Robust Quality Management for Differentiated Imprecise Data Services*

Mehdi Amirijoo, Jörgen Hansson
Dept. of Computer and Information Science
Linköping University, Sweden
{meham,jorha}@ida.liu.se

Sang H. Son
Dept. of Computer Science
University of Virginia, Charlottesville, Virginia, USA
son@cs.virginia.edu

Svante Gunnarsson
Dept. of Electrical Engineering
Linköping University, Sweden
svante@isy.liu.se

Abstract

Several applications, such as web services and e-commerce, are operating in open environments where the workload characteristics, such as the load applied on the system and the worst-case execution times, are inaccurate or even not known in advance. This implies that transactions submitted to a real-time database cannot be subject to exact schedulability analysis given the lack of a priori knowledge of the workload. In this paper we propose an approach, based on feedback control, for managing the quality of service of real-time databases that provide imprecise and differentiated services, given inaccurate workload characteristics. For each service class, the database operator specifies the quality of service requirements by explicitly declaring the precision requirements of the data and the results of the transactions. The performance evaluation shows that our approach provides reliable quality of service even in the face of varying load and inaccurate execution time estimates.

1. Introduction

Lately the demand for real-time data services, provided by real-time databases (RTDBs), has increased and applications used in, e.g. manufacturing, web servers, and e-commerce, are becoming increasingly sophisticated in their data needs. In these applications it is desirable to process user requests within their deadlines using fresh data. In dynamic systems, such as web servers and sensor networks with non-uniform access patterns, the workload of the databases cannot be precisely predicted and, hence, the databases can become overloaded. As a result, deadline misses and freshness violations may occur during the transient overloads. To address this problem we propose a quality of service (QoS) sensitive approach, based on imprecise computation [13], to guarantee a set of requirements on the behavior of the database, even in the presence of unpredictable workloads.

In this paper we employ the notion of imprecise computation [13] on transactions as well as data, i.e., we allow data objects to deviate, to a certain degree, from their corresponding values in the external environment. However, only using imprecise computations will not by itself solve the problems caused by transient overload, as there is an upper limit of the amount of resources that can be traded off for QoS. Instead of attempting to provide service to all the workload submitted to the RTDB we may service only a subset, representing the most important parts, of that workload. Previous work in service differentiation in real-time systems [3] and RTDBs [10, 8] focus on the importance or criticality of the transactions. Transactions are usually classified into classes with regard to their importance and it is assumed that the more important classes receive the best QoS. We consider the importance of a class and the QoS that the class requires to be disjoint and, hence, importance and QoS are two orthogonal entities. This is in contrast to less general approaches, e.g. value-driven scheduling [3], where deadline miss ratio of important transactions is lower than less important transactions. For example, consider an embedded vehicle control application [7] where there is a set of tasks with different importance. The fuel ignition task

* This work was funded, in part by CUGS (the National Graduate School in Computer Science, Sweden), CENIIT (Center for Industrial Information Technology) under contract 01.07, NSF grants CCR-0098269 and IIS-0208758, and ISIS (Information Systems for Industrial Control and Supervision).

1052-8725/04 $20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

is very important whereas engine monitoring tasks are considered to be less important. However, the fuel ignition task may require less precise data as compared to the engine monitoring tasks which may require very precise data. Consequently, the QoS demand of the monitoring tasks is higher although they are less important compared to the ignition task.

In this paper we present an architecture to manage QoS, defined in terms of data precision and transaction precision, to support service differentiation of multiple classes. To the best of our knowledge this is the first paper describing performance management of multiple classes in real-time databases that support imprecise computation at transactions and data object level. As the first contribution we present a QoS specification model supporting orthogonality between importance and QoS. The expressive power of the QoS specification model allows a database operator[1] or database designer to specify not only the desired nominal system performance, but also the the worst-case system performance and system adaptability in the face of unexpected failures or load variation. The second contribution is an architecture and two algorithms, based on feedback control scheduling [16, 14], for managing the QoS as given by the QoS specification. The performance studies show that the suggested algorithms give a robust performance of RTDBs, in terms of transaction and data precision, even for transient overloads and with inaccurate execution time estimates of the transactions. We achieve resource isolation among the different classes, and we show that during overloads transactions are rejected in a strictly hierarchical fashion based on importance. Finally, the experimental results show that our approach supports orthogonality between class importance and class QoS needs.

The rest of this paper is organized as follows. A problem formulation is given in section 2. In section 3, the assumed database model is given. In section 4, we present an approach for QoS management and in section 5, the results of performance evaluations are presented. In section 6, we give an overview on related work, followed by section 7, where conclusions and future work are discussed.

2. Problem Formulation

In our database model, data objects in an RTDB are updated by update transactions, e.g. sensor values, while user transactions represent user requests, e.g. complex read-write operations. We apply the notion of imprecision at data object and user transaction level. Increasing the resources allocated to the update transactions results in greater quality of data (QoD) as the imprecision of the data objects decreases. Similarly, increasing the resources for user transactions results in greater quality of user transactions, for brevity referred to as quality of transaction (QoT), as the imprecision of the results produced by user transactions decreases. Intuitively, sufficiently precise data values stored in the database are regarded to have no effect on the result of a transaction. If temporal consistency constraints are satisfied then such imprecision is admissible.

Let $SVC = \{svc^1, \dots, svc^c, \dots, svc^{|SVC|}\}$ denote the set of service classes and $|SVC|$ denote the number of service classes. User transactions are classified into service classes based on their importance, where the first level svc^1 holds the most important or critical transactions, the second level svc^2 holds the less important transactions and so on. We introduce the notion of transaction error (denoted te_i), inherited from the imprecise computation model [13], to measure the quality of a transaction T_i. Here, the quality of the result given by a transaction depends on the processing time allocated to the transaction. The transaction returns more precise results, i.e. lower te_i, as it receives more processing time. Further, for a data object stored in the RTDB and representing a real-world variable, we can allow a certain degree of deviation compared to the real-world value. If such deviation can be tolerated, arriving updates may be discarded during transient overloads. To measure data quality we introduce the notion of data error (denoted de_i), which gives an indication of how much the value of a data object d_i stored in the RTDB deviates from the corresponding real-world value, which is given by the latest arrived transaction updating d_i. Note that the latest arrived transaction updating d_i may have been discarded and, hence, d_i may hold the value of an earlier update transaction.

