JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 30, 765-786 (2014)

DOI:10.1688/JISE.2014.30.3.14

Energy-Efficient Real-Time Scheduling of Tasks With Abortable Critical Sections*

JUN WU+ AND KAI-LONG KE
Department of Computer Science and Information Engineering
National Pingtung Institute of Commerce
Pingtung, 900 Taiwan
E-mail: junwu@npic.edu.tw; leon.kidd@gmail.com

In this paper, an energy-efficient scheduling algorithm, called ceiling-based conditional abortable scheduling (CB-CAS) algorithm, is proposed to schedule periodic hard real-time tasks in a non-ideal DVS processor. Based on the schedulability analysis, CB-CAS calculates a proper processor speed for task execution so that the energy consumption can be reduced without violating the timing constraints of tasks. For saving more energy, we also assume that the critical sections of tasks are abortable, which is a strategy originally proposed to reduce priority inversions. In this paper, CB-CAS introduces a conditional abort rule and a dynamic speed adjustment method to work with the rate monotonic scheduling algorithm and the priority ceiling protocol so that the energy consumption could be reduced further. Whenever two tasks are conflicting for the same resource, CB-CAS will examine the cost of blocking the higher-priority task and the cost of aborting the lower-priority task. CB-CAS will abort the lower-priority task and adjust the processor speed dynamically only if it is more energy efficient. The schedulability analysis and the properties of CB-CAS are given in this paper. The capabilities of CB-CAS were also evaluated by a series of experiments, for which we have some encouraging results.

Keywords: real-time systems, dynamic voltage scaling, energy-efficient scheduling, task scheduling, task synchronization, abortable critical sections

1. INTRODUCTION

A dynamic voltage scaling (DVS) processor can operate tasks at different speeds (*i.e.*, frequencies) by adjusting its supply voltage. The energy consumption can be reduced when tasks are executed at a lower processor speed. However, the slowdown of the processor speed will cause an impact on performance due to the late completion of tasks. On the other hand, the late completion of a real-time task can be allowed as long as the task meets its timing constraints. Such an observation provides a strong demand in energy-efficient scheduling of real-time tasks in a DVS processor.

In the past decade, many excellent energy efficient real-time scheduling algorithms have been proposed. A comprehensive survey can be found in [1]. Most work focuses on independent tasks, relatively little work has been done in the presence of task synchronization. However, in many real cases, tasks are dependent due to resource sharing. In this

Received October 21, 2013; revised December 13, 2013; accepted February 4, 2014.
Communicated by Cho-Li Wang.
* The preliminary version of this work has been presented at the 3rd International Symposium on Advances in Embedded Systems and Applications, 25-27 June, 2012, Liverpool, UK.
* This work was supported in part by the National Science Council of Taiwan, under Grants NSC-101-2221-E-251-005 and NSC-102-2221-E-251-004.
+ Corresponding author.

paper, we assume that resources are accessed by tasks in a mutually exclusive manner. We define the part of code dealing with accessing a shared resource as critical section. To synchronize real-time tasks with critical sections, many excellent approaches have been proposed, such as the well-known *priority inheritance protocol* (PIP) [2], *priority ceiling protocol* (PCP) [2], and *stack resource policy* (SRP) [3], *etc.*

Based on various assumptions on system and/or task models, researchers have explored the DVS scheduling of tasks under synchronization constraints (*e.g.*, [4-14]). Different from the past work, we assume that the critical sections are abortable which is a strategy originally developed to reduce priority inversions [15-19]. Priority inversion is the phenomenon where a higher-priority task is blocked by lower-priority tasks. It occurs whenever a higher-priority task is attempting to access a shared resource that has been currently used by another lower-priority task. When critical sections are abortable, it provides an opportunity for saving more energy. This is because we can examine the energy consumption for the blocking of the higher-priority task and the aborting of the lower-priority task. According to the examining results, the lower-priority task can be aborted if the aborting is more energy efficient than the blocking. Note that we also need a dynamic speed adjustment method to calculate a proper processor speed for the aborting and the aborted tasks so that the energy consumption can be reduced without violating the timing constraints of tasks.

In this paper, we propose an energy-efficient scheduling algorithm, called *ceiling-based conditional abortable scheduling* (CB-CAS) algorithm, to schedule a set of periodic real-time tasks under synchronization constraints of the shared resources. CB-CAS is a preemptive priority driven scheduling algorithm with fixed priority assignment and ceiling-based concurrency control. In particular, CB-CAS introduces a conditional abort rule to work with *rate-monotonic scheduling* (RMS) [20] algorithm and PCP. Based on the schedulability analysis of RM, CB-CAS calculates a lowest possible processor speed for task execution such that the energy consumption can be reduced without violating the timing constraints of tasks. Whenever a higher-priority task is requesting a shared resource that is currently accessed by an abortable critical section of another lower-priority task, the conditional abort scheme will abort the lower-priority task if the aborting can save more energy. CB-CAS will also slow down the processor speed dynamically whenever more energy saving can be obtained. The schedulability analysis and the properties of CB-CAS are given in this paper. It is shown that the timing constraints of tasks can be met under CB-CAS. Furthermore, CB-CAS is deadlock free and the number of blockings and abortings are bounded. The capabilities of CB-CAS were also evaluated by a series of experiments, for which we have some encouraging results. In particular, it demonstrates the strengths of CB-CAS in reducing energy consumption.

The rest of this paper is organized as follows. Section 2 summarized the related work of this research. Section 3 provides the system model and the problem definition. Sections 4 proposes the scheduling algorithm for real-time tasks with abortable critical sections. Section 5 provides the properties of our proposed CB-CAS. Section 6 provides experimental results. Finally, Section 7 is the conclusion.

2. RELATED WORK

Making critical section abortable could reduce the number of priority inversions,

especially for tasks with higher priorities. Huang *et al.* [15], Tokuda *et al.* [16], and Shu *et al.* [17] proposed real-time concurrency control protocols based on the *priority abort scheme* (PAS) [15], in which a critical section can be aborted by any task which has a higher priority. However, such an approach could cause unnecessary aborts thus degrading the schedulability of the system. Takada and Sakamura [18] proposed the *ceiling abort protocol* (CAP) to make the critical sections abortable under PCP. Unfortunately, a large number of aborts could occur at the same time under CAP, for which the system performance significantly degrades. Later, Lam and Ng [19] proposed another PCP-based protocol, called *conditional abortable priority ceiling protocol* (CA-PCP) to solve the problem. Based on the existing system utilization, the length of the abortable critical section, and the usage of resources, CA-PCP defines a condition to handle the aborting such that the schedulability can be guaranteed.

In the recent decades, dynamic voltage scaling for real-time systems has received considerable attention. Chen and Kao [1] have presented an extensive survey of energy-efficient DVS scheduling of real-time tasks. However, relatively little work has been done in the presence of task synchronization with shared resources (*e.g.*, [4-14]. For tasks with non-preemptible critical section, Zhang and Chanson [4] first proposed a dynamic priority scheduling algorithm, called *dual speed* (DS) algorithm. Under DS, tasks are executed initially at a static low speed and the processor speed is switched to a high speed dynamically as soon as a task is blocked. The low and high speeds of DS are calculated based on the sufficient schedulability condition of the *earliest deadline first* (EDF) algorithm [20]. Hence, the timing constraints of tasks can be met while energy consumption can be reduced. Later, some extensions of DS have been proposed [5, 6]. In particular, Lee *et al.* [5] explored the problem with a tighter schedulability analysis, and proposed a multi-speed extension of DS, call *multi-speed* (MS) algorithm, to achieve further energy savings. For tasks with preemptible critical sections, Jejurikar and Gupta [8] proposed an approach, called *uniform slowdown with frequency inheritance* (USFI) algorithm. Under USFI, each task is assigned to be executed at a static speed which is calculated by taking the worst-case blocking time into account. If a task blocks other tasks, it will inherit and to be executed at the highest speed of the blocked tasks during the blocking. In the recent years, researchers also explored the DVS scheduling and synchronization of real-time tasks in multiprocessor/multi-core environments (*e.g.*, [21-27]).

3. SYSTEM MODEL AND PROBLEM DEFINITIONS

3.1 System Model

In this paper, we will explore the energy-efficient real-time task scheduling on a *non-ideal* DVS processor[1] which supports a set of K discrete available speeds[2] $\mathcal{S} = \{s_1, s_2, \ldots, s_K\}$, where $s_1 < s_2 < \ldots < s_K$. Let s_{min} and s_{max} denote the minimum and the maximum available speed (*i.e.*, $s_{min} = s_1$ and $s_{max} = s_K$), respectively. Without loss of generality, we assume that the s_{max} is 1 and all other speeds are normalized with respect to the maximum speed s_{max}.

