Cache Leakage Control Mechanism for Hard Real-Time Systems

Jaw-Wei Chi†, Chia-Lin Yang†, Yi-Jung Chen†, Jien-Jia Chen‡
Department of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan
{b89021, yangc, d91015}@csie.ntu.edu.tw†
jjchen@newslab.csie.ntu.edu.tw‡

ABSTRACT

Leakage energy consumption is an increasingly important issue as the technology continues to shrink. Since on-chip caches constitute a major portion of the processor's transistor budget, several leakage control policies have been proposed to reduce cache leakage. However, these policies introduce performance unpredictability thereby not suitable for hard real-time applications that require the timing constraint is met in all cases. In this paper, we propose the first approach to apply existing low leakage circuit techniques on hard real-time applications. The proposed timing-aware cache leakage control mechanism exploits task slack time to turn cache lines into the low-leakage state provided that the timing constraint is met. The experimental results show that the proposed cache leakage control policy achieves comparable leakage reduction to the leakage control policy that aggressively turns cache lines into low-leakage modes without considering the timing constraint.

Categories and Subject Descriptors

B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids

General Terms

Algorithm, Design, Performance

Keywords

Cache leakage control policy, Hard real-time system

1. INTRODUCTION

Power consumption is becoming a critical design issue of embedded systems due to the popularity of portable devices such as cellular phones and personal digital assistants. As the technology continues to shrink, leakage power is becoming a dominant factor to overall CPU energy [13]. Reducing leakage energy can be done by exploiting task idle time to shut down the CPU completely [4, 5, 9, 10] or individual micro-architecture component, for example, caches [7, 20] and branch predictors [11]. Previous works on applying shutting down techniques to hard real-time systems only focus on turning off a CPU completely [4, 5, 9, 10]. We are not aware of any research work that applies micro-architectural leakage reduction techniques to hard real-time systems. This paper is the first attempt to bridge this gap.

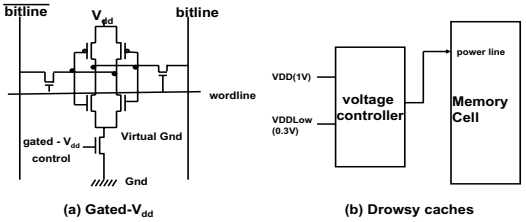


Figure 1: Leakage reduction circuits: (a) gated-Vdd, and (b) drowsy caches.

In this paper, we target at on-chip cache leakage reduction. On-chip caches constitute a major portion of the processor's transistor budget and account for a significant share of leakage. In fact, leakage is projected to account for 70% of the cache power budget in 70nm technology [13] . Therefore, reducing cache leakage power consumption is important for reducing a processor's total leakage. Two types of circuit techniques have been proposed to reduce cache leakage: Gated-Vdd [20] and drowsy caches [7]. Figure 1 shows the circuit of gated-Vdd and drowsy caches. The gated-Vdd technique turns off a cache line completely to save maximum leakage power, but the loss of state exposes the system to incorrect turn-off decisions which result in significant performance penalty. The drowsy cache technique uses a small supply voltage to retain the data in a memory cell at the low-leakage state [7, 14]. Therefore, the drowsy cache technique reduces leakage less than the gated-Vdd technique, but it incurs much less penalty when accessing a memory cell at the low-leakage state. The delay to switch a memory cell from the low-leakage state to the active state is called wake-up overhead. To decide when to turn a cache line into the low-leakage state is called a

CASES'07, September 30–October 3, 2007, Salzburg, Austria.

cache leakage control policy. Existing control policies are in two broad categories: application-sensitive and application-insensitive policies. Application-sensitive policies use the feedback from applications to perform leakage control, e.g., DRI-cache [22] turns off I-cache lines based on I-cache miss rates. Application-insensitive policies periodically put cache lines into the low-leakage mode regardless of applications' behavior, e.g., the simple policy in [7] turns all the cache lines in a cache to the drowsy mode at every fixed period. None of these control policies provide precise timing control. For a hard real-time system that requires the system to meet the timing constraint in all cases, even slight performance degradation could cause catastrophic system breakdown.

Contribution

In this paper, we propose the first timing-aware cache leakage control mechanism for hard real-time systems. To achieve energy savings with hard real-time guarantee, we exploit both static and dynamic slack to tolerate delay caused by accessing low-leakage cache lines.

Unlike previous works that choose between the drowsy cache or gated-Vdd, our scheme allows the joint use of both techniques. We exploit task-level information to manage cache lines of idle and active tasks differently. For cache lines allocated to an active task, due to short idle period between accesses, only the drowsy cache technique is considered. Cache lines of an active task are turned into the drowsy mode periodically, and waken up when they are accessed. The period to turn all cache lines to the drowsy mode is referred to as the drowsy window size. A smaller drowsy window size leads to higher leakage savings at the cost of higher wake-up overheads. Our timing-aware cache leakage control mechanism chooses the smallest drowsy window size provided that the timing constraint is met. For cache lines allocated to idle tasks, we seek opportunities to turn cache lines off completely to get more leakage gain as long as the penalty of fetching data from the lower level memory hierarchy does not cause the violation of timing constraint. In summary, this paper makes the following contributions:

1. We propose the first timing-aware cache leakage control mechanism that allows a hard real-time system to take advantage of existing cache leakage reduction techniques.
2. Our timing-aware cache leakage control scheme exploits task-level information to allow the joint use of drowsy caches and gated-Vdd techniques. For cache lines allocated to idle tasks, we seek opportunities to turn cache lines off completely to get more leakage gain.
3. Our timing-aware leakage control mechanism has the capability to adjust the drowsy window size dynamically with hard real-time guarantee, while previous works could only use a fixed drowsy window size throughout program execution[7, 14]. Run-time drowsy window resizing allows us to choose a smaller window size whenever the system has more slack time to achieve more leakage reduction.