For a service class svc^c we can then specify the desired QoS of svc^c in terms of te_i that the transactions in svc^c produce and the data error of the data objects that transactions in svc^c read. We observe that a data object may be accessed by several transactions in different service classes and, hence, there may be different precision requirements put upon the data item. It is clear that we need to ensure that the data error of the data object complies with the needs of all transactions and, consequently, any data error conflicts must be resolved by satisfying the needs of the transaction with the stronger requirement. If the access patterns of the transactions are known in advance we may use that information to keep the data objects precise. However, in dynamic systems with unpredictable access patterns we must rather predict the access patterns of the transactions during run-time and adapt the precision of the data objects such that the transactions accessing them have precise readings.

Finally, it is important that the te_i of terminated transactions in the same service class does not vary significantly from one transaction to another. Here it is emphasized that large deviations between te_i must be minimized to ensure

1 By a database operator we mean an agent, human or computer, that supervises and operates the database, including setting the QoS.

IEEE COMPUTER SOCIETY

QoS fairness. In summary the goal of this work is to: (i) establish a model for expressing QoS requirements in terms of data error and transaction error for each service class, (ii) develop an architecture and a set of algorithms for managing data error and transaction error such that given QoS specifications for each service level are satisfied, and (iii) minimize the variation of transaction error among admitted transactions, i.e., maximize QoS fairness.

3. Data and Transaction Model

We consider a main memory database model, where there is one CPU as the main processing element. We consider the following data and transaction models. In our data model, data objects can be classified into two classes, temporal and non-temporal [17]. For temporal data we only consider base data, i.e., data objects that hold the view of the real-world and are updated by sensors. A base data object d_i is considered temporally inconsistent or stale if the current time is later than the timestamp of d_i followed by the absolute validity interval avi_i of d_i, i.e. $currenttime > timestamp_i + avi_i$. For a data object d_i, let data error $de_i = \Phi(cv_i, v_j)$ be a non-negative function of the current value cv_i of d_i and the value v_j of the latest arrived transaction that updated d_i or that was to update d_i but was discarded. The function Φ may for example be defined as the absolute deviation between cv_i and v_j, i.e., $de_i = |cv_i - v_j|$, or the relative deviation as given by $de_i = \frac{|cv_i - v_j|}{|cv_i|}$. To capture the QoD demands of the different service classes we model the data error as perceived by the transactions in svc^c with $de_i \times def^c$ where def^c denotes the data error factor of the transactions in svc^c. The greater def^c is, the greater does the transactions in svc^c perceive the data error. We define the weighted data error as $wde_i = de_i \times def_{d,i}$, where $def_{d,i}$ is the maximum data error factor of the transactions accessing d_i.

Update transactions arrive periodically and may only write to base data objects. User transactions arrive aperiodically and may read temporal and read/write non-temporal data. User and update transactions (T_i) are composed of one mandatory subtransaction m_i and $|O_i| \geq 0$ optional subtransactions $o_{i,j}$, where $o_{i,j}$ is the jth optional subtransaction of T_i. For the remainder of the paper, we let $t_{i,j}$ denote the jth subtransaction of T_i. Since updates do not use complex logical or numerical operations, we assume that each update transaction consists only of a single mandatory subtransaction, i.e., $|O_i| = 0$. We use the milestone approach [13] to transaction imprecision. Thus, we divide transactions into subtransactions according to milestones. A mandatory subtransaction is completed when it is completed in a traditional sense. The mandatory subtransaction gives an acceptable result and must be computed to completion before the transaction deadline. The optional subtransactions may be processed if there is enough time or resources available. While it is assumed that all subtransactions of a transaction T_i arrive at the same time, the first optional subtransaction (if any) $o_{i,1}$ becomes ready for execution when the mandatory subtransaction, m_i, is completed. In general, an optional subtransaction, $o_{i,j}$, becomes ready for execution when $o_{i,j-1}$ (where $2 \leq j \leq |O_i|$) completes. We set the deadline of every subtransaction $t_{i,j}$ to the deadline of the transaction T_i. A subtransaction is terminated if it has completed or has missed its deadline. A transaction T_i is terminated when $o_{i,|O_i|}$ completes or one of its subtransactions misses its deadline. In the latter case, all subtransactions that are not completed are terminated as well.

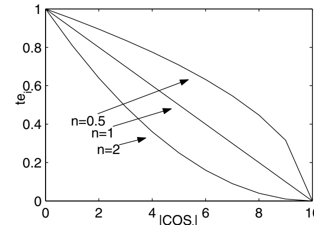

Figure 1. Contribution of $|COS_i|$ to te_i.

For a user transaction T_i, we use an error function [5] to approximate its corresponding transaction error given by $te_i(|COS_i|) = \left(1 - \frac{|COS_i|}{|O_i|}\right)^{n_i}$ where n_i is the order of the error function and $|COS_i|$ denotes the number of completed optional subtransactions. By choosing n_i we can model and support multiple types of transactions showing different error characteristics (see Figure 1).

4. Approach

Below we describe an approach for managing the performance of an RTDB in terms of transaction and data quality. First, we start by defining QoS and how it can be specified. An overview of the feedback control scheduling architecture is given, followed by issues related to modeling of the architecture and design of controllers. We refer to the presented approach as Robust Quality Management of Differentiated Imprecise Data Services (RDS).

4.1. Performance Metrics and QoS specification

We apply the following steady-state and transient state performance metrics [14] to each service class. $Terminated^c(k)$ denotes the set of terminated transactions in service class svc^c during the interval $[(k-1)T, kT]$, where T is the sampling period. For the rest of this paper, we sometimes drop k where the notion of time is not important.

1052-8725/04 $20.00 © 2004 **IEEE**

IEEE COMPUTER SOCIETY

- The average transaction error of admitted user transactions,

$$ate^c(k) = 100 \times \frac{\sum_{i \in Terminated^c(k)} te_i}{|Terminated^c(k)|}(\%)$$

gives the precision of the results produced by user transactions.

- The data precision requirement of user transactions is given using def^c.
- Data precision is manipulated by managing the data error of the data objects, which is done by considering an upper bound for the weighted data error given by the maximum weighted data error $mwde$. An update transaction T_j is discarded if the weighted data error of the data object d_i to be updated by T_j is less or equal to $mwde$ (i.e. $wde_i \leq mwde$). If $mwde$ increases, more update transactions are discarded, degrading the quality of data. Setting $mwde$ to zero results in the highest data precision, while setting $mwde$ to one results in lowest data precision allowed.
- Overshoot M_p^c is the worst-case system performance in the transient system state (see Figure 2) and it is given in percentage. Overshoot is applied to ate^c.
- Settling time T_s^c is the time for the transient overshoot to decay and reach the steady state performance (see Figure 2), hence, it is a measure of system adaptability, i.e., how fast the system converges towards the desired performance. Settling time is applied to ate^c.
- To measure QoS fairness among admitted transactions, we introduce the standard deviation of transaction error,

$$sdte^c(k) = \sqrt{\frac{\sum_{i \in Terminated^c(k)} (100 \times te_i - ate^c(k))^2}{|Terminated^c(k)| - 1}}.$$

which gives a measure of how much the transaction error of terminated transactions deviates from the average transaction error.

