schiefer and G. Valentin. DB2 Universal Database Performance Tuning. *IEEE Data Engineering Bulletin*, 22(2):12–19, 1999.

[29] 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.

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

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

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

[33] Trition sensor sampling. http://www.jamstec.go.jp/jamstec/TRITON/future/pdf/Table5.3.pdf.

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

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

[36] 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.

[37] 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.