[1] There are two types of DVS processors have been considered in the literature: ideal and non-ideal. An ideal DVS processor can be operated at any speed in the range from the minimum to the maximum available speed, while a non-ideal DVS processor has only discrete speeds. Nowadays, most DVS processors are non-ideal, and the ideal DVS processors are only for theoretical analysis purpose.

[2] Note that K can be considered as a constant and it is known in a prior.

The power consumption of a DVS processor can be defined as a function $PC(s_i)$ of the adopted processor speed s_i. Various power consumption functions of practical DVS processors have been modeled in the literatures, such as [4-8, 28, 29]. In particular, Chen and Kuo [28] reported that the power consumption function $PC(s_i)$ of Intel XScale is approximated by $0.08+1.52s_i^3$ Watt. The amount of CPU cycles executed in a time interval is linear of the processor speed. Let $s(t)$ be the processor speed at time t. The amount of CPU cycles completed in a time interval $(t_1, t_2]$ can be represented as $\int_{t_1}^{t_2} s(t)dt$. Hence, the energy consumption $EC_{(t_1, t_2]}$ in a time interval $(t_1, t_2]$ can be calculated by $\int_{t_1}^{t_2} PC(s(t))dt$.

Note that, in this paper, we ignore some overheads incurred for scheduling tasks, such as the time and energy consumption for speed-switching and context-switching (due to preemptions), because the overheads are relatively low compared to the cost of executing tasks. For example, Rajan *et al.* [30] have shown that the speed-switching overhead is about 1-2% of the entire energy consumption. However, when the overheads are significant and can not be ignored, researchers have proposed excellent approaches for reducing or eliminating speed switching and/or preemptions [31-33]. Furthermore, in the recent years, the intercore communication overhead has been considered in the literature for multicore environments. Particularly, Wang *et al.* have proposed excellent approaches for removing the intercore communication overhead on multiprocessor system-on-chips when tasks have precedence constraints [21-24].

3.2 Task Model

In this paper, we are interested in energy-efficient scheduling of a set of n periodic hard real-time tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ in a uniprocessor system. We assume that tasks are preemptive and every task τ_i has a fixed priority P_i and a worst-case computation time C_i. Note that the value of C_i is given by assuming the task is executed at the maximum processor speed s_{max}. A task is a template of its instance and each instance will be instantiated for every request of the task. The requests of a task τ_i will arrive regularly for every period T_i. Let $\tau_{i,j}$ denote the jth instance of task τ_i. Every task instance $\tau_{i,j}$ should be completed no later than its deadlines which is defined as its arrival time plus the relative deadline D_i of the task τ_i. We assume in this paper that tasks are well formed which satisfies $0 \le (C_i) \le D_i \le T_i$, $\tau_i \in \mathcal{T}$.

We consider tasks are dependent due to exclusive access to shared resources. Assume that the system consists of a set of m shared resources $\mathcal{R} = \{r_1, r_2, \ldots, r_m\}$, and each task might access one or more resources during its execution. To ensure the resources can be accessed in a mutually exclusive manner, we assume that resources are guarded by binary semaphores. A task is said to be executed in its *critical section* when it has granted the access right to a resource. Note that the executions of critical sections are preemptible. Let $\mathcal{Z}_i = <z_{i,1}, z_{i,2}, \ldots, z_{i,n_i}>$ be the list of critical sections of task τ_i, where $z_{i,j}$ is the jth critical section of task τ_i and it corresponds to the code segment between the jth locking operation and its corresponding unlocking operation. For any two critical sections $z_{i,x}$ and $z_{i,y}$ of a task τ_i, $z_{i,x} \subset z_{i,y}$ denotes that the execution of $z_{i,x}$ is entirely contained in $z_{i,y}$. We also called $z_{i,x}$ and $z_{i,y}$ are *nested* critical sections if $z_{i,x} \subset z_{i,y}$ or $z_{i,y} \subset z_{i,x}$. The locks of nested critical sections have to be unlocked in the reverse order from which they were granted. In this paper, we assume critical sections are *properly nested* means that for any two critical sections $z_{i,x}$ and $z_{i,y}$, either $z_{i,x} \subset z_{i,y}$, $z_{i,y} \subset z_{i,x}$, or their executions

do not overlap (*i.e.*, $z_{i,x} \cap z_{i,y} = \varnothing$) [2].

Different from the past work, we assume that the critical sections are abortable. Each critical section $z_{i,j}$ consists of an *abortable segment* $z_{i,j}^{a}$ followed by an *unabortable segment* $z_{i,j}^{u}$. A task can be aborted when it is executed in the abortable segment of a critical section, however, the execution of the aborted task has to be restarted later from the beginning of the abortable segment. Such a re-execution of an aborted task is considered as a cost for aborting. In contrast, once a task enters the unabortable segment of a critical section, it cannot be aborted and it will be executed to the end (the preemption is allowed.). Recall that we assumed that critical sections are properly nested. When two critical sections are nested, for simplicity, we assume that the inner critical section is unabortable and is included in the unabortable segment of the outer critical section.

3.3 Problem Definition

Let *lcm* be the least common multiple of all tasks' periods (also called hyperperiod). Since the task set $\mathcal{T}$ repeats an identical execution trace for every hyperperiod, we only need to examine the time interval (0, *lcm*] for analyzing the schedulability of the entire schedule [20]. The research problem of this paper is defined as follows:

Problem 3.1 Given a set of *n* periodic hard real time tasks $\mathcal{T}$ with abortable critical sections, and a set of *m* shared resources $\mathcal{R}$. The problem is to schedule $\mathcal{T}$ and to synchronize their accesses of shared resources $\mathcal{R}$ on a non-ideal DVS processor which supports a set of *K* discrete speeds $\mathcal{S}$. During a given time interval (0, *lcm*], the objective of this problem is to generate a schedule such that all tasks can meet their timing requirements (*i.e.*, ***their deadlines*) *and the energy consumption*** $\int_0^{lcm} PC(s(t))dt$ is minimized.

The problem is $\mathcal{NP}$-hard even if the DVS processor only supports a single speed and the critical sections are unabortable [34].

4. CEILING-BASED CONDITIONAL ABORTABLE SCHEDULING (CB-CAS) ALGORITHM

In this section, the *ceiling-based conditional abortable scheduling* (CB-CAS) algorithm is proposed to schedule periodic real-time tasks with abortable critical sections in a non-ideal DVS processor. In Section 4.1, we shall present the rules for task scheduling and synchronization (including the conditional abort rule) of CB-CAS. In Sections 4.2 and 4.3, the derivation of the proper processor speed for task executions and the dynamic speed adjustment method are presented. Finally, an example is given in Section 4.4 to illustrate our proposed CB-CAS.

4.1 Rules for Task Scheduling and Synchronization

The rules of CB-CAS for task scheduling are the same as those of RMS algorithm [20]. We now present those rules as follows:

- **Rule 1 (Priority Assignment):** Each task is assigned a fixed priority according to its

period. In particular, a task with a shorter period is assigned a higher priority.

- **Rule 2 (Task Scheduling):** The task which has the highest priority among all ready tasks can be executed on the processor. If a task does not attempt to access any resource, the task can preempt the execution of any task with a lower priority, whether or not the priorities are assigned or inherited. (Priority inheritance will be defined later.)

The rules of CB-CAS for task synchronization are defined as follows:

- **Rule 3 (Priority Ceiling):** When a task τ_i attempts to lock (*i.e.*, access) a resource r_j which is currently unlocked, the priority of τ_i must be higher than it system ceiling PL_i^*; otherwise, the lock request is blocked.
- **Rule 4 (Priroity Inheritance):** A task τ_i uses its assigned priority, unless it locks some resources and blocks higher-priority tasks. If τ_i blocks one or more higher-priority tasks, it inherits the highest priority of the tasks blocked by τ_i. When a task unlocks a resource, it resumes the priority it had at the point of obtaining the lock on the resource. Moreover, the priority inheritance is transitive.