- Admission Percentage, $ap^c = \frac{|Admitted^c(k)|}{|Submitted^c(k)|}(\%)$, where $|Admitted^c(k)|$ is the number of admitted transactions and $|Submitted^c(k)|$ is the number of submitted transactions in svc^c.

We define QoD in terms of $mwde$ and an increase in QoD refers to a decrease in $mwde$, while a decrease in QoD refers to an increase in $mwde$. Similarly, we define QoT for a service class svc^c in terms of ate^c. QoT for a service class svc^c increases as ate^c decreases, while QoT decreases as ate^c increases. The QoS specification is given in terms of def^c and a set of target levels in the steady-state

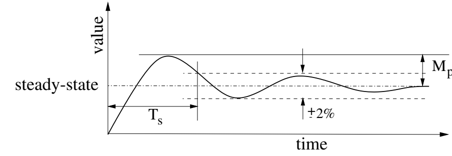

Figure 2. Definition of settling time (T_s) and overshoot (M_p)

or references ate_r^c for ate^c. Turning to QoD requirement specification, for service class svc^c we want that $de_i \leq \xi$, where ξ is an arbitrary data error. Assume that several transactions, including those in svc^c, with different precision requirements access d_i. Then it must hold that $def_{d,i} \geq def^c$, since $def_{d,i}$ is the maximum data error factor of the transactions accessing d_i. d_i is least precise when $mwde$ is equal to one and, hence, $de_i \times def^c \leq de_i \times def_{d,i} = wde_i \leq 1$. From this we conclude that $de_i \leq \frac{1}{def^c}$. So by setting def^c to $\frac{1}{\xi}$ we satisfy the QoD requirement of the transactions in svc^c. The following example shows a specification of QoS requirements: $\{ate_r^1 = 30\%, def^1 = 1, T_s^1 \leq 60s, M_p^1 \leq 35\%\}, \{ate_r^2 = 20\%, def^2 = 3, T_s^2 \leq 60s, M_p^2 \leq 35\%\}, \{ate_r^3 = 40\%, def^3 = 0.3, T_s^3 \leq 60s, M_p^3 \leq 35\%\}, \{ate_r^4 = 50\%, def^4 = 0.1, T_s^4 \leq 60s, M_p^4 \leq 35\%\}$. Note that the transactions in svc^1 are more important than the transactions in svc^2, however, the QoS requirement for svc^1 is less than the QoS requirement for svc^2, i.e., $ate^1 > ate^2$ and $def^1 < def^2$. This shows the orthogonality of importance and QoS requirements.

4.2. QoD Classes

We need to make sure to meet the data precision requirements of the transactions in all service levels. An initial approach would be to simply block a transaction accessing a data object that does not satisfy the precision requirement. However, this may lead to many deadline misses and, hence, decreased performance. To lower the blocking time we must rather classify the data objects according to the precision requirements of the transactions accessing them, where each class of data objects represents a data precision requirement. The data classification must be adaptive since the access patterns of the transactions may change during run-time. A sporadic transaction with high precision requirement may access a data object only once during the entire operation of the RTDB. Keeping the precision of that data object at a high level may result in a waste of resources, since few or no update transactions are discarded.

Conceptually, when a user transaction with a very high data precision requirement accesses a data object, we classify the data object to a higher QoD class representing

IEEE COMPUTER SOCIETY

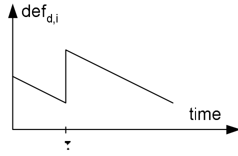

Figure 3. QoD classification of the data object d_i

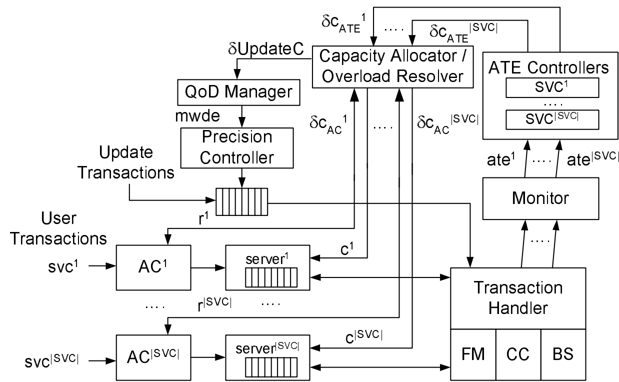

Figure 4. QoS management architecture using feedback control

greater precision. If after a while, no transaction with equal or greater precision requirement accesses that data object, then we move the data object to a lower QoD class, representing a lower precision requirement. Once a transaction T_i accesses a data object d_i where the data error factor def^c of T_i is greater than the current data error factor $def_{d,i}$ of d_i, then we set $def_{d,i}$ to def^c. Hence, d_i is raised to the QoD class representing def^c. The data error factor $def_{d,i}$ then decreases linearly over time, moving to lower QoD classes until a transaction with a higher data error factor accesses d_i, as shown in Figure 3. At time τ a transaction in svc^c accesses d_i with def^c greater than $def_{d,i}$ and, hence, $def_{d,i}$ is set to def^c in order to adapt to the new precision requirement. After time τ no more transactions with higher precision requirements than $def_{d,i}$ access d_i and therefore the precision requirement of d_i is relaxed by lowering $def_{d,i}$. This way the system is adaptive to changes in access patterns.

4.3. Feedback Control Scheduling Architecture

The architecture of our QoS management scheme is given in Figure 4. Update transactions have higher priority than user transactions and are upon arrival ordered in an update ready queue according to the earliest deadline first (EDF) scheduling policy (for an elaborate discussion on EDF see e.g. [3]). To provide individual QoS guarantees for each user transaction class we have to enforce isolation among the classes by bounding the execution time of the transactions in each class. This is achieved by using deferrable servers [3], which enables us to limit the resource consumption of transactions, while lower average response time is enforced for the transactions in higher service classes. Let $server^c$ denote the server for svc^c and c^c denote the capacity of $server^c$. We assign priorities to the servers according to their importance, i.e., $server^c$ has higher priority than $server^{c+1}$. Upon activation $server^c$ serves any pending user transactions in its ready queue within the limit of c^c or until no more user transactions are waiting, at which point $server^c$ becomes suspended and $server^{c+1}$ becomes active. Note, $server^c$ is reactivated if new user transactions arrive and c^c is greater than zero. The capacity is replenished with the sampling period T.