We evaluate the proposed leakage control scheme on 8 real applications. The experimental results show that with tight deadlines, the simple policy in [7] causes high deadline miss ratio. (e.g., with 1% static slack [1], the deadline miss ratio [2] is up to 97.6%.) This confirms our assertion that existing leakage reduction techniques are not suitable for hard real-time applications, and a timing-aware leakage control scheme is a must. When static slack is 1% where the simple policy achieves 89.7% to 90.6% leakage saving, the proposed scheme achieves less leakage savings than the simple policy (78.4% to 86.9%) in order to satisfy the timing constraint. As task slack increases, the leakage savings of the proposed method approaches that of the simple policy. With 20% of static slack, our scheme even achieves up to 1.3% more leakage savings than the simple policy. This energy advantage provided by the proposed scheme comes from run-time drowsy window resizing. The experimental results also show that the joint use of the drowsy cache and gated-Vdd technique provides up to 2.8% more leakage reduction compared to that of adopting the drowsy technique alone.

The rest of the paper is organized as follows. Section 2 describes the previous works. Section 3 gives the background of this work. Section 4 describes the basic system model discussed in this paper. The proposed timing-aware cache leakage management policy is described in Section 5. Section 6 describes the architectural and OS support for the proposed scheme. The experimental results are discussed in Section 7, and Section 8 offers our conclusions and future work.

2. PREVIOUS WORK

Several works have been proposed to reduce dynamic or leakage power of real-time systems [19, 9, 10, 17]. To reduce CPU leakage energy, Martin et al. [17] propose a new scheduling algorithm that combines DVS (Dynamic Voltage Scaling) and adaptive body biasing to simultaneously optimize both dynamic and leakage power consumption in real-time systems. Niu et. al. [19] propose a method that combines DVS and CPU shut-down to minimize the overall energy consumption for hard real-time systems. Jejurikar et. al. [9, 10] present a procrastination scheduling technique to maximize the duration of idle intervals by keeping the processor in a sleep/shutdown state even if there are pending tasks, within the constraints imposed by performance requirements. All the above works that minimize CPU leakage consumption for hard real-time systems only focus on turning off a CPU completely.

3. BACKGROUND

In this section, we introduce existing cache leakage control policies and the cache locking algorithm adopted in our work.

3.1 Cache Leakage Control Policies

There are two broad categories of cache leakage control policies: application-sensitive policies [22, 23, 18] and appli-

[1]Static slack = 1 - $\sum_i^n \frac{W_i}{P_i}$, where W_i and P_i are the WCET and period of a task i among n tasks in a task set.

[2]Deadline miss ratio = $\frac{N_{miss\_tasks}}{N_{total\_task}}$, where N_{miss_tasks} is the number of tasks that missed their deadline, and N_{total_tasks} is the total number of executed tasks.

cation insensitive policies [12, 7, 14]. Application sensitive techniques decide when and where to do leakage control according to the feedback from applications. DRI-cache [22] employs gated-Vdd and reduces I-cache leakage power by resizing the cache according to the variation of I-cache miss rates. Zhang et al. [23] present a compiler approach that turns off the cache lines of code regions that would not be accessed for a long period of time. In [18], the upper bound of leakage power reduction in caches is estimated, and a prefetching scheme that combines both drowsy caches and the gated-Vdd technique is proposed to approximate optimal cache leakage reduction. Contrast to application sensitive policies, application insensitive policies periodically turn cache lines into the low-leakage state regardless of applications' behavior. Cache decay [12], which adopts the gated-Vdd circuit, turns off a cache line completely when it is not accessed for a period of time. Simple policy in [7] that cooperates with the drowsy cache circuit turns all cache lines into the drowsy mode periodically. This is also the policy we adopt in this paper for managing cache lines allocated to active tasks. We call this policy as Drowsy+Simple in this paper. Because Drowsy+Simple incurs more performance degradation in instruction caches than in data caches, Kim et al. [14] propose an architectural control mechanism that cooperates with drowsy caches for instruction cache leakage reduction without significant impact on execution time. The control technique in [14] divides an instruction cache into regions called *banks*, and cache lines are waken up from the drowsy mode in the granularity of bank. A bank prediction scheme is also proposed to reduce the performance overhead.

Based-on the above discussion, we know that both application -sensitive and application-insensitive leakage control techniques introduce unpredictable performance overhead and thereby not suitable for hard real-time applications.

3.2 Cache Locking Technique

For real-time applications, analyzing the worst case execution time (WCET) is critical. The use of caches complicates the WCET analysis due to the unpredictability in cache behavior. In a multitasking environment, the unpredictable cache behavior comes from both inter- and intra-task interference. One way to cope with this problem is cache locking [21], which restricts cache usage so as to eliminate both inter- and intra-task interference. Cache locking loads and locks cache contents to ensure that cache contents remain unchanged during program execution. The ability to lock cache contents is available in several commercial processors, e.g., PowerPC 440 and ARM946E-S. In ARM946E-S, a set of instructions for cache locking are provided, and cache locking is achieved by programming the instructions with the memory addresses of contents to be locked [2].