Rules 3 and 4 of CB-CAS are developed based on the well-known *priority ceiling protocol* (PCP) [2]. In particular, Rules 3 and 4 are the priority ceiling rule and the priority inheritance rule of PCP. For the priority ceiling rule (*i.e.*, Rule 3), a priority ceiling is assigned to each resource, which is the highest priority of tasks that may access the resource. It allows a task τ_i to enter a critical section only if τ_i's priority is higher than its system ceiling PL_i^* which is the highest priority ceiling of resources currently locked by tasks other than τ_i. The rationale behind the setting of priority ceiling and the ceiling rule is to ensure that when a task τ_i preempts the current task and executes one of its critical section $z_{i,x}$, the priority of τ_i is guaranteed to be higher than the priorities of all the preempted critical sections during the execution of the critical section $z_{i,x}$. For the priority inheritance rule (*i.e.*, Rule 4), whenever a task τ_i blocks higher-priority tasks, τ_i will temporarily inherit the highest priority of the tasks currently blocked by τ_i. This rule prevents to prolong the blocking time of the higher-priority tasks because the medium-priority tasks are not allowed to preempt the task τ_i.

When critical sections are abortable, aborting becomes an option for handling resource conflicts. Hence, we propose the conditional abort rule as follows:

- **Rule 5 (Conditional Abort):** When a task τ_i attempts to lock a resource r_j which is currently locked by another task τ_k, τ_k is aborted and the lock request of τ_i is granted if the following three conditions are hold: (1) τ_k is being executed in the abortable segment $z_{k,l}^a$ of a critical section $z_{k,l}$, (2) $P_i = PL_i^*$, and (3) $a_i < b_i$; otherwise, the lock request is blocked. Where a_i and b_i are the completed computation amount of $z_{k,l}^a$ and the incompleted computation amount of $z_{k,l}$, respectively.

Whenever a task τ_i attempts to lock a resource r_j, Rule 5 will be performed if r_j is currently locked; otherwise, Rule 3 will be performed. Note that both Rule 3 and Rule 5 can be done in constant time. We now present the rationale for the design of the conditional abort rule. Suppose that τ_i is the task currently being executed on the processor, and τ_i is attempting to access a resource r_j which is currently locked by another lower-

priority task τ_k. If τ_k is being executed in an abortable segment $z^a_{k,l}$, there are two options should be considered:

Option 1: Let τ_i to abort τ_k.

By choosing this option, the current task τ_i won't be blocked by a lower priority task, *i.e.*, τ_k. However, the abortable segment $z^a_{k,l}$ has to be re-executed. This re-execution is considered as a cost charged to τ_i by the aborted task τ_k. Let a_i be the re-execution cost which is the completed computation amount of the aborted $z^a_{k,l}$. Note that the priority of τ_k has to resume its previous priority if it has inherited the priority of a higher-priority task due to a blocking.

Option 2: Let τ_k to block τ_i.

When this option is chosen, the current task τ_i will be blocked by τ_k. Let b_i be the incomplete computation amount of $z_{k,l}$, which is considered as a cost for blocking of τ_i. The value of b_i is $|z_{k,l}| - a_i$, where $|z_{k,l}|$ is the length (computation amount) of $z_{k,l}$. Note that τ_i's priority will be inherited by τ_k until τ_k unlocks r_j.

Rule 5 makes a decision between the above two options according to the two conditions: $P_i = PL^*_i$ and $a_i < b_i$. In particular, τ_i will abort τ_k if $P_i = PL^*_i$ and $a_i < b_i$; otherwise, τ_i will be blocked. The first condition $P_i = PL^*_i$ is to ensure that the condition $P_i > PL^*_i$ is hold[3] when τ_i aborts τ_k and then its lock request for r_j is granted. The second condition $a_i < b_i$ is to ensure that the aborting is more energy-efficient than the blocking because the re-execution time is shorter than the blocking time.

4.2 Task Execution Speed

Based on the well-known schedulability analysis of RMS and PCP, we will derive a processor speed level, denoted by s^*, for task execution. When tasks are scheduled under RMS and PCP, the following theorem provides a sufficient (*i.e.*, worst case) condition that characterizes the schedulability of a given task set.

Theorem 4.1 (Sha *et al.* [2]) A set of n periodic hard real-time tasks (sorted in non-decreasing order of their periods) using the priority ceiling protocol can be scheduled by the rate-monotonic scheduling algorithm if the following conditions are satisfied:

$$\forall i, 1 \le i \le n, \sum_{k=1}^{i} \frac{C_k}{T_k} + \frac{B_i}{T_i} \le i(2^{1/i} - 1). \tag{1}$$

Where B_i is the worst-case blocking time of a task τ_i, which can be calculated in a priori [2]. When aborting becomes an option, the worst-case re-execution time of an aborted task is also needed for schedulability analysis. Suppose that τ_k is being executed in an abortable segment $z^a_{k,l}$. We assume that τ_k is aborted by τ_i, the time for re-executing the aborted critical section does not exceed $|z^a_{k,l}|$, where $|z^a_{k,l}|$ represents the length (*i.e.*, time) of $z^a_{k,l}$. Since $z^a_{k,l}$ is the abortable segment of a critical section $z_{k,l} \in \mathcal{Z}_k$, the maximum re-execution time of task τ_k is no more than $\max_{z_{k,l} \in \mathcal{Z}_k}\{|z^a_{k,l}|\}$. However, the period of task τ_i is less than the period of task τ_k, such an aborting may occurs at most $\lceil T_k/T_i \rceil$ times. Thus, the

[3] The condition $P_i > PL^*_i$ is original used for PCP to prevent transitive blocking and deadlock.

re-execution cost of τ_k due to the abortings from τ_i is no more than $\max_{z_{k,l} \in \mathcal{Z}_k}\{|z_{k,l}^a|\} \lceil T_k/T_i \rceil$. Note that τ_i's instances will arrive at least $\lfloor T_k/T_i \rfloor$ times within τ_k's period.

Let A_i be the maximum re-execution cost charged to an instance of τ_i by a lower priority task (which might be aborted by τ_i). The value of A_i is defined as follows:

$$A_i = \max_{\substack{\tau_k \in \text{AccessedSet}(\text{ResourceSet}(\tau_i)) \\ \wedge P_i > P_k}} \{0, \frac{|z_{k,l}^a| \lceil T_k / T_i \rceil}{\lfloor T_k / T_i \rfloor} | z_{k,l} \in \mathcal{Z}_k\}. \tag{2}$$

Where AccessedSet(r_i) and ResourceSet(τ_i) are the set of tasks which may access to resource r_j and the set of resources which are accessed by task τ_i, respectively. When tasks have abortable critical sections, the following theorem shows that tasks can be feasibly scheduled by CB-CAS if they are executed at processor speed s^*.

Theorem 4.2 A set of n dependent periodic tasks with abortable critical sections (sorted in non-decreasing order of their periods) can be feasibly scheduled by CB-CAS with the execution speed s^*, where

$$s^* = \max_{\forall i, 1 \le i \le n} \{s^{*,i}\} \tag{3}$$

and

$$s^{*,i} = \min_{s_j \in \mathcal{S}} \{s_j \mid \frac{\sum_{k=1}^{i} \frac{C_k}{T_k} + \frac{\max\{B_i, A_i\}}{T_i}}{i(2^{1/i} - 1)} \le s_j\}. \tag{4}$$

Proof: This theorem can be proved in two parts. First, consider Eq. (1) as shown in Theorem 4.1. Because the definition of the worst-case computation time (*i.e.*, C_i, for $1 \le i \le n$) is assumed that tasks are running at the maximum processor speed, it is easy to shown that tasks can be scheduled by RMS and PCP at a specific processor speed s_j if the following conditions are satisfied:

$$\forall i, 1 \le i \le n, \frac{\sum_{k=1}^{i} \frac{C_k}{T_k} + \frac{B_i}{T_i}}{i(2^{1/i} - 1)} \le s_j. \tag{5}$$

Second, also in Eq. (1), the worst-case blocking time (*i.e.*, B_i) is take into account for schedulability analysis. When critical sections are abortable, however, the aborting becomes a new option for handling resource conflicts. When task τ_i is attempting to lock a resource r_j which is currently locked by a lower priority task τ', there are two possible cases: (1) task τ_i will be blocked by task τ', and the blocking time is no more than B_i; (2) task τ' will be aborted by task τ_i, and the re-execution time is no more than A_i. Since the cost (time) for blocking and aborting are mutually exclusive, *i.e.*, they cannot occur at the same time. Eq. (1) will also be satisfied when the term B_i is replaced by $\max\{B_i, A_i\}$ when tasks have abortable critical sections and they are scheduled by CB-CAS. Based on the two parts we have discussed, this theorem is correct. ❑

The processor speed s^* is calculated off-line (before the runtime) by using Eqs. (3)

and (4), as shown in Algorithm 4.1. The time complexity of the calculation is $O(n^2m)$, where n and m are the number of tasks and resources in the system, respectively. Note that both the calculation of A_i and B_i can be done in $O(nm)$ time. Also note that the given task set is unschedulable even at the maximum processor speed if Algorithm 4.1 returns failure.

