Energy-Efficient Tasks scheduling Algorithm for Dual-core Real-time Systems

Wann-Yun Shieh
Chang Gung University
Computer Science and Information Engineering
Tao-Yuan, Taiwan
wyshieh@mail.cgu.edu.tw

Bo-Wei Chen
Chang Gung University
Computer Science and Information Engineering
Tao-Yuan, Taiwan
m9729012@stmail.cgu.edu.tw

***Abstract*— Real-time embedded devices have been widely used in our daily life. To satisfy the performance requirements, most current designs tend to apply the dual- or multi-core processor architecture in the systems. Such systems, however, usually have low power consumption demands. Therefore the Dynamic Voltage Scaling (DVS) technique has been included in most designs. In this paper, we focus our study on the energy-efficient task scheduling algorithm for the dual-core real-time systems. Our goal is to minimize the system's energy consumption and maintain the performance of task execution at the same time. To achieve this goal, we propose two approaches: off-line and on-line. For the off-line approach, we propose an Integer Linear Programming (ILP) based algorithm to find the optimal scheduling. For the on-line approach, we propose a heuristic algorithm. The experimental results show that the energy consumption can be reduced effectively by the heuristic algorithm, and is close to the optimal bounds obtained by the ILP model.**

Keywords-component; energy-efficient task scheduling; dual-core real-time system; integer linear programming

I. INTRODUCTION

Many mobile or portable devices, like car information systems, smart phones, CULV laptops etc, become more and more popular in our daily life. These devices usually need to execute the real-time operations like multimedia communications, digital signal processing, and video-stream displaying etc. To satisfy the performance requirements, most of their designs tend to apply the dual- or multi-core processor architecture. However, these devices still use the battery as the major power supply. Therefore the low-power design issues for multi-core real-time systems become more and more important.

To address this problem, many studies proposed the energy-aware task scheduling algorithms to arrange the executions for all real-time tasks on multiple cores (typical is dual) [4][7]. These approaches first calculate the utilization of incoming tasks. They defined the utilization of tasks as the ratio of time the tasks spending on executing their works in a time interval. The time interval is usually measured from the time the task released to its deadline. Through the evaluation of task utilization, they can predict whether the cores will become idle in that interval. In other words, if a task is predicted that it will finish the execution in one of cores before the deadline, then we can let it work on that core by a lower voltage to save the power consumption. Though decreasing a core's voltage will delay a task's finish time, the overall energy consumption can still be reduced by the equations of *Energy* (E) = *Power* (P) × *Execution time* (T) and $P = \alpha \times C_L \times V^2 \times f$, where α is the switching probability, C_L is the load capacitance, and V is the supply voltage. This is so called the dynamic voltage scaling (DVS) technique [2][3].

By using the DVS technique in the energy-aware task scheduling algorithm, each task will be scheduled to one of cores to perform the execution by its own supply voltage. Such a utilization-based approach is more suitable for periodic tasks. This is because we can obtain the information about task release time, period, workload, and deadline of each periodic task upon its first arrival, and we can use them to make an optimal scheduling strategy in advance. For aperiodic tasks, however, the only information we have is each incoming task's workload and deadline, and we have to use them to make low-power scheduling decisions on the run-time.

We use a simple example to show the scheduling effects on aperiodic tasks. We assume the number of cores is two, and each core can be supplied by two voltage levels for DVS. Table 1 shows the properties of six tasks. We assume that their arrival times are 0, 0, 1, 1, 4, and 4, respectively. The simplest task scheduling is to use the as-soon-as-possible (ASAP) policy without DVS. When each task arrives, a scheduler will issue it to the core of which the workload is the lowest. The result after scheduling is shown in Fig. 1(a). We can see that both cores have slack time left after all tasks finish their executions.