To decide the contents to be locked in a cache, the algorithm in [21] selects the locked contents of the cache so as to minimize the CPU utilization. The CPU utilization U is estimated by

$$U = \sum_{i=1}^{n} \frac{W_i}{P_i} \quad (1)$$

, where W_i denotes the WCET of task i and P_i denotes the period of task i. Therefore, the algorithm tries to lock data that are accessed in the worst-case execution path and have high access frequencies. The algorithm targets at a set-associative cache in a multitasking environment. They first identify data mapped to the same cache set, and sort these data in a non-decreasing order of $n_{load(i,j)}/P_i$ ratio, where $n_{load(i,j)}$ is the number of access to the j-th data of task i along the worst-case execution path. For an N-way cache, the first N data are selected to be locked in the cache.

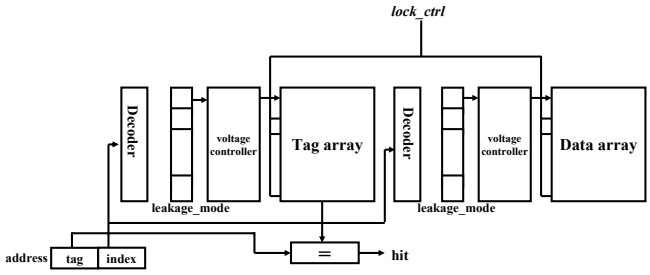


Figure 2: Baseline cache architecture.

4. SYSTEM MODEL

The system consists of a task set of the n periodic real-time tasks. These tasks are independent tasks and preemptable. Tasks are denoted as $\mathcal{T} = \{\tau_0, \tau_1, ..., \tau_n\}$, where $\mathcal{T}$ denotes the task set and τ_i denotes the i-th task of n tasks. Each τ_i has its own period P_i and its WCET W_i. We assume a task's deadline is its period. Tasks are scheduled using the EDF (Earliest Deadline First) scheduling policy. A task with earlier deadlines gets higher priority. The scheduler has two queues: waiting queue ($Q_{waiting}$) and ready queue (Q_{ready}). The waiting queue contains the completed tasks, and the ready queue contains the running and preempted tasks. The task that is currently running is the active task, and the task that is preempted or completes is the idle tasks. The schedualibitlity of a task set is tested by the CPU utilization U defined by Eq.(1). If U is less than 100%, the task set is said to be schedulable.

The baseline cache architecture that supports cache locking described in Section 3.2 is shown in Figure 2. The *lock_ctrl* signal indicates whether a cache line can be replaced or not. We select instructions to be locked in the instruction cache based on the locking algorithm described in Section 3.2. Each cache line is associated with the leakage_mode bits to select the supply voltage. A cache line can be turned into either the state-preserving mode (i.e. drowsy caches) or state-destructive mode (i.e. the gated-Vdd circuit). We use the terms drowsy mode and state-preserving mode interchangeably in this paper. A cache line switches to the active state once it is accessed.

5. TIMING-AWARE CACHE LEAKAGE CONTROL

The objective of the proposed leakage management method is to determine the drowsy window size for active tasks and the leakage mode for idle tasks, provided that the timing constraint is not violated. In this section, we present our leakage management method for both active and idle tasks.

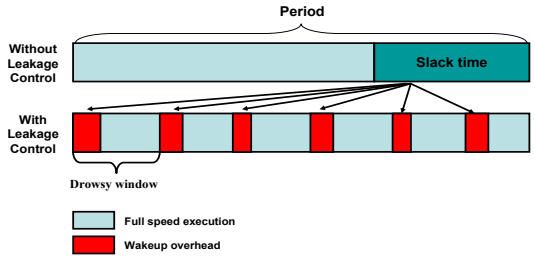


Figure 3: Illustration of utilizing task slack time to tolerate the performance overhead caused by activating cache lines in the low-leakage mode.

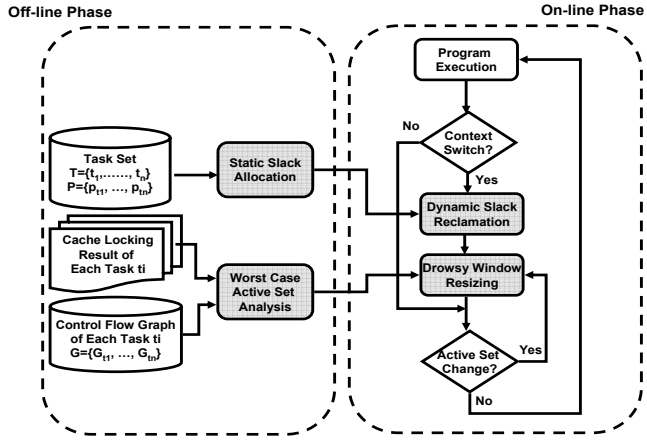
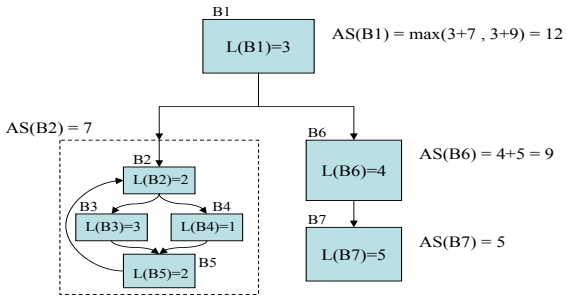


Figure 4: Overview of the timing-aware cache leakage control mechanism for active tasks.

5.1 Leakage Control Scheme for Active Tasks

The leakage control scheme for active tasks is based on the Drowsy+Simple policy proposed in [7]. Different from Drowsy+Simple that uses fixed drowsy window size, the leakage control scheme for active tasks adjusts drowsy window size dynamically with hard real-time guarantee. The drowsy window size affects leakage savings and the performance overhead caused by waking-up drowsy cache lines. With a smaller window size, cache lines are set to the drowsy mode more frequently thereby achieving higher leakage reduction, but it also causes more wake-up overheads. As illustrated in Figure 3, to meet the timing constraint, the total wake-up overheads cannot exceed a task's slack. Therefore, our leakage control scheme is to decide the smallest drowsy window size so as the timing constraint is met. That is, the wake-up overhead of all drowsy windows does not exceed the total slack time. The slack time of a task comes from two sources. One is called static slack that is computed based on the WCET. The other is called dynamic slack which is due to variations of task execution time.