Algorithm 4.1 Execution Speed Calculation

Given tasks are in non-decreasing order of their periods.

1: $s^* \leftarrow s_{min}$;
2: **for** ($i = 1$; $i \le n$; i++) **do**
3: $s^{*,i} \leftarrow s_{max} + 1$;
4: $s'' \leftarrow \dfrac{\sum_{k=1}^{i} \frac{C_k}{T_k} + \max\{B_i, A_i\}}{i(2^{1/i} - 1)}$;
5: **for** ($j = K$; $j \ge 1$; j– –) **do**
6: **if** (($s'' \le s_j$) $\wedge$ ($s_j < s^{*,i}$)) **then**
7: $s^{*,i} \leftarrow s_j$;
8: **end if**
9: **end for**
10: **if** ($s^{*,i} \leftarrow s_{max}$) **then**
11: **return** *failure*; // the task set is unschedulable
12: **end if**
13: **if** ($s^{*,i} > s^*$) **then**
14: $s^* \leftarrow s^{*,i}$;
15: **end if**
16: **end for**
17: **return** s^*;

4.3 Dynamic Speed Adjustment

According to Theorem 4.2, all tasks are executed at the speed s^* initially, which is to ensure that the timing constraints of tasks can be satisfied. However, the calculation of speed s^* is based on the schedulability analysis with the worst-case blocking cost or aborting cost (*i.e.*, $\max\{B_i, A_i\}$). It provides a chance to save more energy when the actual cost of blocking or aborting is less than the worst cases. Algorithm 4.2 is our proposed *dynamic speed adjustment* method which is to execute tasks at a lower processor speed whenever it is possible.

Let M_i be the maximum value of B_i and A_i for every task $\tau_i \in \mathcal{T}$, which can be calculated off-line. When a task τ_i aborts a lower priority task τ_k, lines 1-5 show that how the execution speed is adjusted. Suppose that task τ_k is being executed in an abortable segment $z^a_{k,l}$. Let C'_i be the incomplete computation amount of task τ_i and a_i be the completed computation amount of the abortable segment $z^a_{k,l}$. In particular, a_i is the re-execution cost for the aborting. Lines 4 and 5 show that the execution speeds of both τ_i's remaining computation and τ_k's re-execution are slow down from s^* to s'. It is obvious that the remaining computation time of task τ_i is no more than $(C'_i + \max\{B_i, A_i\})/s^*$ if it is

still being executed at speed s^*. Note that Theorem 5.1 (as shown in the next section) shows that the maximum number of abortings or blockings is no more than 1 for every task instance. If τ_i aborts τ_k, the remaining computation of task τ_i is no more than $C_i' + a_i$. Hence, we can find a lower processor speed $s' = \min_{s_j \in \mathcal{S}}\{s_j \mid \frac{s^*(C_i' + a_i)}{C_i' + M_i} \le s_j\}$ as shown in line 3, where $M_i = \max\{B_i, A_i\}$.

Algorithm 4.2 Dynamic Speed Adjustment Method

When a task τ_i aborts a lower-priority task τ_k which is currently being executed in an abortable segment $z_{k,l}^a$.

1: $C_i' \leftarrow$ the incomplete computation of τ_i;
2: $a_i \leftarrow$ the completed computation amount of $z_{k,l}^a$;
3: $s' \leftarrow \min_{s_j \in \mathcal{S}}\{s_j \mid \frac{s^*(C_i' + a_i)}{C_i' + M_i} \le s_j\}$
4: Set the execution speed of task τ_i as s';
5: Set the re-execution speed of the aborted segment of task τ_k as s';

When a task τ_i is blocked by a lower-priority task τ_k which is currently being executed in a critical section $z_{k,l}$.

6: **if** (τ_k is being executed in an abortable segment $z_{k,l}^a$) **then**
7: $b_i \leftarrow$ the incomplete computation amount of $z_{k,l}$;
8: **else** // in this case, τ_k is being executed in an unabortable segment $z_{k,l}^u$
9: $b_i \leftarrow$ the incomplete computation amount of $z_{k,l}^u$;
10: **end if**
11: $s' \leftarrow \min_{s_j \in \mathcal{S}}\{s_j \mid \frac{s^*(C_i' + b_i)}{C_i' + M_i} \le s_j\}$;
12: Set the execution speed of task τ_i as s';
13: Set the execution speed of task τ_k as s' until τ_k exits $z_{k,l}^u$;

When a task τ_i finishes the execution of its last critical section

14: **if** (there is no blocking and aborting was occurred during its execution) **then**
15: $C_i' \leftarrow$ the incomplete computation of τ_i;
16: $s' \leftarrow \min_{s_j \in \mathcal{S}}\{s_j \mid \frac{s^* C_i'}{C_i' + M_i} \le s_j\}$;
17: Set the execution speed of task τ_i as s';
18: **end if**

Lines 6-13 show the speed adjustment when a task τ_i is blocked by a lower priority task τ_k. Lines 6-10 calculate b_i which can be considered as the actual blocking time of τ_i. When task τ_k is being executed in an abortable segment $z_{k,l}^a$, the actual blocking time is the incomplete computation amount of $z_{k,l}$ (the incomplete computation amount of $z_{k,l}^a$ plus $|z_{k,l}^u|$). On the other hand, the actual blocking time is the incomplete computation amount of an unabortable segment $z_{k,l}^u$ when task τ_k is being executed in $z_{k,l}^a$. Because a task instance can be blocked or aborts another task for at most once (as shown in Theorem 5.1), line 11 calculates a lower speed $s' = \min_{s_j \in \mathcal{S}}\{s_j \mid \frac{s^*(C_i' + b_i)}{C_i' + M_i} \le s_j\}$ for the execution of τ_i's remaining computation. Note that the execution speed of τ_k is also assigned as s' until it exits $z_{k,l}^u$.

Lines 14-18 consider another possible for saving more energy. If there is no blocking or aborting was occurred after a task τ_i exits its last critical section. We adjust the execution speed of τ_i's remaining computation to a lower speed s' which is the lowest processor speed such that $s^* C_i'/(C_i' + M_i) \le s_j, \forall s_j \in \mathcal{S}$.

Algorithm 4.2 shows the three cases for adjusting the processor speed dynamically In all the three cases, the time complexity of the calculation of the adjusted low speed s' is $O(K)$, where K is the number of the available speeds of the DVS processor and it can be considered as a constant. In other words, the adjusted processor speed can be calculated in constant time.

4.4 Example

We shall illustrate CB-CAS with the following example:

Example 4.1 Consider two tasks τ_1 and τ_2, as shown in Fig. 1 and Table 1, which are scheduled by CB-CAS on a non-ideal DVS processor. We assume that the processor supports 10 discrete speeds $\mathcal{S} = \{s_1, s_2, \ldots, s_{10}\}$, where $s_1 < s_1 < \ldots. < s_{10}$. In particular, we assume that $s_1 = 0.1$, $s_2 = 0.2$, $s_9 = 0.9$, and $s_{10} = 1$.

The priority of τ_1 is higher than the priority of τ_2, *i.e.*, $P_1 > P_2$, because $T_1 < T_2$. Suppose that both τ_1 and τ_2 may access a resource r_x. In particular, tasks τ_1 and τ_2 may access r_x in their critical section $\tau_{1,1}$ and $\tau_{2,1}$, respectively, where $|z_{1,1}| = |z^a_{1,1}| + |z^u_{1,1}| = 1 + 1 = 2$ and $|z_{2,1}| = |z^a_{2,1}| + |z^u_{2,1}| = 1 + 2 = 3$. According to the ceiling rules, the priority ceiling of r_x is equal to the priority of τ_1, *i.e.*, $PL_x = P_1$. According to Algorithm 4.1, τ_1 and τ_2 will be scheduled at processor speed $s^* = s_7 = 0.7$. Fig. 2 shows the schedule results. Note that we only consider the schedule from time 0ms to 50ms. Because 50ms is the hyperperiod of τ_1 and τ_2, they will repeat an identical execution trace for every hyper-period [20].