Instead of immediately issuing the task to a core on its arrival, we let it wait in a queue first. When any one of cores finishes all of allocated tasks (we call this time the scheduling point), the scheduler will wake up to schedule the tasks waiting in the queue by predicting the workloads in two cores from current time to the worst deadline. Take the tasks in Table 1 for example. At time 0, task 1 and task 2 will be allocated to core 1 and core 2, initially. Because the deadline of task 2 is at time 4, we can let core 2 work in a lower voltage level, say half of the original one, to perform task 2's execution, as shown in Fig. 1(b). (Here we assume the execution time will become double as the voltage reduced half times.) The first scheduling point happens at time 2 because core 1 has task 1's work done. Recall that task 3 and task 4 arrived at time 1. Therefore the scheduler will pick them up from the queue and schedule them to core 1 and core 2 afterwards. Note that due to the deadline

constraints, they cannot be executed by a lower voltage. The scheduled result is shown in Fig. 1(c). The last scheduling point happens at time 5, where task 3 finishes the execution earlier than task 4. At this point, the scheduler will determine which cores task 5 and task 6 should be issued to, and by which voltage levels. We show one of scheduling results in Fig. 1(d).

In Fig. 1(d), we can find that each task can be issued dynamically to different cores by different supply voltages with all deadline constraints satisfied. If we let the high voltage be the 6 Volts and the low voltage be the 3 Volts, by replacing the variables with real values in the energy equations mentioned above, we can obtain that the energy consumption is reduced byabout 23.3% from Fig.1(a) to Fig. 1(d).

From above example, we conclude that the key design issue of a runtime energy-aware scheduling algorithm is, at each scheduling point, which cores should the waiting tasks be issued to, and which voltage level should be selected to perform each task's execution.

Table 1. Task Table

Task	Deadline	Workload
T_1	2	2
T_2	4	2
T_3	6	3
T_4	8	3
T_5	11	3
T_6	11	2

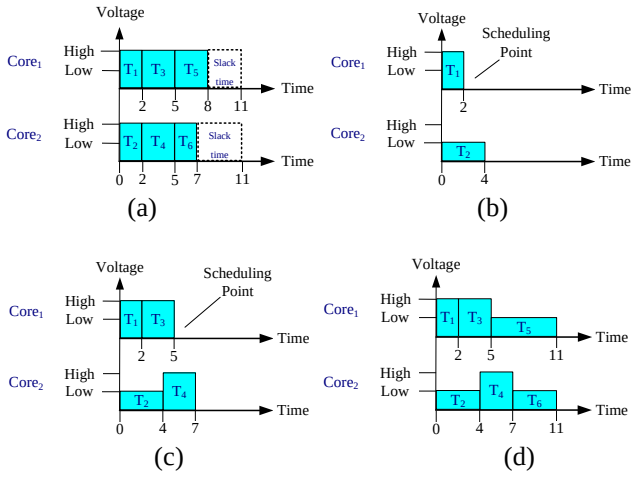


Fig. 1 The task scheduling of Table. 1

II. RELATEWORK

Many studies about real time task scheduling have been proposed. Most of them focused on static task scheduling. These static task scheduling approaches can also be partitioned into two types by their constraints; one is the time constraint [13][16], and the other is the resource constraint [12][16]. The time constraint is to limit the total execution time of function units for each task during scheduling, and the resource constraint is to limit the number of function units used in each task. Both of them have to obtain total tasks' detail information like release time, deadline, and execution time before scheduling; that's why we call them static task scheduling.

In recent years, however, the requirements of dynamic task scheduling have been addressed [4][8][7][17]. In [4], the authors proposed an algorithm to balance the periodic task loads on multi-cores and adjusted the number of active cores to reduce leakage power. In [8], the authors proposed a periodic task scheduling algorithm to reduce the system-level energy consumption. They delayed the execution time of some tasks to reduce the I/O device waiting time, thus reduce the energy consumption for I/O device to stay in idle state. In [7], the authors proposed an algorithm to schedule periodic tasks on common deadline for reduce energy consumption on multi-cores. In [17], the authors proposed an ILP solution for aperiodic tasks on unique core. They adjusted the execution voltage in a task at different time to reduce energy consumption.

