QeDB: A Quality-Aware Embedded Real-Time Database *

Woochul Kang, Sang H. Son, and John A. Stankovic
Department of Computer Science
University of Virginia
{wk5f,son,stankovic}@cs.virginia.edu

Abstract

QeDB is a database for data-intensive real-time applications running on flash memory-based embedded systems. Currently, databases for embedded systems are best effort, providing no guarantees on its timeliness and data freshness. Moreover, the existing real-time database (RTDB) technology can not be applied to these embedded databases since they hypothesize that the main memory of a system is large enough to hold all database, which can not be true in data-intensive real-time applications. QeDB uses a novel feedback control scheme to support QoS in such embedded systems without requiring all data to reside in main memory. In particular, our approach is based on simultaneous control of both I/O and CPU resource to guarantee the desired timeliness. Unlike existing work on feedback control of RTDB performance, we actually implement and evaluate the proposed scheme in a modern embedded system. The experimental results show that our approach supports the desired timeliness of transactions while still maintaining high data freshness compared to baseline approaches.

1 Introduction

Recent advances in sensor technology and wireless connectivity have paved the way for next generation real-time applications that are highly data-driven, where data represent real-world status. For many of these applications, data from sensors are managed and processed in application-specific embedded systems with certain timing constraints. For example, control units of an automobile collect and process large volume of real-time data not only from internal sensors and devices [29], but also from external environments such as nearby cars and intelligent roads [14][1]. As another example, PDAs carried by firefighers for search-and-rescue task collect real-time sensor data from the burning building and peer firefighters; it also processes the data to check the dynamically changing status of the fire scene and alert the potential danger to firefighters in a timely fashion [2][3].

An embedded database [28] is an integral part of such applications or application infrastructures. Unlike traditional DBMSs, database functionality is delivered as part of the application (or application infrastructure) and they run with or as part of the applications in embedded systems. However, current off-the-shelf embedded databases such as Berkeley DB [7] and SQLite [8] are unaware of timing and data freshness requirements, showing poor performance in these applications. Unfortunately, applying the existing RTDB approaches to these embedded databases have problems since they do not consider the constraints and characteristics of the embedded systems. In particular, most previous RTDBs have been main-memory databases and ignore the effect of accessing data in persistent secondary storages such as high-capacity flash memories, which is de facto standard in modern embedded systems. In resource constrained embedded systems, it is not always feasible to have large enough main-memory to hold the entire data, especially when applications are data-intensive. Even with the advancement of processors, memory, and storage, these embedded systems are relatively resource-constrained due to cost, form factor, battery lifetime, and the increased size of data and applications. In main-memory database systems, requests to a RTDB incur only CPU load without I/O and the timeliness of transactions are determined only by CPU loads. In contrast, in non-main-memory database systems, the response time of transactions are determined not only by CPU load, but also by I/O latency, making previous RTDB approaches non-effective.

To address this problem, we have designed and implemented a real-time embedded database (RTEDB), called QeDB(Quality-aware Embedded Database). In our earlier work, we proposed an I/O-aware approach in guaranteeing QoS in RTEDBs [19]. However, it is based on simulation and some assumptions are too rigid to implement in embedded systems with general purpose OS such as embedded Linux. Hence, there were limitations in modeling real system behaviors and workloads. QeDB has been designed as an embedded database for embedded systems with a secondary storage such as high-capacity flash memory. It has been implemented by extending Berkeley DB [7], which is a popular open-source embedded database, to support the required QoS. To the best of our knowledge, this is the first paper on providing QoS for embedded databases with a real implementation.

The contributions of this paper are as follows,

1. real-time transaction model in RTEDBs,
2. an architecture based on feedback control to satisfy a given QoS specification using Multiple-Inputs/Multiple-Outputs (MIMO) control technique,
3. the implementation and evaluation of the proposed approach on a real embedded device with realistic workloads.

*This research work was supported by NSF CNS-0614886 and KOSEF WCU Project R33-2008-000-10110-0.

In embedded systems, the database runs on the same system as a part of the real-time application. Therefore, in QeDB, a real-time transaction is defined as a real-time task that accesses data through its RTEDB with optional transaction guarantees. For example, real-time transactions of firefighter's PDA run periodically to retrieve the up-to-date sensor data by issuing requests to underlying RTEDB and also perform computation tasks such as checking the potential dangers, and finding safe paths. Because real-time transactions consist of both data accesses and computation with the retrieved data, the changes in the runtime environment of RTEDB can affect the timeliness of both I/O and computation in transactions. For instance, as will be seen in Section 3, decreasing the size of the buffer increases not only the average I/O response time, but also the CPU load because more frequent buffer management activities, e.g., searching for least-recently-used pages, are required due to low buffer hit ratio. This close interaction of computation and I/O operations for data access implies that the eventual QoS of transactions can be guaranteed only when both I/O and computation are considered simultaneously.

In QeDB, the metrics of QoS are tardiness of transactions and freshness of data, which may pose conflicting requirements. QeDB achieves the desired QoS using a feedback control technique. Feedback control has recently been applied to manage RTDB performance in the presence of dynamic workloads [10][18]. In particular, QeDB exploits a Multiple Inputs/Multiple Outputs (MIMO) modeling and controller design technique to capture the close interactions of multiple inputs of the system (CPU load and buffer hit ratio) and the multiple system outputs (I/O tardiness and computation tardiness). The MIMO controller adjusts both CPU load and buffer hit ratio simultaneously.

The evaluation results demonstrate that MIMO control of QoS in QeDB is significantly more effective than baseline approaches. In particular, QeDB makes a better negotiation between the timeliness of transactions and the freshness of data by providing proper amount of resources in a robust and controlled manner.

The rest of the paper is organized as follows. Section 2 presents the overview of real-time transaction and data model in QeDB. Section 3 describes the QeDB architecture and feedback control system. Implementation issues are discussed in Section 4. Section 5 shows the details of the evaluation settings and presents the evaluation results. Related work is presented in Section 6 and Section 7 concludes the paper and discusses future work.

2 Overview of QeDB

2.1 System Model

QeDB targets real-time embedded systems having a high-capacity flash memory as a secondary storage. Figure 1 shows the software stack of an embedded system, which runs a real-time application with support from a RTEDB. A buffer is located in the main memory and it is a cache between the slow secondary storage and the CPU. The buffer is global, and shared among transactions to reduce the data access time. An I/O request from application(s) for a data object incurs I/O operations to flash memory only if the data object is not found in the buffer.

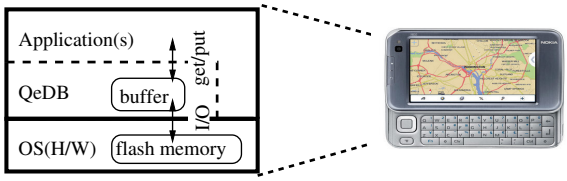


Figure 1. A real-time application with support from a RTEDB.

2.2 Data and Transactions

Unlike traditional DBMSs, QeDB does not support complex query processing on data. Instead, QeDB is a key/value store, which supports efficient and concurrent accesses to data[1]. While the interface *put(key k_1,value v)* is used for the storage of data *v* with key k_1, the interface *get(key k_2)* is used for the retrieval of data with key k_2. Operations *get* and *put* involve mostly I/O operations between the buffer and the secondary storage to fetch and flush data. However, it also involves computation such as manipulating buffer cache, looking up indexes, and locking data and index pages. In this paper, *I/O operations* refer to *put* and *get* operations in QeDB, which include not only raw I/O operations to flash memory, but also computation required for the I/O operations.

Data objects in QeDB can be classified into two classes, temporal and non-temporal data. Temporal data objects are updated periodically by *update transactions*. For example, an update transaction is issued when a new sensor reading becomes available. In contrast to update transactions, *user transactions* may read and modify both temporal and non-temporal data objects. All transactions are *canned transactions*, whose operations are pre-defined at design time of the application. Their operations are hard-coded into the applications, and invoked dynamically at runtime. The characteristics of a transaction such as execution time, and access pattern are known at the design time. However, the workload and data access pattern of the whole database is unpredictable and changes dynamically because the invocation frequency of each transaction is unknown and multiple transactions execute concurrently. Hence, their response time can be unpredictable. Transactions access data through QeDB and transactional properties such ACID (atomicity, consistency, isolation, and durability) between data accesses are provided if they are specified by the applications.

Program 1 shows an example of a transaction that is invoked periodically in a PDA of firefighters to check the structural integrity of the burning building[2]. Note that it not only

[1] QeDB uses the same data storeage and retrieval mechanism of underlying Berkeley DB without much extension.

[2] Some details are not shown for clarity and readability.

has I/O operations to get/put data, but also computation that checks the integrity of the structure. The logical consistency of the data accesses can be guaranteed by enclosing all or part of the transaction with *begin_transaction* and *commit (or abort)*. However, logical consistency is not required for all transactions and it is application-dependent.

Program 1 A transaction checking the integrity of a structure.

```
trx_check_structure_integrity()
{
   /* A list of keys to sensor data to process */
   DBT key_displacement_sensors={Sen_1,Sen_2,..., Sen_n}

   /* memory to copy in sensor data */
   DBT data_sensors[NUM_SENSORS];

   /* Some computation */
   init();

   /* perform I/Os by reading in data from the DB */
   for (i=0; i< NUM_SENSORS; i++){
       data_sensors[i]= get(key_displacement_sensors[i]);
   }
   /* computation */
   status = analyze_integrity(data_sensors);

   /* another I/O */
   put(key_status, status);
}
```

2.3 Real-Time Transactions

Transactions can be classified into *real-time transactions* and *non-real-time transactions*. Real-time transactions are real-time tasks, which have deadlines on their completion time, and they have higher priority than non-real-time transactions. For instance, the transaction in Program 1 should report the status of the structural integrity of the burning building within a specified deadline; otherwide, the firefighters may lose a chance to escape from the dangerous place that might collapse. We apply soft deadline semantics, in which transactions still have value even if they miss their deadline. For instance, having late report on the status of the building is better than having no report due to the abortion of the transaction. Soft deadline semantics have been chosen since most data-intensive real-time applications accessing databases are inherently soft real-time applications. Because of concurrent accesses to data and their complex interactions such as locking in databases, hard real-time is hard to achieve. The primary focus of this paper is the QoS management that dynamically minimizes the tardiness of these real-time transactions at runtime.

3 QoS Management in QeDB

Next, we describe our approach for managing the performance of QeDB in terms of QoS. First, we start by defining performance metrics in Section 3.1. An overview of the feedback control architecture of QeDB is given in Section 3.2, followed by the description of the QoS controller design.

3.1 Performance Metrics

The goal of the system is to maintain QoS at a certain level. The most common QoS metric in real-time systems is deadline miss ratio. The deadlines of transactions are application-specific requirement on the timeliness of the transactions, and the deadline miss ratio indicates the ratio of tardy transactions to the total number of transactions. However, it turned out deadline miss ratio is problematic in RTEDBs because the rate of transaction invocation in embedded databases is very low compared to conventional database systems, which handle thousands of transactions per second. For example, a real-time transaction of firefighter's PDA, which checks the status of the building, can be invoked on a per-second basis [16]. With this small number of transactions, the confidence interval of deadline miss ratio can be very wide [11]. Instead, QeDB controls the QoS based on the average tardiness of the transactions. For each transaction, we define the tardiness by the ratio of response time of the transaction to its respective (relative) deadline.

$$tardiness = \frac{response\ time}{deadline}. \tag{1}$$

Another QoS metric, which may pose conflicting requirements, is data freshness. In RTDBs, validity intervals are used to maintain the temporal consistency between the real-world state and sensor data in the database [18]. A sensor data object O_i is considered fresh, or temporally consistent, if *current time-timestamp(O_i)* $\leq$ *avi(O_i)*, where $avi(O_i)$ is the absolute validity interval of O_i. For O_i, we set the update period $P_i = 0.5 \times avi(O_i)$ to support the sensor data freshness [31]. QeDB supports the desired data freshness in terms of perceived freshness (PF) [18].

$$PF = \frac{N_{fresh}}{N_{accessed}}, \tag{2}$$

where N_{fresh} represents the number of fresh data accessed by real-time transactions, and N_{access} represents total number of data accessed by real-time transactions. When overloaded, the data freshness could be traded off to improve the tardiness as long as the target freshness is not violated.

3.1.1 I/O deadline and CPU deadline

The tardiness of a transaction is determined by the response time of both I/O operations and the computation in the transaction. In particular, in a data-intensive real-time application, the I/O response time is a critical factor. The tardiness of a transaction in Equation 1 tells how much a system is overloaded, but it does not tell which resource is overloaded; it can be either I/O or CPU. Therefore, the deadline of a transaction is divided into *I/O deadline* and *CPU deadline* to measure the tardiness of I/O and CPU activities separately. In a transaction, *I/O deadline* and *CPU deadline* are the maximum total time allocated to all I/O operations and all computation activities, respectively. Initially, the I/O deadline and the CPU

deadline of a transaction are determined based on the profiled minimum execution time of I/O operations, $EXEC_{i/o}$, and the computation activities, $EXEC_{cpu}$, in the transaction. $EXEC_{i/o}$ includes the overhead which is proportional to the number of I/O operations, e.g., looking up the buffer cache, locking index/data pages, and etc., but it does not include actual I/O time to access data in flash memory since the buffer hit ratio is assumed 100%. $EXEC_{cpu}$ is the minimum execution time of the transaction except the $EXEC_{i/o}$. Given $EXEC_{i/o}$ and $EXEC_{cpu}$, the slack time of a transaction can be obtained:

$$(EXEC_{i/o} + EXEC_{cpu}) \times sf = deadline, \quad (3)$$

where sf is a slack factor, and it should be greater than one for a transaction to be schedulable in the given system. Hence, the I/O deadline and CPU deadline are set initially as follows;

$$deadline_{i/o} = EXEC_{i/o} \times sf, \quad (4)$$
$$deadline_{cpu} = EXEC_{cpu} \times sf \quad (5)$$
$$= deadline - deadline_{i/o} \quad (6)$$

The definition of tardiness in Equation 1 is extended to tardiness in I/O and CPU as follows:

$$tardiness_{i/o} = \frac{response\_time_{i/o}}{deadline_{i/o}} \quad (7)$$

$$tardiness_{cpu} = \frac{response\_time_{cpu}}{deadline_{cpu}} \quad (8)$$

However, assigning the same static slack factor for both I/O and CPU deadline can be problematic since the ideal slack times for I/O operations and computation change as the system status changes. For example, when one of the resource is overloaded while the other is not, it would be desirable to allocate more slack time to the resource since the other resource is under-utilized. To this end, QeDB dynamically adjusts I/O and CPU deadlines in each sampling period by Algorithm 1.

Algorithm 1: Run-time adaptation of deadlines.

Input: average $tardiness_{i/o}$ and $tardiness_{cpu}$
Input: $\tau_{i/o}$ and τ_{cpu}
if $tardiness_{i/o} \geq tardiness_{cpu}$ **then**
  $\tau_{i/o} + +$; $\tau_{cpu} = 0$;
  $\delta d = \alpha \times \tau_{i/o}$;
  increase $deadline_{i/o}$ by $\delta d\%$
else
  $\tau_{cpu} + +$; $\tau_{i/o} = 0$;
  $\delta d = \alpha \times \tau_{cpu}$;
  decrease $deadline_{i/o}$ by $\delta d\%$
end
$deadline_{cpu} = deadline - deadline_{i/o}$

In Algorithm 1, I/O and CPU deadlines are adjusted by $\delta d\%$ on every sampling period. In a normal state, δd is set to a small number to prevent the high oscillation of deadlines; α is set to 1 in our testbed. However, as a specific resource is being overloaded for consecutive sampling periods, δd increases multiplicatively to speed up the adaptation of deadlines. In the presence of a QoS controller, the overloading of a specific resource happens consecutively when the QoS controller can not adjust CPU or I/O load any further.

3.2 QoS Management Architecture

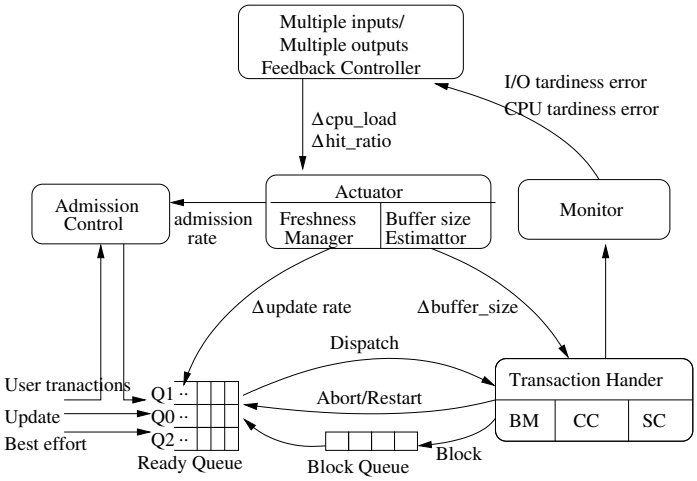


Figure 2. QeDB Architecture.

Figure 2 shows the architecture of QeDB that consists of the MIMO feedback controller, actuator, performance monitor, admission controller, buffer manager (BM), concurrency controller (CC), and scheduler (SC).

The figure shows three separate queues in the ready queue. Temporal data updates are scheduled in Q0 and receives the highest priority. Q1 handles real-time user transactions. Non-real-time transactions in Q2 has the lowest priority, and they are dispatched only if Q0 and Q1 are empty. Transactions in each queue is scheduled in FCFS manner. As a user transaction arrives, it is required to finish by the time equal to the sum of the current time and (relative) deadline. For concurrency control, we apply 2PL (two phase locking) provided by Berkeley DB underlying QeDB. Transactions can be blocked, aborted, and restarted due to data conflicts.

In assigning priorities to transactions, transaction timeliness and and data freshness pose conflicting requirements. If user transactions are given a higher priority than temporal updates, the transaction timeliness can be improved at the cost of the potential freshness reduction and vice versa [9]. In QeDB, we give a higher priority to temporal data updates to preserve the freshness of data. However, the timeliness of user transactions are still guaranteed by the feedback control loop, which regulates the rate of update transactions and the buffer size.

The performance monitor computes the I/O and CPU tardiness, i.e., the difference between the desired I/O (and CPU) response time and the measured I/O (and CPU) response time at every sampling period. Based on the errors, the feedback controller computes the required buffer hit ratio adjustment

(Δhit_raio) and CPU load adjustment (Δcpu_load). The actuator estimates the required buffer size adjustment and update rate adjustment based on Δhit_ratio and Δcpu_load. Finally, the buffer manager and the freshness manager adjusts the buffer size and update rates of temporal data.

3.2.1 Feedback Control Procedure

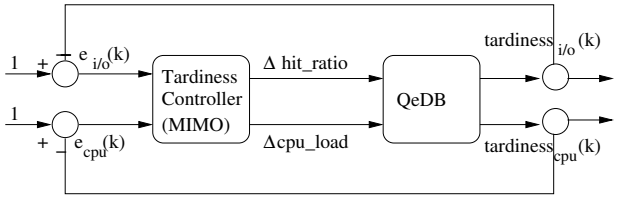


Figure 3. Tardiness Control Loop.

The goal of the feedback controller shown in Figure 3 is to support the transaction response time equal to its deadline, which requires the desired tardiness to be 1. The overall feedback control procedure is as follows.

1. At the k^{th} sampling instant, the tardiness errors $e_{i/o}(k)$ and $e_{cpu}(k)$ are computed respectively for I/O tardiness and CPU tardiness.

2. Based on $e_{i/o}(k)$ and $e_{cpu}(k)$, the MIMO controller computes the control signal Δhit_ratio and Δcpu_load. Unlike a *Single Input/Single Output* (SISO) controller, the MIMO controller computes control signals simultaneously considering both I/O tardiness and CPU tardiness.

3. The actuator translates Δhit_ratio to $\Delta buffer\_size$. QeDB maintains a linear model that correlates the buffer size to the buffer hit ratio [12]. This linear model is updated at each sampling period since locality of data accesses changes dynamically at run-time. $\Delta buffer\_size$ is achieved by changing the buffer size according to this model. Changing the buffer size also changes CPU load as will be shown in the next Section. Therefore, Δcpu_load is adjusted after applying a new buffer size.

4. The Δcpu_load is achieved by adjusting the update rates of *cold* temporal data. Update transactions have small impact on the buffer hit ratio since they access only one data object. For cost-effective temporal data updates, the access update ratio $AUR[i]$ is computed for each temporal data d_i; $AUR[i]$ is defined as $\frac{Access\ Frequency[i]}{Update\ Frequency[i]}$. If $\Delta cpu\_load < 0$, the update rates of a cold data object, which is accessed infrequently, are adjusted from $p[i]$ to $p[i]_{new}$ [17]. The adjustment changes CPU load by $(p[i]_{new} - p[i])/p[i])$. This update period adjustment repeats to a subset of cold data until $\Delta cpu\_load \geq 0$ or no more freshness adaptation is possible.

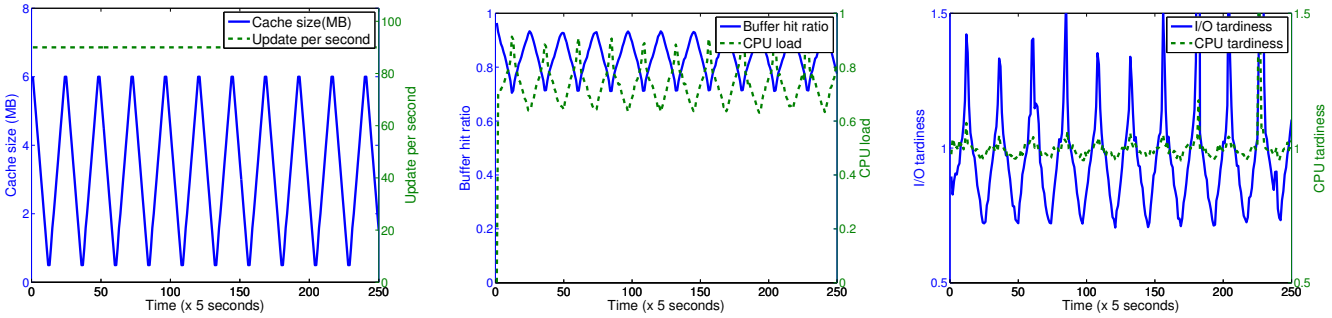
3.3 System Modeling and Controller Design

In this section, we take a systematic approach to designing the tardiness controller.

The first step in the design of a feedback control loop is the modeling of the controlled system [15]; QeDB in our study. Unlike previous work [10][18], which have single-input, single-output (SISO), the QeDB in this paper has multiple inputs (hit_ratio and cpu_load) and multiple outputs ($tardiness_{i/o}$ and $tardiness_{cpu}$). We may choose to use two separate SISO models for each pair of control input and system output; one SISO model for Δhit_ratio and $tardiness_{i/o}$, the other model for Δcpu_load and $tardiness_{cpu}$. However, if an input of a system is highly affected by another input, then a MIMO model should be considered [13] since having two SISO models can not capture the interaction between different control inputs and system outputs.

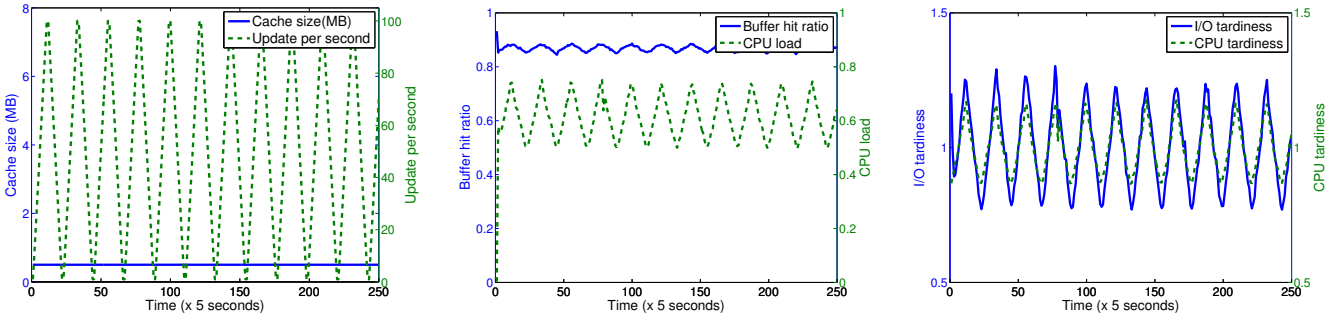
Before the actual *system identification* [22] of QeDB, we performed a series of experiments on a testbed to understand the interaction between multiple control inputs and multiple system outputs. (The details of the testbed and workloads will be described in Sections 4 and 5, respectively.) First, Figure 4 shows the results of varying the cache size while the update rate of temporal data is fixed. It shows that changing the buffer hit ratio via varying cache size also changes the CPU load, and they are inversely proportional; increasing buffer hit ratio decreases the CPU load, and vice versa. This interaction is because I/O operations of QeDB involves not only raw I/O to the flash memory, but also computation such as searching for data pages in a buffer, locking data/index pages, and finding a least-recently-used page. Therefore, when the buffer hit ratio is low, the CPU load increases proportionally to find LRU pages, and allocate a buffer space for new pages from the secondary storage. In the next experiment, the update rate of temporal data is varied while the buffer size is fixed. Figure 5 shows the results. Update transactions are computation-intensive, and it is expected to change only CPU load. The result in Figure 5-(b) matches this expectation. While buffer hit ratio is affected a little bit by varying update rates, the effect is negligible. However, Figure 5-(c) shows that changing CPU load by adjusting update rates affects both I/O tardiness and CPU tardiness; both I/O tardiness and CPU tardiness are proportional to CPU load. This is because the I/O operations in QeDB involves lots of computation themselves and the increased CPU load make them preempted more frequently by high priority update transactions. Moreover, increasing the update rates of temporal data increases the lock confliction rate, making I/O operations to wait for the locks. These results show that a MIMO model is required to capture those close interactions of multiple inputs and multiple outputs of QeDB.

In the actual system identification of QeDB, two inputs are changed simultaneously with relatively prime cycles on the same testbed. The relatively prime cycles are used to fully stimulate the system by applying all different combination of



(a) Cache size vs. update rate (b) Buffer hit ratio vs. CPU load (c)I/O and CPU tardiness

Figure 4. Varying cache size.



(a) Cache size vs. update rate (b) Buffer hit ratio vs. CPU load (c)I/O and CPU tardiness

Figure 5. Varying update rate.

two inputs. The result, which is not shown due to space limitations, is used for system modeling. The form of linear time invariant model for QeDB is shown in Equation 9, with parameters $\mathbf{A}$ and $\mathbf{B}$.

$$\begin{bmatrix} tardiness_{i/o}(k+1) \\ tardiness_{cpu}(k+1) \end{bmatrix} = \mathbf{A} \cdot \begin{bmatrix} tardiness_{i/o}(k) \\ tardiness_{cpu}(k) \end{bmatrix} + \mathbf{B} \cdot \begin{bmatrix} hit\_ratio(k) \\ cpu\_load(k) \end{bmatrix} \quad (9)$$

Because QeDB is modeled as a MIMO system, $\mathbf{A}$ and $\mathbf{B}$ are 2x2 matrices. In our study, the model has $\mathbf{A} = \begin{bmatrix} 0.275 & -0.266 \\ -0.158 & 0.601 \end{bmatrix}$, and $\mathbf{B} = \begin{bmatrix} -0.255 & 1.980 \\ 0.120 & 0.784 \end{bmatrix}$ as its parameters. All eigenvalues of $\mathbf{A}$ are inside the unit circle, hence, the system is stable [15]. The accuracy metric R^2 $\left(= 1 - \frac{variance(\text{experimental value - predicted value})}{variance(\text{experimental value})}\right)$ is 0.844 and 0.866 for I/O and CPU tardiness, respectively. This shows that the above model is accurate enough since $R^2 \geq 0.8$ is considered acceptable [15].

For QeDB, we choose to use a proportional integral (PI) control function given by,

$$U(k) = K_P \cdot E(k) + K_I \cdot \sum_{j=1}^{k-1} E(j), \quad (10)$$

where K_P and K_I are controller gains. We used the linear quadratic regulator (LQR) technique to determine control gains, which is accepted as a general technique for MIMO control [15]. The details of our MIMO controller design procedure can be found in [19].

Finally, the buffer affects the choice of sampling interval because the buffer hit ratio changes slowly after adjusting its size. If the sampling interval is too short, controlling buffer size may not make full effect until the next sampling period, thus, wasting the control effort. Conversely, if the sampling interval is too long, the speed of control will be slow. Our experiments showed that 5 second sampling interval makes a good trade-off between the two conflicting requirements.

4 Implementation

In this sectoin, we describe the system components used in the implementation of QeDB, and show some implementation issues not directly related to QoS control.

4.1 Hardware and Software

The hardware platform used in the testbed is Nokia N810 Internet tablet [6]. The summarized specification of the testbed is given in Table 1. We chose this platform because N810 represents typical modern embedded systems that QeDB is aiming for; it has small main memory space compared to large flash memory space, limiting the the application of main memory-based RTDB technologies[3]. The flash memory in N810 has 2KB page size and 128KB erase block size.

The operating system of N810 is *Maemo*, which is a modified version of GNU/Linux slimmed down for mobile devices[4]. The flash memory of N810 can be accessed through

[3] In N810, the remaining main memory space for user applications is less than 30MB.

[4] Maemo is based on GNU/Linux 2.6.21 kernel.

Table 1. H/W specification of the testbed.

CPU	400MHz TI OMAP 2420
Memory	128 MB RAM
Flash storage	256 MB (on-board), 2GB (SD card)
Network	IEEE 802.11b/g, Bluetooth 2.0

MTD (Memory Technology Device) [5] and JFFS2 (Journaling Flash File Systems, version 2) [4]; database files in QeDB are stored in the flash device via this JFFS2 filesystem. MTD provides a generic interface to access flash devices, which includes routines for read, write, and erase. JFFS2 is a log-structured file system designed for use on flash devices in embedded systems. Rather than using a translation layer on flash devices to emulate a normal hard drive, it places the filesystem directly on the thin MTD layer. JFFS2 organizes a flash device as a log which is an continuous medium, and data are always written to the end of the log. In a flash device old data cannot be overwritten before erasing, so the modified data must be written out-of-place. In a background process, JFFS2 collects these old garbage data.

QeDB is an extension of Berkeley DB, which is a popular open-source embedded database. Berkeley DB provides robust storage features as traditional database systems, such as ACID transactions, recovery, locking, multi-threading for concurrency, and single-master replication for high availability. However, Berkeley DB does not provide the QoS support in terms of tardiness and freshness, which is the main objective for the design of QeDB.

4.2 Implementation Issues

We discuss several implementation issues and challenges for implementing QeDB.

Avoiding Double Buffering QeDB uses file system to store data. When using file systems with a database, the read data is double-buffered in both the file system buffer cache and the DBMS buffer cache. Double buffering not only wastes memory space, but also make I/O response time unpredictable; DBMS have no control over the filesystem layer buffer cache, since it is controlled by the operating system. QeDB's dynamic buffer adjustment scheme cannot achieve its goal in the presence of double-buffer since changing the buffer size at QeDB only affects the buffer in DBMS layer. Unfortunately, Berkeley DB for Linux, underlying QeDB, does not support direct I/O, or bypassing filesystem's buffer cache. QeDB solves this problem by making a separate partition for database files, and disabling buffer cache of that partition at the file system level. The buffer cache in the file system was disabled by applying modification to the JFFS2 code in the Linux kernel.

Dynamic Voltage/Frequency Scaling (DVFS) Modern embedded systems such as Nokia N810 exploit DVFS technique to save power and prolong its lifetime. With DVFS, a CPU has several discrete frequency modes. Nokia N810 has 4 frequency modes and the operating frequency is adjusted automatically by a kernel space module based on the current CPU usage; if current CPU load is more than a threshold, the frequency of the CPU is switched to the higher frequency mode, and vice versa. Since both DVFS controller of OS and QoS controller in QeDB try to control the CPU load, the end result becomes unpredictable. Therefore, we currently disable the DVFS by setting the frequency to the highest one. Having multiple performance controller working on the same resource poses an interesting research question on their interaction. We reserve this as our future work.

5 Evaluation

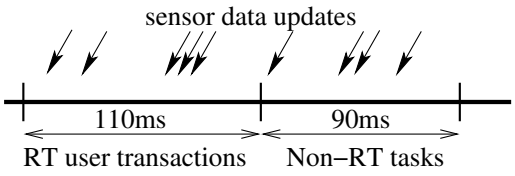


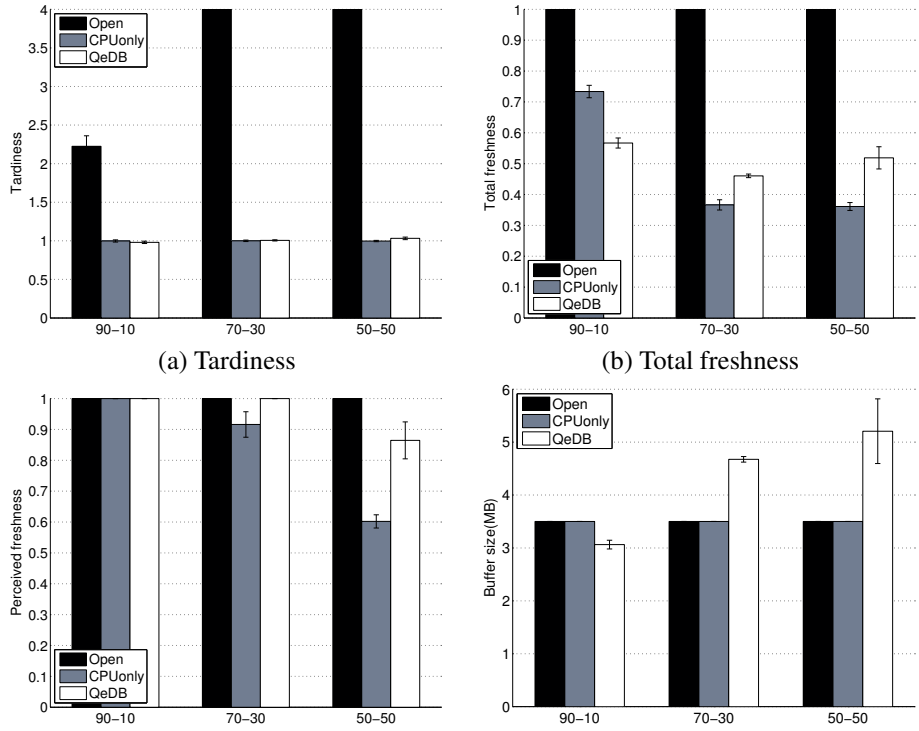
Figure 6. Tasks in the experiment.

For evaluation, a firefighting scenario from [3] is adapted, and simulated on our testbed. In this scenario, a PDA carried by a firefighter collects sensor data via wireless communication, and a periodic real-time task running on the PDA checks the status of the burning building such as the possibility of collapse, explosion, the existence of safe retreat path, etc.

A PDA carried by a firefighter is simulated by a N810 Internet tablet and a stream of sensor data from a building is simulated by a separate 3.0 GHz Linux desktop. Two devices are connected via wireless Ethernet. The desktop simulates 1024 set of sensors located in the building by continuously sending update reports to the N810. The report period of each set of sensors is uniformly distributed in [1 sec, 10 sec]. In the N810, each update report from the desktop initiates an update transaction to the underlying QeDB. Each set of sensors takes 1KB in storage space, totaling 1MB for temporal database in the N810. The N810 has another 7MB data for non-temporal data such as maps of the building, and engineering data to analyze the status of the building[5]. On every $200ms$ period, a new QeDB user transaction is instantiated by a local timer in the N810. The user transaction has a similar structure to Program 1 and it simulates the operation of checking the status of the building. The profiled minimum response time, or the execution time, is $31 \pm 0.52ms$ for $EXEC_{I/O}$, and $36 \pm 0.3ms$ for $EXEC_{CPU}$ with 99% confidence.

The deadline of the user transaction is set to $110ms$ and the remaining $90ms$ is reserved for non-real-time tasks, which are still important for proper operation of the system. For example, our experiment shows that the keypad and GUI components are not working smoothly if the user transaction takes

[5]These are raw data sizes. The total storage and memory overhead to keep this raw data in the database system is more than the twice of the raw data.



(a) Tardiness (b) Total freshness

(c) Perceived freshness (d) Buffer size

Figure 7. Average performance with X-Y access patterns.

Table 2. Tested approaches.

Open	Pure Berkeley DB
CPUonly	SISO control of update rates
QeDB	MIMO control

longer than $110ms$. This is similar to having aperiodic servers in real-time scheduling to handle aperiodic tasks. Figure 6 shows the tasks in one period; this period repeats in the experiment. In the experiment, we do not set a specific requirement on the freshness of the temporal data to observe how much trade-off is made to make real-time transactions timely.

For performance comparisons, we consider three approaches shown in Table 2. *Open* is the basic Berkeley DB without any special performance control facility. However, the FCFS scheduling with 3 differentiated queues as in Section 3.2 are still used to process real-time transactions faster than the others. Thus, Open represents a state-of-art database system. In contrast, *CPUonly* represents a RTDB having a QoS management scheme [17], which is not I/O-aware. In CPUonly, as the tardiness of transactions deviates from the desired tardiness, only the CPU workload is adjusted by changing update rates of temporal data. I/O workload is not dynamically controlled. This scheme is originally designed for main-memory RTDBs that have no or negligible I/O workload. This approach uses a SISO model and controller; the CPU utilization is the control input and the transaction tardiness is the system output. A PI controller is used for CPUonly. Finally, *QeDB* is our approach, which controls both I/O and CPU tardiness using a MIMO controller.

5.1 Average performance

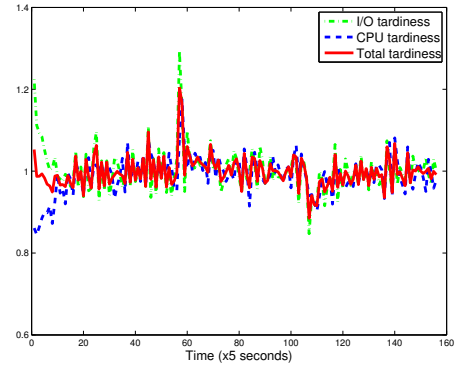
Computing systems show different behavior for different workloads. In this experiment, workloads are varied by applying different data access patterns. The effect of data contention is tested using $x-y$ data access scheme. In $x-y$ access scheme, $x\%$ of data accesses are directed to $y\%$ of the data in the database. For instance, with 90-10 access pattern, 90% of data accesses are directed to 10% of data in the database, thus, incurring data contention on 10% of entire data. We tested the robustness of our approach by applying three different $x-y$ access patterns.; 90-10, 70-30, and 50-50 data access patterns.

The average performance for each approach with each data access pattern is shown in Figure 7; the confidence interval is 99%. Figure 7-(a) shows that both CPUonly and QeDB achieve the tardiness goal in all data access patterns. In contrast, Open does not achieve the goal in any data access pattern. Actually, the tardiness of Open could not be measured in 70-30 and 50-50 data access patterns because the system did not respond. This is the situation, in which real-time tasks take all resources, preventing low-priority tasks from proceeding.

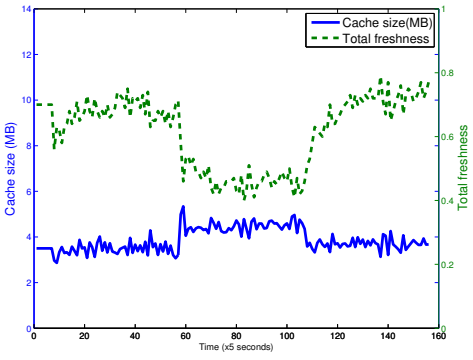
Even though both CPUonly and QeDB achieve the tardiness goal, the trade-offs, which they make, are very different. Figure 7-(b) shows the total tardiness, which is the ratio of fresh data to the entire data in the database. CPUonly shows lower data freshness compared to QeDB, and this is even more evident when we consider the perceived freshness of data as shown in Figure 7-(c). In 90-10 access pattern, the total size of locality is small, and transactions are CPU-bounded since most data access requests can be handled in the buffer with high buffer hit ratio. However, as the data access

spreads widely as in 50-50 access pattern, the size of locality becomes larger, and the transactions become I/O-bounded since the small buffer of CPUonly incurs low hit ratio. In case of CPUonly, the tardiness is controlled only by adjusting the freshness of temporal data, regardless of which resource is getting overloaded. Therefore, CPUonly have to lower the freshness of data excessively, which can be problematic if an application requires high data freshness. For example, the total freshness of CPUonly drops from 0.73 ± 0.02 to 0.37 ± 0.1 when the access pattern changes from 90-10 to 50-50, which is more than 50% degradation of the data freshness. In contrast, QeDB achieves more stable data freshness in all data access patterns. As the size of locality is getting larger, QeDB increases the size of buffer as in Figure 7-(d), while still maintaining high data freshness. For instance, the buffer size increase about 64% and the total freshness drops 18% when the access pattern changes from 90-10 to 50-50. If we consider the perceived freshness of data, the difference between CPUonly and QeDB is even higher. The perceived freshness drops only 14% in QeDB while it drops 40% in CPUonly.

5.2 Transient Performance



(a) Tardiness



(b) Buffer size and freshness

Figure 8. Transient performance of QeDB.

The average performance is not enough to show the performance of dynamic systems. Transient performance such as settling time should be small enough to satisfy the requirements of applications. In this experiment, the workload increases by introducing a disturbance. For example, consider a situation where peer firefighers opportunistically exchange their temporal data when they are close enough to communicate with each other. This opportunistic data exchange incurs both additional I/O and CPU load to process it. In the experiment, the disturbance is a periodic transaction that lasts for 50 sampling periods. The periodic transaction retrieves 1KB data page with $10ms$ interval.

Figure 8-(a) shows the tardiness of real-time transactions. The corresponding buffer hit ratio and data freshness at the same time are shown in Figure 8-(b). The disturbance starts at the 55^{th} sampling period and ends at the 105^{th} sampling period. In Figure 8-(a), we can see that the tardiness increases suddenly at the 55^{th} sampling period. However, the tardiness stabilizes within 3 sampling periods. When the disturbance disappears at the 105^{th} sampling period, it takes 8 sampling periods to stabilize. These long settling times are the result of controller tuning in the controller design phase in Section 3.3. We can reduce the settling times by choosing control parameters for more aggressive control. However, aggressive control results in the higher overshoots and fluctuations in controller inputs (the cache size and update rates). Changing the cache size too frequently has a problem in our system since it causes more read/write operations to and from the flash memory. In particular, frequent write operations can incur significant I/O overhead. For details on controller tuning, users are referred to [19]

6 Related Works

Most RTDB work is based on simulations. Only a few works [9][17] have actually been implemented and evaluated in real database systems. However, most of them are not available publically, or outdated. QeDB has been developed to address this problem. Moreover, all previous implementations take main memory-based RTDB approaches, limiting their application to a broader range of systems, resource-constrained systems in particular. In contrast, QeDB does not require that all data reside in main memory.

In embedded systems domain, several DBMS implementations [20][21] are available. Most of the them are flash memory-based DBMSs, and exploit the peculiar characteristics of flash memory to optimize its resource consumption. Some DBMSs [26][27] target extremely resource-constrained embedded systems such as sensor platforms. However, these DBMSs provide only basic features for accessing data such as indexing. Some features of DBMSs such as concurrency control and guaranteeing logical consistency are usually not supported in these DBMSs. Therefore, their performance optimization is at the flash device level. Moreover, even though they provide efficient mechanisms on accessing data, their approaches are basically best-effort; they do not assure any guarantees on its performance. Unlike these DBMSs, QeDB guarantees the QoS goals set by applications via feedback control even in the presence of dynamically changing workload.

Feedback control have been actively applied to manage the performance of various systems such as a web server [23], real-time middleware [24], caching service [25], and email

server [30]. However, these approaches are not directly applicable to RTDBs, because they do not consider RTDB-specific issues such as data freshness. Moreover, these approaches consider only single resource, e.g., contention in CPU, for QoS management. Unlike these approaches, QeDB manages multiple resources, which have close interaction, via MIMO feedback control.

7 Conclusions and Future Work

In this paper, we proposed a novel feedback control architecture to support the desired tardiness of transactions in RTEDBs. Unlike previous feedback control of RTDB performance, our approach controls multiple resources simultaneously, which have close interaction, to provide more efficient and robust control behavior. We showed the feasibility of the proposed feedback control architecture by implementing and evaluating it on a modern embedded system. Our evaluation shows that the simultaneous control of I/O and CPU resource can make a better negotiation between the timeliness of transactions and the freshness of data by providing proper amount of resources in a robust and controlled manner. In the future, we plan to enhance the feedback control scheme to include other QoS metrics such as power, which is critical in mobile embedded systems.

References

[1] CarTALK2000 Project, http://www.cartalk2000.net/, 2007.

[2] Communication and Networking Technologies for Public Safety,. National Institute of Standards and Technology, http://w3.antd.nist.gov/comm_net_ps.shtml, 2008.

[3] Fire Information and Rescue Equipment (FIRE) project, http://fire.me.berkeley.edu/, 2008.

[4] JFFS2: The Journalling Flash File System, version 2, http://sources.redhat.com/jffs2/, 2008.

[5] Memory Technology Device Subsystem for Linux, www.linux-mtd.infradead.org/, 2008.

[6] Nokia N-Series, http://www.nseries.com/, 2008.

[7] Oracle Berkeley DB, http://www.oracle.com, 2008.

[8] SQLite, http://www.sqlite.org, 2008.

[9] B. Adelberg. *STRIP: A Soft Real-Time Main Memory Database for Open Systems*. PhD thesis, Stanford University, 1997.

[10] M. Amirijoo, J. Hansson, and S. H. Son. Specification and management of QoS in real-time databases supporting imprecise computations. *IEEE Transactions on Computers*, 55(3):304–319, March 2006.

[11] L. Bertini, J. C. B. Leite, and D. Mosse. Statistical qos guarantee and energy-efficiency in web server clusters. In *ECRTS '07: Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 83–92, Washington, DC, USA, 2007. IEEE Computer Society.

[12] K. P. Brown, M. J. Carey, and M. Livny. Goal-oriented buffer management revisited. *SIGMOD Rec.*, 25(2):353–364, 1996.

[13] Y. Diao, N. Gandhi, and J. Hellerstein. Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache web server. In *Network Operations and Management*, April, 2002.

[14] W. Enkelmann. Fleetnet - applications for inter-vehicle communication. In *Intelligent Vehicles Symposium, 2003. Proceedings. IEEE*, 2003.

[15] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. Wiley IEEE press, 2004.

[16] X. Jiang, N. Y. Chen, J. I. Hong, K. Wang, L. Takayama, and J. A. L. Siren: Context-aware computing for firefighting. In *In Proceedings of Second International Conference on Pervasive Computing*, pages 87–105, 2004.

[17] K.-D. Kang, J. Oh, and S. H. Son. Chronos: Feedback control of a real database system performance. In *RTSS*, 2007.

[18] K.-D. Kang, S. H. Son, and J. A. Stankovic. Managing deadline miss ratio and sensor data freshness in real-time databases. *IEEE Transacctions on Knowledge and Data Engineering*, 16(10):1200–1216, October 2004.

[19] W. Kang, S. H. Son, J. A. Stankovic, and M. Amirijoo. I/O-aware deadline miss ratio management in real-time embedded databases. In *The 28th IEEE Real-Time Systems Symposium (RTSS)*, Dec, 2007.

[20] G.-J. Kim, S.-C. Baek, H.-S. Lee, H.-D. Lee, and M. J. Joe. LGeDBMS: A small DBMS for Embedded System with Flash Memory. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, 2006.

[21] S.-W. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007.

[22] L. Ljung. *Systems Identification:Theory for the User 2nd edition*. Prentice Hall PTR, 1999.

[23] C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son. A feedback control approach for guaranteeing relative delays in web servers. In *RTAS '01: Proceedings of the Seventh Real-Time Technology and Applications Symposium (RTAS '01)*, 2001.

[24] C. Lu, X. Wang, and C. Gill. Feedback control real-time scheduling in orb middleware. In *RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 37, Washington, DC, USA, 2003. IEEE Computer Society.

[25] Y. Lu, T. F. Abdelzaher, and A. Saxena. Design, implementation, and evaluation of differentiated caching services. *IEEE Trans. Parallel Distrib. Syst.*, 15(5):440–452, 2004.

[26] G. Mathur, P. Desnoyers, D. Ganesan, and P. J. Shenoy. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *SenSys*, 2006.

[27] S. Nath and A. Kansal. FlashDB: Dynamic self-tuning database for NAND flash. In *The International Conference on Information Processing in Sensor Networks (IPSN)*, 2007.

[28] A. Nori. Mobile and embedded databases. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1175–1177. ACM, 2007.

[29] D. Nystrom, A.Tesanovic, C.Norstrom, J. Hansson, and N.-E. Bankestad. Data management issue in vehicle control systems: a case study. In *ECRTS'02*, 2002.

[30] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. *Real-Time Syst.*, 23(1-2):127–141, 2002.

[31] K. Ramamritham, S. H. Son, and L. C. Dipippo. Real-time databases and data services. *Real-Time Systems*, 28(2-3):179–215, 2004.