

# Power-Aware Data Buffer Cache Management in Real-Time Embedded Databases \*

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

## Abstract

*The demand for real-time data services in embedded systems is increasing. In these new computing platforms, using traditional buffer management schemes, whose goal is to minimize the number of I/O operations, is problematic since they do not consider the constraints of those platforms such as limited energy and distinctive underlying storage. In particular, due to asymmetric read/write characteristic of flash memory, minimum buffer misses neither coincide with minimum power consumption nor minimum I/O deadline miss ratio. In this paper we propose a power-aware buffer cache management scheme for real-time databases whose secondary storage is a flash memory. We focus on the problem of guaranteeing the performance goal in terms of both I/O power consumption and I/O deadline miss ratio. To address this problem, we propose logical partitioning of the global buffer pool into read and write buffer pools, and dynamic feedback control of read/write buffer pool sizes to satisfy both performance goals. We have shown through an extensive evaluation that our approach satisfies both performance goals in a variety of workloads and access patterns with considerably smaller size of buffer pools compared to baseline approaches.*

## 1 Introduction

A number of emerging applications require real-time data services to handle large amounts of data in a timely fashion. With continued miniaturization and increasing computational power, we see ever growing demand for real-time data services in embedded systems that run sophisticated and intelligent control software. Running real-time data services in these embedded systems pose different challenges because of distinctive characteristics of embedded systems, both in H/W and S/W.

One major challenge is resource constraints of embedded systems. In particular, power consumption is a major concern for embedded systems. The availability of data services in battery-powered embedded systems is clearly limited by the

amount of available energy. For instance, we may consider a real-time database running on a mobile computer carried by a firefighter in a burning building [1]. It collects data from nearby sensors and responds to queries on the burning building while the firefighter tries to find a pathway to save people inside the building. In this situation, the real-time database should not only provide responses to the queries within their deadlines but it also needs to control its power consumption to satisfy the dynamically changing power constraints. As another example, we may consider a habitat monitoring system with a wireless sensor network [19]. The gateway mote of the system, which is powered by rechargeable battery with solar panels, collects data from sensor nodes and provides real-time data service to remote monitoring authorities; this system should consume less power than that which can be harvested from the solar panels. As shown in these examples, many real-time embedded systems need predicted and bounded behavior not only in timely processing of data, but also in resource consumption, power consumption in particular. Unfortunately, traditional real-time database systems (RTDBS) have guaranteed only timely processing of data [2][10], ignoring resource constraints of embedded systems.

	Read		Write	
NAND Flash [18]	15 $\mu$ s	0.74 $\mu$ J	200 $\mu$ s	9.9 $\mu$ J

**Table 1. The response time and energy consumption of a read and a write operation in flash memory**

Another problem of applying existing RTDBS technologies to real-time embedded systems comes from the peculiar properties of underlying H/W of embedded systems. Above all, the secondary storage for embedded database is quite different. In most modern real-time embedded systems, data is stored in high capacity NAND-type flash memory. Unlike disks, the access time to flash memory is not affected by mechanical parts, thus highly predictable. These desirable characteristics make flash memory more suitable for RT-EDBS. However, flash memory has very distinctive operational properties such as asymmetrical read/write response time and power consumption as shown in Table 1. Due to

\*This research work was funded in part by NSF-CNS-0614886

these distinctive properties of flash memory, using existing storage management techniques such as data buffer cache management designed for high buffer hit ratio should be re-considered. Until now, most research on data buffer management is centered around how to minimize the number of I/Os between the buffer and the secondary persistent storage by maximizing buffer hit ratio. However, as explained above, real-time embedded systems have different performance metrics such as deadline miss ratio and power consumption. Satisfying these performance objectives is more important than achieving maximum buffer hit ratio. Optimizing buffer hit ratio neither guarantees minimum I/O power consumption nor minimum I/O deadline miss ratio.

In our earlier work [11], we presented an I/O-aware deadline miss ratio management scheme in flash memory-based embedded systems, which showed that adjusting buffer size was very effective in controlling the I/O deadline miss ratio and preventing I/O overload. While our initial work was encouraging, it did not consider power issues in buffer management.

In this paper, we propose a power-aware buffer cache management scheme for RTEDBS whose secondary storage is flash memory. In particular, the performance objectives for I/O, which includes both power consumption and deadline miss ratio, are set by an application, and these objectives are achieved by dynamically tuning the sizes of buffers with a feedback control loop. For effective control of I/O workloads, the global buffer pool is logically partitioned into buffer pools for updated pages and non-updated pages; they are highly related to write I/O workload and read I/O workload, respectively. By partitioning the buffer space according to the type of pages, we can provide fine control on I/O workloads to meet the multiple performance goals simultaneously. The interaction between the two performance metrics is captured by applying *multiple-input/multiple-output* (MIMO) modeling and control technique.

The evaluation results demonstrate that our approach gives robust and controlled behavior in terms of guaranteeing both the power consumption and the miss ratio in I/O under a variety of workloads, access patterns, and even in the presence of transient overloads. In addition, the performance goals are achieved with significantly smaller consumption of the buffer space than the baseline approaches, in which only either I/O deadline miss ratio or I/O power consumption is considered.

The rest of this paper is organized as follows. In Section 2, we place our approach in context by examining relevant work in related areas. In Section 3, the system model for RTEDBS is presented. The details of our buffer management scheme is described in Section 4. Section 5 shows our simulation settings and presents evaluation results, which quantifies the benefits of our buffer management scheme. Finally, Section 7 concludes the paper.

## 2 Related Work

The work of this paper is based on two distinctive but related fields; flash memory-based DBMS and goal-oriented adaptive buffer management.

### 2.1 Flash Memory-Based DBMS

A flash device, NAND-type flash device in particular, has been increasingly adopted as data storage media for a wide spectrum of embedded systems. Unlike disks, the random access time to flash memory is several orders of magnitude faster and highly predictable. However, some distinctive characteristics of flash memory makes it very inefficient for a disk-based DBMS to run on top of flash memory. Those distinctive characteristics of flash memory can be summarized as follows. First, all read and write operations happen at page granularity, which is typically 512-4096 bytes, and the pages are organized into blocks, which is typically 16-128 Kbytes. A page can only be updated after erasing the entire block, which it belongs to; a selective update of a particular data page is not possible. Second, read and write operations have asymmetric response time and power consumption as shown in Table 1.