The transaction handler manages the execution of the transactions. It consists of a freshness manager (FM), a unit managing the concurrency control (CC), and a basic scheduler (BS). The FM checks the freshness before accessing a data object, using the timestamp and the absolute validity interval of the data. We employ two-phase locking with highest priority (2PL-HP) [1] for concurrency control. 2PL-HP is chosen since it is free from priority inversion and has well-known behavior. We consider two different scheduling algorithms as basic schedulers: (i) Earliest Deadline First (EDF), where transactions are processed in the order determined by their absolute deadlines, and (ii) Highest Error First (HEF) [2], where transactions are processed in the order determined by their transaction error, i.e., the next transaction to run is the one with the greatest transaction error. For both basic schedulers (EDF and HEF) the mandatory subtransactions have higher priority than the optional subtransactions and, hence, scheduled before them. We refer to RDS_{EDF} when EDF is used as a BS, correspondingly to RDS_{HEF} when HEF is used as BS.

At each sampling instant kT, the controlled variables ate^c are monitored and fed into the ATE Controllers, which compare the performance references ate^c_r with ate^c to get the current performance errors. Based on this each ATE Controller computes a requested change δc^c_{ATE} to c^c. If ate^c is higher than ate^c_r, then a positive δc^c_{ATE} is returned, requesting an increase in the capacity so that ate^c is lowered to its reference. The requested changes in capacities are given to the Capacity Allocator, which distributes the capacities according to the class level. During overloads it may not be possible to accommodate all requested capacities. Instead, the QoD is lowered, resulting in more discarded update transactions, hence, more resources can be allocated for user transactions. If the lowest data quality

1052-8725/04 $20.00 © 2004 **IEEE**

IEEE COMPUTER SOCIETY

is reached and no more update transactions can be discarded, the amount of capacity r^c that is not accommodated is returned to the admission controller, which rejects transactions with a total execution time of r^c. Now, a controller computes a change in capacity such that $ate^c(k+1)$ equals ate^c_r. This change in capacity is based on an observed $ate^c(k)$, which depends on the admitted load and $c^c(k)$. However, due to the unpredictable workload applied on the database, the admitted load may increase (for example consider the most important service class where no transactions are rejected). To suppress large overshoots we react in advance by informing the Overload Resolver to modify the capacities when the current admitted load is greater than the admitted load during the previous sampling interval. If for a service class svc^c, the execution time of the admitted transactions in the current period is greater than the previous period then an increase in capacity δc^c_{AC} equal to the difference in admitted execution time is requested for svc^c. The capacities of $svc^c, \ldots, svc^{|SVC|}$ are then recomputed if the increase in capacity can be accommodated. If δc^c_{AC} cannot be accommodated then more transactions in svc^c, corresponding to δc^c_{AC} are rejected.

We have modeled the controlled system using $\mathcal{Z}$-transform theory [6]. The transfer function of the model describing ate in terms of δc^c_{ATE} is given by $P(z) = \frac{ate(z)}{\delta c^c_{ATE}(z)} = \frac{G_c}{z-1}$, where G_c is the derivative of the function relating ate^c and c^c at the vicinity of ate^c_r. We have tuned G_c by measuring ate for different capacities and taking the slope at ate_r. The ATE Controller is implemented using a P-controller tuned with root locus [6].

4.4. Data and Transaction Error Management

4.4.1. Capacity Allocation Figure 5 shows how c^c, r^c, and $mwde$ are computed. To simplify the presentation of the algorithms we denote the execution time of update transactions with the capacity of the update transactions, although there is no server for update transactions. Let $c^{Update}(k)$ denote the capacity, or equivalently the execution time, of the update transactions. Since the arrival patterns of the update transactions are varying, we take the moving average of the update transaction capacity to smoothen out great variations (line 1). We start allocating capacities with respect to the service levels, starting with svc^1. The requested capacity $c^c_{req}(k+1)$ of service class svc^c is the sum of the previously assigned capacity $c^c(k)$ and the requested change in capacity $\delta c^c_{ATE}(k+1)$ (line 4). If the total available capacity is less than the requested capacity then we degrade QoD by calling **ChangeUpdateC** along with how much update capacity to free (lines 5-7). See Section 4.4.2 for an elaborate description on **ChangeUpdateC**. If the requested capacity still cannot be accommodated, we enforce the capacity adjustment by rejecting more transactions (lines 10-11). One key concept in the capacity allocation algorithm is that we employ an optimistic admission policy. We start by admitting all transactions and in the face of an overload we reject transactions until the overload is resolved. This contrasts against pessimistic admission policy where the admission percentage is increased as long as the system is not overloaded. We believe that the pessimistic admission policy is not suitable as some critical transactions, i.e. the transactions in higher service classes, are initially discarded. Continuing with the algorithm description, if the requested capacity can be accommodated, we try to reduce the number of rejected transactions (lines 12-21). Finally, after the capacity allocation we check to see whether there is any spare capacity c_s, and if so we upgrade QoD within the limits of c_s. If QoD cannot be further upgraded, i.e. $mwde = 0$, then we distribute c_s among the servers (lines 23-30).

4.4.2. QoD Management The precision of the data is controlled by the QoD Manager by setting $mwde(k)$ depending on the requested change in update capacity $\delta c^{Update}(k)$. Rejecting an update results in a decrease in update capacity. We define the gained capacity, $gc(k) = \sum_{T_i \in Discarded(k)} eet_i$, as the capacity gained due to the result of rejecting one or more updates during period k. $Discarded(k)$ is the set of discarded update transactions during the period $[(k-1)T, kT]$ and eet_i is the estimated execution time of the update transaction T_i. In our approach, we profile the system and measure gc for different $mwde$s and linearize the relationship between these two, i.e. $mwde = \mu \times gc$. Further, since RTDBs are dynamic systems such that the behavior of the system and environment is changing, the relation between gc and $mwde$ is adjusted on-line. This is done by measuring $gc(k)$ for a given $mwde(k)$ during each sampling period and updating μ. Having the relationship between gc and $mwde$, we introduce the function $h(\delta c^{Update}(k)) = \mu \times (gc(k) - \delta c^{Update}(k))$, which returns an $mwde$ given $\delta c^{Update}(k)$ and the linear relation between gc and $mwde$. Since $mwde$ cannot be greater than one and less than zero we use the function,

$$f(\delta c^{Update}(k)) = \begin{cases} 1, & h(\delta c^{Update}(k)) > 1 \\ h(\delta c^{Update}(k)), & 0 \le h(\delta c^{Update}(k)) \le 1 \\ 0, & h(\delta c^{Update}(k)) < 0 \end{cases}$$

to enforce this requirement. Now that we have arrived at f it is straightforward to compute $mwde$, as shown in Figure 6. The function **ChangeUpdateC** returns the estimated change in update capacity given the requested change $\delta c^{Update}(k)$.