At time 0, both τ_1 and τ_2 are arrival, task τ_1 starts its execution because its high priority. At time 1.428ms, τ_1 enters its critical section and exits at time 4.284ms. Note that the computation time of a task τ_i is C_i/s_j when it is executed at a speed s_j because the original value of C_i is given by assuming it will be executed at the maximum processor speed. Hence, 1ms becomes 1.428ms when $s^* = 0.7$. According to Algorithm 4.2, *i.e.*, the dynamic speed adjustment method, τ_1's execution speed should be lowered to $s' = \min_{s_j \in \mathcal{S}}\{s_j \mid \frac{s^* C_1'}{C_1' + max\{B_1, A_1\}} \le s_j\} = 0.2$ (where τ_1's incomplete computation $C'_1 = 1$) because

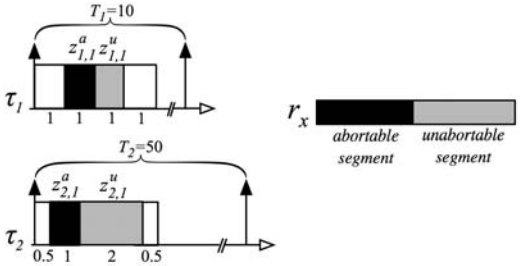


Fig. 1. Example task instances.

Table 1. Parameters of example tasks.

Task	$T_i = D_i$	C_i	B_i	A_i
τ_1	10	4	3	1
τ_2	50	4	0	0

time unit: microsecond

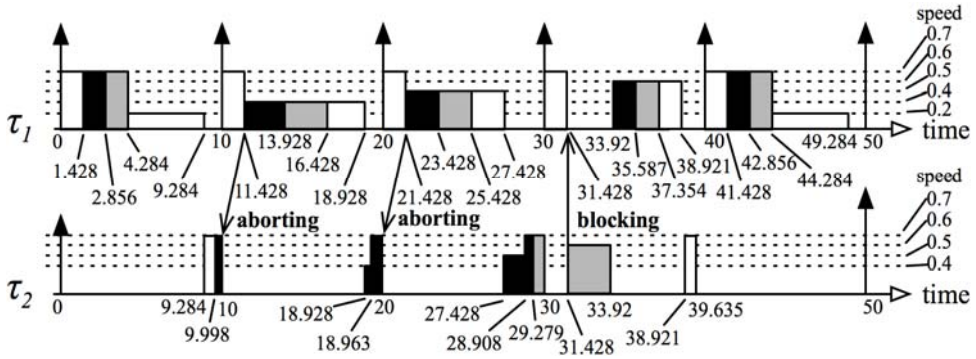


Fig. 2. A CB-CAS schedule.

there is no blocking or aborting was occurred when τ_1 exits its last critical section. Thus, τ_1 is executed to the end at speed 0.2. At time 9.284ms, τ_1 finishes its execution and τ_2 starts its execution and it locks r_x successfully at time 9.998ms.

At time 10ms, a new instance of τ_1 arrives and preempts the execution of τ_2 due to its high priority. At time 11.428ms, τ_1 attempts to lock r_x which has been locked by τ_2. According to the conditional abort rule (*i.e.*, Rule 5), τ_1 aborts τ_2, since τ_2 is being executed in an abortable segment and $a_1 < b_1$ (*i.e.*, the re-execution cost for aborting is less than the blocking cost), where $a_1 = 0.002 \times 0.7 = 0.014$ and $b_1 = 3 - a_1 = 2.986$. According to Algorithm 4.2, the execution speed of τ_1's incomplete computation and the re-execution speed of τ_2 is $s' = \min_{s_j \in \mathcal{S}}\{s_j \mid \frac{s^*(C_1' + a_1)}{C_1' + \max\{B_1, A_1\}} \le s_j\} = 0.4$, where $C'_1 = 3$. At time 18.928 ms, τ_1 finishes its execution and τ_2 resumes from the beginning of its aborted critical section, *i.e.*, $z^a_{2,1}$. Note that this re-execution also use the processor speed 0.4. At time 18.963, the re-execution has been completed and τ_2 resumes to be executed at speed $s^* = 0.7$.

At time 20ms, a new instance of τ_1 arrives and preempts τ_2. Later, at time 21.428, according to Rule 5 and Algorithm 4.2, τ_2 is aborted by τ_1 again and the execution speed of τ_1 and τ_2's re-execution is $s' = \min_{s_j \in \mathcal{S}}\{s_j \mid \frac{s^*(C_1' + a_1)}{C_1' + \max\{B_1, A_1\}} \le s_j\} = 0.5$, where $a_1 = 0.74$, $b_1 = 2.26$ and $C'_1 = 3$. At time 27.428ms, τ_1 finishes its execution and τ_2 resumes from the beginning of its aborted critical section. At time 29.279ms, τ_2 completes its abortable segment and enters its unabortable segment $z^u_{2,1}$.

At time 30ms, a new instance of τ_1 arrives and preempts τ_2, later, τ_1 attempts to lock r_x which has been locked and accessed in an unabortable segment of τ_2. According to Rule 5, τ_1 is blocked by τ_2. Also, according to Algorithm 4.2, the execution speed of the blocking and the τ_1's incomplete computation is $s' = \min_{s_j \in \mathcal{S}}\{s_j \mid \frac{s^*(C_1' + b_1)}{C_1' + \max\{B_1, A_1\}} \le s_j\} = 0.6$, where $b_1 = 1.4953$ and $C'_1 = 3$. At time 33.92ms, τ_2 exits its critical section which blocked τ_1, τ_1 resumes its execution and be executed at speed 0.6. At time 38.921, τ_1 finishes and τ_2 resumes its execution and finishes at time 39.635ms. At time 40ms, τ_1's instance arrives and be executed on the processor, similar to its execution at time 0-9.284ms, τ_1 finishes at time 49.284. ❑

5. PROPERTIES

Lemma 5.1 A task τ_i can be blocked by a lower priority task τ_j only if (1) a resource r_x is locked and being accessed by τ_j; and (2) τ_i is attempting to lock r_x.

Proof: According to the priority ceiling rule (*i.e.*, Rule 3) of CB-CAS, the lemma is correct. ❑

Lemma 5.2 A task τ_j can be aborted by a higher priority task τ_i only if (1) a resource r_x is locked and being accessed in an abortable segment of τ_j; and (2) τ_i is attempting to lock r_x.

Proof: According to the conditional abort rule (*i.e.*, Rule 5) of CB-CAS, the lemma is correct. ❑

Lemma 5.3 When the lock request of a task τ_i on a resource r_x is granted, the priority of τ_i is greater than its system ceiling, *i.e.*, $P_i > PL_i^*$.

Proof: According to the priority ceiling rule (*i.e.*, Rule 3) and the conditional abort rule (*i.e.*, Rule 5), there are two cases that the lock request can be granted if: (1) $PL_i > PL_i^*$, when r_x is unlocked; or (2) $P_i = PL_i^*$ and $a_i < b_i$, when r_x is locked. In the first case, the lemma directly follows. Let τ_j be the task currently locked r_x which is being executed in an abortable segment, in the second case, τ_i will abort task τ_j and then the lock request of τ_i on r_x will be granted.

Since τ_i is one of the tasks which might access to r_x, the priority ceiling of r_x is no less than the priority of τ_i (*i.e.*, $PL_x \geq P_i$). And, the system ceiling of τ_i is no less than the priority ceiling of r_x (*i.e.*, $PL_i^* \geq PL_x$) because r_x is currently locked by a task τ_j other than τ_i. When the conditions $P_i = PL_i^*$ and $a_i < b_i$ are satisfied, it is implied that $PL_i^* = PL_x = P_i$. In other words, r_x is the resource which has the highest priority ceiling of resources currently locked by a task other than τ_i. When τ_j is aborted by τ_i, the value of τ_i's system ceiling will be reduced because r_x becomes unlocked. Hence, $P_i > PL_i^*$ is satisfied. Based on the above two cases, the lemma directly follows. ❑

Theorem 5.1 A task instance can be blocked or abort a lower priority task for at most once.

Proof: Consider three tasks τ_i, τ_a and τ_b where the priority of τ_i is higher than the priorities of τ_a and τ_b, *i.e.*, $P_i > P_a$ and $P_i > P_b$. Suppose that an instance of a task τ_i has been blocked by τ_a or has aborted τ_a because they are conflicting on a resource r_x when the instance of τ_i is initiated. Later, the instance of τ_i can also be blocked by τ_b or abort τ_b only if they are conflicting on another resource r_y. By Lemmas 5.1 and 5.2, the only possibility is that r_x and r_y have been locked by τ_a and τ_b, respectively, when the instance of τ_i is initiated.