These distinctive properties of a flash memory gave birth to several flash-based DBMSs [17][13][12]. Even though they are different in technical details, they follow some common design principles to exploit the properties of flash memory, which includes (1) avoiding direct in-place update to overcome erase-before-write limitation and (2) minimizing write operations at the cost of increased read operations.

Even though flash-based DBMSs give significant performance gain against existing disk-based DBMSs on top of flash-based systems, their approaches are basically best-effort; they do not assure any guarantees on its performance such as power consumption and response time. Unlike these approaches, we guarantee the performance goals set by users or applications by adaptively adjusting system parameters using feedback control.

### 2.2 Adaptive Buffer Management

As diverse applications utilize the data service from DBMSs, the performance goal of each application can vary widely. Most conventional DBMSs provide knobs to tune its operation to meet the diverse goals. The memory size allocated to the buffer is one of the most important knobs, which greatly affects the performance of the system. Since manual tuning of this knob is extremely hard even for database experts, several goal-oriented adaptive buffer management approaches have been proposed [4][3][6]. While different in some technical details, they basically follow the common principles. First, a performance goal is set by an user or an application to each service class; the performance metric for the goal is I/O response time. Second, the actual response

time is compared with the goal, and the difference between the two is used as an input to an estimator that will adjust buffer allocation to the service class in order to move the system closer to its goals. This feedback control loop is repeated at regular intervals.

Our approach follows the same “*observing, estimating and adjusting knob*” approach of previous work. However, our approach has several differences from them. First, our approach can set and achieve multiple performance goals, such as power consumption and I/O deadline miss ratio. Second, we do not partition the available buffer memory into service classes. Instead, we logically partition the buffer memory into updated and non-updated buffers to reflect different properties of write and read operations in flash-based storage.

### 3 System Model for RTEDBS

In this paper, we consider a real-time embedded system having a flash memory as its secondary storage. The flash memory is used for persistent data storage. The data buffer pool is located in main memory and is a cache between the flash memory and the CPU. It is shared between transactions to reduce the data storage access time. The I/O load between main memory and flash memory occurs only when an explicit I/O request is issued for a data object not present in the buffer. Implicit I/O requests such as page faults are not considered because virtual memory is not typically supported in real-time embedded systems for predictability.

In our data model, data objects can be classified into two classes, temporal and non-temporal data. Temporal data objects are updated periodically by *update transactions*. In contrast to update transactions, *user transactions* may read both temporal and non-temporal data objects and modify non-temporal data objects.

Because embedded platforms are assumed for our RT-EDBS, transactions are *canned transactions*, whose characteristics including data requirement and worst-case computation time is known at the design time; embedded systems may have a set of pre-defined queries instead of ad-hoc queries. However, workload and data access patterns of the whole RT-EDBS can be unpredictable and change dynamically because the invocation frequency of each transaction is unknown. Since data requirements are known for each transaction, data requests of each transaction can be gathered before its computation to improve the response time. To this end, we model each transaction as a two-phase operation, an I/O phase and a computation phase. In the I/O phase, data objects for the transaction are brought to the buffer from the flash memory. If all data objects are already present in the buffer, the I/O phase is skipped. In a single transaction, the computation phase can begin only after all its required data objects are present in the buffer. A transaction can commit after the computation phase by updating a copy of the data object in the memory buffer. The time required to update the data object in the buffer is ignored in this transaction model because it is relatively small

compared to flash memory accesses. The buffer manager will *eventually* write the updated pages back to flash memory when the buffer manager is running out of buffer space.

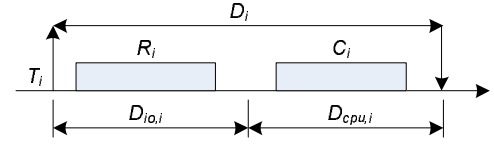


Figure 1. The timing of a typical transaction.

Figure 1 shows the timing of a typical transaction  $i$ , where  $T_i$  is a release time,  $D_i$  is a relative deadline,  $R_i$  is a I/O time,  $C_i$  is a computation time,  $D_{io,i}$  is a relative I/O deadline, and  $D_{cpu,i}$  is a relative CPU deadline. When the I/O deadline is missed, the transaction is aborted and its computation phase is not initiated.

While the deadline of each transaction  $i$  is set by an application in consideration of the worst case I/O time and computation time, I/O deadline and subsequent CPU deadline are dynamically derived from the relative deadline  $D_i$  and deadline miss ratio at time  $t$  as follows,

$$\text{I/O deadline for transaction } i \text{ at time } t = T_i + D_i - C_i \times \left(1 + m_{cpu}(t) \times \frac{D_i - C_i}{C_i}\right), \quad (1)$$

where  $m_{cpu}(t)$  is the CPU deadline miss ratio during the past sampling period. In short, the I/O deadline reflects up-to-date resource status and how much slack time is required for the computation phase to finish within the transaction deadline. For example, when CPU is not overloaded, thus,  $m_{cpu}(t)$  is close to zero, the I/O deadline can be as long as  $T_i + D_i - C_i$ , since CPU phase need only small slack time to complete in time. For more details on I/O deadline assignment scheme, readers are referred to [11].

## 4 Approach

### 4.1 Specification of Performance Objectives

The specification of the performance objectives in RT-EDBS is given as  $\langle \mathcal{P}, \mathcal{M} \rangle$ , where  $\mathcal{P}$  is a power consumption specification and  $\mathcal{M}$  is a deadline miss ratio specification. Since the available energy changes dynamically,  $\mathcal{P}$  may need to be adjusted dynamically; for instance, the available energy at time  $t$  can be affected by workload changes and battery performance degradation. We assume that  $\mathcal{P}$  is set manually by a user or automatically by a higher-level power manager in consideration of the desired lifetime and the dynamics of the available energy. Achieving those target performance objectives requires each sub-component to satisfy its own performance objectives. As a sub-component of RTEDBS, the buffer manager is responsible for managing the I/O-related power consumption and miss ratio. Due to the data-intensive

property of RTEDBS, the I/O performance, both power consumption and miss ratio, has high impact on the overall system performance. In this paper, we assume that static performance objectives for I/O are given in the specification as  $\langle P_{I/O}, M_{I/O} \rangle$  from the RTEDBS at time  $t$ . The buffer manager controls I/O workloads to meet those performance objectives.