IEEE COMPUTER SOCIETY

ComputeCapacity($\delta c^1_{ATE}(k+1), \ldots, \delta c^{|SVC|}_{ATE}(k+1)$)
1: $c^{Update}(k) \leftarrow \alpha \times c^{Update}(k) + (1-\alpha) \times c^{Update}(k-1)$
2: $c(k+1) \leftarrow c^{Update}(k)$ {the total capacity is set to the update capacity}
3: **for** $c = 1$ to $|SVC|$ **do**
4: $c^c_{req}(k+1) \leftarrow \max\left(0, c^c(k) + \delta c^c_{ATE}(k+1)\right)$ {compute the requested capacity}
5: **if** $T - c(k+1) < c^c_{req}(k+1)$ **then** {if the total available capacity if less than the requested capacity}
6: $c(k+1) \leftarrow c(k+1)+$**ChangeUpdateC**$(T - c(k+1) - c_{req}(k+1))$ {try to degrade QoD to free capacity}
7: **end if**
8: $c^c(k+1) \leftarrow \min(T - c(k+1), c^c_{req}(k+1))$ {assign capacity to $server^c$}
9: $c(k+1) \leftarrow c(k+1) + c^c(k+1)$ {update the total capacity}
10: **if** $c^c(k+1) < c^c_{req}(k+1)$ **then** {if the assigned capacity is less than the requested capacity}
11: $r^c(k+1) \leftarrow r^c(k) + c^c_{req}(k+1) - c^c(k+1)$ {reject more transactions}
12: **else if** $r^c(k) > 0$ **then** {if the requested capacity was accommodated and we rejected transactions during the previous period}
13: **if** $T - c(k+1) < r^c(k)$ **then** {if the available capacity is less than the rejected capacity in the previous period}
14: $c(k+1) \leftarrow c(k+1)+$**ChangeUpdateC**$(T - c(k+1) - r^c(k)$ {try to degrade QoD to reject as few transactions as possible}
15: **end if**
16: $r^c(k+1) \leftarrow \max(0, r^c(k) - T + c(k+1))$ {lower the transaction rejection}
17: $c^c(k+1) \leftarrow c^c(k+1) + r^c(k) - r^c(k+1)$ {allocate more capacity to accommodate decrease in transaction rejection}
18: $c(k+1) \leftarrow c(k+1) + r^c(k) - r^c(k+1)$ {update the total capacity}
19: **else** {if the requested capacity is accommodated and we did not reject transactions during the previous period}
20: $r^c(k+1) \leftarrow 0$ {do not reject transactions during the next period}
21: **end if**
22: **end for**
23: $c_s(k+1) \leftarrow T - c(k+1)$ {compute the spare capacity that is not used by the classes}
24: **if** $c_s(k+1) > 0$ **then** {if there is spare capacity}
25: $c_s(k+1) \leftarrow \max(0, c_s(k+1) -$ **ChangeUpdateC**$(c_s(k+1)))$ {try to upgrade QoD as much as possible}
26: **for** $c = 1$ to $|SVC|$ **do**
27: {if there is still spare capacity left after the QoD upgrade then give more capacity to the class servers}
28: $c^c(k+1) \leftarrow c^c(k+1) + \frac{c_s(k+1)}{|SVC|}$ {divide the spare capacity evenly between the classes}
29: **end for**
30: **end if**

Figure 5. The capacity allocation algorithm.

ChangeUpdateC($\delta c^{Update}(k)$)
$mwde(k+1) \leftarrow f(\delta c^{Update}(k))$
Return $\frac{1}{\mu}(mwde(k) - mwde(k+1))$ {return the estimated change in update transaction capacity}

Figure 6. The QoD management algorithm.

5. Performance Evaluation

5.1. Experimental Goals

The performance evaluation is undertaken by a set of simulation experiments, where a set of parameters have been varied. These are: (i) load ($load$), as computational systems may show different behaviors for different loads, especially when the system is overloaded, and (ii) execution time estimation error ($esterr$), since often exact execution time estimates of transactions are not known.

5.2. Simulation Setup

The simulated workload consists of update and user transactions, which access data and perform virtual arithmetic/logical operations on the data. In the experiments, one simulation run lasts for 10 minutes of simulated time. For all the performance data, we have taken the average of 10 simulation runs and derived 95% confidence intervals. We use the QoS specification given in Section 4.1. The workload model of the update and user transactions are described as follows. We use the following notation where the attribute x_i refers to the transaction T_i, and $x_i[t_{i,j}]$ is associated with the subtransaction $t_{i,j}$ of T_i.

Data and Update Transactions. The database holds 1000 temporal data objects (d_i) where each data object is updated by a stream ($stream_i$, $1 \leq i \leq 1000$). The period (p_i) is uniformly distributed in the range (100ms,50s), i.e. $U : (100ms, 50s)$, and estimated execution time (eet_i) is given by $U : (1ms, 5ms)$. The average update value (av_i) of each $stream_i$ is given by $U : (0, 100)$. Upon a periodic generation of an update, $stream_i$ gives the update an actual execution time given by the normal distribution $N : (eet_i, \sqrt{eet_i})$ and a value (v_i) according to $N : (av_i, \sqrt{av_i \times varfactor})$, where $varfactor$ is uniformly distributed in (0,0.5). The deadline is set to $arrivaltime_i + p_i$. We define data error as the relative deviation between cv_i and v_j as given by $de_i = 100 \times \frac{|cv_i - v_j|}{|cv_i|}(\%)$.