Since the priority of τ_i is higher than that of τ_a and τ_b (*i.e.*, $P_i > P_a$ and $P_i > P_b$) and τ_i is one of the task which might access to r_x and r_y, the priority ceiling of r_x and r_y are no less than the priority of τ_i, *i.e.*, $PL_x > P_i$ and $PL_y > P_i$. Assume that r_x has been locked

before r_y is locked. The system ceiling of τ_b is no less than the priority ceiling of r_x (*i.e.*, $PL_b^* \geq PL_x$) when r_x is locked by τ_a. By the ceiling rules, the lock request for r_y cannot be granted if r_x has been locked by τ_a. Because the priority of τ_b is less than its system ceiling, *i.e.*, $P_b > PL_b^*$. On the other hand, similar results can be obtained by assuming that r_y has been locked before r_x is locked. This contradicts the assumption that r_x and r_y have been locked by τ_a and τ_b when the instance of τ_i is initiated. Thus, it is impossible for τ_i to be blocked or abort lower priority tasks for more than once. The theorem follows immediately. ❑

Theorem 5.2 The CB-CAS prevents deadlocks.

Proof: Since CB-CAS inherits the ceiling rules from PCP, a resource r_x can be locked by a task τ_i if and only if r_x is unlocked, and $P_i > PL_i^*$. Consider two tasks τ_i and τ_j where the priority of τ_i is higher than the priority of τ_j, *i.e.*, $P_i > P_j$. Suppose that both of τ_i and τ_j might access to two resources r_x and r_y. The priority ceiling of r_x and r_y are no less than the priority of τ_i. If τ_i has locked a resource r_x or r_y, the system ceiling of τ_j is no less than the priority of τ_i, *i.e.*, $P_j < PL_j^*$, hence, task τ_j cannot lock another resource. Similarly, if τ_j has locked a resource r_x or r_y, the system ceiling of τ_i is no less than the priority of τ_i and τ_i cannot lock another resource. When a resource is locked (no matter which task locked the resources first), it raises the system ceiling and the condition $P_i > PL_i^*$ prevents the so-called *hold-and-wait* situation. Lemma 5.3 shows that the condition $P_i > PL_i^*$ is still satisfied even if τ_i has aborted a lower priority task. Therefore, tasks scheduled by CB-CAS are deadlock free. ❑

6. PERFORMANCE EVALUATION

The experiments described in this section will evaluate the capabilities of CB-CAS in scheduling of periodic hard real-time tasks on a non-ideal DVS processor. We have implemented a simulation to schedule different task workloads. To evaluate the performance of our proposed CB-CAS, we compared the energy consumption with the following approaches:

- **Uniform Slowdown with Frequency Inheritance (USFI) [8]:** USFI computes a uniform slowdown factor (*i.e.*, processor speed) for each task's execution. When a task τ_i blocks other tasks it inherits the maximum slowdown factor of the blocked tasks. We implemented this approach by using RMS and PCP as its scheduling policy and concurrency control protocol. Note that USFI was not designed for tasks with abortable critical sections. We will not abort any task even if the task is being executed in an abortable segment.
- **Conditional Abortable Priority Ceiling Protocol (CA-PCP) [19]:** Based on the schedulability analysis of RMS and PCP, a condition is defined to control the aborts of a task so that the schedulability of the system will not be affected. When tasks are scheduled under CA-PCP, the condition will be adjusted dynamically by taking the system utilization and the length of critical sections (including abortable and unabortable segments) into consideration. Note that CA-PCP was not designed for scheduling of tasks

on a DVS processor. We set the execution speed of tasks as the maximum processor speed.

- **Independent Task Set Transformation (ITST):** We transform the given task set into an independent task set and use RMS to schedule the transformed task set. In particular, for each task τ_i, its worst-case computation time C_i is increased by its worst-case blocking time B_i to result in a transformed task set $\mathcal{T}' = \{\tau_1', \tau_2', \cdots, \tau_n'\}$ where each task is defined by $\tau_i' = \{T_i, D_i, C_i' = (C_i + B_i)\}$. The execution speed of tasks is calculated by $\min_{s_j \in S}\{s_j \mid s_j \geq \sum_{i=1}^{n} \frac{C_i'/T_i}{n(2^{1/n}-1)}\}$.
- **Maximum Speed (MS):** This is a baseline approach. Tasks are scheduled by RMS and PCP, and executed at the maximum processor speed. This baseline approach will not abort any task even if they are being executed in abortable segments.

In the rest of this section, we shall present the performance metrics, data set, and experimental results.

6.1 Performance Metrics and Data Set

In our simulation, the processor speeds and their power consumptions are chosen from Intel XScale [28, 29], as shown in Table 3. The primary performance metric of interest is the energy consumption of tasks, referred to as *EnergyConsum*, which is the sum of the energy consumption of all task instances executed during the simulation time. *EnergyConsum* can be calculated by $\int_0^{simTime} PC(s(t))dt$, where $PC()$, $s(t)$, and *simTime* are the power consumption function, the processor speed at time t, and the simulation time. Note that the value of power consumption of a speed s_i, *i.e.*, $PC(s_i)$, can be found in Table 3.

The parameter settings of the workloads are given in Table 4. For generating feasible task sets, we set the utilization factor of tasks from 0.1 to 0.6 with an increment of 0.1, where the utilization factor is calculated by $\sum_{\tau_i \in \mathcal{T}} (C_i/T_i)$. Each generated task set con-

Table 3. Available speeds and power consumptions for Intel XScale [28, 29].

Speed (MHz)	150	400	600	800	1000
Normalized speed	0.15	0.4	0.6	0.8	1
Power concumption (mW)	80	170	400	900	1600

Table 4. Parameters of workload.

Parameter	*Value*
Utilization factor	(0.1, 0.6) step by 0.1
The number of tasks in the system	20-100
Period	(100, 2000)ms
Worst-case computation time	(10, 300)ms
The number of resources in the system	(5, 10)
The number of resources accessed by a task	(0, 5)
Critical section ratio, *csr*	0.1, 0.3, 0.5
Abortable segment ratio, *asr*	(0, 1) step by 0.2
Simulation time	1,000,000ms

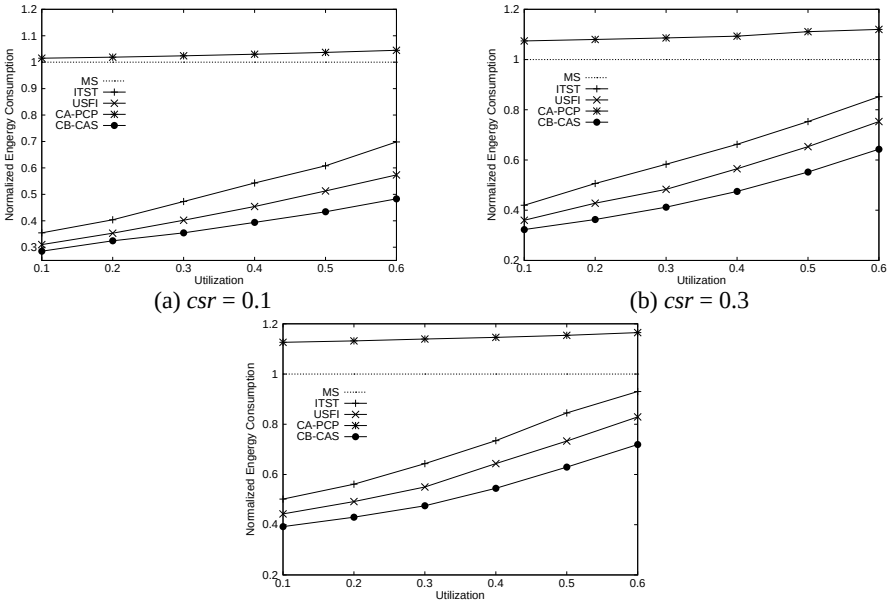
sists of 20 to 100 tasks by normal distribution. The period of a task was selected from 100ms to 2000ms by normal distribution. We assume that tasks' deadlines are equal to their periods, *i.e.*, $D_i = T_i$, $\forall \tau_i \in \mathcal{T}$. The worst-case computation time of a task was selected from 10ms to 300ms by normal distribution. The number of resources is 5 to 10 in the system, and each task will access 0 to 5 resources. Hence, there will be a sufficient number of resource conflicts so that the performance of evaluated approaches could be better understood. The positions and the lengths of the critical sections within each task's execution are selected randomly. However, the length of any critical section $z_{i,j}$ is set randomly from 0 to $csr*C_i$, where *csr* is the *critical section ratio* and is selected from 0.1, 0.3, and 0.5. The length of the abortable segment of any critical section $z_{i,j}$ is no larger than $asr*|z_{i,j}|$, where *asr* is the *abortable segment ratio* and its value is set from 0 to 1 with an increment of 0.2. After the task set was generated, the worst-case computation times were scaled such that the utilization of tasks would not exceed the desired value, and a critical section $z_{i,j}$ was removed if the sum of the lengths of τ_i's critical sections is larger than τ_i's computation time. The simulation time is 1,000,000ms and over 100 task sets per utilization factor, critical section ratio, and abortable segment ratio were tested in the simulation and their results were averaged.