The proposed leakage control scheme contains the off-line phase and on-line phase as shown in Figure 4. In the off-line phase, static slack allocation and the worst case active set analysis are performed. Static slack allocation assigns slack time to each task, and the worst case active set analysis estimates the number of cache lines that can be accessed in a drowsy window in the worst case. In the on-line phase, we perform dynamic slack reclamation and drowsy window resizing. Dynamic slack reclamation reclaims dynamic slacks due to variations of task execution time. Dynamic slack reclamation is performed when context switches occur. Drowsy window resizing is to decide the drowsy window size of each task, and it is performed when context switches occur or the active set changes. Below we describe the proposed scheme in details.



B1,B6,B7 : Normal basic block.
B2,B3,B4,B5 : Merged as one basic block since they are in a loop.

L(Bi): number of locked cache lines touched by Bi.
AS(Bi): Active set size of basic block Bi.
AS(Bi) = max {L(Bi) + Active(Bj)} ,∀Bj ∈ child(Bi)

Figure 5: An example of the control flow graph for the worst case active set analysis.

5.1.1 Off-line Phase

Static Slack Allocation

We first allocate static slack to tasks based on their worst case preemption rates. We allocate less slack time to tasks with higher preemption rates. In our timing-aware leakage control scheme, the slack time of a task cannot be utilized by other tasks until it completes. A task that has a higher preemption rate tends to complete later than other tasks. Therefore, tasks with higher preemption rates are allocated less static slack. Assume for all i, j, if $i < j$, then $P_i < P_j$. The number of preemption $PN(\tau_k)$ of a task τ_k in the worst case is

$$PN(\tau_k) = \sum_{i=1}^{k-1} \lfloor \frac{P_k}{P_i} \rfloor.$$

The static slack time, ρ_k, allocated to a task k is

$$\rho_k = P_k \times (1 - U) \times \frac{1/PN(\tau_k)}{\Sigma_{i=1}^{n} 1/PN(\tau_i)}.$$

Worst Case Active Set Analysis

To estimate the performance overhead caused by activating drowsy cache lines in a drowsy window, we need to predict the number of cache access in a drowsy window. The number of cache lines that can be accessed in a drowsy window in the worst case is all the cache lines that could be accessed in the future. To obtain this information, we first construct the CFG (Control Flow Graph) of a program. In the CFG, each node represents a basic block, and an edge from node a to node b indicates that an execution path exists from basic block a to basic block b. Figure 5 shows an example of the CFG. The worst case active set analysis is performed on the CFG. As shown in Figure 5, each node is associated with $L(B_i)$, which is the number of locked cache lines in basic block B_i. The worst case active set size of each node

B_i, which is denoted by $AS(B_i)$, is the maximal number of locked cache lines that could be accessed from B_i. Therefore, $AS(B_i)$ is calculated by

$$AS(B_i) = \max\{L(B_i) + AS(B_j)\}, \forall B_j \in child(B_i)$$

, where $child(B_i)$ denotes the set of child nodes of B_i. For example, the basic block B_1 in Figure 5 has active set size $AS(B_1) = \max\{L(B_1) + AS(B_2), L(B_1) + AS(B_6)\} = 12$.

Worst case active set analysis is performed at compile time. To convey the worst case active set size to the cache controller, which performs the leakage control, we use a store instruction to write the worst case active set size to the cache controller. Because the drowsy window resizing process is triggered on seeing a change in the worst case active set size, we insert the store instructions only at the loop entry point to prevent frequent drowsy window resizing. As shown in Figure 5, B_2, B_3, B_4 and B_5 form a loop, and the active set size information is recorded on B_2 only.

5.1.2 On-line Phase

Dynamic Slack Reclamation

Dynamic slack is from variations of task execution time, and the collection of the dynamic slack time is performed by the OS when a context switch occurs. The dynamic slack reclamation process used here is similar to the one proposed in [15]. Before we detail dynamic slack reclamation, we first define five notations:

- U_i^{CPU}: the unused CPU budget of τ_i
- W_i^{rem} the remaining WCET of τ_i
- S_i: the slack time of τ_i
- E_i: the execution time of τ_i
- DS: dynamic slack time

When a task arrives (i.e., removed from the waiting queue), U_i^{CPU} and W_i^{rem} are initialized to (WCET + static slack) and WCET, respectively. During the execution of τ_k, U_i^{CPU} is consumed, and W_i^{rem} decreases. W_i^{rem} is updated by the cache controller, and the value is automatically decremented by one at every cycle. Note that we do not claim the slack time of preempted tasks as in [15]. In our scheme, a preempted task could utilize its slack to turn its cache lines into the low-leakage mode during the idle period.