## 4.2 Architecture

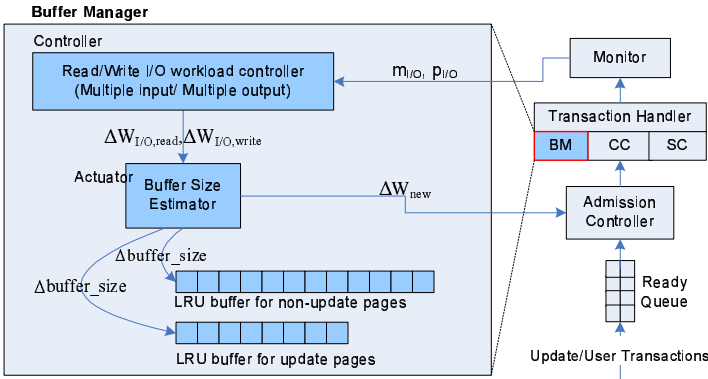


Figure 2. The architecture of buffer manager.

Figure 2 shows the architecture of the RTEDBS and the buffer manager. Transactions issued by sensors (update transactions) and users (user transactions) are placed on the ready queues. The dispatched transactions are managed by the transaction handler which consists of buffer manager (BM), concurrency control (CC), and scheduler (SC). In the SC, update transactions are scheduled in the high priority queue while user transactions are scheduled in the low priority queue. Within each queue, transactions are scheduled using Earliest Deadline First (EDF). Transactions are monitored by the monitor and the statistics of monitored transactions including deadline miss ratio and power consumption are reported to the components that need the information.

The buffer manager is a sub-component of a transaction handler. In our approach, the buffer manager consists of a global buffer pool, a controller, and a buffer size estimator. The global buffer pool is logically partitioned into two buffer pools for updated and non-updated pages, which is called *write buffer* and *read buffer*, respectively. A buffer page in the read buffer can move to the write buffer when the page is updated after its fetch; since the partition is logical no physical copying of the page is required. Each logical buffer pool uses a *LRU* buffer replacement scheme. We may consider using the information from the transaction handler to predict the pattern and choose the best replacement scheme. However, we left the issue of choosing replacement policies for varying access patterns as our future work. In this paper, we only address the problem of buffer allocation.

Because of the complexities inherent in real-world DBMS and workloads, accurately predicting the buffer allocation to

achieve the desired performance objectives is extremely difficult. Therefore, we use a feedback control loop, which adjusts the sizes of two logical buffer pools by monitoring the current system status. The controller gets the current system information from the monitor. The information from the monitor consists of the I/O deadline miss ratio,  $m_{I/O}$ , and the energy consumption from I/O,  $p_{I/O}$ , at sampling period  $t$ . With the information from the monitor, the controller estimates how much read and write I/O workload adjustments are required to meet the target power consumption and the deadline miss ratio. Once the target read/write I/O workloads,  $\Delta W_{I/O,read}$  and  $\Delta W_{I/O,write}$ , are obtained from the controller, the buffer size estimator estimates the size of each buffer pool to meet the target I/O workloads. The sizes of buffer pools are adjusted accordingly.

Because the workload is dynamic and memory is shared by many entities, not just by entities of RTEDBS, but also by other applications and middleware, the maximum available memory space to buffer pool is determined at runtime. In our approach, the sum of read/write buffer pools cannot be bigger than the maximum size of the global buffer pool. If the sum of two logical buffer pools should be bigger than the current maximum size of buffer pool to achieve the performance objectives, a request for additional memory allocation for the buffer pool is made. If additional memory is not allowed for the buffer, then admission control is applied to drop incoming transactions.

## 4.3 Buffer Partitioning

As shown in Table 1, write operations incur higher power consumption and longer response time in flash memory. Therefore, in terms of energy consumption, keeping more updated pages (or dirty pages) in the buffer pool at the expense of evicting non-updated pages can be advantageous. However, if only non-updated pages are evicted from the buffer pool regardless of reference patterns or locality, the scheme degenerates to the scheme where only updated pages reside. This can impair the response time and subsequently the deadline miss ratio of transactions due to high buffer cache miss ratio since most transactions access both updated and non-updated data pages. For instance, keeping index pages in the buffer can be very critical for fast response time of the transaction and minimizing deadline miss ratio. Furthermore, because write operations to the secondary storage are usually made by the buffer manager after transaction commits, they do not have explicit timing constraints and is less related to the deadline misses.

This argument tells us that the read and write workload can affect performance metrics differently, especially when read and write pages have radically different characteristics as in flash memory. Given this situation, the issue of how much buffer should be provided for each updated and non-updated pages to meet the target power consumption and deadline miss ratio becomes important. The sizes of buffer for updated

and non-updated pages respectively should be set to achieve both objectives.

The number of page frames to be allocated to updated and non-updated pages can be determined either by buffer allocation or replacement policies. In traditional approaches, which use the fixed size of buffer cache combined with LRU and its variant replacement schemes, the cost of each buffer page is not differentiated, thus, the number of updated and non-updated buffer pages are determined only by reference patterns or locality. However, this scheme is valid as long as the cost of reading and writing a page to/from the secondary storage is almost equal, which is not true when the secondary storage is flash memory. Some schemes solve the problem with cost-sensitive buffer replacement policy, which considers the cost of page eviction on replacement [9]. In these schemes, the proportion of updated and non-updated pages in the buffer pool are implicitly determined by replacement policies. However, estimating dynamic cost of each page is a very complex problem and sometimes requires additional information from databases. Moreover, the dynamic nature of performance objectives make the cost estimation of each page even harder. Therefore, the runtime cost of running these replacement policies can be non-trivial. This approach is clearly not suitable for resource-constrained embedded systems.