6.2 Simulation Results

For ease of comparison, the energy consumption of evaluated approaches were normalized with respect to the baseline approach, *i.e.*, MS (tasks are always executed at the maximum processor speed.). In the first part of our simulation, we evaluated the effect of the lengths of critical sections on energy consumption. We fixed the abortable segment ratio *asr* to 0.6 and varied utilization factor between 0.1 to 0.6. Figs. 3 (a)-(c) show the experimental results of different approaches, when the critical section ratio *csr* are 0.1, 0.3, and 0.5, respectively.

As shown in Fig. 3, the energy consumption of all evaluated approaches grew with utilization factor. It also shown that our proposed CB-CAS outperforms all others in all cases. In particular, CB-CAS achieved a 71.5% reduction of the energy consumption compared to MS when the utilization of tasks is 0.1 and $csr = 0.1$, as shown in Fig. 3 (a). This is because MS always execute tasks at the maximum processor speed, *i.e.*, 1000MHz with power consumption of 1600mW, while CB-CAS could execute tasks at a lower processor speed. Since the value of the execution speed s^* of CB-CAS is dominated by the utilization of tasks and the maximum value of the worst-case blocking time and the re-execution cost of each task τ_i, *i.e.*, $\max\{B_i, A_i\}$, as we can observe in Eq. (3) and Eq. (4), and the worst-case blocking time and the re-execution cost of a task are highly related to the lengths of critical sections. CB-CAS could execute tasks at a lower processor speed, when the utilization is low and the lengths of critical sections are short. In fact, in our simulation, almost every task instance of all the generated task sets is executed at the minimum processor speed, *i.e.*, 150MHz with power consumption of 80mW, when utilization is 0.1 and $csr = 0.1$. Hence, a significant energy saving (compared to MS) is obtained for tasks scheduled by CB-CAS.

In Fig. 3, the performance ranking is CA-PCP, MS, ITST, USFI, and CB-CAS (from the worst to the best). Both of MS and CA-PCP use the maximum processor speed



(a) *csr* = 0.1 (b) *csr* = 0.3

(c) *csr* = 0.3

Fig. 3. Normalized energy consumption with varying utilization factor when abortable segment factor is 0.6.

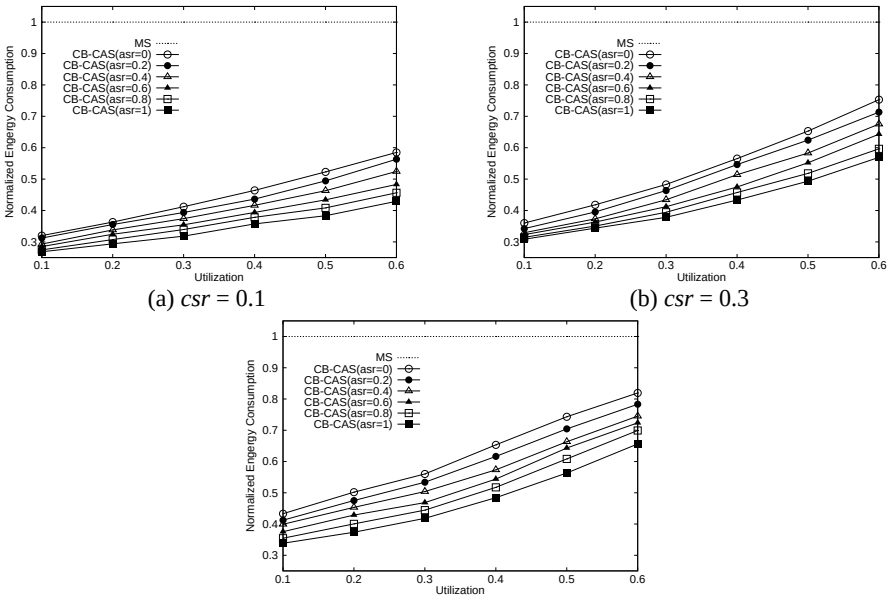
for task executions. However, CA-PCP performs worse because its additional re-execution time for the aborted critical sections. Hence, the aborting of tasks without scaling the processor speed properly will result in a higher energy consumption. As the lengths of critical sections increased with the critical section ratio *csr*, the resource conflicts were also occurred more frequently, *i.e.*, the more number of blockings and aborting will alos be increased. As the results, CA-PCP performs even worse for tasks with higher *csr*, as shown in Fig. 3 (c).

Compare to CA-PCP and MS, other three approaches (*i.e.*, ITST, USFI, and CB-CAS) use lower processor speeds for task executions such that the energy consumption is reduced. In particular, the execution speeds of tasks are calculated based on the utilization of tasks (*i.e.*, the utilization factor) under ITST, USFI, and CB-CAS. As the utilization increased with the utilization factor, the calculated execution speeds are also getting higher. Since the simulation results were normalized to MS (which uses the maximum processor speed for task executions), the energy consumption of ITST, USFI, and CB-CAS grew with utilization, as shown in Fig. 3.When critical section ratio *csr* is increased, the number of resource conflicts is also increased. Hence, the performance of ITST, USFI, and CB-CAS are getting worse, as shown Figs. 3 (a)-(c).

The ITST performs better than CA-PCP and MS because it uses a lower processor speed to execute tasks. However, its performance is worse than USFI and CB-CAS. It is because ITST transforms tasks by adding the worst-case blocking time to tasks' computation time, *i.e.*, $C'_i = C_i + B_i$. In other words, ITST assumes the blocking will be occurred for every instance of tasks. Although ITST does not perform well, it provides an upper bound of the energy consumption for scheduling and synchronization tasks with shared

resources. In other words, any well-designed approach would not perform worse than ITST in energy consumption.

The performance of USFI outperforms all approaches except our proposed CB-CAS. It is because the design of USFI does not take abortable critical sections into consideration. Under our proposed CB-CAS, tasks are executed at a lower processor speed and tasks might be blocked or abort lower-priority tasks (depend on which one consumes less energy), when tasks are conflicting on the same resources. The results of CA-PCP have shown that the aborting will result in more energy consumption because the abortable segment of an aborted critical section has to be re-executed. However, compare to CA-PCP, our proposed CB-CAS aborts a task only if it is more energy-efficient. Also note that CB-CAS performs the dynamic speed adjustment method to derive a proper processor speed for saving more energy. As a result, CB-CAS outperforms all others in all cases.



(a) *csr* = 0.1 (b) *csr* = 0.3

(c) *csr* = 0.5

Fig. 4. Normalized energy consumption with varying utilization factor for CB-CAS with different settings of abortable segment factors.

In the second part of our simulation, the effect of the lengths of aborable segments on energy consumption is evaluated for our proposed CB-CAS. Fig. 4 shows the experimental results with different settings of the abortable segment ratio. As the lengths of critical sections are increased with *csr*, the energy consumption is getting higher (as shown in Figs. 4 (a)-(c)) because the number of blockings and abortings is also increased. Note that the blocking time and the re-execution time for aborted tasks are also increased with *csr*. When *asr* = 0, CB-CAS degrades to have a blocking option only when a resource conflict occurs. Hence, the results for CB-CAS with *asr* = 0 is worst. When *asr* is increased, the aborting becomes another option for handling resource conflicts. Because

CB-CAS aborts lower-priority tasks only if it is more energy efficient, and a proper processor speed is calculated by the dynamic speed adjustment method, the energy consumption is reduced with the increasing of the length of abortable segments.

7. CONCLUSION

In this paper, we are interested in energy-efficient scheduling of hard real-time tasks with abortable critical sections on a non-ideal DVS platform. We assume that real-time tasks are preemptive, periodic and fixed priority. Different from the past work, we assume that a critical section consists of an abortable segment and an unabortable segment. In particular, a task can be aborted when it is being executed in an abortable segment. Such a scheduling problem is $\mathcal{NP}$-hard and its objectives is to minimize the energy consumption of tasks while the timing constraints are met.