When τ_i is preempted or completes, we first consume the dynamic slack (DS) from unused CPU budget of the tasks in $Q_{waiting}$ with earlier deadlines. Then, we update U_i^{CPU} of task τ_i. DS is estimated by the following equation:

$$DS = \sum_{\tau_k \in Q_{waiting}} U_k^{CPU}.$$

If DS is greater than E_i, U_i^{CPU} is not consumed. Otherwise, the CPU budget is updated using the following formula.

$$U_i^{CPU} = U_i^{CPU} - (E_i - DS).$$

Therefore, the slack time that a task can use to compensate the wake-up overheads is

$$S_i = (U_i^{CPU} - W_i^{rem}) + DS.$$

Drowsy Window Resizing

The process of drowsy window resizing is to decide the smallest drowsy window size such that the timing constraint is met. Drowsy window resizing is performed when a context switch occurs or when the current active set is changed. To decide the drowsy window size of the scheduled task, we have to find the smallest drowsy window size with the wake-up overhead that is not larger than the task's available slack. Therefore, the drowsy window size is the smallest window size that satisfies the following inequality:

$$\lceil \frac{W_i^{rem}}{wsize} \rceil \times S_{active(i)} \times OH \leq S_i \quad (2)$$

, where $wsize$ denotes the window size, $S_{active(i)}$ denotes the worst case active set size of task τ_i, and OH denotes the number of cycles to wake up a drowsy cache line.

5.2 Leakage Control Scheme for Idle Task

The cache lines of idle tasks could be turned into the state-preserving or state-destructive mode. When a context switch occurs, the OS decides the leakage mode of the task that is switched out during its idle period based on its idle time and available slack. The slack S_i and idle period I_i of a task τ_i that is preempted or completes are given below:

$$\begin{cases} S_i = U_i^{CPU} - W_i^{rem} \\ I_i = BCET(\tau_{curr}) \quad \texttt{preempted tasks} \end{cases}$$

$$\begin{cases} S_i = \rho_i \\ I_i = T_{arrive}(\tau_i) - T_{enter\_q}(\tau_i) \quad \texttt{completed tasks} \end{cases}$$

, where $BCET(\tau_{curr})$ is the best case execution time of the current active task, $T_{arrive}(\tau_i)$ is the next arrival time of τ_i, and $T_{enter\_q}(\tau_i)$ is the time τ_i entering the waiting queue.

To decide the leakage mode of an idle task, we need to evaluate the performance overhead ($P_{overhead}(M_i)$) and the energy overhead ($E_{overhead}(M_i)$) of a low-leakage mode M_i, where M_i is either the state-preserving or state-destructive mode. $P_{overhead}(M_i)$ and $E_{overhead}(M_i)$ are:

$$P_{overhead}(M_i) = N_{wake} \times D_{wake}(M_i)$$

$$E_{overhead}(M_i) = N_{wake} \times E_{wake}(M_i)$$

, where N_{wake} denotes the number of times to wake up cache lines in the low-leakage mode, and $D_{wake}(M_i)$ and $E_{wake}(M_i)$ denote the delay and energy overhead to wake up cache lines in the low-leakage mode M_i. For the state-preserving mode, the wake-up latency is 2-cycle when both the tag and data array are in the drowsy mode, and the wake-up energy is the energy required to charge a drowsy cache line from the drowsy state to the active state. For the state-destructive mode, the wake-up overhead is the latency and energy to access the next level memory hierarchy.

To turn an idle task's cache lines into a low-leakage mode M_i, the task must have

(1) $P_{overhead}(M_i) \leq S_i$, and

(2) $E_{overhead}(M_i) \leq E_{leak\_reduction}(M_i)$

, where $E_{leak\_reduction}(M_i)$ denotes the leakage reduction obtained by applying low-leakage mode M_i. $E_{leak\_reduction}(M_i)$ is derived from the following formula:

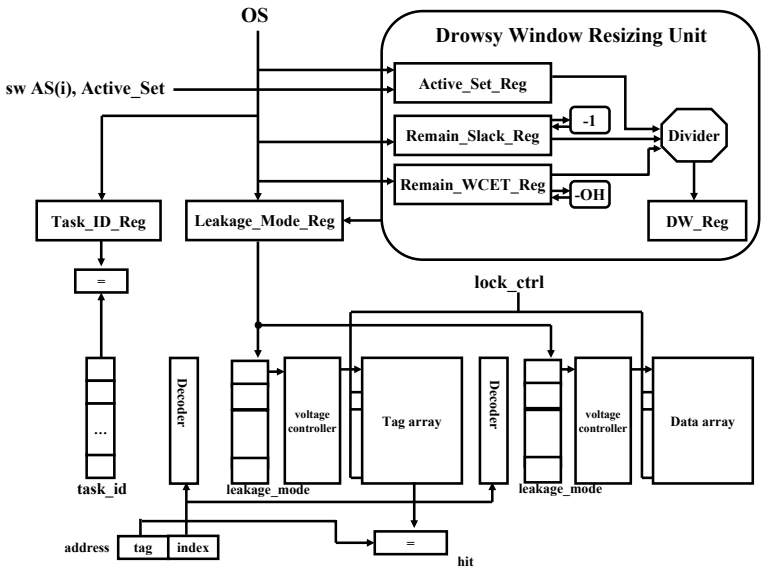


Figure 6: Hardware support for the proposed timing-aware leakage control scheme.

$E_{leak\_reduction}(M_i) =$
$(E_{leak\_active} - E_{leak\_low}(M_i)) \times I_{idle} - E_{overhead}(M_i)$

, where $E_{leak\_active}(M_i)$ and $E_{leak\_low}(M_i)$ denote the leakage energy of cache lines in the active and low-leakage mode M_i, respectively. I_{idle} is the idle length of the idle task.

To determine the leakage mode of idle tasks, we evaluate the performance overhead and leakage reduction achieved by both the gated-Vdd and drowsy cache circuits. The low-leakage mode which achieves the most leakage reduction while meeting the timing constraint is selected as the low-leakage mode of an idle task.

6. ARCHITECTURAL AND OS SUPPORT

This section describes the required modifications to the cache controller and the operating system where a context switch is handled, including (1) task-aware leakage control, (2) drowsy window resizing and (3) restoring cache contents of cache lines that are put into the state-destructive mode.