#### 4.4 Control Loop Design

Because of the complexities of DBMS and unpredictability of workloads, it is extremely difficult to predict the proper sizes of buffer for updated and non-updated pages to achieve the performance goals, if not impossible. Using feedback controllers has shown to be effective for such real-time systems with unpredictable workload [10][16]. Difference equation models of the feedback control are independent of load assumptions and consequently more suitable for systems where load statistics are difficult to obtain or where the load does not follow a distribution that is easy to handle analytically.

In this section, we present the design of our feedback control loop that controls read and write workloads to satisfy the desired I/O power consumption and the miss ratio.

##### 4.4.1 System Modeling

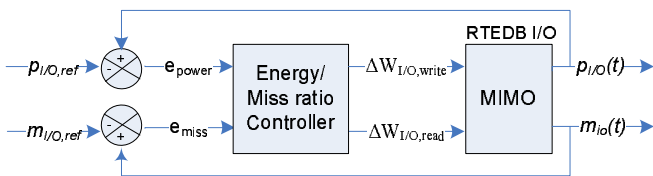


Figure 3. The feedback control loop.

The first step in the design of feedback control loop is the modeling of the controlled system [7]; the I/O sub-component of RTEDBS in our study. Since both I/O power consumption and miss ratio are affected by both read workloads and

write workloads, the I/O of RTEDBS is inherently *Multiple Inputs/Multiple Outputs* (MIMO). Therefore, to capture the close interaction between the multiple inputs [5],  $W_{I/O,write}$  and  $W_{I/O,read}$ , and the multiple outputs,  $p_{I/O}$  and  $m_{I/O}$ , we have chosen to use a MIMO model as shown in Figure 3.

Another issue in modeling a computing system is its non-linearity. Complex systems such as the I/O subsystem in RTEDBS can show a non-linear response to inputs. For example, the I/O deadline miss ratio behaves quite differently when the I/O is saturated from when it is not saturated. However, the system can be approximated quite closely with linear time invariant models such as the ARX model by choosing an operating region where the system's response is approximately linear [7]. Even when the system's response is highly non-linear, the system can be modeled with linear models by dividing the operating region into several sub-operating regions, where each region is approximately linear; in this case, adaptive control techniques such as gain scheduling [7] can be used for control. To this end, we choose to use linear time invariant model, which is shown in (2) with parameters **A** and **B**.

$$\begin{pmatrix} p_{I/O}(k+1) \\ m_{I/O}(k+1) \end{pmatrix} = \mathbf{A} \cdot \begin{pmatrix} p_{I/O}(k) \\ m_{I/O}(k) \end{pmatrix} + \mathbf{B} \cdot \begin{pmatrix} W_{I/O,write}(k) \\ W_{I/O,read}(k) \end{pmatrix}. \quad (2)$$

Because the I/O of RTEDBS is modeled as a MIMO system, **A** and **B** are  $2 \times 2$  matrices. A RTEDBS simulator which will be introduced in Section 5 was used for *system identification* [15] to get **A** and **B**. In the system identification, relatively prime sine wave workloads for read and write were applied simultaneously to get the parameters. In our study, the RTEDBS model has  $\mathbf{A} = \begin{pmatrix} 0.5914 & 0.0760 \\ -0.0006 & 0.0469 \end{pmatrix}$ , and  $\mathbf{B} = \begin{pmatrix} 0.1200 & 0.3364 \\ -0.0006 & 0.1501 \end{pmatrix}$  as its parameters. All eigenvalues of **A** are inside the unit circle; hence, the system is stable [7].

In terms of system order, note that we model the I/O of RTEDBS as a first-order system; the current outputs are determined by their inputs and outputs of the last sample. As we show later, the accuracy of the model is satisfactory and, hence, the chosen model order is sufficient for our purposes.

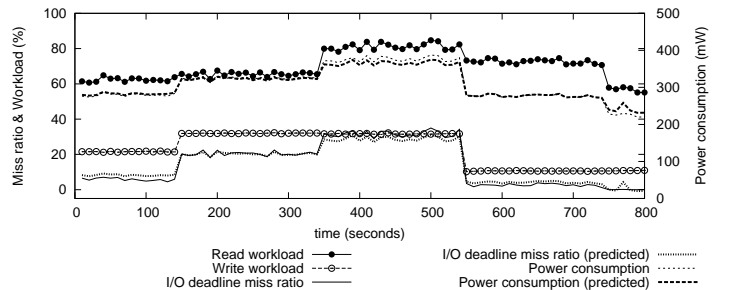


Figure 4. Model validation.

The model is validated by comparing the experimental result to what the model predicts. Figure 4 plots the

experimental response of the RTEDBS and the prediction of the model. We can see that the model gives highly accurate predictions. The accuracy metric  $R^2 = 1 - \frac{\text{variance}(\text{experimental value} - \text{predicted value})}{\text{variance}(\text{experimental value})}$  is 0.97 and 0.96 for the deadline miss ratio and the power consumption respectively. Usually,  $R^2 \geq 0.8$  is considered acceptable [7]. With regard to  $R^2$  and multi-step validation in Figure 4, the suggested first-order linear model is considered acceptable.

#### 4.4.2 Controller Design

For its robustness and simplicity, we choose to use a proportional integral (PI) control function given by,

$$U(k) = K_p \cdot E(k) + K_I \cdot \sum_{j=1}^{k-1} E(j), \quad (3)$$

where

$$U(k) = \begin{bmatrix} W_{I/O,read}(k) \\ W_{I/O,write}(k) \end{bmatrix}, \quad (4)$$

$$E(k) = \begin{bmatrix} p_{I/O,ref}(k) - p_{I/O}(k) \\ m_{I/O,ref}(k) - m_{I/O}(k) \end{bmatrix}, \quad (5)$$

and  $K_p$  and  $K_I$  are controller gains. (6)

Even though  $P$  controller is simpler than  $PI$  controller, our experiment shows that the system is not stabilized by  $P$  controllers. At each sampling instant  $k$ , the controller computes the control input  $U(k)$  by monitoring the control error  $E(k)$ .

One important design consideration in computing systems such as I/O in RTEDBS, which have a stochastic nature, is to control the trade-off between short settling times and overreacting to random fluctuations. If a controller is too aggressive, then the controller over-reacts to this random fluctuation. To this end, we choose to use the linear quadratic regulator (LQR) technique to find optimal control gains, which is accepted as a more general technique for MIMO systems [7]. LQR allows us to better negotiate the trade-offs between speed of response and over-reaction to random fluctuation by selecting appropriate weighting parameters. The obtained controller gains using LQR technique are  $\mathbf{K}_P = \begin{pmatrix} 0.3486 & -0.9611 \\ -0.7653 & 0.1324 \end{pmatrix}$  and  $\mathbf{K}_I = \begin{pmatrix} 0.0948 & -0.6519 \\ -0.1534 & 0.1802 \end{pmatrix}$ . For more details on LQR technique, readers are referred to [7].

Finally, in terms of sampling interval, we sample every 10 seconds. Our experiment shows that sampling intervals shorter than 10 seconds make the system unstable since it takes several seconds for the buffer hit ratio to change after adjusting the buffer size. Because of the relatively long sampling interval and the simple controller design, the overhead of our approach on the system is negligible.

#### 4.5 Read/Write Buffer Size Adjustment

Once target  $W_{I/O,read}$  and  $W_{I/O,write}$  are given from the controller, the target buffer hit ratios of the read buffer,

$HIT_{read}$ , and the write buffers,  $HIT_{write}$ , can be estimated from Equation 7 and 8, respectively.

$$HIT_{read} = 1 - \frac{W_{I/O,read}}{AW_{I/O,read}}, \quad (7)$$

$$HIT_{write} = 1 - \frac{W_{I/O,write}}{AW_{I/O,write}}, \quad (8)$$

where  $AW_{I/O,read}$  and  $AW_{I/O,write}$  are *applied read workload* and *applied write workload*, respectively. Unlike the real workload,  $W_{I/O}$ , *applied I/O workload*,  $AW_{I/O}$ , is the amount of I/O requests generated by transactions; among the I/O requests, only the requests that miss the buffer cache incur real I/O activities. Both  $W_{I/O}$  and  $AW_{I/O}$  are the ratio of amount of I/O requests to the maximum bandwidth of the secondary storage.

Finally, the sizes of read and write buffer to achieve these target buffer hit ratios can be estimated with an estimation function  $\lambda$ , which models the relation between the buffer size and the buffer hit ratio. In this paper, we use a linear approximation technique that was shown to be very effective if the linear approximation is updated regularly with latest buffer hit ratio and buffer size information [3].

## 5 Experiments and Results

The main objective of the experiments is to test the effectiveness of separating read and write workload by measuring how accurately the performance goals are achieved. Schemes that only control either power or I/O deadline miss ratio with a unified buffer pool is compared to our scheme. For experiments, we developed a simulator that models the proposed RTEDBS. Various workloads were applied to the simulator to test its performance.

### 5.1 Simulation Settings

To set proper simulation parameters, a firefighting scenario in urban high-rises is considered. The following scenario is adapted from [1].

In this scenario, a building has a *wireless sensor network* (WSN) that is composed of smoke and temperature sensors and radio beacon nodes that convey critical information to firefighters and occupants. Each firefighter has a head-mounted display (HMD) and a computer attached to his or her SCBA tank or in their turnout coat. The computer runs a RTDB, which has non-temporal building layout data and real-time sensor data from the building's WSN. Because of the vulnerability of network connections to the external database at the fire scene, each RTDB processes queries locally using local data, instead of depending on a database at back-end. Queries are invoked periodically and aperiodically on occurrences of specific events, and the results are visualized on the HMD with a floor plan image.

### 5.1.1 Database Model

The data in the system can be divided into 4 different categories: (1) sensor data from environments such as temperature, smoke, and motion, (2) sensor data from firefighters such as remaining oxygen level, motion, and current location, (3) information on each location, which includes geographical coordinates and material of walls, and finally, (4) digital map for localization and indoor navigation. (1) and (2) are temporal data, and (3) and (4) are non-temporal data. In our model, each category of data is managed by a relation; *SensorEnviro*, *SensorFireFighter*, *Location*, and *FloorPlan* relations manage respective category of data. The size of each relation depends on the number of sensors, deployed firefighters, and the size of the building. We assume that the size of each relation is 500, 500, 2000, and 3000 pages, respectively. Aside from the data managed by the database, the mobile system needs floorplan images of the building for graphical display.

### 5.1.2 Update Transactions

Parameter	Value
Update interval ( $P_i$ )	<i>Uniform</i> (1sec, 100sec)
$EET_i$	<i>Uniform</i> (2ms, 4ms)
# data object access/update	1
Update CPU load	$\approx 50\%$
Update write I/O load	$\ll 50\%$

**Table 2. Update transaction settings.**

The update stream updates only temporal data in *SensorValues* relation. The update period,  $p_i$ , follows a uniform distribution *Uniform*(1sec, 100sec); data from geographically close sensors are updated more frequently. The expected execution time ( $EET$ ) of an update transaction is uniformly distributed in the range (2ms, 4ms), excluding writing response time to the secondary storage. The actual execution time is determined by writing response time at run-time. Update transactions have no deadlines and they always have higher priority than any user transaction. The default settings shown in Table 2 generate about 50% CPU load and less than 50% applied I/O write workload,  $AW_{I/O,write}$ .

### 5.1.3 User Transactions

Parameter	Value
$EECT_i$	<i>Uniform</i> (3ms, 5ms)
Actual exec. time	<i>Normal</i> ( $EET_i$ , $\sqrt{EET_i}$ )
$EETI_i$	$\#AccessData \times ReadAccessTime/page$
Relative deadline	$(EECT_i + EETI_i) \times slack\ factor$
Slack factor	<i>Uniform</i> (5, 10)
Query mix	Type-I,-II,-III with equal arrival rates
Selectivity	Type-I,-II,-III equally 2%
Update probability	$\ll 1\%$

**Table 3. User transaction settings.**

A user transaction accesses both temporal and non-temporal data and possibly updates non-temporal data. User

transactions are read-intensive with small probability of update. In the firefighting scenario, user queries run periodically and aperiodically on occurrences of specific events to provide a building-wide situation and to alert potential dangers. For example, the queries may include “*Find locations where CO/CO<sub>2</sub> level is higher than the threshold within 10 meters from my location.*” and “*Find any motion detection within 10 meters from my location.*”. To model these operations, three different query workloads are employed in our experiments, (I)-*selection*, (II)-*nested-index join*, and (III)-*nested-loop join*. The selectivity of each operation is defined by the parameter  $Selectivity_I$ ,  $Selectivity_{II}$  and  $Selectivity_{III}$ . Our workload represents three different type of memory access patterns, which is typically found in a DBMS; looping, random access, and scanning.

The execution of user transactions consists of an I/O phase and a computation phase. The expected execution time ( $EECT_i$ ) of the computation phase is given by the uniform distribution, *Uniform*(3ms, 5ms). The expected execution time of the I/O phase ( $EETI_i$ ) is proportional to the number of data to access. Details of user transaction settings are shown in Table 3.

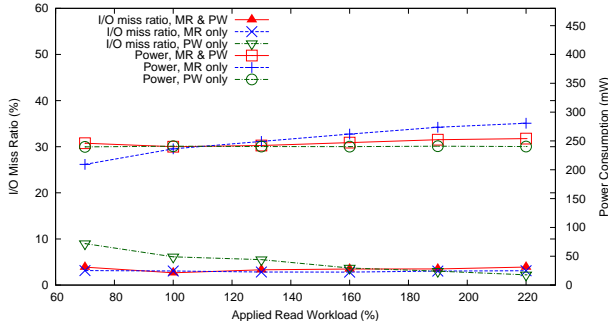
### 5.1.4 H/W Models

A NAND flash memory is assumed for persistent data storage. Read operations occur in the unit of a page. The size of a page is set to 4Kbytes. The actual cost of flash access is determined by the interface to the flash chip. We assume a *Flash Translation Layer* (FTL) [8] to interface the NAND flash chip. The FTL provides a disk like interface, which includes the capability to read and write a page directly without worrying about erase-before-write constraint. However, FTL internally needs to deal with the characteristics of the underlying flash device, incurring high overhead in flash accesses; the runtime overhead of FTL varies across manufacturers [17]. Moreover, combined with the limitation of data buses, the total access cost to flash memory ranges from several times to several thousands times of the raw flash memory access cost [14]. In consideration of these overhead, the cost of flash memory access is modeled as approximately 20 times of the raw flash memory access cost. The response times are set to 300 $\mu$ s and 3000 $\mu$ s, respectively for reading and writing a page, and each page read and write operation takes 14.8 $\mu$ J and 198 $\mu$ J, respectively. The energy consumption of I/O is modeled as  $\#flash\_read \times 14.8\mu J + \#flash\_write \times 198\mu J$ .

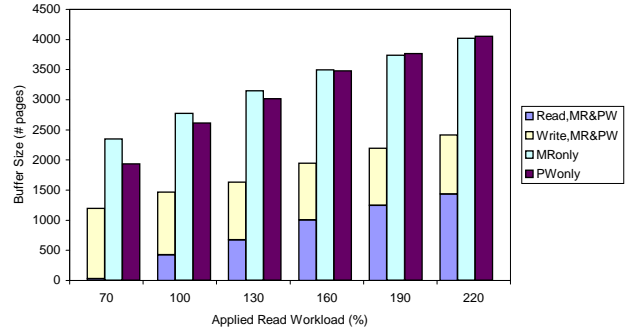
Each firefighter has a computer powered by a battery. The battery’s capacity is 1,600mAh and operates at 3.7V; these numbers are typically found in commercial PDAs.

## 5.2 Baselines

To our best knowledge, the issue of simultaneous control of power consumption and deadline miss ratio have not been

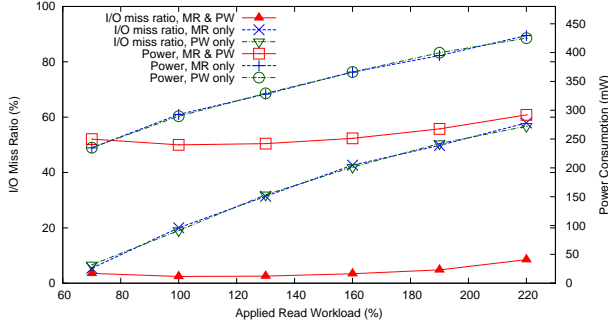


(a) Miss ratio and Power Consumption

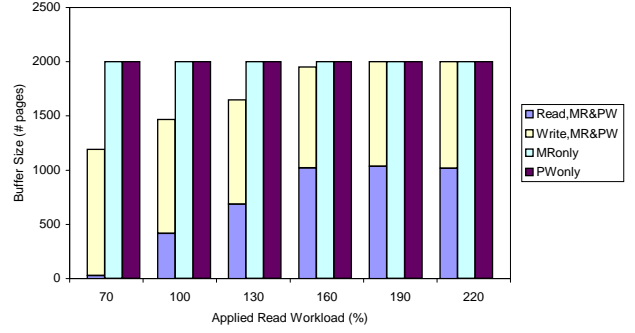


(b) Buffer size

Figure 5. Average performance when varying read workload with no limit on buffer size.



(a) Miss ratio and Power Consumption



(b) Buffer size

Figure 6. Average performance when varying read workload with maximum buffer size of 2000.

studied in real-time databases, and for buffer management in particular. Therefore, we compare our scheme (**MR&PW**) with the following baseline schemes.

**MISS RATIO ONLY (MRonly):** This scheme does not partition the buffer pool into read/write buffers. It has only one global buffer pool. The size of the buffer is adjusted via feedback control loop to satisfy I/O deadline miss ratio; the power consumption is not controlled. This scheme is similar to [3][4]. The main difference from them is the performance metric; while the goal of those approaches are I/O response time, *MRonly* uses I/O deadline miss ratio as a performance goal. For comparison to our approach, the I/O of RTEDBS is modeled by a first-order *Single-Input/Single-Output* (SISO) model; The I/O workload is the control input and the I/O deadline miss ratio is the system output. A *PI* controller is used.

**POWER-ONLY (PWonly):** This scheme is the same as *MRonly* except that it controls only power consumption instead of I/O deadline miss ratio.

### 5.3 Results

Each simulation is run at least 10 times and its average and 95% confidence interval is taken; confidence intervals are not plotted unless it deviates more than 10% from the average.

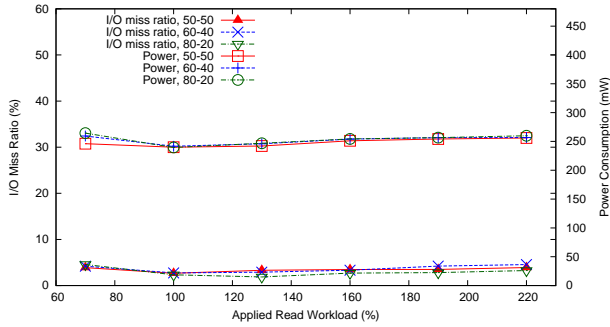
For experiments, the reference I/O miss ratio and the power consumption are set to 3% and 240mW, respectively.

With 240mW I/O power consumption, each fire-fighter's mobile computer can operate approximately for 2 and a half hours.

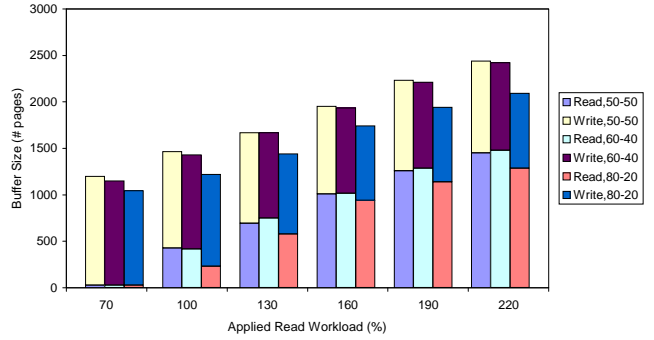
#### 5.3.1 Experiment 1: Varying Loads

Computational systems show different behavior for different workloads, especially when overloaded. In this experiment, read workloads are varied by applying increasing number of user transactions;  $AW_{I/O,read}$  is changed from 70% to 220%. The arrival rate of update transactions follows the default settings and does not change through experiments.

The first experiment examines the case when the available memory to buffer pool has no limitation; the buffer can expand/shrink without limitation in its size. The results are shown in Figure 5. It shows that our approach effectively achieves both the desired I/O deadline miss ratio and the power consumption. On the contrary, *MRonly* and *PWonly* achieve only I/O deadline miss ratio and power consumption, respectively. In case of *MRonly*, the power consumption increases monotonically as the applied read workload increases. For instance, the power consumption increases about 45% when  $AW_{I/O,read}$  changes from 70% to 220%. Similarly, even though *PWonly* achieves the desired power consumption, the desired I/O deadline miss ratio is not satisfied. Moreover, the performance of the baseline approaches are achieved with high cost as shown in Figure 5-b; both baseline approaches require about two times more buffer space to achieve those performances. Since the baseline approaches



(a) Miss ratio and Power Consumption



(b) Buffer size

Figure 7.  $x - y$  data access patterns with varying workload.

do not differentiate read and write workloads, the size of the total buffer pool is determined to accommodate pages, which is critical to achieve the desired performance goals; in this experiment, keeping a certain number of updated pages are critical to achieve the performance goals. In contrast, since our approach logically partitions the buffer pool into the read and the write buffers, the increase of  $AW_{I/O,read}$  affects only the size of read buffers while the size of write buffer stays almost constant, which is the minimum to achieve the desired performance goals.

In practice, the available main-memory space for the buffer can be quite limited in embedded systems. In the next set of experiments, the size of the buffer is set to hold 2000 data pages in maximum ( $4KB \times 2000 = 8MB$ ). The performance gap between our approach and baselines is even more evident in this case as shown in Figure 6. Since the baseline approaches reach the maximum size of the buffer through every case of the read workloads, the buffer can not be adjusted to achieve the performance goals, thus resulting in rapid increase of both I/O deadline miss ratio and the power consumption. In contrast, since the required size of buffer increases slowly in our approach, the maximum buffer size is reached only after  $AW_{I/O,read}$  is more than 190%. Even after reaching the maximum buffer size, both I/O deadline miss ratio and power consumption increase less rapidly than the baseline approaches.

### 5.3.2 Experiment 2: Varying Data Access Patterns

The I/O workload is highly affected by data access patterns. By default, we assumed that all relations are uniformly selected for user transactions. However, the data access patterns can be different from a uniform access pattern. In this experiment, the effect of data contention is tested using  $x - y$  access scheme as described in [10]. In the  $x - y$  access scheme,  $x\%$  of data accesses are directed to  $y\%$  of the data in the database. For instance, with 90-10 access pattern, 90% of data accesses are directed to 10% of data in the database, thus, incurring data contention on 10% of entire data.

We test the robustness of our approach by applying three different  $x - y$  access patterns; 80-20, 60-10, and 50-50 data access patterns. As shown in Figure 7-(a), our scheme

achieves the performance goals in all three different access patterns. However, the buffer size to achieve the same performance with different access patterns are a little bit different as shown in Figure 7-(b). As the degree of data contention increases, the smaller size of the buffer is enough to achieve the same degree of I/O deadline miss ratio and power consumption; 80-20 consumes about 15% less buffer space than 50-50 in all workloads.

Our results demonstrate that the proposed buffer management scheme is robust enough to cope with different data access patterns.

### 5.3.3 Experiment 3: Transient Performance

The average performance is not enough to show the performance of dynamic systems like RTEDBS. Transient performance such as settling times should be small enough to satisfy the requirements of applications.

In this experiment, the read workload is changed suddenly to observe the transient behavior of our scheme. Initially a 70% read workload is applied to the system. At 290 seconds, user transactions surge suddenly as a step function to have 220% applied read workload.

Figure 8 shows the result. We can see that both power consumption and I/O deadline miss ratio increase instantly at 290 seconds. However, the power consumption stabilizes within two sampling periods, and the I/O deadline miss ratio stabilizes within one sampling period. The relatively long settling time of the power consumption is not problematic in most cases since the average power consumption is more meaningful than the transient power consumption unless the power consumption requirement changes frequently. However, even though I/O deadline miss ratio stabilizes within one sampling interval, it may not be satisfactory for some applications since the sampling interval is relatively long (10 seconds). We may tune the system to be more responsive to the changes of the workload by reducing the sampling interval. However, a short sampling interval can make the system too sensitive to the stochastic components of the workload. These kinds of trade-offs between the responsiveness to workload changes and the sensitivity to stochastic components are inevitable in the design of a feedback controller for computing systems [7]. We

leave this issue as future work.

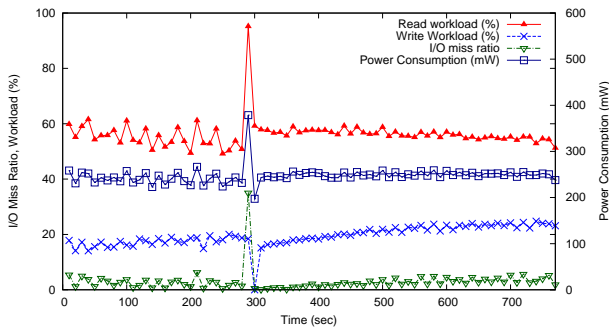


Figure 8. Sudden surge of read workload.

## 6 Conclusion

In this paper, we focused on the problem of guaranteeing the performance goals in the buffer manager of real-time embedded databases in terms of both I/O power consumption and I/O deadline miss ratio. Due to dramatically different properties of read and write operations of the flash memory, which is a de-facto standard in mobile and embedded systems, optimizing buffer hit ratio does not always guarantee minimum power consumption or I/O deadline miss ratio.

To address this problem, we have proposed logical partitioning of a buffer pool into a read and write buffers, and dynamic feedback control of read/write buffer sizes to satisfy the performance goals. The partitioning of a buffer pool enables us to effectively separate read and write workload, which have very different impact on system behavior both in terms of power consumption and deadline miss ratio. Unlike previous approaches, our approach uses a MIMO modeling and control technique to capture the close interactions of multiple inputs and outputs.

Using a detailed RTEDBS simulation model, we studied the performance of our approach under various workloads and data access patterns. For comparison purpose, we also examined two SISO approaches, which consider only single performance goal with a unified buffer pool. The experimental results show that our approach gives robust and controlled behavior in terms of guaranteeing both the desired power consumption and the I/O deadline miss ratio for diverse workloads and data access patterns, even in the presence of transient overloads. In particular, the results show that using MIMO approach to capture the interaction between multiple performance metrics can enable saving scarce resources of embedded systems, e.g., buffer space.

With the increase of the demand for real-time data services in flash-based embedded systems, the significance of providing guarantees on their power consumption as well as response times will increase. The work in this paper is the first attempt that addresses this problem.

## References

- [1] Fire Information and Rescue Equipment (FIRE) project, <http://fire.me.berkeley.edu/>.
- [2] M. Amirijoo, S. H. Son, and J. Hansson. QoS adaptation for achieving lifetime predictability of WSN nodes communicating over satellite links. In *Fourth International Conference on Networked Sensing Systems (INSS)*, June, 2007.
- [3] K. P. Brown, M. J. Carey, and M. Livny. Goal-oriented buffer management revisited. *SIGMOD Rec.*, 25(2):353–364, 1996.
- [4] J.-Y. Chung, D. Ferguson, G. Wang, C. Nikolaou, and J. Teng. Goal-oriented dynamic buffer pool management for database systems. Technical report, IBM RC19807, October, 1995.
- [5] Y. Diao, N. Gandhi, and J. Hellerstein. Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache web server. In *Network Operations and Management*, April, 2002.
- [6] D. F. Ferguson, L. Georgiadis, C. Nikolaou, and K. Davies. Goal oriented, adaptive transaction routing for high performance transaction processing systems. In *International Conference on Parallel and Distributed Information Systems (PDIS)*, 1993.
- [7] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. Wiley IEEE press, 2004.
- [8] INTEL. Understanding the flash translation layer (FTL) specification. application note ap-684, December 1998.
- [9] J. Jeong and M. Dubois. Cost-sensitive cache replacement algorithms. In *HPCA '03*, 2003.
- [10] 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*, 16(10):1200–1216, October 2004.
- [11] W. Kang, S. H. Son, J. A. Stankovic, and M. Amirijoo. I/O-aware deadline miss ratio management in real-time embedded databases. In *The 28th IEEE Real-Time Systems Symposium (RTSS)*, Dec, 2007.
- [12] G.-J. Kim, S.-C. Baek, H.-S. Lee, H.-D. Lee, and M. J. Joe. LGeDBMS: A small DBMS for Embedded System with Flash Memory. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 1255–1258, 2006.
- [13] S.-W. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007.
- [14] S. Lin, D. Zeinalipour-Yazti, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Efficient indexing data structures for flash-based sensor devices. *Trans. Storage*, 2(4):468–503, 2006.
- [15] L. Ljung. *Systems Identification: Theory for the User 2nd edition*. Prentice Hall PTR, 1999.
- [16] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Syst.*, 23(1-2):85–126, 2002.
- [17] S. Nath and A. Kansal. FlashDB: Dynamic self-tuning database for NAND flash. In *The International Conference on Information Processing in Sensor Networks (IPSN)*, 2007.
- [18] SAMSUNG. Samsung K9K1G08R0B 128M x 8bit NAND Flash Memory.
- [19] L. Selavo, A. Wood, Q. Cao, T. Sookoor, H. Liu, A. Srinivasan, Y. Wu, W. Kang, J. Stankovic, D. Young, and J. Porter. Luster: wireless sensor network for environmental research. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, 2007.