Based on the well-known rate-monotonic scheduling (RMS) algorithm and the priority ceiling protocol (PCP), we proposed a novel approach, called *ceiling-based conditional abortable scheduling* (CB-CAS), for scheduling tasks with abortable critical sections. When two tasks are conflicting on the same resource, CB-CAS introduces a conditional abort rule to abort the lower-priority task if it is being executed in an abortable segment and it is more energy efficient. Based on the schedulability analysis, a lowest possible processor speed is calculated for task executions. We also propose a dynamic speed adjustment method to slow down the processor speed as low as possible such that the energy consumption could be reduced further. Our theoretic analysis shows that CB-CAS is deadlock free and the maximum number of abortings or blockings is no more than one. The capabilities of CB-CAS were evaluated by a series of experiments, for which we have some encouraging results. When tasks are scheduled by CB-CAS, the schedulability can be predicted while the energy consumption can be reduced.

ACKNOWLEDGEMENT

The authors would like to acknowledge the two anonymous referees for their valuable comments in improving the quality of the paper.

REFERENCES

1. J.-J. Chen and C.-F. Kuo, "Energy-efficient scheduling for real-time systems on dynamic voltage scheduling (DVS) platforms," in *Proceedings of IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2007, pp. 28-38.
2. L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, Vol. 39, 1990, pp. 1175-1185.
3. T. P. Baker, "Stack-based scheduling for realtime processes," *Journal of Real-Time Systems*, Vol. 3, 1991, pp. 67-99.
4. F. Zhang and S. T. Chanson, "Processor voltage scheduling for real-time tasks with

non-preemptible sections," in *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, 2002, pp. 235-245.
5. J. Lee, K. Koh, and C.-G. Lee, "Multi-speed DVS algorithms for periodic tasks with non-preemptible sections," in *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2007, pp. 459-468.
6. A. M. Elewi, M. H. A. Awadalla, and M. I. Eladawy, "Energy efficient real-time scheduling of dependent tasks sharing resources," in *Proceedings of International Conference on Computer Engineering and Systems*, 2008, pp. 273-242.
7. R. Jejurikar and R. Gupta, "Energy aware task scheduling with task synchronization for embedded real time systems," in *Proceedings of International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2002, pp. 164-169.
8. R. Jejurikar and R. Gupta, "Energy aware task scheduling with task synchronization for embedded real time systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 25, 2006, pp. 1024-1037.
9. Y.-W. Pan, "Energy-efficient task synchronization for real-time systems on dynamic voltage scaling (DVS) platforms," Master Dissertation, Department of Computer Science and Information Engineering, National Pingtung Institute of Commerce, Pingtung, Taiwan, 2009.
10. K.-L. Kao, "DVS scheduling of real-time tasks with abortable critical sections," Master Dissertation, Department of Computer Science and Information Engineering, National Pingtung Institute of Commerce, Pingtung, Taiwan, 2011.
11. J. Wu, "A prediction-based approach for energy-efficient DVS scheduling of dependent real-time tasks," in *Proceedings of the Working in Progress Section of the IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011, pp. 33-36.
12. Y.-S. Chen, L.-P. Chang, and T.-W. Kuo, "Multiprocessor frequency locking for real-time task synchronization," in *Proceedings of ACM Symposium on Applied Computing*, 2008, pp. 289-293.
13. J. Wu, "BTS-SRP: an energy-efficient concurrency control protocol for embedded real-time systems," in *Proceedings of International Conference on Advances in Computer Science and Engineering*, 2013, pp. 200-203.
14. J. Wu and K.-L. Kao, "Energy-efficient scheduling of real-time tasks with abortable critical sections," in *Proceedings of International Symposium on Advances in Embedded Systems and Applications*, 2012, pp. 1788-1793.
15. J. Huang, J. A. Stankovic, K. Ramamritham, and D. Towsley, "On using priority inheritance in real-time databases," in *Proceedings of IEEE Real-Time Systems Symposium*, 1991, pp. 210-221.
16. H. Tokuda and T. Nakajima, "Evaluation of real-time synchronization in real-time Mach," in *Proceedings of the USENIX Mach Symposium*, 1991, pp. 213-221.
17. L.-C. Shu and M. Young, "A mixed locking/abort protocol for hard real-time systems," in *Proceedings of IEEE Workshop on Real-Time Operating Systems and Software*, 1994, pp. 102-106.
18. H. Takada and K. Sakamura, "Real-time synchronization protocols with abortable critical sections," in *Proceedings of International Workshop on Real-time Computing Systems and Application*, 1994, pp. 4852.

19. K.-Y. Lam and J. K.-Y. Ng, "A conditional abortable priority ceiling protocol for scheduling mixed real-time tasks," *Journal of Systems Architecture*, Vol. 46, pp. 573-585.
20. C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the Association for Computing Machinery*, Vol. 20, 1973, pp. 46-61.
21. Y. Wang, D. Liu, Z. Qin, and Z. Shao, "Optimally removing inter-core communication overhead for streaming applications on MPSoCs," *IEEE Transactions on Computers*, Vol. 62, 2013, pp. 336-350.
22. Y. Wang, H. Liu, D. Liu, Z. Qin, Z. Shao, and E. H.-M. Sha, "Overhead-aware energy optimization for real-time streaming applications on multiprocessor system-on-chip," *ACM Transactions on Design Automation of Electronic Systems*, Vol. 16, 2011, pp. 14:1-14:32.
23. Y. Wang, D. Liu, Z. Qin, and Z. Shao, "Memory-aware optimal scheduling with communication overhead minimization for streaming applications on chip multiprocessors," in *Proceedings of IEEE Real-Time Systems Symposium*, 2010, pp. 350-359.
24. Y. Wang, D. Liu, M. Wang, Z. Qin, and Z. Shao, "Optimal task scheduling by removing inter-core communication overhead for streaming applications on MPSoC," in *Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010, pp. 195-204.
25. X. J. R. Jeyaseelan, "DVS scheduling in multi core real time system," *International Journal of Advanced Research in Computer Engineering and Technology*, Vol. 2, 2013, pp. 524-529.
26. S. Pagani and J.-J. Chen, "Single frequency approximation scheme for energy efficiency on a multi-core voltage island," Technical Report (KIT-MTA-2013-0001), Department of Informatics, Karlsruhe Institute of Technology, Germany, 2013.
27. N. Anne and V. Muthukumar, "Energy aware scheduling of aperiodic real-time tasks on multiprocessor systems," *Journal of Computing Science and Engineering*, Vol. 7, 2013, pp. 30-43.
28. G. Chen, K. Huang, J. Huang, C. Buckl, and A. Knoll, "Effective online power management with adaptive interplay of DVS and DPM for embedded real-time system," in *Proceedings of the Euromicro Conference on Digital System Design*, 2013, pp. 881-889.
29. D. Rajan, R. Zuck, and C. Poellabauer, "A dual speed approach to workload-aware voltage scaling," Technical Report (TR-2006-05), Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, USA, 2006.
30. B. Mochocki, X. S. Hu, and G. Quan, "Transition-overhead-aware voltage scheduling for fixed-priority real-time systems," *ACM Transactions on Design Automation of Electronic Systems*, Vol. 12, 2007, Article No. 11.
31. F. Muhammad, B. M. Khurram, F. Muller, C. Belleudy, and M. Auguin, "Precognitive DVFS: minimizing switching points to further reduce the energy consumption," in *Proceedings of Work-In-Progress Session of the Real-Time and Embedded Technology and Applications Symposium*, 2008, pp. 9-12.
32. A. L. Mohan and A. S. Pillai, "Dynamic voltage scaling with reduced frequency switching and preemptions," *International Journal of Electrical and Electronics En-*

gineering, Vol. 1, 2011, pp. 10-14.
33. A. K. Mok, "Fundamental design problems for the hard real-time environment," Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

Jun Wu (吳卓俊) is an Assistant Professor in the Department of Computer Science and Information Engineering at National Pingtung Institute of Commerce, Pingtung, Taiwan. His research interests include real-time embedded systems and database systems. He received Best Paper Award from the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) in 2005. He received his BS degree in Computer Science and Information Engineering from I-Shou University, Kaohsiung, Taiwan, in 1996. He received his MBA degree in Information Management from National Yunlin University of Science and Technology, Yunlin, Taiwan, in 1998. He received his Ph.D. degree in Computer Science and Information Engineering from National Chung Cheng University, Chiayi, Taiwan, in 2004. He is a member of the IEEE.

Kai-Long Ke (柯凱朧) also known as Kai-Long Kao, received his BS and MS degrees in Computer Science and Information Engineering from National Pingtung Institute of Commerce, Pingtung, Taiwan, in 2010 and 2011, respectively. His research interests include real-time systems and energy-efficient scheduling.