To support task-aware leakage control, we add an additional field to record the corresponding task id of a cache line as shown in Figure 6. The current task id is stored in the Task_ID_Reg register. A cache line is allowed to switch its leakage mode only when its task id matches the current task id. The Leakage_Mode_Reg register keeps the leakage mode for the current task. A write to the Leakage_Mode_Reg register shall trigger the update of the leakage_mode bits for the cache lines of the current task.

To support drowsy window resizing, the cache controller is enhanced with the Drowsy Window Resizing Unit as shown in Figure 6. The drowsy window resizing unit uses three registers, Remain_WCET_Reg, Remain_Slack_Reg and Active_Set_Reg, to store W_i^{rem}, S_i, $S_{active(i)}$ in Eq.(2). Remain_WCET_Reg is automatically decremented by one at every cycle, and Remain_Slack_Reg is automatically decremented by OH when a drowsy cache line is waken up. The values in Remain_WCET_Reg and Active_Set_Reg are saved /restored during context switch. When a context switch occurs, the OS first determines the leakage mode of the current task during its idle period through the algorithm described in Section 5.2. The OS then performs dynamic slack reclamation described in Section 5.1 to set the value of the Remain_Slack_Reg register for the newly scheduled task. Note that the Active_Set_Reg register is updated during program execution when the active set changes. Obtaining the drowsy window size, according to Eq.(2), requires one multiply and divide operations. To reduce hardware resources, this calculation is transformed into two divide operations as shown in the following equation:

$$wsize = \frac{W_i^{rem}}{\frac{S_i}{S_{active(i)} \times OH}}$$

Since OH is fixed at two, we use a shift operation to perform $S_{active(i)} \times OH$. According to the divider implementation provided in [6], the divide latency is 16 cycles. So the cache controller takes 32 cycles to estimate the drowsy window

Table 1: Simulated architecture parameters.

Processor Core	
Instruction Window	16-RUU, 16-LSQ
Issue width	1 instruction per cycle, in-order issue
Functional Unit	4 IntALU, 1 IntMult Div
	1 FPALU, 1 FPMult Div
	2 mem ports
Memory Hierarchy	
L1 I-cache	Size 8KB, 2-way, 16B block size
L2 cache	Size 32KB, 4-way, 32B block size 8-cycle access latency
Memory	12-cycle access latency
Energy Parameter	
Process Technology	0.07um
Supply Voltage	0.9V
Temperature	100^oC

Table 2: Task sets characterization.

Name	Description	Code size	WCET
Small task set (Total code size 7608 bytes)			
jfdctint	JPEG integer implementation of the forward DCT	3296	19087
crc	cyclic redundancy code example program	1400	142088
ludcmp	Linear equations by LU decomposition	2336	16607
matmult	Matrix multiplication	576	12555
Medium task set (Total code size 9192 bytes)			
qurt	Computation of roots of quadradic equations	1200	4038
minver	Matrix inversion	3656	11281
jfdctint	JPEG integer implementation on DCT	3296	18969
fftl	FFT Cooly-Turkey algorithm	1040	8685

size. Since the cache is still accessible to the CPU during the drowsy window resizing process, this does not incur any performance overhead. One way to avoid expensive divide operations for calculating the drowsy window size is to use a table-lookup approach. We leave it as the future work. The resulting drowsy window size is stored in DW_Reg.

To be able to restore locked cache contents for cache lines that are put into the state-destructive mode, we set the tag array into the drowsy mode only. For a cache access that results in a tag hit but a data miss, a request of the hit tag address is issued to the lower level of the memory hierarchy to fetch the data, and the *lock_ctrl* signal is turned off to allow data write-back.

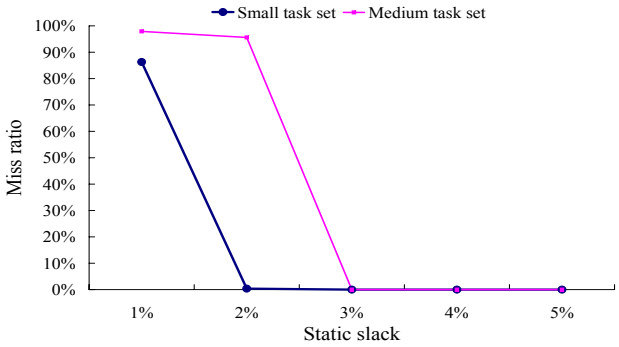


Figure 7: Deadline miss ratio of Drowsy+Simple.

7. EXPERIMENTAL RESULTS

For cache leakage evaluation, we use the HotLeakage tool set [24]. HotLeakge is developed based on the Wattch [3] tool set. HotLeakage explicitly models the effects of temperature, voltage, and parameter variations, and has the ability to recalculate leakage currents dynamically as temperature and voltage changed at runtime due to operating conditions, DVS techniques, etc. To simulate multi-tasking workloads, we modified HotLeakage to allow multiple programs executing simultaneously. We also implement the EDF scheduler. In our experiment, cache locking is performed on L1 I-cache. Since we also put cache tag into the drowsy mode, the performance overhead of accessing a drowsy line is set to 2 cycles according to [16], and the power overhead is 0.3mW. The access delay of the L2 cache is set to 8 cycles. The detailed processor and memory hierarchy parameters are shown in Table 1. We implement two leakage control mechanisms, the Drowsy+Simple scheme proposed in [7], and the proposed timing-aware leakage control scheme (TALC). For the Drowsy+Simple scheme, we determined the drowsy window size through exhaustive simulations and chose the best one on the average, 1000-cycle [7]. The cache lines allocated to idle tasks are turned into the drowsy mode immediately when a context switch occurs.