User Transactions. Each $source_i$ generates a transaction T_i, consisting of one mandatory subtransaction and $|O_i|$, uniformly distributed between 1 and 10, optional subtransaction(s). The estimated (average) execution time of the mandatory and the optional ($eet_i[t_{i,j}]$) subtransactions is given by $U : (1ms, 4ms)$. The estimation error $esterr$ is used to introduce execution time estimation error in the average execution time given by $aet_i[t_{i,j}] = (1 + esterr) \times eet_i[t_{i,j}]$. Further, upon generation of a transaction, $source_i$ associates an actual execution time to each subtransaction $t_{i,j}$, given by $N : (aet_i[t_{i,j}], \sqrt{aet_i[t_{i,j}]})$. The deadline is set to $arrivaltime_i + eet_i \times slackfactor$. The slack factor is uniformly distributed according to $U : (10, 20)$. In the performance evaluation we consider four service classes,

1052-8725/04 $20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

where 20%, 20%, 30%, and 30% of the workload is distributed to $svc^1, \dots, svc^4$, respectively.

5.3. Baseline

To the best of our knowledge, there has been no earlier work on techniques for managing data imprecision and transaction imprecision, satisfying QoS or QoD requirements for differentiated services. For this reason we compare RDS_{EDF} and RDS_{HEF} with a baseline, called Admit-All, where all transactions are admitted and scheduled with EDF, and no QoD management is performed. This way we can study the impact of the workload on the system, e.g., how ate is affected by increasing workload.

5.4. Experiment 1: Results of Varying Load

We measure ate, ap, $mwde$, and $sdte$ and apply loads from 40% to 700%. The execution time estimation error is set to zero (i.e. $esterr = 0$). Figure 7 shows ate, ap, and $mwde$. The dashed lines indicate references, while the dashed-dotted lines give the overshoot. The confidence intervals for ap and ate are less than $\pm 11.21\%$, the confidence interval for $mwde$ is less than $\pm 2.07\%$, and the confidence interval for $sdte$ is less than $\pm 6.07\%$.

As we can see from Figure 7(a), ate of the service classes increases with increasing load. Admit-All violates the QoS specification as ate^c is greater than $ate^c_r \times (100 + M_p)$, while RDS_{EDF} and RDS_{HEF} produce ate reaching the references and, hence, satisfying the QoS specification. As the load increases the admission percentage of the lowest service class svc^4 decreases and more transactions are rejected. This means that when ap^4 becomes zero then ate^4 is equal to zero as we always measure the average transaction error over admitted transactions. From Figure 7(a) we see that ate^4 starts decreasing at loads equal to 180%, but starts increasing again at 240% load. We have tuned the ATE Controllers for loads around 100%; the effect of this is a delay until ap^4 reaches zero as the load increases. As the capacity assigned to the server in svc^4 is zero for loads greater than 240% and ap^4 does not decrease to zero instantaneously, then average ate^4 becomes greater than zero. This is further shown in Section 5.6 where we discuss the transient performance of the algorithms.

Turning to admission percentage in Figure 7(b), we observe the strict hierarchic admission policy where the lowest service class suffers at the expense of the higher service classes. ap^4 starts decreasing at 140% load and when all transactions in svc^4 are rejected, ap^3 starts decreasing. Note that the graphs show the average ap over the entire run and we cannot have zero ap as we start with 100% admission percentage at the start of the run. If we would take the average of ap^4 at steady-state then we would observe a zero

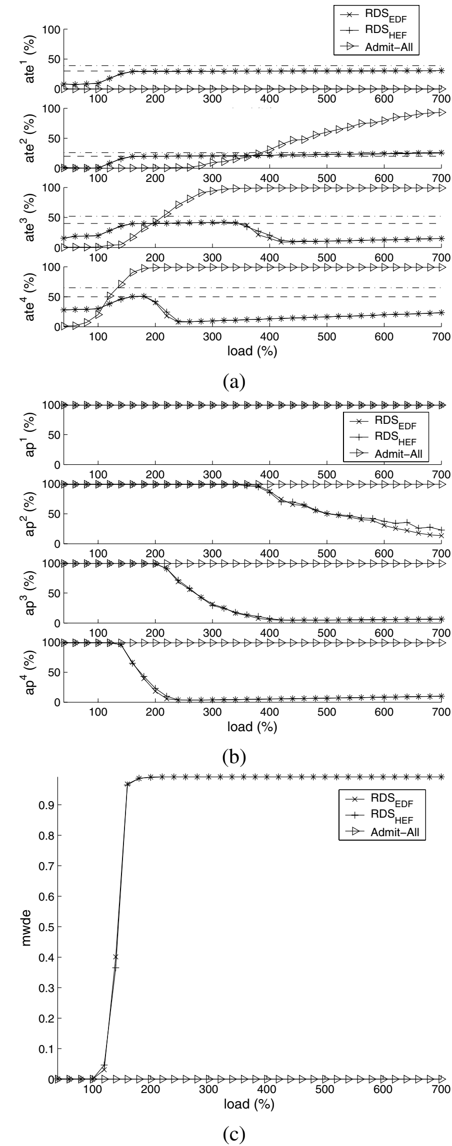

Figure 7. Experiment 1: Varying load

average ap^4 for loads greater than 240%. Turning to Figure 7(c) we see that $mwde$ starts increasing at 100% load, trying to lower the update load so that no user transactions are rejected. Studying $sdte$ we note that the difference between RDS_{HEF} and RDS_{EDF} is not significant. However in all cases RDS_{HEF} provides a lower $sdte$ than RDS_{EDF}, with the maximum difference of 6.73% observed.

From the QoS specification (see Section 4.1) we see that the transactions in svc^1 are more important than the transactions in svc^2, however, the QoS requirement for svc^1 is less than the QoS requirement for svc^2, i.e., $ate^1 > ate^2$. Figure 7 shows that ate^1 is greater than ate^2 (i.e. the QoS requirement of svc^1 is lower than the QoS requirement of svc^2)

1052-8725/04 $20.00 © 2004 **IEEE**
IEEE COMPUTER SOCIETY

and at the same time the admission percentage of svc^1 is greater or equal to the admission percentage of svc^2 (i.e. transactions in svc^2 are rejected in favor of the transactions in svc^1). This result clearly shows that RDS supports orthogonality between importance and QoS requirement. In summary we have shown that RDS_{HEF} and RDS_{EDF} provide robust and reliable performance that is consistent with the QoS specification for varying load, as ate of all classes has been less than the specified overshoot. The admission mechanism enforces the desired strict hierarchic admission policy, and RDS_{HEF} provides more QoS fairness than RDS_{EDF}. The latter is consistent with observations from the experiments where service differentiation was not used [2].