These studies[4][7][8][17], however, were not suitable for the aperiodic task scheduling on multi-cores architectures. In the unique-core architecture, the scheduler considers only the time space to allocate tasks and determines their execution voltages. But in the multi-core architectures, the scheduler needs more considerations about core selection. This is because different core-selection policies will have different effects on the problems of load unbalancing between cores, or task deadline-miss rates. These issues become challenging for aperiodic tasks, and they are what we addressed in this paper.

III. PRELIMINARIES

We show the preliminaries for the ILP model. At first, we will give the basic definitions about the inputs of the model, including the tasks, and the processor cores. Then we will show the architectural model about the cores, tasks, and the scheduler. Finally, we will show the power model about the DVS.

A. Basic Definitions

Given an aperiodic task set $T = \{task_1, task_2, task_3 \ldots task_k\}$ and a core set $C = \{core_1, core_2\}$, our goal is to schedule the tasks on the cores to finish their executions on time and minimize energy consumption. Each task arrives upon its release time (R_i), and contains its properties including the predicted execution time (e_i) and the deadline (D_i). The release time is the time a task is forked to the processor. The predicted execution time can be obtained by evaluating how many operations in a task should be executed by the highest supply voltage on any one of cores. All operations in a task should be finished before the deadline. We also assume that the execution of the task is non-preemptive. Without loss of generality, we assume the number of cores is two, and each core can supply two level voltages (6V, 3V) for task execution.

B. Architecture model

The architectural model between the cores, tasks, and the scheduler is shown in Fig. 2. When a task is released, it will be inserted in the ready queue first. The scheduler will wake up when a core finishes the executions of the tasks allocated to it previously. At this time one core will become idle, and we call this time a scheduling point. Note that at each of this point, the scheduler will schedule all waiting tasks to both of the cores depending on each core's *free time slots* and *total workload* of tasks currently in the ready queue. The *free time slots* on each core can be counted from the current time to the worst deadline, which is the farthest deadline among the waiting tasks in the ready queue. The *total workload* is the accumulation of predicted execution time for all tasks waiting for scheduling. After scheduling, the ready queue becomes empty to wait for other incoming tasks, and the scheduler will sleep until the next time to be awakened.

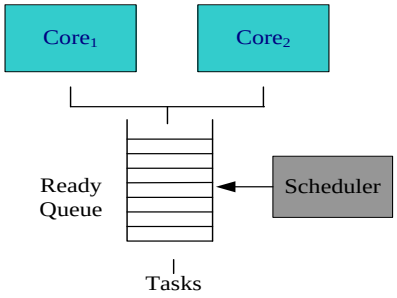


Fig. 2 Architecture model

C. Power model

We assume the core has two working modes, low supply-voltage mode and high supply-voltage mode. Each working mode will let a core have different power consumption and different delays in task execution. The relationship between the power consumption and the supply voltage follows the rule of (Eq. 1), where α is the switching probability, C_L is the load capacitance, and v is the supply voltage. From (Eq. 1), if the core needs T time slots to execute the task, we can obtain the energy consumption as shown in (Eq. 2). On the other hand, the relationship between the delayed execution time and the supply voltage follows the rule of (Eq. 3), where the v is the task executed voltage. That is, we assume that the execution time will become double as the voltage reduced half times.

$$Power\ consumption = \alpha \times C_L \times v^2 \quad \text{(Eq. 1)}$$

$$Energy\ consumption = Power\ consumption \times Execution\ time \quad \text{(Eq. 2)}$$

$$Execution\ time = \begin{cases} 1 \times predicted\ execution\ time\ \ ....\ \ v = high \\ 2 \times predicted\ execution\ time\ \ ....\ \ v = low \end{cases} \quad \text{(Eq. 3)}$$

Other variables which will be used in the following ILP model and the heuristic algorithm are defined in Table 2.