The benchmarks used in this work are from the SNU real-time benchmark suite [1]. The benchmark programs are C sources which are collected from numerical calculation programs and DSP algorithms. We mix multiple applications together to form two multi-tasking workloads, the small task set and the medium task set. The small task set has about 7KB total code size, and the medium task set has about 9KB total code size. Details of the workloads are listed in Table 2. The WCET of each task is measured with cache locking. To generate varying execution time, we use the method similar to [8]. We assume the BCET of a task as a percentage of its WCET. In our experiments, the (BCET/WCET) ratio is set to 0.95. The execution time of each task instance is generated by a normal distribution with mean $\mu = (WCET + BCET)/2$ and standard deviation $\rho = (WCET - BCET)/6$. The task instance is forced to terminate once it's execution time is expired.

We first show the deadline miss ratio of Drowsy+Simple to demonstrate the importance of designing a timing-aware leakage control algorithm. We adjust the period of each task to achieve 1%, 2%, 3% , 4% and 5% static slack. Figure 7 shows the ratio of tasks missing deadlines with different static slack. We can see that the Drowsy+Simple scheme has high ratio of tasks missing their deadlines with low static slack. For the small task set, the miss ratio is 86.3% and 0.4% when static slack is 1% and 2%, respectively. For the medium task set, the miss ratio is up to 97.9% and 95.6% when the static slack is 1% and 2%, respectively. Drowsy+Simple has higher miss ratio in the medium task set than in the small task set. The medium task set has larger total code size and has more instructions locked in the cache than those of the small task set. Therefore, the Drowsy+Simple scheme incurs more performance degradation in the medium task set than in the small task set. Although Drowsy+Simple only misses the deadlines in the cases with a tight schedule, this is still not acceptable for a hard real-time system that requires the system to always meet the timing constraint. This confirms our assertion that existing leakage reduction techniques are not suitable for hard real-time applications. Our timing-aware leakage control algorithm is guaranteed to meet the timing constraint, therefore, the miss ratio is zero in all cases.

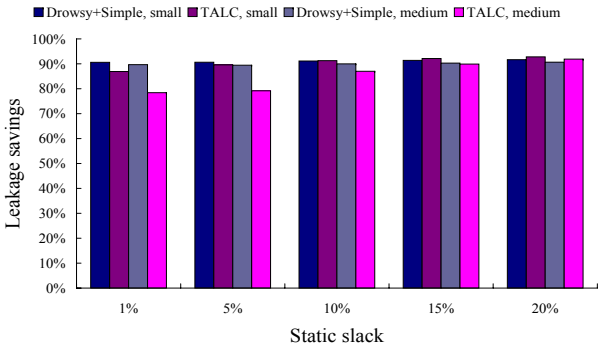


Figure 8: Evaluation of leakage reduction.

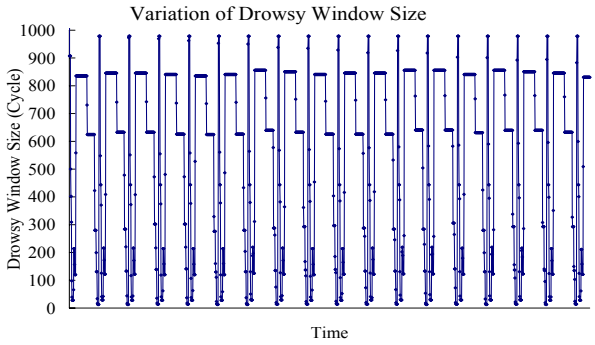


Figure 9: Variation of drowsy window size for the small task set with 20% static slack.

Figure 8 compares the energy savings achieved by our TALC scheme vs. the Drowsy+Simple mechanism with 1%, 5%, 10%, 15% and 20% static slack. Note that for fair comparison, in this set of experiments, the TALC scheme turns the cache lines of idle tasks into the drowsy mode only. We present experimental results for the small and medium task sets separately. When static slack is 1% where Drowsy+Simple has 86.3% and 97.6% of tasks missing their deadlines with the small and the medium task set, in order to satisfy the timing constraint, the TALC scheme achieves less energy savings than Drowsy+Simple. From Figure 8, we also observe that TALC achieves less leakage reduction with the medium task set than the small task set. Since TALC assumes the worst case active set for drowsy window resizing, it could overestimate the wake-up delay. For the medium task set, the overestimation is more serious than the small one since the medium task has larger code size and longer worst-case execution path. A more precise active set analysis scheme could help alleviate this problem. We leave this as the future work. As the slack time increases, the energy savings achieved by TALC approaches Drowsy+Simple. With 20% static slack, the proposed scheme has 1.1% and 1.3% more leakage savings than Drowsy+Simple with the small and medium task set, respectively. This energy advantage provided by TALC over Drowsy+Simple comes from run-time drowsy window resizing. Figure 9 shows the profiling of drowsy window size with 20% static slack for the small task set . The drowsy window size is sampled every 2000-cycles. We could see that the drowsy window size of TALC ranged from 13 cycles to 979 cycles while Drowsy+Simple fixed the window size to 1000-cycle.

To evaluate the effect of turning off cache lines of idle tasks completely, we create a new task set that has sufficient length of idle period to take advantage of the state-destructive mode. To lengthen the idle period, we can increase both static and dynamic slack. To increase static slack, we set 20%, 30%, 40%, 50% and 60% static slack in this set of experiments. To increase dynamic slack, we prolong a task's WCET by increasing the number of iterations executed by the task's major subroutines on the worst-case execution path. Since the BCET/WCET ratio remains 0.95 as the original setup, a task gains more dynamic slack with increasing WCET. The experimental results of this new task set is shown in Table 3. In Table 3, TALC-drowsy denotes the TALC scheme adopting only the drowsy circuits, while

Table 3: Leakage savings of TALC-drowsy and TALC-dual.