5.5. Experiment 2: Results of Varying esterr

In Experiment 1 (Section 5.4) we examined the behavior of the algorithms for varying load and we assumed that we had no execution time estimation error. In reality exact execution times are not available and for this reason we evaluate the behavior of the algorithms when increasing the execution time estimation error. We measure ate, ap, and $mwde$ and apply 300% load. The execution time estimation error $esterr$ is varied between -0.4 and 3 with steps of 0.2. This way we examine the effects of both overestimation and underestimation of the execution time. Figure 8 shows ate, ap, and $mwde$. The confidence interval for ap is less than 5.86%, the confidence interval for ate is less than $\pm 8.60\%$, and the confidence interval for $mwde$ is $\pm 0.00\%$. Ideally, the performance of the RTDB should not be affected by execution time estimation errors. This corresponds to no or little variations in ate, ap, and $mwde$ as $esterr$ changes. As shown in Figure 8, ate, ap, and $mwde$ do not change significantly with varying $esterr$. From above we conclude that RDS_{HEF} and RDS_{EDF} are insensitive to changes to execution time estimation error as ate, ap, and $mwde$ do not change significantly with varying $esterr$. This means that RDS conforms to inaccurate execution times, satisfying the QoS specification.

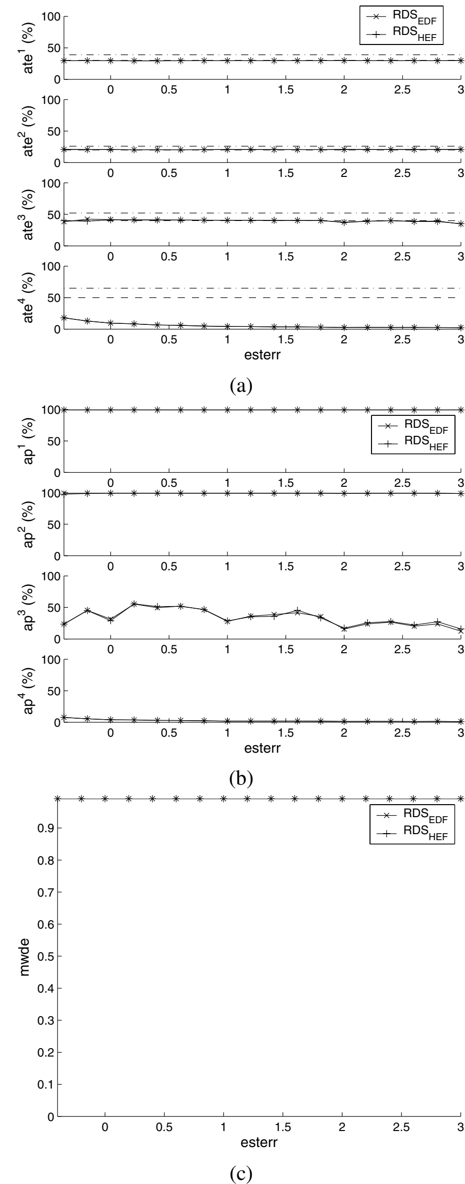

Figure 8. Experiment 2: Varying execution time estimation error

5.6. Experiment 3: Transient Performance

Studying the average performance is often not enough when dealing with dynamic systems. Therefore we study the transient performance of RDS_{EDF}. We do not include the performance of RDS_{HEF} as it has similar behavior to RDS_{EDF}. We measure ate, ap, and c. The load is set to 200% and $esterr$ set to zero. Figure 9 shows the transient behavior of RDS_{EDF}. The dash-dotted line indicates overshoot, whereas the dashed lines represent references. The overshoot for $ate^1, \ldots, ate^3$ are measured to be 2.18% (ate^1 equal to 30.65% at time 20s), 33.24% (ate^2 equal to 26.65% at time 15s), and 38.21% (ate^3 equal to 55.28% at time 10s), while the overshoot requirement by the QoS specification is 35% for all classes. Hence, the overshoot requirement for the most important service classes svc^1 and svc^2 are satisfied. The overshoot for svc^4 is undefined, since the steady-state value of ate^4 is zero.

As we can see, ate^4 reaches 100% and it takes a while until ate^4 becomes zero. Using feedback control we are able to react to changes only when the controlled variable has changed and, hence, when ate^4 is greater than ate^4_r, the ATE Controller for svc^4 computes a positive δc^4_{ATE},

IEEE COMPUTER SOCIETY

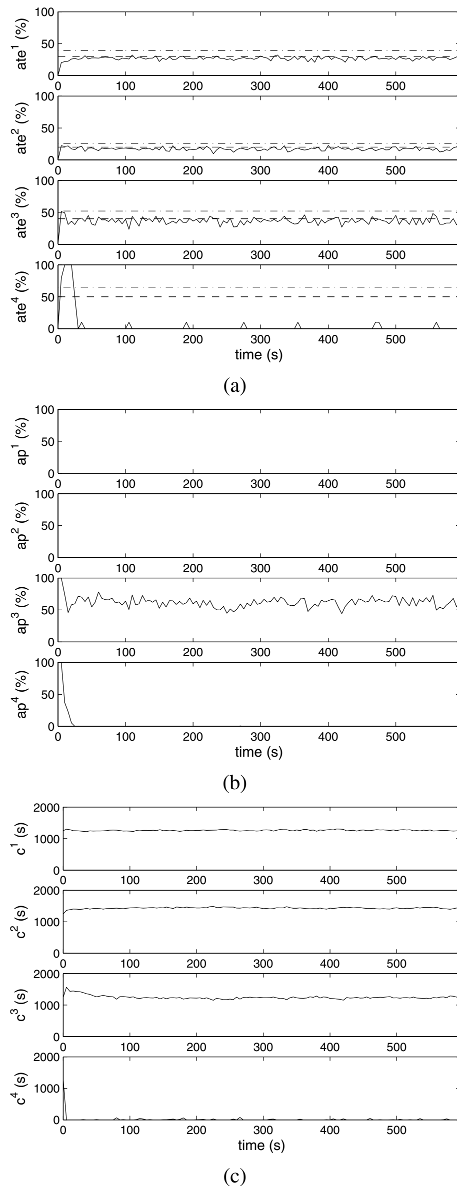

Figure 9. Experiment 3: Transient performance

requesting for more capacity. As the assigned capacity of $server^4$ is less than the requested capacity (c^4 is near zero), transactions are rejected instead according to the capacity allocation algorithm (see Figure 5). The rejection rate increases as δc^4_{ATE} increases and, hence, a larger δc^4_{ATE} results in improved suppression of ate^4 and faster convergence to zero. The magnitude of δc^4_{ATE} increases as the magnitude of P-controller parameter increases. Now, we have tuned the controllers when the load is 100% and considering the applied load in the experiment is 200%, the magnitude the P-controller parameter is not sufficiently large to efficiently suppress ate^4.

