Reducing Deadline Misses and Power Consumption in Real-Time Databases

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

***Abstract*—In data-intensive real-time embedded applications, it is desirable to process data service requests in a timely manner using fresh data, consuming less power. However, related work is relatively scarce. In this paper, we present an effective approach to decrease both the deadline miss ratio and power consumption by merging similar real-time transactions, while systematically adapting the data freshness. In a simulation study, our approach considerably reduces deadline misses and power consumptions compared to the state-of-the-art baselines, supporting the required data freshness.**

I. Introduction

The demand for real-time data services in embedded systems is increasing. For example, small-footprint embedded databases, e.g., [1], [2], [3], [4], are developed to support real-time embedded applications, e.g., energy-efficient avionics, medical devices, firefighting, real-time engine diagnosis, and traffic management by roadside units. In such applications, it is important to process real-time transactions and queries (read-only transactions) in a timely manner using fresh (i.e., temporally consistent) data that represent the current real-world status, while saving power.

However, achieving this objective is challenging. It is known that optimal energy scheduling is intractable even in a single processor with dynamic voltage and frequency scaling (DVFS) and a low-power idle state [5]. The problem becomes harder in RTDBs due to additional challenges. The timeliness of user transactions, data freshness, and power saving requirements may compete with each other. In RTDBs, usually dedicated update transactions are used to periodically refresh temporal data, e.g., sensor readings, to ensure the freshness. User transactions analyze them to provide real-time data services. If higher priority is given to user transactions, their timeliness can be improved at the cost of the decreased data freshness or vice versa. Simply consuming more computational resources and power to support the timeliness and data freshness is not desirable either. User transactions may arrive aperiodically depending on the current real-world status, e.g., the traffic or weather conditions. Also, transactions may get aborted and restarted due to data/resource conflicts unknown *a priori*, incurring deadline misses [6]. Thus, in this paper, we explore effective heuristics to decrease both the deadline miss ratio and power consumption in RTDBs.

In a database system, multiple queries often access common data. For example, traffic data at busy intersections or severe weather data are usually accessed more often. In fact, large data access skews are prevalent [7]. For example, the well-known 80/20 rule indicates that 20% of data are accessed for 80% of accesses. Thus, multiple queries can be merged into a single query to avoid repeated data access and processing. However, a direct application of this approach may result in many deadline misses in RTDBs, since queries should be delayed to get batched together. For instance, Lang et al. [8] intentionally delay queries to combine them in non-RTDBs. They decrease the energy consumption by up to 54% at the cost of a 43% increase in the average response time.

To combine real-time queries without jeopardizing the timeliness, our approach schedules the transaction with the highest priority (e.g., the earliest deadline transaction) first, while scanning the ready queue sorted in non-ascending priority order backwards to merge similar user transactions together in the background.[1] Thus, our approach avoids duplicate data access and processing as much as possible without disrupting high priority transactions at or near the head of the queue. Thus, it is compatible with any priority-driven scheduling algorithm.

To further decrease the deadline miss ratio and power consumption, we also extend the adaptive freshness management scheme [9]. It gracefully adapts the data freshness within a range specified by a database administrator (DBA) or an RTDB application designer aware of the data needs in a specific RTDB application, e.g., traffic control or fire detection, by leveraging the (near) future data access pattern analyzed by our real-time query aggregation scheme, incurring little additional overhead.

Our real-time query aggregation and adaptive data freshness management schemes cooperate with each other to mitigate the additional RTDB challenges discussed before. First, our query aggregation technique merges multiple aperiodic user transactions that access common data when they arrive simultaneously upon, for example, a traffic incident or fire breakout. By decreasing the user transaction workload, it meets more user transaction deadlines, incurring less data/resource conflicts among them. Second, our approach for adaptive freshness management decreases the update load, which could be substantial in RTDBs [10], [11], [9], to further decrease

[1] Although read operations of transactions as well as queries are merged in our approach, our approach is called real-time query aggregation to be consistent with the well accepted term, query aggregation [8].

deadline misses and power consumptions. Moreover, by reducing both user and update workloads, our approaches alleviate data/resource contention between user and update transactions too. In sum, they considerably relieve the tension among the competing requirements for the user transaction timeliness, data freshness, and power saving in RTDBs.

Race-to-idle and never-idle are two major approaches for power saving. In the race-to-idle method, the processor runs at the maximum speed to enter a low-power idle state as early as possible. On the other hand, in the never-idle scheme, the processor speed is continuously adapted to meet the performance requirement with less power consumptions. In this paper, we take a race-to-idle approach to reduce the processor power consumption due to the decreasing effectiveness of the never-idle approach based on, for example, DVFS [12]. At runtime, our approach processes real-time update and user transactions at the highest processor speed, while reducing the RTDB workload via real-time query aggregation and data freshness adaptation. When the RTDB is idle with no update or user transaction to execute, our approach switches to a low-power processor idle mode based on the idle interval length estimated considering the data update periods and recent user transaction arrival pattern.

Despite the importance, related work on RTDB power management is relatively scarce [3], [2]. A summary of our key contributions and novelty follows:

- Our real-time query aggregation scheme reduces both the miss ratio and power consumption rather than doing trade-offs between them.
- The miss ratio and power consumption are further reduced by systematically adapting the freshness by exploiting not only the current but also the future data access pattern found by the real-time query aggregation scheme.
- It is generally applicable to RTDB power management, since it does not assume a constrained or specialized transaction/query model.
- It requires neither any special hardware nor extensive system modeling and tuning. It only needs low-power idle states supported by almost all processors today.
- Our approach is configurable and relatively easy to use. For instance, a DBA may choose to only support real-time query aggregation to save power, while avoiding any freshness adaptation. Also, s/he just needs to set only a few parameters for real-time query aggregation and freshness adaptation considering RTDB application semantics.

For performance evaluation, we have done a simulation study modeled after real-world RTDB applications, e.g., air traffic control, fire detection, and engine diagnosis [10], [11], [2]. Our approach decreases the deadline miss ratio and dynamic power consumption compared to the tested baselines, which represent and enhance state-of-the-art RTDBs, by up to approximately 38% and 52%, respectively. Even when the allowed real-time query aggregation and freshness adaption are limited to minimal degrees,[2] our approach reduces the miss ratio and power consumption by up to roughly 18% and 37%, respectively, while supporting the desired data freshness.

The remainder of this paper is organized as follows. Related work is discussed in Section II. The supported transaction types, data freshness requirements, and the power-aware RTDB architecture are discussed in Section III. In Section IV, our approach for deadline miss ratio and power consumption reductions in RTDBs is discussed. In Section V, the performance of our approach and baselines is evaluated via an extensive simulation study. Finally, the paper is concluded and future work is discussed in Section VI.

II. Related Work

Generally, research on power/energy management in database systems is relatively new. It is known that [8] is the first to provide concrete techniques for energy-efficient query processing. It explicitly delays queries for combined processing, while supporting DVFS. It has been followed by other projects on database power/energy management in data centers including [13], [14], [15], [16]. In these approaches, the database energy consumption is reduced for the increased response time or decreased throughput. However, naively slowing real-time transactions down in RTDBs for energy efficiency may incur many deadline misses as discussed before. Neither do they support real-time transaction scheduling, concurrency control, or data temporal consistency. Sensor network databases [17], [18] support relatively simple in-network data processing, e.g., sensor data aggregation, to mainly optimize communication costs. In [19], efficient data freshness management is explored when data are retrieved from wireless sensors in sequence specialized for rescue or tactical situations. However, real-time query aggregation and RTDB power management are not considered.