Static slack	TALC-drowsy	TALC-dual	Difference
20%	90.9%	93.3%	2.4%
30%	91.7%	94.2%	2.5%
40%	92.9%	95.6%	2.7%
50%	93.9%	96.6%	2.7%
60%	94.2%	97.0%	2.8%

TALC-dual denotes the TALC scheme adopting both the drowsy and gated-Vdd circuits. The results show that turning off cache lines of an idle task achieves up to 2.8% more leakage savings than that of TALC-drowsy.

8. CONCLUSION AND FUTURE WORK

In this paper, we present a timing-aware cache leakage control scheme for hard real-time system. The basic idea of the proposed algorithm is to utilize system slack to tolerate the performance overhead caused by activating cache lines in the low-leakage mode. Our scheme allows the joint use of drowsy and gated-Vdd circuits. Furthermore, the proposed timing-aware leakage control algorithm is able to adjust the drowsy window size dynamically with hard real-time guarantee. The experimental results show that with a tight schedule, the proposed scheme achieves comparable leakage savings with Drowsy+Simple while providing timing guarantee. With sufficient static slack (e.g., 20%), our scheme achieves up to 1.3% more leakage savings than Drowsy+Simple. This energy advantage provided by the proposed scheme comes from run-time drowsy window resizing. With the task set that has opportunities to put cache lines into the state-destructive mode for idle tasks, the proposed scheme achieves up to 2.8% more leakage savings than the proposed scheme with the drowsy mode only.

The proposed scheme can be improved in two ways. First, we plan to adopt a history-based prediction scheme to provide more precise active set estimation than the worst case active set analysis method presented in this paper. Second, we are designing a table-lookup approach to avoid expensive divide operations required for drowsy window resizing.

9. ACKNOWLEDGMENTS

This work is supported in part by research grants from Excellent Research Projects of National Taiwan University

95R0062-AE00-07, and ROC National Science Council NSC 95-2221-E-002-360 and NSC 96-2752-E-002-008-PAE.

10. REFERENCES

[1] Snu real-time benchmarks. In *http://archi.snu.ac.kr/realtime/benchmark/index.html*.

[2] ARM946E-S. http://www.samsung.com/products/semiconductor /asic/ipcorelibrary/intellectureproperties/processorcores /armcores/ddi0201_a946es.pdf.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual international symposium on Computer architecture (ISCA' 00)*, 2000.

[4] J.-J. Chen, H.-R. Hsu, and T.-W. Kuo. Leakage-aware energy-efficient scheduling of real-time tasks in multiprocessor systems. In *Proc. the 12th IEEE Real-Time and Embedded Technology and Applications Symposiums (RTAS '06)*, 2006.

[5] J.-J. Chen and T.-W. Kuo. Procrastination for leakage-aware rate-monotonic scheduling on a dynamic voltage scaling processor. In *Proc. of Conference on Languages, Compilers, and Tools for Embedded Systems 2006(LCTES '06)*, 2006.

[6] A. Cortex-R4F. http://www.arm.com/pdfs/cortex-r4f

[7] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: Simple techniques for reducing leakage power. In *Proceedings of the 29th annual international symposium on Computer architecture 2002(ISCA' 02)*, 2002.

[8] R. Jejurikar and R. Gupta. Integrating preemption threshold scheduling and dynamic voltage scaling for energy efficient real-time systems. In *Proceedings of the 10th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '04)*, 2004.

[9] R. Jejurikar and R. Gupta. Dyanmic slack reclamation with procrastination scheduling in real-time embedded systems. In *Proceedings of the 42nd Design Automation Conference (DAC '05)*, 2005.

[10] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceeding of the 41st Design Automation Conference (DAC '04)*, 2004.

[11] P. Juang, K. Skadron, M. Martonosi, Z. Hu, D. W. Clark, P. W. Diodato, and S. Kaxiras. Implementing branch-predictor decay using quasi-static memory cells. *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 1, p.180-219, 2004.

[12] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th annual international symposium on Computer architecture 2001(ISCA' 01)*, 2001.

[13] N. S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *IEEE Computer*, 36.

[14] N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge. Drowsy instruction caches: Leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture(Micro-35)*, 2002.

[15] W. Kim, J. Kim, and S. Min. A dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack time analysis. In *Proceedings of the conference on Design, automation and test in Europe (DATE '02)*, 2002.

[16] Y. Li, D. Parikh, and Y. Zhang. State-preserving vs. non-state-preserving leakage control in caches. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '04)*, 2004.

[17] S. Martin, K. Flautner, T. Mudge, and D. Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessor under dynamic workloads. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design (ICCAD '02)*, 2002.

[18] Y. Meng, T. Sherwood, and R. Kastner. On the limits of leakage power reduction in caches. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, 2005.

[19] L. Niu and G. Quan. Reducing both dynamic and leakage energy consumption for hard real-time systems. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems (CASES '04)*, 2004.

[20] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '00)*, 2000.

[21] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of the 23rd IEEE Real-time Systems Symposium (RTSS '02)*, 2002.

[22] S.-H. Yang, B. Falsafi, M. D. Powell, K. Roy, and T. N. Vijaykumar. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA-7)*, 2001.

[23] W. Zhang and J. S. Hu. Compiler-directed instruction cache leakage optimization. In *Proc. the 35th Annual International Symposium on Microarchitecture (MICRO-35)*, 2002.

[24] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects.