Hindawi Publishing Corporation
Mobile Information Systems
Volume 2015, Article ID 657252, 14 pages
http://dx.doi.org/10.1155/2015/657252

Research Article

QoS Management for Embedded Databases in Multicore-Based Embedded Systems

Woochul Kang[1] and Jaeyong Chung[2]

[1]*Embedded Systems Engineering Department, Incheon National University, Incheon 406-772, Republic of Korea*
[2]*Electronic Engineering Department, Incheon National University, Incheon 406-772, Republic of Korea*

Correspondence should be addressed to Jaeyong Chung; jychung@inu.ac.kr

Received 11 June 2015; Accepted 27 September 2015

Academic Editor: Francesco Palmieri

With ubiquitous deployment of sensors and network connectivity, amounts of real-time data for embedded systems are increasing rapidly and database capability is required for many embedded systems for systematic management of real-time data. In such embedded systems, supporting the timeliness of tasks accessing databases is an important problem. However, recent multicore-based embedded architectures pose a significant challenge for such data-intensive real-time tasks since the response time of accessing data can be significantly affected by potential intercore interferences. In this paper, we propose a novel feedback control scheme that supports the timeliness of data-intensive tasks against unpredictable intercore interferences. In particular, we use multiple inputs/multiple outputs (MIMO) control method that exploits multiple control knobs, for example, CPU frequency and the Quality-of-Data (QoD) to handle highly unpredictable workloads in multicore systems. Experimental results, using actual implementation, show that the proposed approach achieves the target Quality-of-Service (QoS) goals, such as task timeliness and Quality-of-Data (QoD) while consuming less energy compared to baseline approaches.

1. Introduction

Recently, database functionality is increasingly embedded into mobile and embedded platforms for systematic management of a large amount of real-time data such as sensor streams. For example, autonomous cars need to process a large volume of real-time data from sensors in real time [1]. In such systems, real-time tasks with intensive database accesses are required to provide a certain level of Quality-of-Service (QoS), such as task timeliness and data freshness.

In our previous work [2], we presented a real-time embedded database, called QeDB, that supports the timeliness of data-intensive tasks using the control-theoretic QoS management architecture. With the feedback control loop, QeDB can achieve the desired QoS by adapting its control knobs based on QoS errors. Hence, a precise system model is not required at the design time. Our previous work assumed single-core platforms for the QoS management. However, modern embedded systems are increasingly moving towards multicore platforms and they pose a huge challenge since concurrent data accesses might cause contention at non-CPU resources, such as memory and I/O [3–9]. There are potential intercore interferences as the data accesses from one core could also be influenced by the requests from the other CPU cores. As a result, the response time of a data-intensive task can be delayed significantly due to the bottleneck in accessing data.

To handle such unpredictable intercore interferences, we might consider *Quality-of-Data (QoD) scaling* as a primary control knob for the QoS management. With the QoD scaling, the incoming sensor updates are selectively dropped by the admission controller to control the workload. By scaling down QoD, the workload for data accesses is reduced, rendering less intercore contention for data accesses. However, a major disadvantage of QoD scaling is that its applicability is limited by users' QoD requirements. Hence, the QoS goals might not be satisfied if QoD is saturated at its maximum or minimum. Moreover, QoD scaling is not useful if a task's response time is dominated by computation, not by accessing data.

In this paper, we propose an efficient QoS management approach, in which multiple complementing control knobs are exploited simultaneously to handle highly unpredictable workloads in multicore platforms. In our approach, the limitation of QoD scaling is complemented by exploiting *Dynamic Voltage/Frequency Scaling (DVFS)* [10]. Unlike QoD scaling, DVFS is more appropriate to control the speed of tasks when the workload is less data-intensive. Further, DVFS has a wide range of operating region. For example, ARM-based Exynos 5422 mobile processor supports 19 frequency/voltage levels ranging from 200 MHz to 2.0 GHz. The two distinctive control knobs are combined using a novel *multiple inputs multiple outputs* (MIMO) control architecture. This MIMO control architecture can capture the interrelationships between the multiple control knobs and the system outputs and generates proper combinations of the multiple control signals according to the varying workloads.

We implement the proposed approach in an actual multicore-based embedded device by extending our previous work. The evaluation results demonstrate that the proposed QoS management approach is more effective in QoS enforcement than applying either DVFS or QoD scaling alone. Our approach can achieve the QoS goals with significantly smaller power consumption, particularly when workloads are data-intensive and have high chance of intercore interferences for accessing data.

The rest of this paper is organized as follows: in Section 2, we summarize our previous work on QeDB including its transaction model and QoS management architecture. In Section 3, we discuss the effect of intercore interferences on data-intensive real-time tasks. In Section 4, we present our approach to QoS management. In Section 5, the performance evaluation settings and results are presented. Related work is presented in Section 6 and Section 7 concludes the paper.

2. Overview of QeDB

Our current work extends our previous work on QeDB [2]. QeDB is a key/value store database for data-intensive real-time applications running on embedded devices. We briefly introduce QeDB for the discussion of the following sections.

2.1. Data and Real-Time Transactions. Data objects in QeDB can be categorized into *temporal* and *nontemporal* data. Temporal data objects are updated by *update transactions* when new sensor readings become available. *User transactions* are tasks that perform computation using data objects in the database. User transactions consume both temporal and nontemporal data objects. Algorithm 1 is an example of user transaction that performs real-time analysis by accessing sensor data in QeDB. Instead of supporting complex queries, data objects in QeDB can be accessed through *get (key)* and *put (key, value)* interfaces. Data objects are identified using keys.

User transactions in QeDB can specify its desired response time or deadline, according to their timing constraints. We call such user transactions as *real-time transactions*. If a real-time transaction is periodic, its periodic instances are supposed to meet the deadline. QeDB only supports soft real-time semantics, and, hence, missing deadlines decreases the QoS (Quality-of-Service) but does not jeopardize correct system behavior.

```
rt_task_begin:
  /* a list of keys for fresh sensor data */
  DBT key_sensors = {s_1,s_2,...,s_n};
  /* memory buffer for sensor data */
  DBT data_sensors[MAX_SENSORS];
  /* access data through embedded database */
  for key in key_sensors {
      data_sensors[i++] = get(key);
  }
  /* computation for analysis */
  analyze_risk(data_sensors);
rt_task_end:
```

Algorithm 1: An example of data-intensive real-time task.

2.2. QoS Metrics. Since typical embedded systems do not have many transactions, *deadline miss ratio* is not a stable metric for QoS management [11]. Hence, we define the tardiness of a transaction as follows:

$$\text{tardiness} = \frac{\text{response time}}{\text{deadline}}. \quad (1)$$

For example, a task is tardy if its response time is greater than its deadline. In QeDB, the average *tardiness* of real-time transactions is used as a QoS metric to quantify the timeliness of the transactions. Each transaction performs both computation and data accesses. Therefore, a transaction's response time is affected by both its data accessing activities and computation activities. To this end, we define the following parameters for a transaction J:

(i) *Data response time* R_{data} is the total response time for J to access data objects in the database. The response time of each database access is measured by instrumenting *put* and *get* methods. It should be noted that put/get methods have computational overheads to access data, such as processing indexes.

(ii) *Computation response time* R_{comp} is the response time for J to perform computational activities, excluding database accesses.

(iii) *Transaction response time* R is the response time of J. Hence, R equals $R_{\text{comp}} + R_{\text{data}}$.

Another important QoS metric for data-intensive real-time systems is the *Quality-of-Data (QoD)*. In real-time systems, the result of real-time tasks depends on both the logical correctness and the temporal correctness. The result is temporally correct only if the real-time data from sensors are *fresh* enough. A temporal data object O_i is considered *fresh* or temporally consistent, if its timestamp is less than its *absolute validity interval (avi)*. In this work, we define QoD as the ratio

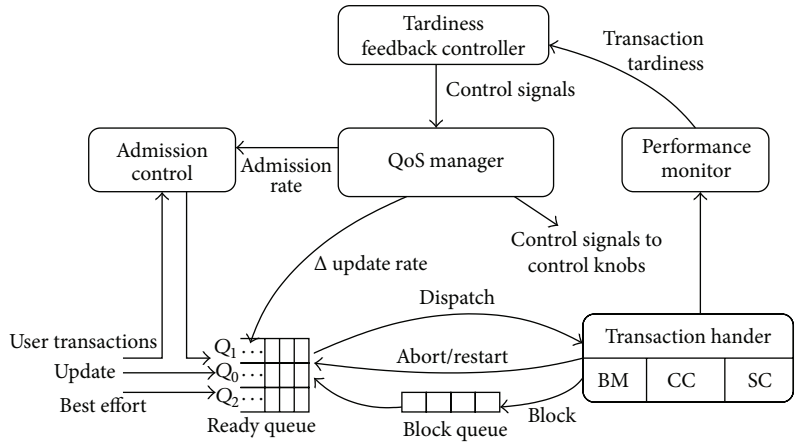


Figure 1: QoS management architecture of QeDB [2].

of the number of fresh data objects N_{fresh} to the total number of temporal data objects N_{temporal}:

$$\text{QoD} = \frac{N_{\text{fresh}}}{N_{\text{temporal}}}. \quad (2)$$

Since the higher QoD is desirable as far as the system is not overloaded, users or applications specify only the minimum QoD, denoted in QoD_{min}, as a QoS specification.

2.3. QoS Management Architecture. QeDB supports the desired QoS through its QoS management architecture shown in Figure 1.

The architecture follows the feedback control principle, and hence it exploits the closed loop of continuous "*monitoring*" and "*control.*" The *transaction handler* includes the core engine of the underlying embedded database, and it processes admitted transactions. At every sampling period, the *performance monitor* computes the average tardiness of real-time transactions. The *tardiness feedback controller* generates control signals by taking the difference between the target tardiness and the current average tardiness. The *QoS manager* enforces these control signals by using available control knobs in the system. In our previous work, we changed the rate of sensor updates and subsequent QoD using the *admission controller* to control the system's overheads. In recent embedded platforms, the QoS controller might exploit other control knobs, such as DVFS. In the following sections, we discuss how to exploit these multiple control knobs in multicore environments.

3. Motivation: Intercore Interferences in Multicore Systems

Modern embedded systems are increasingly moving towards multicore platforms. We might consider scheduling real-time transactions onto a dedicated CPU core to avoid scheduling interferences from non-real-time tasks. However, contention for shared resources, such as memory and I/O, cannot be avoided completely in multicore platforms. In particular, the interferences between the cores pose significant challenge for data-intensive real-time applications, in which predictable system behavior is highly required.

To illustrate the problem, a microbenchmark is performed in a multicore embedded platform. In the benchmark, a real-time task is invoked periodically on every 100 ms, and it is scheduled to run in CPU core #1. The task is a transaction, as shown in Algorithm 1, that performs computational analysis by actively accessing real-time data in the database. At the same time, a stream of independent best-effort transactions are executed in the other CPU cores to interfere the real-time transaction. These best-effort transactions access different databases in the system. For this benchmark, the QoS management mechanism in Figure 1 is deactivated. We use the CPU affiliation feature of the testbed platform to assign the transactions to different CPU cores. Transactions in each active CPU core are scheduled according to real-time FIFO policy. The details of the testbed platform are discussed in Section 5.

In the benchmark, the real-time transaction's data response time R_{data} and computation response time R_{comp} are measured under varying CPU clock speed. Figure 3(a) shows the result when the real-time transaction is executed without interfering best-effort transactions in the other CPU cores. Figures 3(b) and 3(c), respectively, plot the results when the best-effort transactions are scheduled in the other 1 and 3 CPU cores. The results show that the task response time of the real-time transaction is increased significantly as more CPU cores are used to execute best-effort transactions. For instance, at 2.0 GHz, the task response time of the transactions is increased from 0.21 to 0.36 and to 0.42, respectively, when the other 1 and 3 CPU cores are involved in interfering real-time transactions. However, it should be noted that the computation response times R_{comp}'s of the real-time transactions are not much affected by the intercore interferences. Only the data response times R_{data}'s of the real-time transactions are increased from 0.05 to 0.16 and to 2.3, respectively, in Figures 3(a), 3(b), and 3(c). These results demonstrate that the response time of data-intensive real-time transactions can be affected significantly by intercore

interferences. Further, these intercore interferences at the shared resources are hard to predict and pose significant challenges for data-intensive real-time applications.

The potential presence of intercore interferences changes the characteristics of the workload. For example, in Figure 3(a), the real-time transactions are more computation-oriented when no interfering transactions run at other CPU cores. The ratio between R_{comp} and R_{data} is about 0.8 : 0.2. Therefore, changing the CPU speed using DVFS can be effective in controlling the total response time of the transaction. However, when transactions have high intercore interferences from other CPU cores, as in Figure 3(c), the ratio between R_{comp} and R_{data} is changed to about 0.5 : 0.5. In this situation, changing the computation speed using DVFS has limited effect on the task response time. For instance, in Figure 3(a), the normalized response time of 0.4 is achieved at CPU core frequency of 1100 MHz. In contrast, in Figure 3(c), the normalized response time of 0.4 cannot be supported even at 2000 MHz CPU frequency, which is the maximum CPU frequency.

We performed the second microbenchmark experiment to understand the impact of QoD scaling when transactions incur high intercore interferences. In the experiment, we measure the response time of the real-time transactions while the QoDs of the transactions are varied from 10% to 100%. We can decrease the QoD of temporal data by increasing the update intervals of sensors. During the experiment, the CPU frequency is fixed at 1.0 GHz. Figure 4 shows that the data response time R_{data} of the real-time transactions is affected significantly by QoD. For instance, in Figure 4, decreasing the QoD from 100% to 50% reduces R_{data} from about 0.5 to 0.3. This result shows that decreasing QoD is an effective method to reduce the chance of intercore interferences in multicore systems.

4. QoS Management for Multicore Systems

In this section, we propose the QoS management approach that exploits multiple control knobs to handle highly dynamic workloads in multicore environments.

4.1. Metric to Quantify Intercore Interference. As seen in Section 3, the workload characteristic of a transaction can be significantly affected by intercore interferences. As a consequence, the effectiveness of the control knobs, for example, DVFS and QoD scaling, also changes according to the varying workloads. The QoS management architecture in Figure 1 is supposed to coordinate these multiple control knobs under such highly variable multicore environments. To this end, we define drr *(data response ratio)* as a metric that characterizes transaction J's workload state:

$$\text{drr} = \frac{R_{data}}{R_{comp} + R_{data}}. \tag{3}$$

drr of a transaction is a ratio of data response time to the total response time. In this paper, we assume real-time transactions, in which the data access pattern is not varying much between their repeating periods. Therefore, significant changes in drr imply the presence of intercore interferences. For instance, drr gets higher as more intercore interferences occur. We further define drr^{norm} as J's *nominal data response ratio* that represents the minimal drr:

$$\text{drr}^{norm} = \frac{R_{data}^{norm}}{R_{data}^{norm} + R_{comp}^{norm}}, \tag{4}$$

in which R_{data}^{norm} and R_{comp}^{norm}, respectively, are transaction J's R_{data} and R_{comp} profiled while no interfering tasks are executed in the other CPU cores. Therefore, the gap between drr and drr^{norm} can be used as an indicator that tells how much a transaction is delayed due to tardy data accesses. In multicore-based real-time systems, intercore interferences are the major source of tardy data accesses.

4.2. Feedback Control Procedure. The primary goal of QoS management is to support the transaction response time equal to the desired response time. Further, another goal is to exploit multiple control knobs properly, considering the dynamic workloads of multicore systems. Since we have two control goals, we need to provide at least two control inputs to control them. For example, if real-time transactions are tardy due to intercore interferences, we need a control knob that effectively reduces the intercore interferences. Conversely, if the transactions are tardy because of slow computation activities, we need another control knob to speed up the computation. Given a task, one available control knob that significantly affects its computation response time is the processor speed. The higher the processor speed, the shorter the response time of the task. In modern embedded processors, the processor speed can be controlled by changing processor frequency using DVFS. Regarding data response time, we can exploit *QoD scaling* as a control knob. Since the higher QoD is translated into the more frequent accesses to temporal data, the data response time of a transaction is highly affected by QoD.

To achieve these multiple goals using multiple control knobs, we propose to exploit the MIMO (multiple inputs/multiple outputs) control loop shown in Figure 5. The overall feedback control steps are as follows:

(1) The desired transaction tardiness, tard^{target}, and the desired data response time, drr^{target}, are set. Typically, we may set them to 1 and drr^{norm}, respectively. By setting drr^{target} to drr^{norm}, we require the system to maintain the minimal drr against potential intercore interferences.

(2) At the kth sampling instant, the average tardiness error $e_{tard}(k)$ and the drr error $e_{drr}(k)$ are computed for real-time transactions.

(3) According to $e_{tard}(k)$ and $e_{drr}(k)$, the tardiness controller computes the control signals Δfreq and ΔQoD. The MIMO controller computes the control signals simultaneously considering both the transaction tardiness and the data response ratio.

(4) The QoS manager changes the CPU core frequency to achieve Δfreq.

(5) ΔQoD is achieved by adjusting the update rates of temporal data objects.

4.3. Feedback Control Loop Design. In this paper, we take a systematic approach to designing the feedback controller.

4.3.1. System Modeling and Verification. The first step in designing a feedback controller is to construct a model that captures the target system's properties. In this study, the QeDB running on a multicore system is the target system.

As discussed in previous sections, the goals of the QoS management are to support the desired transaction tardiness while preventing excessive intercore interferences in multicore environments. To achieve these multiple control goals using multiple control knobs, we exploit a MIMO model. The form of MIMO linear time-invariant model for QeDB is shown in

$$\begin{bmatrix} \text{tardiness}\,(k+1) \\ \text{drr}\,(k+1) \end{bmatrix} = \mathbf{A} \cdot \begin{bmatrix} \text{tardiness}\,(k) \\ \text{drr}\,(k) \end{bmatrix} + \mathbf{B} \cdot \begin{bmatrix} \text{freq}\,(k) \\ \text{QoD}\,(k) \end{bmatrix}. \tag{5}$$

The model parameters **A** and **B** are 2×2 matrices because the system has two inputs and two outputs.

We may choose to use two separate single input/single output (SISO) models, one SISO model to relate CPU frequency to transaction tardiness and another SISO model to relate QoD to drr. However, if system inputs affect multiple outputs, then a MIMO model should be considered to capture the interaction between the different control inputs and system outputs [12]. For instance, in our system, changing QoD affects both the transaction tardiness and drr.

In the actual system identification of QeDB, two inputs are varied simultaneously. The relatively prime cycle inputs are used to fully stimulate the system by applying all different combinations of the two inputs. Figure 6 shows the result of the system identification. The model parameters obtained through the system identification are $\mathbf{A} = \begin{bmatrix} 0.8504 & -0.0066 \\ -0.1449 & 0.3882 \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} -0.1983 & 0.1485 \\ 0.2448 & 0.7762 \end{bmatrix}$. These parameters quantify the interaction between the control inputs and the system outputs. For instance, the two components of **B**'s first row have different signs and this means that the CPU frequency and the QoD scaling drive the tardiness of transactions in different directions. One widely used metric to quantify the model accuracy is R^2, where $R^2 = 1 - \text{variance(experimental value} - \text{predicted value)/variance(experimental value)}$. The R^2's of our model are 0.908 and 0.823 for transaction tardiness and drr, respectively. In general a model with $R^2 \geq 0.8$ is considered valid [13].

4.3.2. Controller Design. The closed-loop model is constructed as follows:

$$\begin{bmatrix} \mathbf{e}\,(k+1) \\ \mathbf{e}_I\,(k+1) \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{I} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{e}\,(k) \\ \mathbf{e}_I\,(k) \end{bmatrix} - \begin{bmatrix} -\mathbf{B} \\ \mathbf{0} \end{bmatrix} \mathbf{u}\,(k) + \begin{bmatrix} \mathbf{I} - \mathbf{A} \\ \mathbf{0} \end{bmatrix} \mathbf{r}, \tag{6}$$

where $\mathbf{r} = \begin{bmatrix} 1 & \text{drr}^{\text{target}} \end{bmatrix}^T$. In this model, the control error vector $\mathbf{e}(k)$ and the acumulated control error vector $\mathbf{e}_I(k)$ are used as the state vector. For the robustness against disturbance and simplicity, we choose to apply a *proportional integral (PI)* control function, given by

$$\mathbf{u}\,(k) = \begin{bmatrix} \text{freq}\,(k) \\ \text{QoD}\,(k) \end{bmatrix} = -\begin{bmatrix} \mathbf{K}_P & \mathbf{K}_I \end{bmatrix} \begin{bmatrix} \mathbf{e}\,(k) \\ \mathbf{e}_I\,(k) \end{bmatrix}, \tag{7}$$

where $\mathbf{K}_P$ and $\mathbf{K}_I$, respectively, are proportional and integral controller gains. $\mathbf{K}_P$ and $\mathbf{K}_I$ are 2×2 matrices. At each sampling instant k, the performance monitor calculates the control error

$$\mathbf{e}\,(k) = \begin{bmatrix} 1 - \text{tardiness}\,(k) \\ \text{drr}^{\text{target}} - \text{drr}\,(k) \end{bmatrix} \tag{8}$$

and the accumulated control error

$$\mathbf{e}_I\,(k+1) = \mathbf{e}_I\,(k) + \mathbf{e}\,(k). \tag{9}$$

Using $\mathbf{e}(k)$ and $\mathbf{e}_I(k)$, the control law in (7) computes the controller input $\mathbf{u}(k)$.

The properties of the closed-loop system, such as the settling time, the overshoot, and the stability, are determined by the control gains $\mathbf{K}_P$ and $\mathbf{K}_I$. We obtained the control gains using *linear quadratic regulator (LQR)* technique that minimizes the cost function J:

$$J = \sum_{k=0}^{\infty} \begin{bmatrix} \mathbf{e}\,(k) & \mathbf{e}_I\,(k) \end{bmatrix} \cdot \mathbf{Q} \cdot \begin{bmatrix} \mathbf{e}\,(k) \\ \mathbf{e}_I\,(k) \end{bmatrix} + \mathbf{u}\,(k)^T \cdot \mathbf{R} \cdot \mathbf{u}\,(k), \tag{10}$$

where the weighting matrices **Q** and **R** quantify the cost of control error and the cost of control effort, respectively. Since minimizing the transaction tardiness is the primary goal of the QoS management, we put the higher weight to the tardiness control error e_{tard} compared to the data response ratio error e_{drr} by choosing $\mathbf{Q} = \text{diag}(1, 1/10, 1, 1/10)$. The first and the second elements of **Q** quantify the cost of control errors e_{tard} and e_{drr}, respectively. Once weighting matrices **Q** and **R** are determined, MATLAB commands *dlqr* can be used to get the controller gains. The controller gains obtained through *dlqr* are $\mathbf{K}_P = \begin{bmatrix} 0.396 & -0.062 \\ -0.107 & -0.067 \end{bmatrix}$ and $\mathbf{K}_I = \begin{bmatrix} 0.058 & -0.035 \\ -0.025 & -0.041 \end{bmatrix}$.

We can analytically prove the stability of the closed-loop system in (6) by showing that the poles of the closed-loop system are all within the unit circle [13]. In (6), the poles are the eigenvalues of

$$\begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{I} & \mathbf{I} \end{bmatrix} - \begin{bmatrix} -\mathbf{B} \\ \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{K}_P & \mathbf{K}_I \end{bmatrix}. \tag{11}$$

Table 1: Hardware specification.

CPU	Exynos 5422 octa cores
Processor frequency	200 MHz–2.0 GHz (19 levels)
Memory	2 GByte RAM
Storage	32 GByte eMMC
Network	IEEE 802.11b/g/n wireless LAN

By applying $\mathbf{K}_P$ and $\mathbf{K}_I$ to (11), we can get the poles of the closed-loop system, which are 0.38, $0.87 \pm 0.02i$, and 0.93. These poles are all within the unit circle, and this proves that the designed closed-loop system is analytically stable. However, an actual system might manifest different behavior, and hence we need to verify the stability of the system in empirical manner too. In Section 5, we verify the empirical stability of the proposed system through actual evaluation.

4.4. Implementation. The proposed QoS management approach and baselines are implemented by extending QeDB [2]. QeDB internally exploits Berkeley DB as a transaction handler. Berkeley DB [14] provides low-level database features, such as storage management, multithreading for concurrent data processing, locking, and recovery. However, the original Berkeley DB does not support QoS, such as task tardiness and freshness of temporal data. QeDB extends Berkeley DB with QoS management architecture shown in Figure 1. Originally, QeDB only supports the QoD scaling through admission control. This work integrates the QoD scaling with hardware-supported DVFS. In each real-time task, every access to data is performed by invoking Berkeley DB's *put* and *get* methods. These data access methods are instrumented to monitor the response time and data ratio.

5. Evaluation

In this section, we introduce the testbed used for the experiment and present the goals and results of the evaluation.

5.1. Evaluation Testbed and Settings. The hardware platform for the testbed is Odroid-XU3 evaluation board [15]. The specification of the board is shown in Table 1. The Exynos 5422 SoC of Odroid-XU3 has 4 Cortex-A15 cores and 4 Cortex-A7 cores. During the evaluation, 4 Cortex-A7 cores are turned off to exclude the effect of heterogeneous cores. Exynos 5422 has 19 DVFS voltage/frequency steps. The power consumption of the system is measured in real time using Odroid Smart Power [15].

For performance evaluation, we simulate the adapted search-and-rescue scenario from [16] on our testbed. In the scenario, a mobile device, carried by a firefighter, collects streams of sensor readings from nearby sensors. The Odroid-XU3 device is used to simulate fire-fighter's mobile device. Sensor streams from the building are simulated by 3.0 GHz quad-core i-7 Linux desktop. The sensor readings were obtained from realistic simulation using CFAST (the Consolidated Model of Fire and Smoke Transport) simulator [17]. Total 1024 sensors are recorded using the simulation, and each sensor's reporting period follows the uniform distribution ranging from 1 to 10 seconds. During the evaluation, the desktop sends sensor streams from the trace to the mobile device. When a new sensor reading arrives to the device, an update transaction is invoked to store the sensor data.

Table 2: Tested approaches.

Open	Pure Berkeley DB with *OnDemand* DVFS governor
DVFSonly	Embedded database supporting QoS via DVFS control
QoDxxx	Embedded database supporting QoS via QoD scaling control; CPU core frequency is set to xxx Mhz
MIMO	Embedded database supporting QoS with MIMO control of DVFS and QoD scaling

Table 3: Transaction workload types.

None	No interfering best-effort transactions in the other CPU cores.
C90-D10	Computation-intensive workload ($R^{\text{norm}}_{\text{comp}} : R^{\text{norm}}_{\text{data}} = 9 : 1$).
C50-D50	Balanced workload ($R^{\text{norm}}_{\text{comp}} : R^{\text{norm}}_{\text{data}} = 5 : 5$).
C10-D90	Data-intensive workload ($R^{\text{norm}}_{\text{comp}} : R^{\text{norm}}_{\text{data}} = 1 : 9$).

At the mobile device, one Cortex-A15 core is assigned for real-time transactions/tasks as shown in Figure 2. A real-time transaction is invoked periodically on every 100 ms to simulate the real-time analysis of the building state such as the direction of fire, possibility of explosion, and safe retreat paths. We set the real-time transaction's workload to have $R^{\text{norm}}_{\text{comp}} : R^{\text{norm}}_{\text{data}} = 5 : 5$. The deadline of the real-time transaction is set to 50 ms. The slack time is used for aperiodic jobs, such as updating the GUI and updating sensor data. The minimum QoD is set to 0.5, implying that maximum 50% of incoming sensor updates can be dropped.

The other 3 CPU cores are assigned for aperiodic best-effort transactions/tasks. These best-effort transactions are supposed to generate various intercore interferences according to workload types. Table 3 shows the workload types of best-effort transactions with different ratios between $R^{\text{norm}}_{\text{comp}}$ and $R^{\text{norm}}_{\text{data}}$. *C90-D10* is the most computation-intensive, and, conversely, *C10-D90* is the most data-intensive. Each best-effort transaction's $R^{\text{norm}}_{\text{data}}$ and $R^{\text{norm}}_{\text{comp}}$ are adjusted by changing the number of data object accesses and the loop counts of a dummy computation loop. However, all transactions are configured to have almost equal nominal response time R^{norm}, which is $R_{\text{data}} + R_{\text{comp}}$. At each core, a best-effort transaction is invoked continuously, and its consecutive invocations are separated by a uniformly distributed time interval between 50 ms and 150 ms.

The real-time transactions and the best-effort transactions are assigned to their respective CPU cores using the processor affinity feature of Linux. We do not assign particular CPU cores to update transactions. Hence, update transactions can be assigned to any CPU cores according to underlying operating system's scheduling policy.

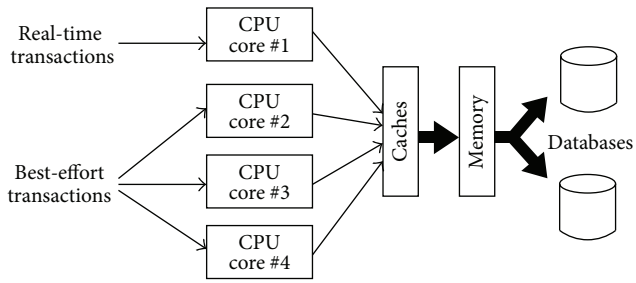
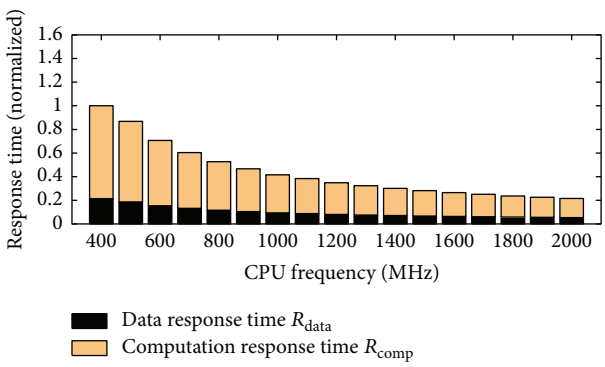
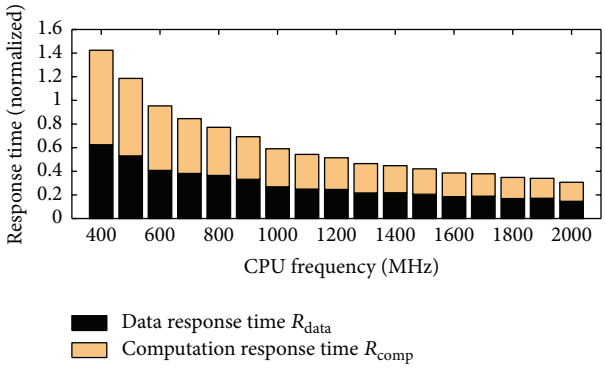


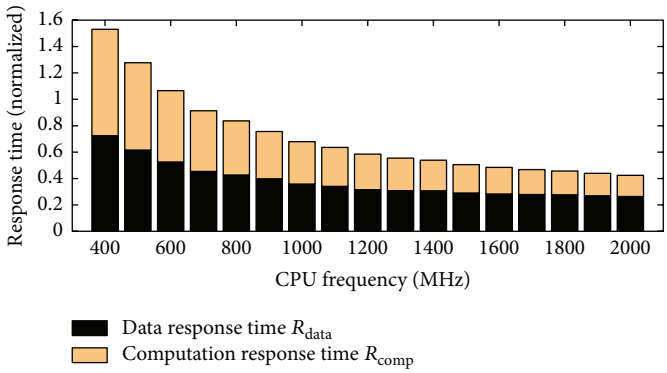
Figure 2: Concurrent database accesses.



(a) Real-time transaction run at CPU core 1 without interfering CPU cores



(b) CPU core 2 interferes real-time transactions at core 1



(c) CPU cores 2–4 interferes real-time transactions at core 1

Figure 3: Task response time with varying CPU speed (QoD = 100%).

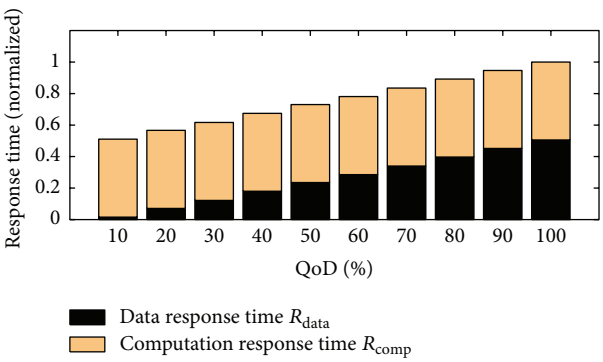


Figure 4: Task response time with varying QoD while the best-effort transactions run at 3 CPU cores.

5.2. Evaluation Goals and Baselines. The objectives of the performance evaluation are (1) to verify that the proposed approach can support the QoS specification under various conditions and (2) to test the effectiveness of the proposed QoS management approach.

For the first objective, we investigate the behavior of the proposed system under various conditions, where a set of parameters are varied. We vary the following parameters: (1) the workload characteristics of interfering tasks and (2) the number of interfering CPU cores. For the second objective, we compare the proposed QoS management approach with several state-of-the-art baseline approaches. For performance evaluation, we consider 4 approaches shown in Table 2. *Open* is the Berkeley DB without QoS support. In Open,

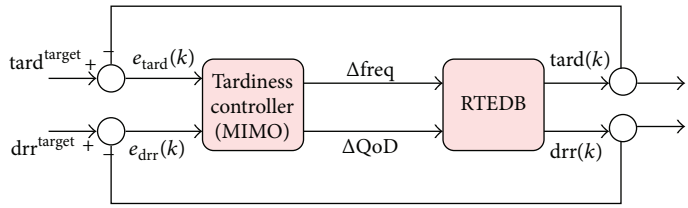


Figure 5: Tardiness control loop.

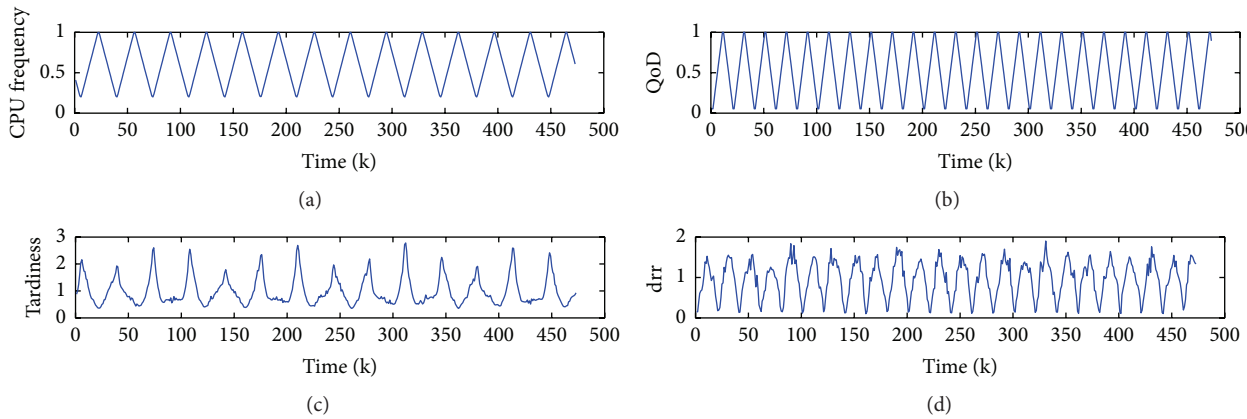


Figure 6: System identification.

the operating system's DVFS governor is set to *OnDemand*, in which the CPU frequency is adjusted to maintain its CPU utilization within the boundary between 20% and 90%. Thus, Open represents the state-of-the-art embedded databases with nominal power management support from underlying operating systems. DVFSonly and QoDxxx represent QeDB supporting transaction tardiness using a single input/single output (SISO) controller. In *DVFSonly* and *QoDxxx*, the tardiness of real-time transactions is controlled only through DVFS and QoD scaling, respectively. Since QoD scaling does not adjust CPU frequency dynamically, QoDxxx's CPU frequency is set to xxx MHz. Finally, *MIMO* is the proposed QoS management approach that supports the transaction tardiness using the MIMO controller integrating DVFS and QoD scaling.

5.3. Average Performance. In this experiment, the average performance of the proposed approach is investigated under various conditions.

5.3.1. Data-Intensive versus Computation-Intensive Workloads. In this experiment, we test the performance of each approach when different workloads, shown in Table 3, are applied to interfere the real-time transactions in one CPU core.

Figure 7 shows the results. As shown in Figure 7(a), both DVFSonly and MIMO closely support the target tardiness of real-time transactions in all interfering workload types. In contrast, Open and QoDxxx do not satisfy the tardiness goal in most interfering workload types. QoD scaling approaches cannot achieve the tardiness goal since, as shown in Figure 7(b); their QoD is saturated at either the minimum, which is 0.5, or the maximum, which is 1. This result demonstrates the limitation of scaling QoD. For DVFSonly and MIMO, the target tardiness is satisfied at the cost of increased CPU frequency as shown in Figure 7(c). In particular, the CPU frequency of DVFSonly increases rapidly as the more data-intensive workloads are applied. This shows that intercore interferences for accessing data have significant impact on the tardiness of real-time transactions. In contrast, MIMO's CPU frequency increases slowly as the workload becomes more data-intensive. This is because MIMO exploits not just DVFS but also QoD scaling. Figure 7(b) shows that more QoD degradation occurred in MIMO as the workloads become more data-intensive. It should be noted that MIMO's QoD is saturated at the minimum, which is 0.5, when *C10-D90* workload is applied. However, unlike QoDxxx, MIMO achieves the tardiness goal since it can exploit DVFS as another control knob.

Figure 7(d) shows the average power consumption of different approaches. When no interfering workload is applied, the power consumption of all approaches, except QoD1800, is not much different. However, as more data-intensive workloads are applied, the power consumption of Open and DVFSonly increases rapidly. For example, Open consumes about 2.7 times more power when C10-D90 workload is applied. This shows that intercore interferences result in significant power consumption. Unlike Open and DVFSonly, however, MIMO's power consumption increases slowly compared to other approaches. This is because MIMO can maintain relatively lower CPU frequency by reducing intercore interferences using QoD scaling.

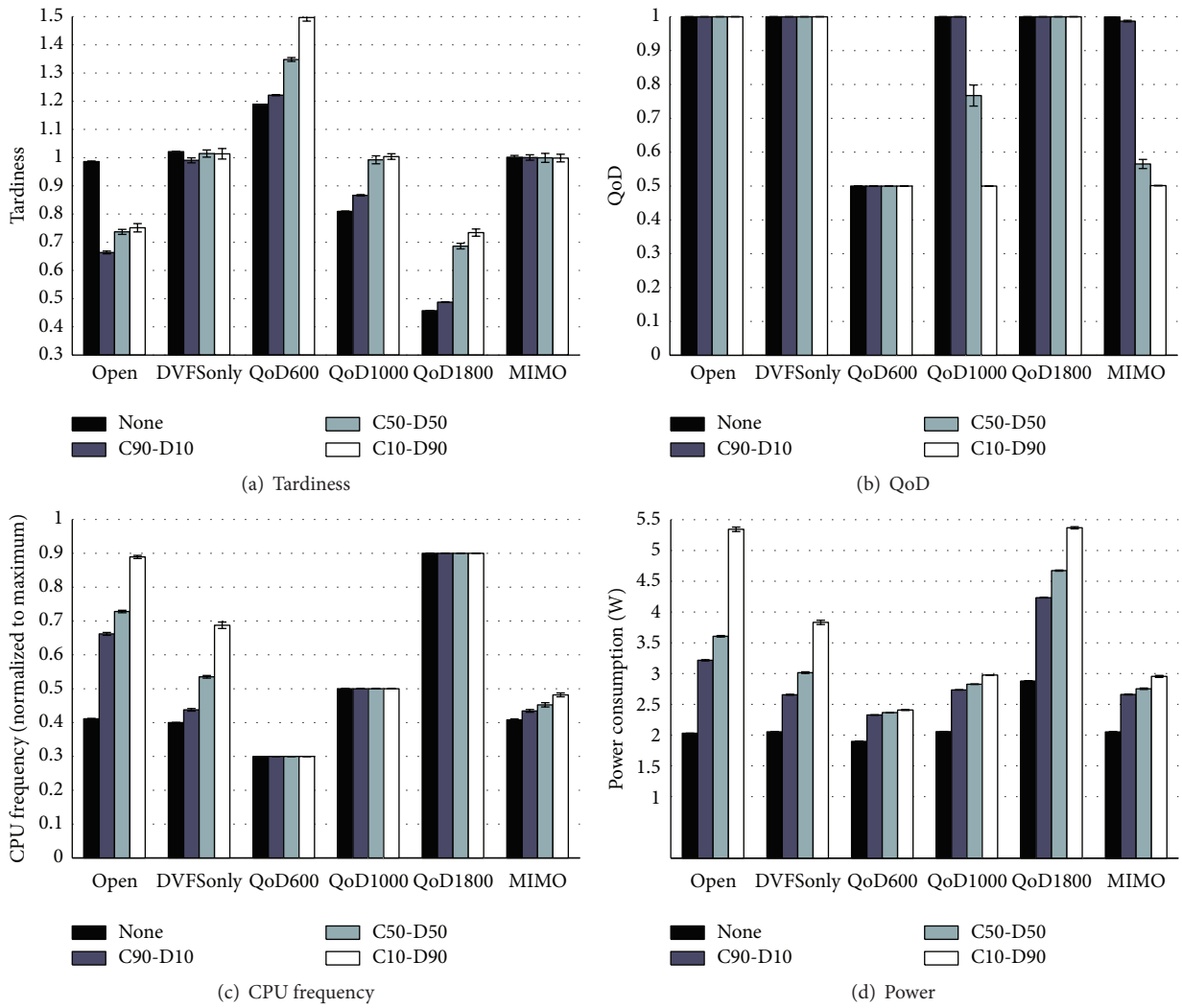


Figure 7: Average performance with varying interfering workload patterns.

5.3.2. Varying Number of Interfering CPU Cores. In this experiment, we change the number of interfering CPU cores while real-time transactions are executed in one CPU core.

Figure 8 shows the result when computation-intensive workload *C90-D10* is applied. In Figure 8, increasing the number of interfering CPU cores has not much impact on the performance of real-time transactions. For instance, each approach, except Open, shows very similar tardiness, QoD, and CPU frequency regardless of the number of interfering CPU cores. Further, in Figure 8(d), the power consumption is gradual and proportional to the number of interfering CPU cores. For instance, in Figures 8(b) and 8(c), when 3 CPU cores are used to interfere real-time transactions, MIMO maintains the maximum QoD while CPU frequency is increased no more than 5%. These results demonstrate that when workloads are computation-intensive, the chances of intercore interferences are low, and the power consumption is proportional to the number of active CPU cores.

Figure 9 shows the results when *C10-D90*, which is data-intensive, is applied. The result shows that increasing the number of interfering CPU cores has significant impact on the performance when the workload is data-intensive. For instance, DVFSonly requires about 72% higher CPU frequency to achieve the tardiness goal when 3 CPU cores are used to interfere real-time transactions. Further, in Figure 9(d), the power consumption increases exponentially for Open and DVFSonly. In contrast, MIMO requires less than 20% increase of CPU frequency at the cost of degrading QoD to the minimum to achieve the tardiness goal. By combining DVFS and QoD scaling, MIMO incurs gradual power increases. This is because MIMO reduces the intercore interferences by decreasing QoD as shown in Figure 9(b).

5.4. Transient Performance. For real-time applications, average performance is not enough to describe their dynamic behavior. Transient performance such as settling time and overshoot should be small enough. In this experiment, we introduce sudden intercore interferences in order to observe the transient behavior of the tested approaches. Initially, real-time transactions are running in one CPU core without

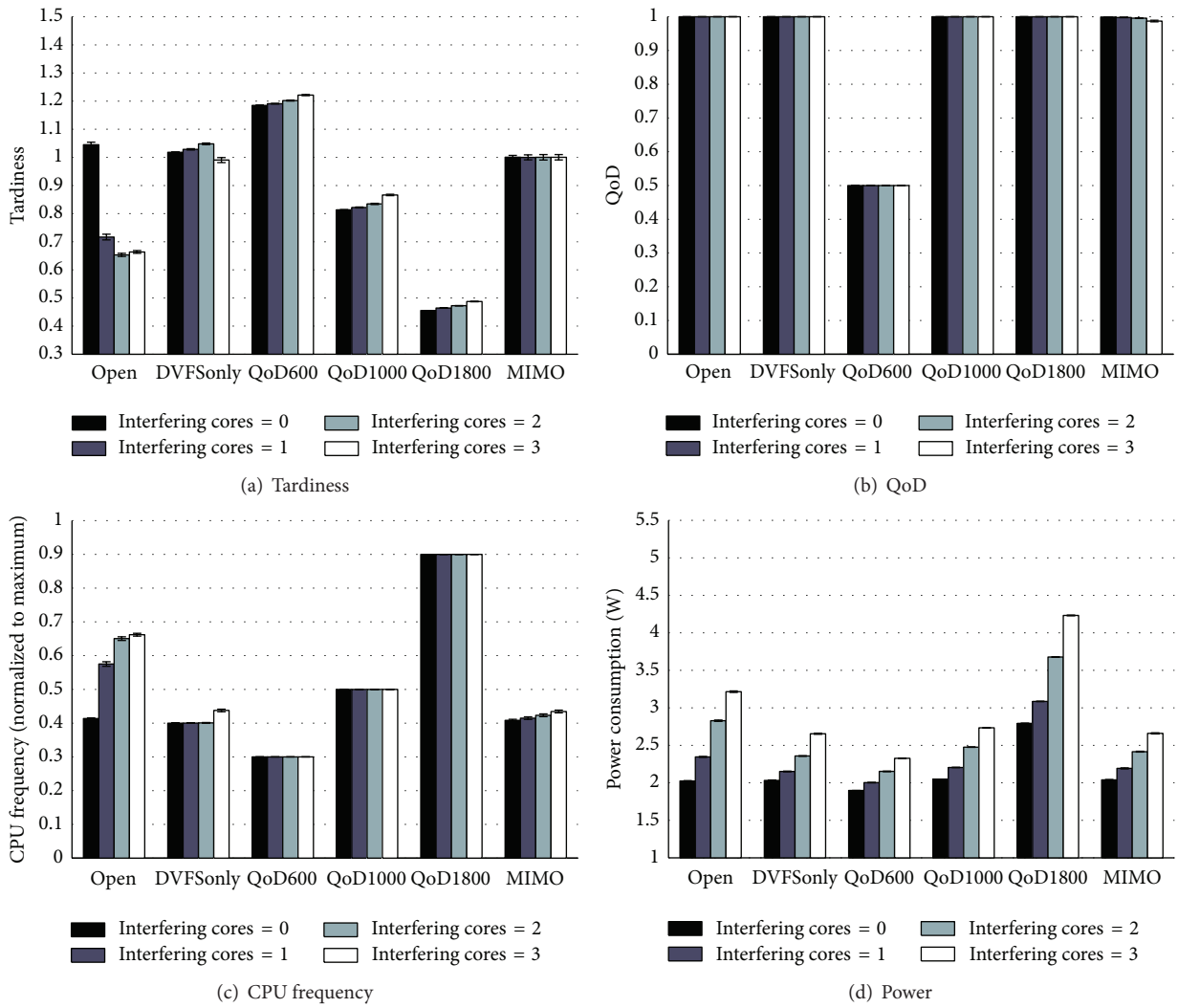


Figure 8: Average performance with varying number of interfering cores (C90-D10 workload is applied).

interferences from the other CPU cores. At the 150th sampling instant, disturbance is introduced by executing best-effort transactions in the other 3 CPU cores. The disturbance persists until the 400th sampling period. The best-effort transactions' workload type is *C50-D50*.

Figure 10 shows the transient behavior of the tested approaches. All approaches, except Open in Figure 10(a), support the desired tardiness using the QoS management architecture of QeDB. These approaches react against the disturbance within 3 sampling periods to achieve the target transaction tardiness. Their overshoots, which are the maximum deviations from the QoS goal, are less than 20%. In Figure 10(b), DVFSonly supports the desired tardiness by increasing CPU frequency by 37%. In Figure 10(c), the QoD1000 does not achieve the target tardiness initially because its QoD is saturated at the maximum. However, while the disturbance is injected, it achieves the target tardiness by lowering QoD. This shows that QoD saturation severely limits the applicability of the QoD scaling technique. Both DVFSonly and QoD1000 do not control drr, and hence drr increases significantly while the disturbance is injected. For instance, DVFSonly's drr increases from 0.28 to 0.44 during the disturbance period. This high drr implies that the real-time transactions' data accesses are delayed due to intercore interferences. In MIMO, we can control drr by setting $\text{drr}^{\text{target}}$ properly. In Figures 10(d) and 10(e), $\text{drr}^{\text{target}}$ is set to 0.28 and 0.40, respectively. According to $\text{drr}^{\text{target}}$, MIMO shows different behavior. When $\text{drr}^{\text{target}}$ is 0.28, which is drr^{norm}, MIMO's controller maintains $\text{drr}^{\text{target}}$ by significantly lowering QoD against the disturbance. On the other hand, the increase of CPU frequency is less than 10%. This means MIMO exploits QoD scaling more aggressively since the transactions are tardy due to intercore interferences. If a user wants to maintain high QoD, MIMO can be configured to resemble DVFSonly by setting $\text{drr}^{\text{target}}$ high. In Figure 10(e), MIMO's $\text{drr}^{\text{target}}$ is set to 0.40 and its reaction against the disturbance is similar to DVFSonly's. When $\text{drr}^{\text{target}}$ is 0.40, MIMO maintains QoD as high as 0.96 against the disturbance. The target tardiness is mostly achieved by increasing CPU frequency; the CPU frequency is increased by about 34%.

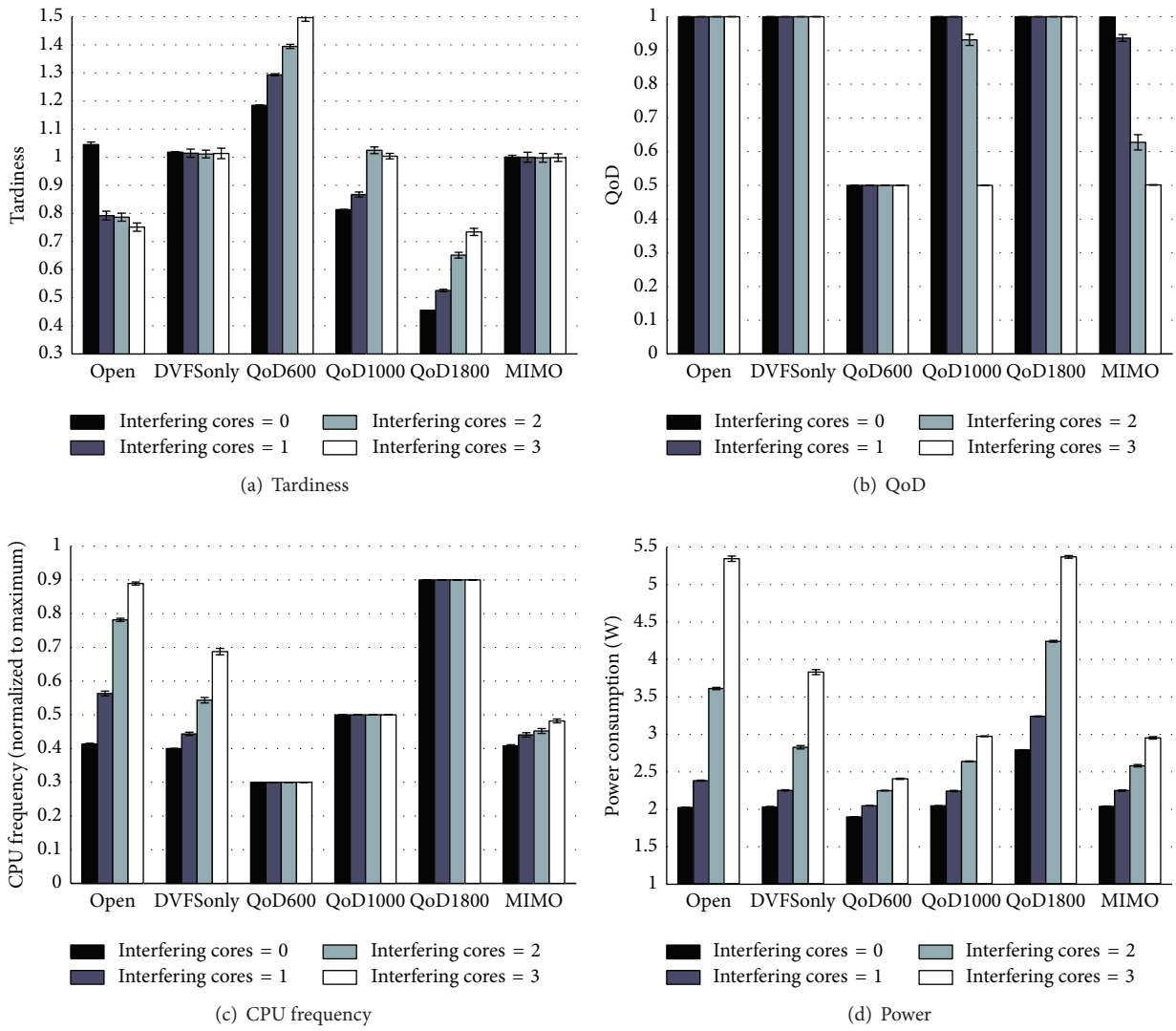
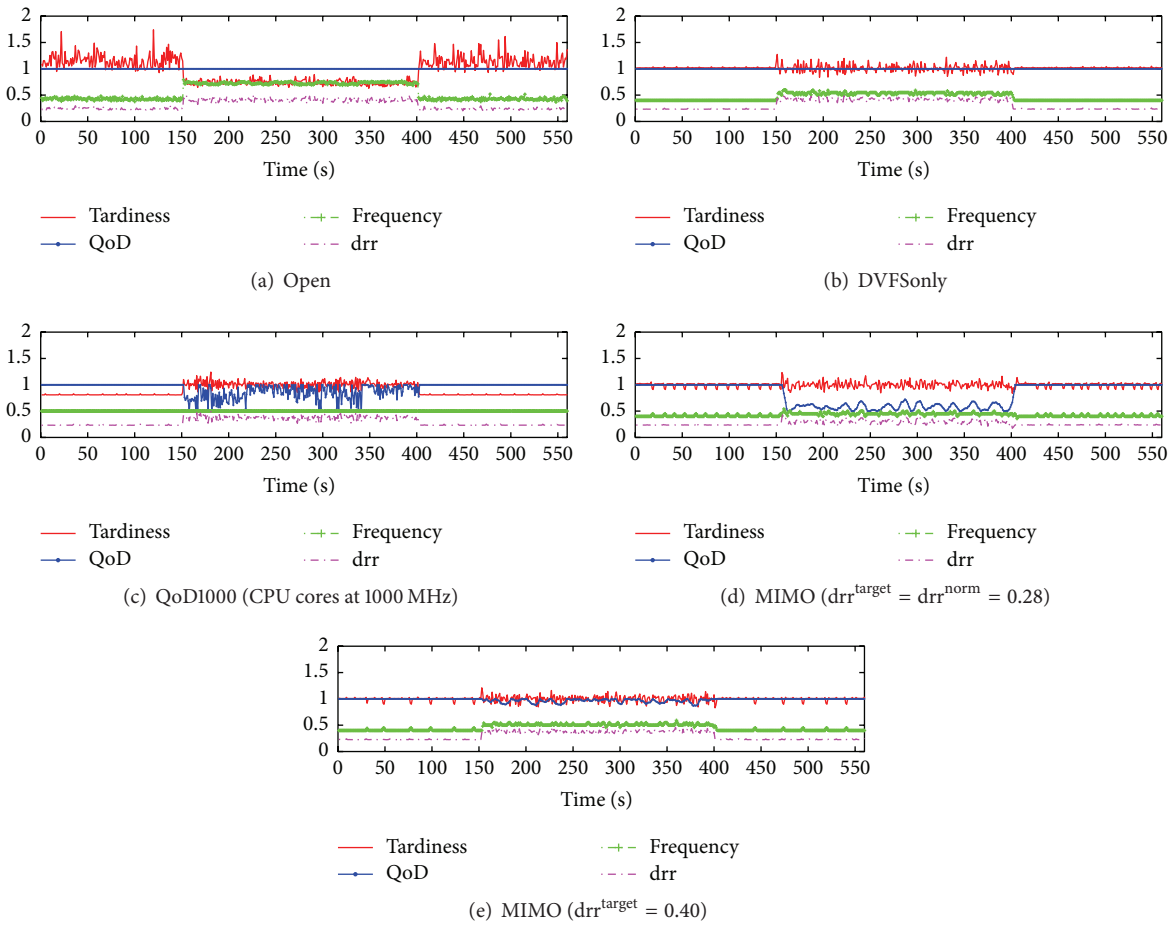


Figure 9: Average performance with varying number of interfering cores (C10-D90 workload is applied).

6. Related Works

Prior research demonstrates that, in multicore environments, the contention for shared resources might cause performance anomalies [7–9]. In particular, existing databases show poor performance in multicore systems due to the interference between cores to access data. Hence, developing databases for multicore machines has drawn intense research effort [3–6]. Papadopoulos et al. proposed to exploit *helper cores* to efficiently prefetch data needed by working threads [4]. Johnson et al. removed locking contention from existing storage managers [5]. Salomie et al. proposed to partition the multicore machine and used existing databases in a replicated configuration as if the multicore machine was a distributed system [3]. These works target high-performance server environments and their primary goal is to achieve high throughput. Further, they try to change the implementation of a specific DBMS to better exploit multiple cores. Unlike these works, we focus on supporting predictable data access response time in multicore embedded systems and our approach is not tailored for specific DBMS implementation.

QoD scaling via active load shedding [18] has been applied to real-time databases (RTDBs) [19, 20] and stream management systems (DSMSs) [21, 22] for performance management at runtime. A common approach for load shedding is to drop incoming data updates under overloading situation. For instance, Amirijoo et al. exploited imprecise computation on data to allow data objects to deviate from true value to a certain degree [20]. However, the applicability of load shedding is highly application-dependent and its range is limited by applications' requirements. Hence, for many applications, QoD scaling via load shedding is hard to be a primary control knob to support the desired performance. In our work, we use QoD scaling together with DVFS to reduce potential intercore interferences. These two control knobs complement each other.



(a) Open

(b) DVFSonly

(c) QoD1000 (CPU cores at 1000 MHz)

(d) MIMO ($drr^{target} = drr^{norm} = 0.28$)

(e) MIMO ($drr^{target} = 0.40$)

FIGURE 10: Transient behavior.

There has been a large amount of previous works to use DVFS to save processor power while still supporting the timeliness of tasks [10, 23, 23–25]. Yao et al. first gave theoretic exploration of DVFS for real-time tasks considering a set of aperiodic tasks [26]. For non-real-time systems without specific deadlines, performance metrics such as CPU utilization have been used [27, 28]. These approaches exploit a simple feedback mechanism based on the chosen performance metric to control processor frequency dynamically. In this work, we showed that the effectiveness of DVFS is diminished when tasks contend to access non-CPU resources in multicore systems. To address this problem, we integrate DVFS with QoD scaling.

Because of its robustness against unpredictable workloads, feedback control theory has been extensively applied for the QoS management of various computing systems, including web servers [29], caching service [30], and email server [31]. Feedback control theory has also been used to support the timeliness of real-time transactions in real-time data services [2, 19, 32]. However, these works do not consider modern multicore environments. In this work, we proposed a novel feedback control mechanism to support transaction tardiness while reducing potential intercore interferences of multicore embedded systems.

7. Conclusions

In this paper, we proposed the QoS management architecture for data-intensive real-time applications running on multicore-based embedded platforms. A novel multi-dimensional feedback control architecture is proposed to support the timeliness of transactions while reducing the effect of potential intercore interferences. Through the proposed control architecture, two distinctive control knobs, which are DVFS and QoD scaling, are controlled simultaneously to support the QoS goals in an efficient and robust manner. We showed the feasibility of the proposed QoS management scheme by implementing and evaluating it on a modern multicore mobile platform. Our evaluation results show that our approach achieves the target QoS goals, such as task tardiness and data quality, while consuming less energy compared to baseline approaches.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2014R1A1A1005781).

References

[1] J. Levinson, J. Askeland, J. Becker et al., "Towards fully autonomous driving: systems and algorithms," in *Proceedings of the IEEE Intelligent Vehicles Symposium (IV '11)*, pp. 163–168, Baden-Baden, Germany, June 2011.

[2] W. Kang, S. H. Son, and J. A. Stankovic, "Design, implementation, and evaluation of a QoS-aware real-time embedded database," *IEEE Transactions on Computers*, vol. 61, no. 1, pp. 45–59, 2012.

[3] T.-L. Salomie, I. E. Subasu, J. Giceva, and G. Alonso, "Database engines on multicores, why parallelize when you can distribute?" in *Proceedings of the 6th ACM Conference on Computer Systems (EuroSys '11)*, pp. 17–30, ACM, April 2011.

[4] K. Papadopoulos, K. Stavrou, and P. Trancoso, "HelperCoreDB: exploiting multicore technology to improve database performance," in *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS '08)*, pp. 1–11, Miami, Fla, USA, April 2008.

[5] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi, "Shore-mt: a scalable storage manager for the multicore era," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT '09)*, pp. 24–35, ACM, Saint Petersburg, Russia, March 2009.

[6] N. Hardavellas, I. Pandis, R. Johnson, and N. Mancheril, "Database servers on chip multiprocessors: limitations and opportunities," in *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR '07)*, pp. 79–87, Asilomar, Calif, USA, January 2007.

[7] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *Proceedings of the 44th Annual IEEE/ACM Symposium on Microarchitecture (MICRO '44)*, pp. 374–385, ACM, Porto Alegre, Brazil, December 2011.

[8] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "MemGuard: memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Proceedings of the IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS '13)*, pp. 55–64, IEEE, Philadelphia, Pa, USA, April 2013.

[9] H. Shah, K. Huang, and A. Knoll, "Timing anomalies in multi-core architectures due to the interference on the shared resources," in *Proceedings of the 19th Asia and South Pacific Design Automation Conference (ASP-DAC '14)*, pp. 708–713, Singapore, January 2014.

[10] T. Ishihara and H. Yasuura, "Voltage scheduling problem for dynamically variable voltage processors," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 197–202, IEEE, Monterey, Calif, USA, August 1998.

[11] L. Bertini, J. C. B. Leite, and D. Mossé, "Generalized tardiness quantile metric: distributed dvs for soft real-time web clusters," in *Proceedings of the 21st Euromicro Conference on Real-Time Systems (ECRTS '09)*, pp. 227–236, IEEE, Dublin, Republic of Ireland, July 2009.

[12] Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. M. Tilbury, "Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server," in *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS '02)*, pp. 219–234, IEEE, April 2002.

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

[14] Oracle Berkeley DB, 2014, http://www.oracle.com/.

[15] HardKernel Products, 2015, http://www.hardkernel.com.

[16] K. Sha, W. Shi, and O. Watkins, "Using wireless sensor networks for fire rescue applications: requirements and challenges," in *Proceedings of the IEEE International Conference on Electro/Information Technology*, pp. 239–244, East Lansing, Mich, USA, May 2006.

[17] R. D. Peacock, W. W. Jones, P. A. Reneke, and G. P. Forney, *CFAST-Consolidated Model of Fire Growth and Smoke Transport (Version 6) User's Guide*, US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 2005.

[18] N. Tatbul, U. Çetintemel, and S. Zdonik, "Staying FIT: efficient load shedding techniques for distributed stream processing," in *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, pp. 159–170, VLDB Endowment, Vienna, Austria, September 2007.

[19] J. Oh and K.-D. Kang, "A predictive-reactive method for improving the robustness of real-time data services," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 5, pp. 974–986, 2013.

[20] M. Amirijoo, J. Hansson, S. Gunnarsson, and S. H. Son, "Enhancing feedback control scheduling performance by online quantification and suppression of measurement disturbance," in *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '05)*, pp. 2–11, San Francisco, Calif, USA, March 2005.

[21] D. J. Abadi, Y. Ahmad, M. Balazinska et al., "The design of the borealis stream processing engine," in *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR '05)*, Asilomar, Calif, USA, January 2005.

[22] StreamBase, http://www.streambase.com/.

[23] J.-P. Halimi, B. Pradelle, A. Guermouche et al., "Reactive DVFS control for multicore processors," in *Proceedings of the IEEE and Internet of Things, IEEE International Conference on and IEEE Cyber, Physical and Social Computing, Green Computing and Communications (iThings/CPSCom-GreenCom '13)*, pp. 102–109, IEEE, Beijing, China, August 2013.

[24] J.-J. Chen and C.-F. Kuo, "Energy-efficient scheduling for real-time systems on dynamic voltage scaling (DVS) platforms," in *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '07)*, pp. 28–38, IEEE, Daegu, Republic of Korea, August 2007.

[25] S. Li and F. Broekaert, "Low-power scheduling with DVFS for common RTOS on multicore platforms," *ACM SIGBED Review*, vol. 11, no. 1, pp. 32–37, 2014.

[26] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced CPU energy," in *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pp. 374–382, IEEE, Milwaukee, Wis, USA, October 1995.

[27] V. Pallipadi and A. Starikovskiy, "The ondemand governor," in *Proceedings of the Linux Symposium*, vol. 2, pp. 215–230, Ottawa, Canada, 2006.

[28] B. Wu and P. Li, "Load-aware stochastic feedback control for DVFS with tight performance guarantee," in *Proceedings of the 20th IEEE/IFIP International Conference on VLSI and System-on-Chip (VLSI-SoC '12)*, pp. 231–236, IEEE, Santa Cruz, Calif, USA, October 2012.

[29] C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son, "A feedback control approach for guaranteeing relative delays in web servers," in *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS '01)*, pp. 51–62, June 2001.

[30] Y. Lu, T. F. Abdelzaher, and A. Saxena, "Design, implementation, and evaluation of differentiated caching services," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 5, pp. 440–452, 2004.

[31] 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 Systems*, vol. 23, no. 1-2, pp. 127–141, 2002.

[32] K.-D. Kang, S. H. Son, and J. A. Stankovic, "Managing deadline miss ratio and sensor data freshness in real-time databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 10, pp. 1200–1216, 2004.