Surprisingly little work has been done on RTDB power management. A novel work [2] is the first to support the desired power consumption and I/O deadline miss ratio, via multi-input, multi-output (MIMO) control, in an RTDB based on flash memory. In [3], a control theoretic approach is developed to support the timeliness of a single periodic real-time transaction run concurrently with a few interfering non-real-time transactions in an embedded database. The power consumption is decreased via dynamic frequency scaling and sensor data dropping in the feedback loop. However, the general applicability of [3] is limited, because it has a constrained transaction model that supports one real-time user transaction only. In [13], DVFS based on proportional and integral (PI) control is applied to decrease the power consumption for I/O bound queries, while supporting the desired throughput in a non-RTDB. However, real-time query aggregation and data freshness adaptation are not considered to reduce the RTDB power consumption. These projects [2], [3], [13] essentially take never-idle approaches that require extensive system

[2]In this set-up, the probability for combining two queries is set to 5% and the bounded data freshness degradation is allowed for only 10% of the data objects. A detailed description is given in Section V.

modeling and tuning, which should be repeated in different platforms. Our approach adopts a more general real-time transaction and data model. Also, it directly considers RTDB power management issues via dynamic power management (DPM) more effective than DVFS.

Power-aware real-time scheduling has been well explored. Related work includes [5], [20], [21], [22], [23] just to name a few. A good survey of power management in hard real-time systems is given in [12]. Although the transaction timeliness, data freshness, and power management in RTDBs are not directly considered in these approaches, basic principles could be applied for more effective RTDB power management. Also, our work could benefit more from previous database research. For example, approximate query processing techniques are developed to produce rough results under overload [24], [25]. The miss ratio and power consumption of our approach could be decreased further, if it is combined with approximate query processing. In [10], [11], update transactions are deferred as much as possible to reduce the workload, meeting the data temporal consistency. Thus, our work is complementary to these approaches.

III. Data Types, Transactions, and System Overview

In this section, the data and transaction types and data freshness requirements considered in this paper are described. Also, an overview of our RTDB architecture is given.

A. Data Types, Transactions, and Deadlines

- **Data Types and Freshness:** In our data service model, there are two types of data: *temporal* and *non-temporal* data. Temporal data, e.g., sensor readings, become outdated as time goes by, because the real world status, e.g., traffic or weather state, may continuously change. The temporal consistency between the real world state and the temporal data in the RTDB is maintained based on the *validity intervals* [6]. A temporal data item O_i is associated with a timestamp that indicates the latest update time. It is considered fresh, i.e., temporally consistent, if (current time $-$ timestamp(O_i) $\leq avi(O_i)$) where $avi(O_i)$ is the absolute validity interval of O_i. On the other hand, non-temporal data, e.g., a vehicle registration number, do not become outdated unless users explicitly modify them. Thus, we focus on managing temporal data in this paper.
- **Transaction Types:** In RTDBs, there are two types of real-time transactions: *update* and *user transactions* [6]. A dedicated update transaction periodically updates O_i at every $P_i = 0.5 \times avi(O_i)$ to maintain the freshness according to the half-half principle [6]. Real-time user transactions arrive *aperiodically* to support, for example, driving route and weather information requests. They are allowed to read temporal data and read/write non-temporal data. In general, a real-time transaction T_i reads and writes sets of data R_i and W_i, respectively. If T_i is an update transaction for O_i, the read set $R_i = \emptyset$ and the write set $W_i = \{O_i\}$. On the other hand, if T_i is a user transaction, R_i consists of one or more temporal/non-temporal data. W_i consists of zero or more non-temporal data. (If T_i is a query, $W_i = \emptyset$.)
- **Deadlines:** The relative deadline of an update transaction is equal to its period. The relative deadlines of user transactions are determined by a specific RTDB application of interest, e.g., transportation management or fire detection. If T_i with a relative deadline D_i is released or arrives at time t, its absolute deadline is $t + D_i$. In this paper, real-time transactions are assigned *firm* deadlines. If all the required read/write operations are completed by $t+D_i$, T_i is committed successfully. Otherwise, it is aborted upon the deadline miss to avoid cascading deadline misses due to intensified data/resource contention.

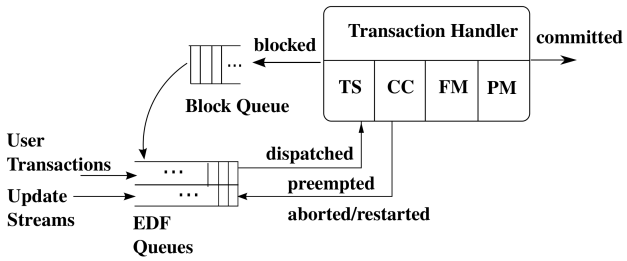


Fig. 1. Power-Aware RTDB Architecture

B. Power-Aware RTDB Architecture

The transaction handler in Figure 1 consists of the transaction scheduler (TS), concurrency controller (CC), freshness manager (FM), and power manager (PM). TS schedules real-time user transactions and data updates. For the clarity of the presentation, in this paper, we employ EDF as the basic scheduling algorithm. In our RTDB architecture shown in Figure 1, two separate EDF queues are used by TS to schedule user and update transactions, respectively. Higher priority is given to update transactions to maintain the data freshness, which is a common practice in RTDBs [6]. Also, we assume that enough resources are available to meet all deadlines of temporal data updates and the freshness requirements enforced by FM.

Thus, in this paper, we mainly focus on reducing the miss ratio of user transactions, while decreasing the power consumption of both user and update transactions via real-time query aggregation and adaptive freshness management performed by TS and FM in our power-aware RTDB architecture that runs on a uniprocessor platform.[3] Also, PM in Figure 1 supports DPM when the system becomes idle.

CC supports the serializability of concurrent transactions. For concurrency control, we support the two phase locking with high priority (2PL-HP) scheme [6]. A data conflict arises, if two transactions access the same data item and at least

[3]A thorough investigation of designing power-aware RTDBs using multi-core processors is reserved for future work discussed in Appendix C.

one of them needs to write it. Under 2PL-HP, a low priority transaction is aborted and restarted upon a data conflict, if it has locked the data causing the (read/write or write/write) conflict. However, it gets blocked, if it is requesting the data already locked by a higher priority transaction in a conflicting manner. A restarted or blocked transaction is moved to the block queue. It is inserted back into the EDF ready queue when the conflicting higher priority transaction(s) commit(s).

IV. Decreasing Deadline Misses and Power Consumptions in RTDBs

Our approach begins to run when the RTDB is initialized. It continues to run until either the DBA explicitly turns it off or shuts the system down. By reducing the user and update workloads, our approach strives to decrease the miss ratio and increase idle intervals to save power. A detailed description follows.

A. Merging Real-Time Queries

Algorithm 1: Real-Time Query Aggregation

input : T_i (i^{th} user transaction in the EDF queue)

```
1  j = i − 1;
2  cnt = 0;
3  while j ≥ 0 and cnt < MaxScan do
4      R_i = Read Set(T_i);
5      R_j = Read Set(T_j);
6      if |R_i ∩ R_j| ≥ θ then
7          R_ij = R_i ∩ R_j;
8      if R_j is merged already then
9          return;
10     i = j;
11     j − −;
12     cnt++;
```

In this paper, we only merge read operations of multiple user transactions. We do not merge update transactions, because each update transaction in an RTDB periodically updates a specific temporal data object in a dedicated manner to maintain the freshness [6]. Neither do we merge the write sets of two user transactions, since any writes done by each transaction should be atomic (all or nothing) and separate from the writes performed by the other transactions [6].

When a user transaction arrives, it is inserted into the i^{th} (≥ 0) place in the EDF queue in Figure 1 based on its deadline. Thus, T_0 is the user transaction with the earliest deadline. In this paper, we support lightweight *incremental* real-time query aggregation. We attempt to aggregate T_i with the transaction(s) in front of it when it is inserted into the EDF queue. Thus, a user transaction with a longer deadline is likely to be aggregated with more user transactions with shorter deadlines. By doing this, we intend to reduce the user transaction load without disrupting transactions with imminent deadlines.

When a user transaction T_i is inserted into the EDF queue, Algorithm 1 is executed for real-time query aggregation. First, R_i and R_j of T_i and T_j where $j = i-1$ in the EDF queue are identified. If $|R_i \cap R_j| \geq \theta$ where θ is the specified threshold, the common data in their read sets are: $R_{ij} = R_i \cap R_j$. In general, a query optimizer in a database system analyzes queries' data accesses for performance optimization. Thus, we exploit the read set information provided by the query optimizer for real-time query aggregation, incurring little additional overhead. Alternatively, real-time transactions are often canned, i.e., predefined, and access specific data elements to enhance the timeliness [6]. In such a case, R_i and R_j are known *a priori*. In both cases, we express R_i and R_j as bit strings of length m that is the maximum transaction size in terms of the total number of data accessed by an arbitrary transaction in the RTDB.[4] We compute R_{ij} by doing a bitwise and operation between R_i and R_j, which is an O(1) time operation.

Second, our algorithm for merging real-time queries check whether R_j has already been merged with the real-time queries ahead of T_j in the EDF queue. If this is true, no more aggregation is needed. Thus, the algorithm returns.

Otherwise, the loop is iterated to further aggregate user transactions for at most $MaxScan$ times where $MaxScan$ is a pre-defined constant to bound the overhead for real-time query aggregation. Therefore, the time complexity of our algorithm for real-time user query aggregation is O(1).

When the user transaction at the head of the EDF queue is executed, the data read by the transaction are shared by the other transactions with later deadlines, which need to access the same data. When a later transaction runs, it uses the common data previously accessed by an earlier deadline transaction as long as they are still fresh. This is another reason to bound the EDF queue scanning for real-time query aggregation by $MaxScan$. Entire EDF queue scanning incurs large overheads. Also, data accessed by earlier transactions may become stale.

B. Adaptive Data Freshness Management

To manage the update workload efficiently, a cost-benefit model for temporal data updates, which is independent of any specific data access pattern, is introduced in [9]. For each temporal data object O_i in the RTDB, the cost is defined as the update frequency, since the computational cost is higher for more frequent updates. The access frequency indicates the benefit of updating O_i. To quantify the cost-benefit relation, the access update ratio (AUR) for O_i, which represents the importance of being fresh, is defined:

$$AUR[i] = \frac{access\ frequency[i]}{update\ frequency[i]} \quad (1)$$

Since temporal data are updated periodically, their update frequencies are already known.

[4] If a certain data item is accessed by T_i, the corresponding bit in R_i is set to 1.

In this paper, we use the *future* access frequency of each temporal data found during real-time query aggregation to update $AUR[i]$ for $O_i \in D$ where D is the set of all temporal data in the RTDB. Thus, the AUR reflects the data access pattern that may vary in time *before* an actual change happens unlike [9], which predicts the future AUR purely based on the recent history.

In our approach, O_i is considered hot, if $AUR[i] \geq 1$. This means the benefit of periodically updating O_i is worth the computational cost represented by the update frequency. Otherwise, it is considered cold. In this paper, D_c and D_h represent the sets of cold and hot data, respectively. Clearly, $D = D_c \cup D_h$ and $D_c \cap D_h = \emptyset$. Also, $|D_c| > |D_h|$, if there is a data access skew. Note that each data item is automatically classified as hot or cold by our approach without requiring any involvement of a DBA or any other user.

Algorithm 2: Adaptive QoD Management

input : D_c, α, σ specified by a DBA

```
1 if t mod P_QoD == 0 then
2     i = 0;
3     while i < |D_c| do
4         P_{i_new} = (1 + σ) × P_i;
5         if P_{i_new} ≤ α × P_{i_min} then
6             P_i = P_{i_new};
7         i++;
```

In this paper, Algorithm 2 is executed at every quality of data (QoD) adaptation period, P_{QoD}, to gracefully adapt the update workload considering the current and near-future data access pattern. In Algorithm 2, $P_{i_{min}}$ is the update period of O_i before any QoD adaptation. If $O_i \in D_c$, the current update period P_i is increased to $(1+\sigma)P_i$ in a QoD adaptation period, if $(1+\sigma)P_i \leq \alpha P_{i_{min}}$ where σ is a parameter used to avoid an abrupt QoD degradation. For example, when σ = 10%, $P_{i_{new}} = 1.1 \times P_i$ after a QoD degradation for O_i. In addition, α (≥ 1) is provided to avoid unbounded QoD degradation. For instance, $P_i \leq 4P_{i_{min}}$ if $\alpha = 4$. Using our approach, a DBA aware of the data needs in a specific RTDB application can set σ and α.

To maintain the freshness of a sensor data item after a possible QoD degradation, we use the *flexible validity interval* (fvi), similar to [9]. Initially, $fvi = avi$ for all data. If the update period P_i for a less critical data object O_i is increased to $P_{i_{new}}$, we set $fvi_{new}(O_i) = 2 \times P_{i_{new}}$ to maintain the freshness of O_i by updating it at every $P_{i_{new}}$. Also, O_i is considered fresh if (current time $-$ timestamp(O_i)) $\leq fvi_{new}(O_i)$.

The current QoD in terms of freshness is defined in the RTDB with N temporal data objects:

$$QoD = \frac{100}{N} \sum_{i=1}^{N} \frac{P_{i_{min}}}{P_{i_{new}}} (\%) \tag{2}$$

Also, the QoD lower bound in a specific RTDB application is:

$$QoD_{LB} = 100 \times \left[(1 - \beta) + \frac{\beta}{\alpha} \right] (\%) \tag{3}$$

where $\beta = |D_c|/|D|$. Thus, $QoD \geq QoD_{LB}$ at any time $t \geq 0$. For example, a DBA can specify that $\beta = 0.4$ such that the QoD of the coldest 40% of the data in the RTDB is allowed to be adapted. For instance, when $\alpha = 4$ and $\beta = 0.4$, $QoD_{LB} = 70\%$. In total, the DBA needs to specify only three parameters for bounded and graceful QoD adaptation: α, β, and σ.

The time complexity of one QoD degradation for $O_i \in D_c$ is O(1). Hence, the total time complexity of QoD adaptation using Algorithm 2 per adaptation period is O($|D_c|$) = O(N).

C. Race-to-Idle in RTDBs

In this section, we reduce the RTDB power consumption via DPM using idle intervals extended by the real-time query aggregation and adaptive freshness management schemes.

TABLE I
C-STATES USED IN THIS PAPER (SOURCE: [23])

State (C_j)	**Power** (P_j)	**Latency** (δ_j)	**Energy** (E_j)
C0 (Run)	1 W	0	0
C1 (Standby)	0.5 W	0.1 ms	0.025 mJ
C2 (Dormant)	0.1 W	2 ms	0.9 mJ
C3 (Shutdown)	0.00001 W	10 ms	5 mJ

For RTDB power management, we consider the advanced configuration and power interface (ACPI) standard that is widely adopted. In ACPI, P states are performance states. P0 supports the highest frequency and voltage. A higher numbered P state spends less power, but provides a lower computational speed due to the reduced frequency and voltage. In contrast, C states are idle states. In the C0 state (active mode), the processor executes instructions normally. No instruction is executed in a low-power state or during a state transition. A transition between the C0 state and C1 state takes relatively negligible time and energy. More power is saved in a higher C state; however, the state transition takes more time and energy as shown in Table I. The table is adopted from a novel work on energy-efficient real-time scheduling [23], which has derived the low-power state model summarized in the table by analyzing the ARM Cortex-A family processors and FreeScale Power architecture. In the table, the instantaneous energy (power) consumption P_j in the state C_j where $j > 0$ is normalized to that in the C0 state. The transition latency of C_j is: $\delta_j = \delta_{0\to j} + \delta_{j\to 0}$ where $\delta_{0\to j}$ is the state transition latency from C0 to C_j and $\delta_{j\to 0}$ is that from C_j to C0. Also, the energy overhead for C_j is: $E_j = E_{0\to j} + E_{j\to 0}$ where $E_{0\to j}$ indicates the energy consumed to switch from C0 to C_j and $E_{j\to 0}$ is the energy spent to shift back from C_j to C0. In this paper, we assume that the break-even time $B_j = \delta_j$ for a low-power state j; that is, an idle interval must be at least as long as δ_j to effectively exploit C_j [23], [12].

Algorithm 3: RTDB Dynamic Power Management

```
while true do
    if busy at time t then
        Process transactions at the highest speed;
    else
        η(t) = release time of the next earliest update − t;
        ψ'(i) = estimated length of the i^th idle interval;
        ℓ(i) = min(η(t), ψ'(i));
        for j = 1; j < N_cs; j++ do
            Find max_{1<j<N_cs} {κδ_j ≤ ℓ(i)}
        if 0 < j < N_cs then
            Switch to the C_j state;
            while idle do
                Stay in the C_j state;
    Compute ψ'(i + 1);
    Switch to the P0 state;
```

In Algorithm 3, our DPM approach performed by PM (Figure 1) in a platform with N_{cs} C-states is summarized. In our approach, the RTDB processes update and user transactions in the P0 state to process them as fast as possible using the highest voltage and frequency.

If the RTDB has no update or user transaction to execute at time t, our DPM scheme computes $\eta(t) = min_{k=1}^{N}(r_k - t)$ where r_k is the release time of the next periodic instance of the update task k that periodically updates temporal data object O_k. Thus, $\eta(t)$ is found in O(N).

At time t, we also estimate the expected length of the i^{th} idle interval since the RTDB system initialization due to no user transaction arrival. To this end, we use an exponentially weighted moving average (EWMA), which is effective to smooth out short-term fluctuations of the trend in a time series [26] (e.g., the lengths of idle intervals observed in time) and subject to much less overheads than machine learning techniques (e.g., [27]) are:

$$\psi'(i) = a \times \psi'(i-1) + (1-a) \times \psi(i-1) \tag{4}$$

where $\psi'(i-1)$ and $\psi(i-1)$ are the previous smoothed idle interval length estimate and the actual length of the most recent idle interval, respectively. In Eq 4, a is the forgetting factor ($0 \leq a \leq 1$). For example, a DBA can set $a = 0.6$ to ensure that the impact of $\psi(i-1)$ is 1% on the smoothed value after 5 idle intervals by recursively solving Eq 4.

The expected length of the i^{th} idle interval is then:

$$\ell(i) = min(\eta(t), \psi'(i)) \tag{5}$$

If the RTDB has no update or user transaction to execute at time t, Algorithm 3 finds $\max_{1<j<N_{cs}} \{\kappa\delta_j \leq \ell(i)\}$ where κ (> 1) is a headroom constant used to compensate for possible errors in estimating $\psi'(i)$ and the overhead of executing Algorithm 3.[5] The CPU then switches to the C_j state that is the estimated deepest low-power state to save power without increasing deadline misses. Since there are a constant number of the C states in a processor, the selection of C_j takes O(1) time.

When the transition to C_j completes at time $t_b (> t)$, the actual idle interval begins. The system stays in the C_j state as long as it is idle. If there is an imminent periodic update job release or a new user transaction arrival at time $t_f (\geq t_b)$, our approach switches back to the P0 state. Thus, the *actual* length of the i^{th} idle interval that excludes any state transition latency is: $\psi(i) = t_f - t_b$. Algorithm 3 derives $\psi'(i+1)$ using Eq 4 based on $\psi'(i)$ and $\psi(i)$. Using $\psi'(i+1)$, Algorithm 3 is re-executed when the system becomes idle again in the future.

In this paper, if $\psi(i) < \delta_j$ $(= B_j)$, we consider that an estimation error has occurred and normalize it to δ_j:

$$e(i) = \begin{cases} (\delta_j - \psi(i))/\delta_j & \text{if } \psi(i) < \delta_j \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

To measure the estimation accuracy, we define the estimation error ratio:

$$P_e = 100 \times N_e / N_{all}\ (\%) \tag{7}$$

where N_e and N_{all} represent the total number of the occurred estimation errors and that of all the state transitions to one of the low-power states, respectively. Also, we measure the average size of the normalized estimation errors:

$$M_e = 100 \times \sum_{i=1}^{N_e} e(i)/N_e\ (\%) \tag{8}$$

Note that no estimation error occurs due to periodic updates, because the periods of temporal data updates in the RTDB are known *a priori*. In Section V, Table I, Eq 7, and Eq 8 are used for performance evaluation.

The time complexity of Algorithm 3 is O(N). Thus, the total time complexity of our approach, which consists of the real-time query aggregation, freshness adaptation, and DPM techniques described in Algorithms $1-3$, is O(N). It is linear in terms of the number of the temporal data in the RTDB but independent of the number of real-time queries. Thus, our approach is applicable to different RTDB applications with various user transaction arrival rates and data access patterns.

V. Performance Evaluation

In this section, the performance of our approach and baselines is thoroughly compared.

A. Simulation Set-Up and Baselines

In this subsection, the simulation model and experimental settings for performance evaluation are discussed. Also, the baselines designed for performance comparisons are described.

TABLE II
SIMULATION SETTINGS FOR DATA AND UPDATES

Parameter	Value
#Data Objects	1000
Update Period	$Uniform[100ms, 50s]$
EET_i	$Uniform[3ms, 6ms]$
Update Load	$\approx 50\%$

1) Simulation Model: Our simulation settings summarized in Tables II and III are similar to the ones used in other RTDB research modeled after data-intensive real-time applications, e.g., air traffic control, fire detection, and engine diagnosis [10], [11], [2], [4].

Our simulation settings for temporal data updates are summarized in Table II. As shown in the table, there are 1000 temporal data objects in our (simulated) RTDB. Each data object O_i ($1 \leq i \leq 1000$) is periodically updated by an update stream $Stream_i$ associated with an estimated execution time EET_i and an update period P_i. EET_i is uniformly distributed in a range [3ms, 6ms]. When a periodic update job is generated, the actual update execution time is derived by applying a normal distribution $Normal(EET_i, \sqrt{EET_i})$ to $Stream_i$ to model potential update time variations.

When no freshness adaptation is performed, the total update workload requires approximately 50% CPU utilization. Also, higher priority is given to updates to maintain the data freshness. Thus, all deadlines of update transactions are met. In the rest of this paper, we only consider the miss ratio of user transactions.

The total load applied to the RTDB is 50% + user transaction load. In this paper, total loads 60%, 70% 80%, 90%, 100%, 110%, and 120% are applied to evaluate the deadline miss ratio and power consumption of our approach and the baselines for different workloads.

TABLE III
SIMULATION SETTINGS FOR USER TRANSACTIONS

Parameter	Value
EET_i	$Uniform[5ms, 20ms]$
Actual Exec. Time	$Normal[EET_i, \sqrt{EET_i}]$
N_{DATA_i}	$EET_i \times$ *Data Access Factor* = [5, 20]
#Actual Data Accesses	$Normal(N_{DATA_i}, \sqrt{N_{DATA_i}})$
Slack Factor	[10, 20]

Table III summarizes the simulation set-up for user transactions. In this paper, a source, $Source_i$, generates a series of real-time user transactions whose inter-arrival time is distributed exponentially. $Source_i$ is associated with EET_i. In this paper, $EET_i = Uniform[5ms, 20ms]$. Using multiple sources, we statistically generate transaction groups with different average execution times and numbers of data accesses. To increase the workload applied to the RTDB, we increase the number of sources. As a result, more user transactions arrive per unit time. When a user transaction is generated, the actual execution time is generated by applying the normal distribution $Normal(EET_i, \sqrt{EET_i})$ to introduce the execution time variance in a series of user transactions produced by $Source_i$.

We derive the average number of data accesses for $Source_i$ in proportion to EET_i; that is, $N_{DATA_i} =$ *data access factor* $\times$ $EET_i = [5, 20]$. Thus, a longer transaction generally accesses more data. When generating a user transaction, the actual number of data accesses of the transaction is determined by applying $Normal(N_{DATA_i}, \sqrt{N_{DATA_i}})$ to introduce a variance among the user transactions generated by $Source_i$.

For a user transaction, *deadline = arrival time + estimated execution time* $\times$ *slack factor* in this paper. A slack factor is uniformly distributed in a range (10, 20). For an update, *deadline = next update period.*

2) Baselines: In this paper, we simulate the RTDB system architecture depicted in Figure 1. The system components for real-time query aggregation, freshness adaptation, and power saving can be selectively turned on/off for performance evaluation purposes. Database query aggregation for energy saving typically requires to delay queries [8], incurring deadline misses as discussed before. Also, most existing RTDB power management schemes [2], [3] rely on the never-idle paradigm that requires complex trade-offs between performance and power conservation. Therefore, we consider the following baselines that represent the current state-of-the-art RTDBs and extend existing database power management schemes for insightful performance comparisons:

- *Power-Unaware RTDB (PU-RTDB):* In this baseline, EDF scheduling and 2PL-HP are supported to process real-time transactions. Update transactions are assigned higher priority than user transactions as discussed in Section III. Our approach and all the baselines apply the same scheduling and concurrency control techniques for fair performance comparisons. However, no query aggregation, freshness adaptation, or power saving is considered, similar to most existing RTDBs. Thus, this baseline represents state-of-the-art RTDBs.
- *Query Aggregation (QA):* In this baseline, we extend the non-RTDB query aggregation scheme presented in [8] by supporting our real-time query aggregation scheme (Section IV-A) to avoid excessive deadline misses. In fact, we have directly applied the query aggregation technique presented in [8] without this extension. We have observed more than 90% of user transactions miss their deadlines even when the total workload is only 60% and user transactions are delayed only until the length of the EDF queue becomes 5. Thus, we use QA instead of [8] for performance comparisons.
- *Freshness Adaptation (FA):* This baseline extends the adaptive QoD management scheme [9] that is power-unaware as discussed in Section IV-B. In [3], QoD adaptation is applied to reduce the power consumption for

[5]Generally, a large κ value provides a lower miss ratio for saving less power or vice versa. In Section V, $\kappa = 1.5$.

running a single real-time user transaction. FA extends [3] in the sense that it supports multiple concurrent real-time user transactions. Further, FA supports the race-to-idle method via DPM instead of DVFS used in [3].

One may argue that QA and FA are just variations of our approach, QA-FA. However, QA and FA considerably extend the state-of-the-art techniques for power-aware RTDBs. Without those extensions, the baselines show relatively poor performance in terms of the miss ratio and power consumption.

TABLE IV
QUERY AGGREGATION AND QOD ADAPTATION PARAMETERS

Parameter	Description	Value
P(QA)	P(query aggregation)	0.05, 0.1, ..., 0.3
α	$P_i \leq \alpha P_{i,min}$	4
β	$\|D_c\|/\|D\|$	0.1, 0.2, ..., 0.5
σ	size of a QoD adaptation	10%
P_{QoD}	QoD adaptation period	5s

In Table IV, the parameters for query aggregation and QoD adaptation are summarized. In this section, QAxx indicates QA with query aggregation probability of $xx\%$. For example, two arbitrary queries can be merged into one query with 5% probability in QA05. For performance evaluation, the query aggregation probabilities 5%, 10%, ..., 30% are used as summarized in Table IV. Data access skews and queries may vary from RTDB application to application. Rather than considering the semantics or requirements of a specific application, the query aggregation probabilities ranging from conservative to moderate values are used for RTDB performance evaluation.

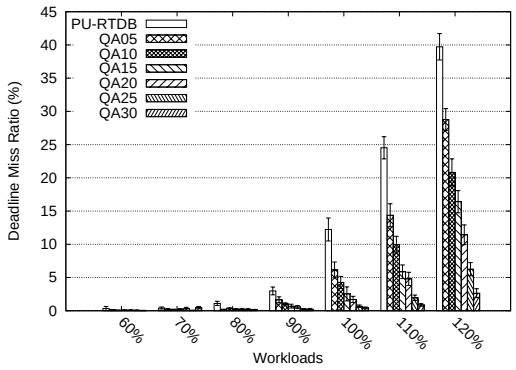
In a similar vein, FAyy indicates that freshness adaptation is allowed for $yy\%$ of the temporal data objects in the RTDB where $yy = 100 \times \beta$ % that indicates the fraction of temporal data whose QoD can be adapted. For performance evaluation, we consider yy = 10%, 20%, ..., 50% to consider potentially different QoD requirements in different RTDB applications as summarized in Table IV. Also, we set α = 4 and σ = 10% for bounded, stepwise QoD adaptation.

Our approach that integrates QAxx and FAyy is indicated by QAxx-FAyy. In this paper, the same workloads and experimental settings are used for fair performance comparisons between the baselines and our approach. Each experimental run takes 10 minutes. Each performance result is the average of 20 runs with 95% confidence intervals.

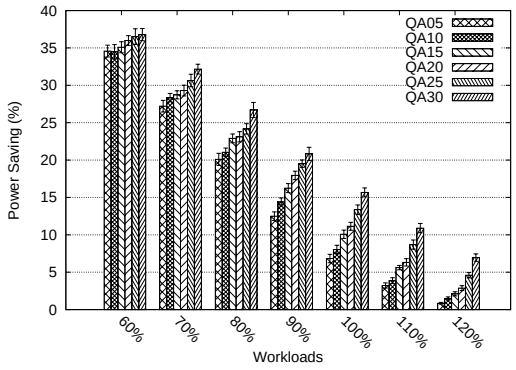
B. Performance Evaluation Results

1) Experiment Set 1 − Impacts of Query Aggregation : Figure 2 compares the deadline miss ratio of PU-RTDB and QA with different $P(QA)$ values in Table IV. It also shows the power saving achieved by QA against PU-RTDB. PU-RTDB and QA provide 100% QoD, since they do not consider freshness adaptation.

In Figure 2(a), the miss ratio of the PU-RTDB ranges between $0.33 \pm 0.3\% - 39.71 \pm 1.97\%$ as the load is increased from 60% to 120%. QA05's miss ratio ranges between



(a) Deadline Miss Ratio



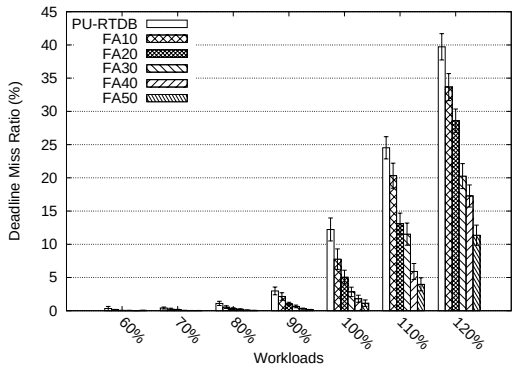
(b) Power Saving

Fig. 2. Deadline Miss Ratio and Power Conservation in Experiment Set 1

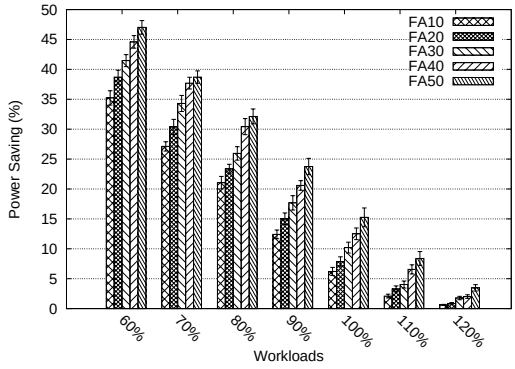
$0.09 \pm 0.08\% - 27.44 \pm 2.54\%$. Thus, compared to PU-RTDB, QA05 reduces the miss ratio by up to approximately 12% when 120% load is applied to the RTDB. Although $P(QA)$ is only 5%, it becomes more effective in terms of reducing the miss ratio when the load increases and more user transactions are merged as a result.

As shown in Figure 2(a), QA30 supports the lowest miss ratio among the tested approaches. Its miss ratio is below 4% even when the load is 120%; it decreases the miss ratio by nearly 36% compared to PU-RTDB. Although the miss ratio generally grows as the load increases, the growth rate is decreased substantially, if more real-time query aggregation is possible. Note that the miss ratio of the tested approaches is non-zero even when the total load is much below 100%, since some transactions may get aborted and restarted due to data/resource conflicts. This indicates the difficulty of processing real-time transactions in RTDBs.

QA considerably decreases the power consumption especially when the load is relatively low as shown in Figure 2(b). The achieved power saving generally decreases as the load increases, because the RTDB should run at the highest speed longer to process more real-time transactions as the load increases. By aggregating real-time queries, it reduces the workload and switches to the low-power mode when the RTDB becomes idle. By doing this, compared to PU-RTDB, QA05 and QA30 decrease the total dynamic power consump-



(a) Deadline Miss Ratio



(b) Power Saving

Fig. 3. Deadline Miss Ratio and Power Conservation in Experiment Set 2

tion by $0.84\% - 34.58\%$ and $6.95\% - 36.78\%$, respectively. Being power-unaware, PU-RTDB cannot decrease the power consumption even when the system is underutilized.

The results in this set of experiments demonstrate that QA is effective in terms of managing the timeliness and power consumption in RTDBs, if it is done without artificially delaying real-time transactions, incurring little overhead. We have measured the CPU utilization for the tested approaches. Compared to PU-RTDB, QA05 and QA30 reduce the utilization by up to 3% and 15%, respectively. We also observe QA's accuracy in estimating the next idle interval length is acceptable. A more detailed discussion of the utilization and estimation accuracy for Experiment Sets $1 - 3$ is given in Appendix A due to space limitations.

2) Experiment Set 2 − Impacts of Freshness Adaptation : In Figure 3(a), the miss ratio of PU-RTDB ranges between $0.33 \pm 0.3\% - 39.71 \pm 1.97\%$ as the load is increased from 60% to 120%.[6] The miss ratio of FA10 ranges between $0.12 \pm 0.071\% - 33.65 \pm 2.03\%$ as shown in Figure 3(a). Compared to PU-RTDB, FA10 reduces the miss ratio by up to approximately 6%.

FA50 supports the lowest miss ratio among the tested approaches. Its miss ratio in Figure 3(a) ranges between $0.05 \pm 0.02\% - 11.36 \pm 1.51\%$, decreasing the miss ratio by up to roughly 28% compared to PU-RTDB. Although the miss ratio grows as the load increases, data freshness adaptation considerably decreases the growth rate. In our approach, the required freshness is always supported for each temporal data item O_i in terms of $fvi[i]$ (or $avi[i]$) if $O_i \in D_c$ (or $O_i \in D_h$). In Experiment Sets 2 and 3, the QoD supported by FA and QA-FA is slightly higher than the required QoD lower bounds because of our periodic QoD adaptation that is bounded and gradual. Due to space limitations, a more detailed discussion of the QoD lower bounds and measured QoD is given in Appendix B.

From Figures 2(a) and 3(a), we observe that QA is more effective than FA is in terms of reducing the miss ratio especially when the load is relatively high. This is because the total update workload is no more than 50% in our simulation set-up, whereas the user transaction load increases from 10% to 70% as the total load increases from 60% to 120%. Thus, QA has more opportunities to aggregate real-time queries as the load increases. Generally, the update load is predetermined in RTDBs; however, user transactions may arrive at any time [6].

From Figures 2(b) and 3(b), we also observe that QA saves more power than FA does when the load is relatively high and vice versa. In general, QA is relatively more desirable in that it does not adapt any quality of service, e.g., QoD, provided by RTDBs to decrease the miss ratio and power consumption. FA, however, is effective too, since it is highly likely that different data have different popularity or importance. Thus, they are complementary and may create synergistic effects, which we evaluate next.

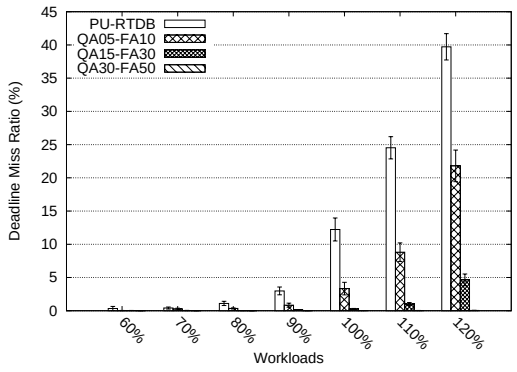
3) Experiment Set 3 − Impacts of Integrated Query Aggregation and QoD Adaptation: As shown in Figure 4, QA05-FA10 that allows minimal real-time query aggregation and freshness adaptation already decreases the miss ratio and power consumption by up to approximately 18% and 37% compared to PU-RTDB.

The highest miss ratio of QA15-FA30, which supports moderate query merging and freshness adaptation, is $4.64 \pm 0.88\%$ for the 120% load. Compared to PU-RTDB, it decreases the miss ratio by approximately 34% when the load is 120%. The miss ratio of QA15-FA30 for 120% load is approximately 2% higher than that of QA30 (Figure 2(b)) and 6% lower than that of FA50 (Figure 3(b)). Also, for 120% load, QA15-FA30 saves roughly 1% less and 2.5% more power than QA30 and FA50 do.

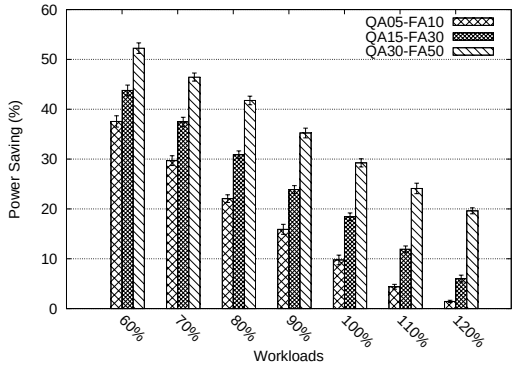
Therefore, the miss ratio and power reductions of QA15-FA30 are comparable to those of QA30 and FA50, which support the highest degree of real-time query aggregation and freshness adaptation in our experiments. This is because QA and FA together decrease both user and update transaction workloads, mitigating the conflicts among the timeliness, freshness, and power saving requirements as discussed in Section I.

The miss ratio of QA30-FA50 is below 0.05% for all the tested RTDB workloads. Thus, the miss ratio of QA30-FA50 is

[6]The miss ratio, power consumption, and utilization results of PU-RTDB are the same as the ones reported in the Experiment Set 1. They are plotted again in Figure 2 just for easier comparisons.



(a) Deadline Miss Ratio



(b) Power Saving

Fig. 4. Deadline Miss Ratio and Power Conservation in Experiment Set 3

invisible in Figure 4(a). Compared to PU-RTDB, it decreases the power consumption by approximately $19\% - 52\%$ as plotted in Figure 4(b).

VI. Conclusions and Future Work

It is desirable yet challenging to process real-time transactions in a timely manner using fresh data, while consuming less power in real-time databases. In this paper, we present an effective approaches for real-time query aggregation and adaptive data freshness management to decrease both deadline misses and power consumptions in RTDBs. The miss ratio and power consumption of our approach are thoroughly compared to those of the baselines that represent and improve state-of-the-art RTDBs. Our approach decreases the deadline miss ratio and power consumption by up to approximately 38% and 52%, respectively. Also, we observe that our approaches complement each other in terms of decreasing the user transaction and temporal data update workloads to decrease the deadline miss ratio and power spending. Finally, future research issues are discussed in Appendix C.

References

[1] "eXtremeDB, a fast, reliable and cost-effective embedded database system for embedded systems and intelligent devices." [Online]. Available: http://www.mcobject.com/emb

[2] W. Kang and S. H. Son, "Power- and time-aware buffer cache management for real-time embedded databases," *Journal of Systems Architecture - Embedded Systems Design*, vol. 58, no. 6-7, pp. 233–246, 2012.

[3] W. Kang and J. Chung, "QoS Management for Embedded Databases in Multicore-Based Embedded Systems," *Mobile Information Systems*, 2015.

[4] T. Gustafsson, H. Hallqvist, and J. Hansson, "A Similarity-Aware Multiversion Concurrency Control and Updating Algorithm for Up-To-Date Snapshots of Data," in *Euromicro Conference on Real-Time Systems*, 2005.

[5] S. Irani, S. Shukla, and R. Gupta, "Algorithms for power savings," *ACM Transactions on Algorithms*, vol. 3, no. 4, 2007.

[6] K. Y. Lam and T. W. Kuo, Eds., *Real-Time Database Systems*. Kluwer Academic Publishers, 2006.

[7] J. Wires, S. Ingram, Z. Drudi, N. J. A. Harvey, and A. Warfield, "Characterizing storage workloads with counter stacks," in *USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[8] W. Lang and J. M. Patel, "Towards eco-friendly database management systems," in *Biennial Conference on Innovative Database Systems Research*, 2009.

[9] 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, Oct. 2004.

[10] M. Xiong, S. Han, K.-Y. Lam, and D. Chen, "Deferrable scheduling for maintaining real-time data freshness: Algorithms, analysis, and results," *IEEE Transactions on Computers*, vol. 57, no. 7, p. 952964, 2008.

[11] S. Han, D. Chen, M. Xiong, K.-Y. Lam, A. K. Mok, and K. Ramamritham, "Schedulability analysis of deferrable scheduling algorithms for maintaining real-time data freshness," *IEEE Transactions on Computers*, vol. 63, no. 4, pp. 979 – 994, 2014.

[12] M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo, "Energy-Aware Scheduling for Real-Time Systems: A Survey," *ACM Transactions on Embedded Computing Systems*, vol. 15, no. 1, 2016.

[13] Z. Xu, X. Wang, and Y. cheng Tu, "Power-Aware Throughput Control for Database Management Systems," in *International Conference on Autonomic Computing*, 2013.

[14] Y.-C. Tu, X. Wang, B. Zeng, and Z. Xu, "A System for Energy-Efficient Data Management," *SIGMOD Record*, vol. 43, no. 1, pp. 21–26, 2014.

[15] M. Kunjir, P. K. Birwa, and J. R. Haritsa, "Peak power plays in database engines," in *International Conference on Extending Database Technology*, 2012.

[16] Z. Xu, Y.-C. Tu, and X. Wang, "Online Energy Estimation of Relational Operations in Database Systems," *IEEE Transactions on Computers*, vol. 64, no. 11, pp. 3223–3236, 2015.

[17] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: An Acquisitional Query Processing System for Sensor Networks," *ACM Transactions on Database Systems*, vol. 30, no. 1, pp. 122–173, 2005.

[18] N. Tsiftes and A. Dunkels, "A Database in Every Sensor," in *ACM Conference on Embedded Networked Sensor Systems*, 2011.

[19] S. Hu and et al., "Data Acquisition for Real-time Decision-making under Freshness Constraints," in *IEEE Real-Time Systems Symposium*, 2015.

[20] M. Völp, M. Hähnel, and A. Lackorzynski, "Has energy surpassed timeliness? - Scheduling energy-constrained mixed-criticality systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2014.

[21] G. Cao and A. A. Ravindran, "Energy Efficient Soft Real-Time Computing through Cross-Layer Predictive Control," in *International Workshop on Feedback Computing*, 2014.

[22] C. Imes, D. H. K. Kim, M. Maggio, and H. Hoffmann, "POET: a portable approach to minimizing energy under soft real-time constraints," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015.

[23] V. Legout, M. Jan, and L. Pautet, "Scheduling algorithms to reduce the static energy consumption of real-time systems," *Real-Time Systems*, vol. 51, no. 2, pp. 153–191, 2015.

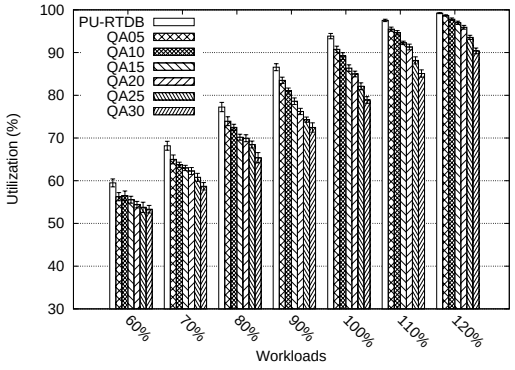
[24] A. Deshpande, Z. Ives, and V. Raman, "Adaptive Query Processing," *Foundations and Trends in Databases*, vol. 1, no. 1, pp. 1–140, 2007.

[25] S. Babu and P. Bizarro, "Adaptive Query Processing in the Looking Glass," in *Conference on Innovative Data Systems Research*, January 2005.

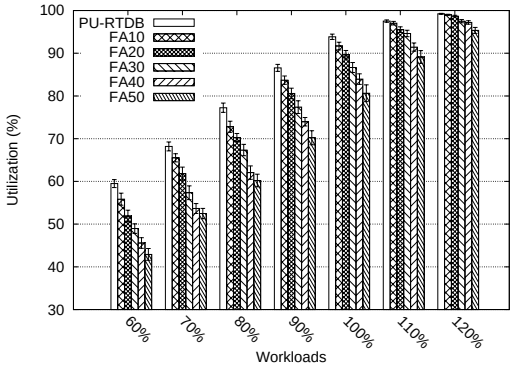
[26] G. Arce, *Nonlinear Signal Processing: A Statistical Approach*. Wiley, 2005.

[27] Y. Zhang, Y. Liu, L. Zhuang, X. Liu, F. Zhao, and Q. Li, "Accurate CPU Power Modeling for Multicore Smartphones," Microsoft, Tech. Rep. MSR-TR-2015-9, 2015.

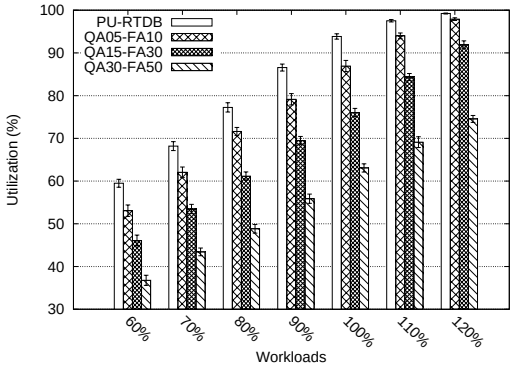
APPENDIX



(a) Experiment Set 1



(b) Experiment Set 2



(c) Experiment Set 3

Fig. 5. CPU Utilization

A. Utilization Measurements and Idle Length Estimation Errors

In Experiment Set 1, the utilization of the PU-RTDB ranges between $59.49\% - 99.23\%$ for the $60\% - 120\%$ loads. In Figure 5(a), QA05 and QA30 reduce the utilization by up to 3% and 15% compared to PU-RTDB. Notably, the magnitude of power saving by QA05 and QA30 in Figure 2(b) is bigger than that of the utilization decrease, because QA or FA switches to a low-power state when the RTDB is idle.

In Experiment Set 1, the estimation error ratio P_e (Eq 7) ranges between $0.25 \pm 0.03\% - 2.04 \pm 0.05\%$. Our estimation accuracy is high in terms of P_e because: 1) the EWMA is effective to track the trend in a time series [26] and 2) the periods of temporal data updates used together with $\psi'(i)$ in Eq 5 are known in advance. Although M_e (Eq 8) ranges between $2.5 \pm 0.46\%$ and $10.39 \pm 0.49\%$, it has little impact on the miss ratio and power consumption because: 1) P_e is low, 2) a 10% estimation error is much smaller than the user transaction execution times and relative deadlines, and 3) the CPU spends only a small amount of time in the C3 state due to δ_3 that is 100 and 5 times longer than δ_1 and δ_2, respectively. More specifically, a 10% estimation error is equal to 0.01ms, 0.2ms, and 1ms with respect to δ_1, δ_2, and δ_3 in Table I. On the other hand, the user transaction execution times and deadlines range between [5ms, 20ms] and [50ms, 400ms], respectively (Table III). Also, when $P(QA) = 0.3$ that provides the highest chance for query aggregation among the tested $P(QA)$ values, the CPU spends only 1.1% and 0.04% of the time in the C3 state for the 60% and 120% loads, respectively. In particular, for the 60% load, QA30 spends approximately 64.2%, 3.34%, 31.35%, and 1.1% of the time in the C0, C1, C2, and C3 states, respectively. When the load is 120%, it spends 93.09%, 1.34%, 5.52%, and 0.04% of the time in the C0, C1, C2, and C3 states, respectively. In Experiment Sets $1-3$, we observe that the amount of the time spent by QA, FA, and QA-FA in the C-states generally decreases in C0, C2, C1, and C3 order.

In Experiment Set 2, FA decreases the utilization by $3\% - 17\%$ compared to the PU-RTDB as shown in Figure 5(b). For the tested workloads, P_e and M_e range between $0.05 \pm 0.01\% - 2.14 \pm 0.07\%$ and $2.46 \pm 0.48\% - 12.08 \pm 0.64\%$, respectively. In FA50, which supports the largest QoD adaptation, the processor spends 4.95% and 0.09% of the time in the C3 state for the 60% and 120% loads, respectively. FA decreases the utilization more for the relatively low loads, because fewer user transactions access temporal data. As a result, more data become cold and subject to freshness adaptation. Thus, it spends more time in the C3 state than QA did in Experiment Set 1 especially when the load is low.

In Experiment Set 3, compared to PU-RTDB, QA05-FA10, QA15-FA30, and QA30-FA50 in Figure 5(c) decrease the utilization by up to approximately 6%, 17%, and 29%, respectively. Due to the effective cooperation between QA and FA, QA-FA decreases the utilization more substantially than QA or FA does separately. P_e ranges between $0.11 \pm 0.01\% - 2.3 \pm 0.04\%$ and M_e between $2.75 \pm 0.59\% - 11.74 \pm 0.45\%$, similar to Experiment Sets 1 and 2. QA-FA spends the largest amount of time in the C3 state among the tested approaches by doing both query aggregation and freshness adaptation. In particular, QA30-FA50 spends 5.49% and 0.6% of the time in the C3 state for the 60% and 120% loads, respectively.

B. QoD Measurements

In our performance evaluation, α = 4 and σ = 10% as described in Section V. In addition, different values of β = $|D_c|/|D|$ = 0.1, 0.2, ..., 0.5, are used. The lower bounds of the QoD for different β values computed using Eq 3 are summarized in Table V.

TABLE V
QOD LOWER BOUNDS WHEN $\alpha = 4$

β	**QoD Lower Bound**
0.1 (FA10)	92.5%
0.2 (FA20)	85%
0.3 (FA30)	77.5%
0.4 (FA40)	70%
0.5 (FA50)	62.5%

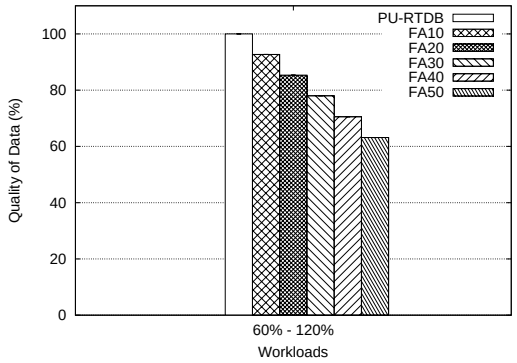


Fig. 6. Quality of Data Supported by QA (Experiment Set 2)

Figure 6 shows the QoD measured in the Experiment Set 2. (The confidence intervals are invisible in Figure 6 for their small sizes.) As shown in the figure, FA always supports the QoD requirements. More specifically, it supports a bit higher QoD than the QoD lower bounds summarized in Table V. This is because the freshness is only adapted by increasing the periods of the data in the set D_c by σ = 10% at each QoD adaptation, which is performed every 5s in our performance evaluation. In Experiment Set 3, we have observed similar QoD results in QA-FA, since the same QoD parameters are used for performance evaluation. In contrast, PU-RTDB and QA always support 100% QoD, since they do not consider adaptive freshness management as discussed previously.

C. Future Work Discussions

There are several possible directions for future research on power-aware RTDBs, which has received relatively little attention despite the importance:

- Enhancing the accuracy in estimating the next idle interval is a key issue for DPM in RTDBs. Machine learning techniques could be applied to enhance the accuracy. However, they are relatively heavy in terms of computation. Moreover, a supervised learning method may perform poorly when the training set used to develop a model does not represent unforeseen idle interval lengths, which may vary in time, well. On the other hand, an unsupervised learning algorithm may take a long time or even fail to derive (learn) unknown parameters necessary for a prediction. In general, the applicability of machine learning to timeliness and power management in RTDBs is largely unknown.
- Effective real-time query optimization, transaction scheduling, and concurrency control techniques may considerably reduce potential data/resource contention and corresponding transaction aborts/restarts. As a result, the deadline miss ratio and power consumption of RTDBs can be decreased. Although transaction scheduling and concurrency control in RTDBs have been studied extensively [6], most existing techniques do not consider power/energy consumptions. It might be possible to adapt/modify them to support power conservation as well. Or, it might be necessary to explore fundamentally different approaches.
- More power could be saved by applying other power management techniques supported by hardware, e.g., DVFS, together with DPM. A related challenge is how to avoid increasing deadline misses or losing opportunities for DPM due to slower real-time query processing caused by the never-idle method. A promising approach could be investigating a hybrid method that seamlessly integrates the never-idle and race-to-idle techniques, e.g., DVFS and DPM, to further decrease the miss ratio and power consumption in RTDBs with little complexity and overhead.
- Multicore processors could be both a blessing and a curse for power-aware RTDBs. At first glance, one may think more transactions/queries could be processed concurrently using multiple cores. However, a naive approach may suffer from severe contention for shared data and resources, e.g., the system bus and memory controllers, among the cores. As a result, it might increase deadline misses or power consumptions in reality. It is largely unknown how to design power-aware RTDBs for multicore platforms. For example, our real-time query aggregation scheme could help the RTDB reduce data/resource contention. However, a difficult problem of assigning real-time transactions/queries to the cores to minimize the data/resource contention, deadline miss ratio, and power consumption remains open.

Note that this list is neither exhaustive nor complete; there could be other important issues not discussed here. However, a key lesson we learned from this work is that it is possible to reduce both deadline misses and power consumptions in RTDBs, while supporting the desired QoD. This approach could be extended to further enhance the timeliness and power efficiency of RTDBs by exploring more effective real-time query optimization, transaction scheduling, concurrency control, and RTDB system design techniques that consider inherent RTDB characteristics, real-time data semantics, or advanced hardware features.

ACKNOWLEDGMENT

We appreciate anonymous reviewers for their help to improve the paper. This work was supported, in part, by NSF grant CNS-1526932.