In other experiments, not presented in this paper, we have observed a significant improvement in ate^4 suppression and faster QoS adaptation when the applied load is equal to the load at which the controllers were tuned. By increasing the magnitude of the P-controller parameter as the applied load increases, better QoS adaptation is achieved. One way to deal with changing system properties, e.g. the load applied on the system, is to use gain scheduling or adaptive control [18], where the behavior of the controlled system is monitored at run-time and controllers adapted accordingly. In our case, the RTDB reacts to the higher applied load by increasing the magnitude of the P-controller parameter such that faster QoS adaptation is achieved. We believe that using gain scheduling, where the parameter of the P-controllers changes according to the current applied load, results in a substantial performance gain with respect to faster QoS adaptation. In our future work we plan to use gain scheduling to update the control parameters.

6. Related Work

Liu et al. [13] and Hansson et al. [9] presented algorithms for minimizing the total error and total weighted error of a set of tasks. The latter cannot be applied to our problem, since we want to control a set of performance metrics such that they converge towards a set of references given by a QoS specification. A query processor, APPROXIMATE [19], produces monotonically improving answers as the allocated computation time increases. The relational database system, called CASE-DB, can produce approximate answers to queries within certain deadlines [15]. Lee et al. studied the performance of real-time transaction processing where updates can be skipped [12]. In contrast to the above mentioned work, we have introduced imprecision at both data object and transaction level and presented QoS in terms of data and transaction imprecision. Rajkumar et al. presented a QoS model, called Q-RAM, for applications that must satisfy requirements along multiple dimensions such as timeliness and data quality [11]. However, they assume that the amount of resources an application requires is known and accurate, otherwise optimal resource allocation cannot be made. Kang et al. used a feedback control scheduling architecture to balance the load of user and update transactions for differentiated services [10] where the database operator can specify miss ratio and utilization requirements. However, in this work performance isolation between classes is not implemented and, consequently, orthogonality in class priority and class QoS requirements cannot be realized. In this paper we have extended our previous work [2] on managing QoS to support service differentiation, including a new QoS specification

model that supports orthogonality between importance and QoS requirements, and a new QoS management algorithm. Feedback control scheduling has been receiving special attention in the past few years [14, 16, 4]. However, none of them have addressed QoS management of imprecise real-time data services.

7. Conclusions and Future Work

In this paper we have argued for the increasing need of RTDBs that are able to react to changes in their environment in a timely manner. In this paper we present an approach for specifying and managing QoS, in terms of transaction and data precision, for differentiated and imprecise real-time data services. The expressive power of the QoS specification model allows a database operator or database designer to specify not only the desired nominal performance, but also the worst-case system performance and system adaptability in the face of unexpected failures or load variation. Further, the QoS specification model allows the database operator to specify the importance and the QoS requirement of the transactions independently. The presented QoS management algorithms, $\mathrm{RDS}_{\mathrm{HEF}}$ and $\mathrm{RDS}_{\mathrm{EDF}}$, give a robust and controlled behavior of RTDBs in terms of transaction and data precision, even for transient overloads and with inaccurate run-time estimates of the transactions.

In the current work all transactions in a service class have the same QoS requirement. In our future work we extend the transaction and service model such that each service class may have multiple QoS requirements. We also plan to implement gain scheduling [18] to update the parameter of the P-controllers such that the control action can be adapted depending on the system properties.

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] M. Amirijoo, J. Hansson, and S. H. Son. Error-driven QoS management in imprecise real-time databases. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2003.

[3] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1997.

[4] G. C. Buttazzo and L. Abeni. Adaptive workload managment through elastic scheduling. *Journal of Real-time Systems*, 23(1/2), July/September 2002. Special Issue on Control-Theoretical Approaches to Real-Time Computing.

[5] J. Chung and J. W. S. Liu. Algorithms for scheduling periodic jobs to minimize average error. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, 1988.

[6] G. F. Franklin, J. D. Powell, and M. Workman. *Digital Control of Dynamic Systems*. Addison-Wesley, third edition, 1998.

[7] T. Gustafsson and J. Hansson. Data management in real-time systems: a case of on-demand updates in vehicle control systems. In *Proceedings of Real-time Applications symposium (RTAS)*, 2004.

[8] J. Hansson, S. H. Son, J. A. Stankovic, and S. F. Andler. Dynamic transaction scheduling and reallocation in overloaded real-time database systems. In *Proceedings of the Conference on Real-time Computing Systems and Applications (RTCSA)*, 1998.

[9] J. Hansson, M. Thuresson, and S. H. Son. Imprecise task scheduling and overload managment using OR-ULD. In *Proceedings of the Conference in Real-Time Computing Systems and Applications (RTCSA)*, 2000.

[10] K.-D. Kang, S. H. Son, and J. A. Stankovic. Service differentiation in real-time main memory databases. In *Proceedings of the International Symposium on Object-oriented Real-time Distributed Computing*, 2002.

[11] C. Lee, J. Lehoezky, R. Rajkumar, and D. Siewiorek. On quality of service optimization with discrete QoS options. In *Proceedings of the 5th Real-Time Technology and Applications Symposium (RTAS)*, 1999.

[12] V. Lee, K. Lam, S. H. Son, and E. Chan. On transaction processing with partial validation and timestamps ordering in mobile broadcast environments. *IEEE Transactions on Computers*, 51(10):1196–1211, 2002. Special issue on Database Management Systems and Mobile Computing.

[13] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. *Proceedings of the IEEE*, 82, Jan 1994.

[14] 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*, 23(1/2), July/September 2002. Special Issue on Control-Theoretical Approaches to Real-Time Computing.

[15] G. Ozsoyoglu, S. Guruswamy, K. Du, and W.-C. Hou. Time-constrained query processing in CASE-DB. *IEEE Transactions on Knowledge and Data Engineering*, 7(6), 1995.

[16] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance managment. *Journal of Real-time Systems*, 23(1/2), July/September 2002. Special Issue on Control-Theoretical Approaches to Real-Time Computing.

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

[18] K. J. Åström and B. Wittenmark. *Adaptive Control*. Addison-Wesley, second edition, 1995.

[19] S. V. Vrbsky and J. W. S. Liu. APPROXIMATE - a query processor that produces monotonically improving approximate answers. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):1056–1068, December 1993.

1052-8725/04 $20.00 © 2004 **IEEE**

IEEE COMPUTER SOCIETY