Table 2. Parameters of a DVS System

R_i	The $task_i$ release time.
D_i	The $task_i$ deadline.
e_i	The $task_i$ execute time when use highest voltage level.
T	Aperiodic Task set T={$task_1$,$task_2$,$task_3$…$task_k$}
$task_i$	Aperiodic $task_i$
v_i	The $task_i$ execution voltage level. v_i = 1 means task execute at high voltage, v_i = 2 means task execute at low voltage.
c_i	The $task_i$ execution at $Core_i$.
L_i	The task$_i$ start execution time.
M_i	The $task_i$ finished execution time.
Q_{ready}	The queue of store tasks which not be schedule.
V	The voltage set v, where the V={v_1,v_2}
C	The core set c, where's the C={c_1,c_2}

IV. ILP MODEL

In this section, we show the ILP model for the dual-core aperiodic task scheduling. To store the information about the scheduling results, we use the decision variable $x_{task_i,c_i,v_i,L_i,M_i}$ .The decision variable $x_{task_i,c_i,v_i,L_i,M_i}$ is an {0, 1} integer variable. If $task_i$ is scheduled on core$_i$ and uses voltage v_j to perform the execution in time interval (L_i,M_i), then $x_{task_i,c_i,v_i,L_i,M_i} = 1$. Otherwise, $x_{task_i,c_i,v_i,L_i,M_i} = 0$.

Recall that the energy consumption of a task running in a core depends on the core's α, c, and v. In this paper, we assume both cores are homogeneous architecture under the same technology. Therefore the values of α and C_L for each core can be considered as the constants. That is, $E_{total} = \sum_{task_i \in T} E_{task_i}$, where the $E_{taski} \propto power \times task_i\ execute\ time$, and the $power \propto v^2$. Using the decision variable $x_{task_i,c_i,v_i,L_i,M_i}$, we write the objective function as follows. Minimize:

$$E_{total} = \sum_{task_i \in T} (v_i^2 \times (M_i - L_i)) \quad \text{(Eq. 4)}$$

For the objective function, there were several constraints need to be followed to ensure the schedule is valid:

(1) Unique constraint

This constraint ensures that every task can be scheduled to only one core for execution with a particular voltage level at one time interval. We represent it as, $\forall$ taski $\in T$, 1<=i<=n,

$$\sum_{c_i=1}^{2}\sum_{v_i=low}^{high}\sum_{L_i=R_i}^{L_i+e_i-1}\sum_{M_i=L_i}^{D_i} x_{task_i,c_i,v_i,L_i,M_i} = 1 \qquad \text{(Eq. 5)}$$

(2) Task overlap constraint

This constraint ensures that at any time each core can execute at most one task. If $task_i$ and $task_j$ were executed at the same $core_i$, then their execution times cannot overlap. We represent it as, given core c_i executed $task_i$ by voltage v_i at time interval (L_i,M_i), and $task_j$ by voltage v_j at time interval (L_i,M_j), if $M_i > L_j$ and $L_j > L_i$, or $M_j > L_i$ and $L_i > L_j$,

$$\sum_{task_i \in T}\sum_{v_i=low}^{high} x_{task_i,c_i,v_i,L_i,M_i} + \sum_{task_j \in T}\sum_{v_j=low}^{high} x_{task_j,c_i,v_j,L_j,M_j} \le 1, \qquad \text{(Eq. 6)}$$

In Eq. 6, the first term in the left side represents all possible schedules for $task_i$ on core c_i, and the second term represents for $task_j$ on the same core. There are two cases that $task_i$'s execution may overlap with $task_j$'s execution: the first case (i.e., $M_i > L_j$ and $L_j > L_i$) is shown in Fig. 3(a), and the second case (i.e., $M_j > L_i$ and $L_i > L_j$) is shown in Fig. 3(b). It should guarantee that in both cases at most one task can be scheduled to its interval for execution.

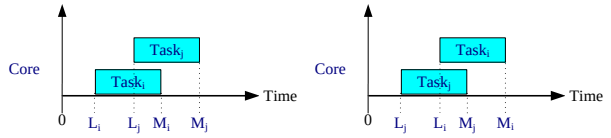


Fig. 3 Task overlap example

(3) Deadline constraint

This constraint ensures that each task should finish its execution before the deadline, no matter what voltages it used. We represent it as, $\forall$ taski$\in T$, 1<=i<=n,

$$M_i \le D_i \qquad \text{(Eq. 7)}$$

V. HEURISTIC ALGORITHM

Considering the time complexity of the ILP approach, we design a heuristic algorithm to schedule the tasks in the runtime. This algorithm will be executed at each scheduling point. We define the schedule point as the time when any one of cores becomes idle and at that time there are tasks waiting in the queue. In other words, between any two scheduling points, all arriving tasks will stay in the queue first. The major processes of the scheduling algorithm include picking up a task from the queue, and determining which core will perform its execution by what voltage. The details of the algorithm are shown in Fig. 4.

In algorithm *Heuristic_Scheduling*, the first step is to select a task which has the earliest deadline from the queue. Then we call a function named *DVS_decision* to decide which voltage should be used in the execution. If the free time space from now on to the task's deadline can tolerate the delay of DVS execution and expanding the task's execution would not cause remaining tasks to miss deadlines, then we can let it work by a low voltage. Next, we call another function named *Core_decision* to decide which core the task should be scheduled to. Here, the fact of load balancing will be the major concern. All above steps will repeat until all waiting tasks have been scheduled on. The details of *DVS_decision* and *Core_decision* are discussed below.

Algorithm Heuristic_Scheduling
Step1: Select a task (say $task_i$) from the queue, which has the earliest deadline;
Step2: Call DVS_decision to decide which supply voltage will be used in $task_i$'s execution;
Step3: Call Core_decision to decide which core will perform $task_i$'s execution;
Step4: Repeat Step1~Step3 until all waiting tasks have been scheduled.

Fig. 4 Heuristic scheduling algorithm

A. *DVS_decision*

In this function, we use two variables to decide a selected task's execution voltage: *Remain_time_slot* and *Expected_DVS_time_slot*. *Remain_time_slot* represents how many free time slots on both of cores from now on can be allocated to the remaining tasks. Assume there are *n* tasks waiting in the queue, and the first farthest, and the second farthest deadlines among these tasks are D_n, and D_{n-1}, respectively. Also we assume $Core_1$ will finish its current task execution at T_n , and $Core_2$ at T_{n-1}. Then *Remain_time_slot* can be calculated by

$$Remain\_time\_slot = (D_n + D_{n-1}) - (T_n + T_{n-1}) \qquad \text{(Eq. 8)}$$

On the other hand, the *Expected_DVS_time_slot* represents the minimal requirement of time space to perform the DVS execution for all waiting tasks. It can be calculated by

$$Expected\_DVS\_time\_slot = \sum_{task_i \in \text{ waiting tasks}} T_{task_i}^{DVS}, \qquad \text{(Eq. 9)}$$

where $T_{task_i}^{DVS}$ is the required time space to perform $task_i$'s execution by DVS.

Take Fig. 5 for example. We assume $Core_1$ will finish its current task up at time 5, and $Core_2$ at time 6. Also we assume at time 5 there are three tasks waiting in the queue, which the deadlines are 9, 11, 12, and the required DVS execution times are 4, 2, 4, respectively. Then we can obtain that *Remain_time_slot*=12 (i.e., (11+12)-(5+6)), and *Expected_DVS_ time_slot*= 10 (i.e., 4+2+4) in this case.

For a task selected from the queue, say $task_i$, if *Remain_time_slot* ≧ *Expected_DVS_time_slot* and one of cores has enough time space to tolerate $task_i$'s DVS execution (i.e., ≧ $T_{task_i}^{DVS}$), then we can let $task_i$ use a low voltage in execution. In this case, delaying $task_i$'s execution would not produce too much "time pressure" on the remaining waiting tasks.

For the contrary cases, i.e., either *Remain_time_slot* ≦ *Expected_DVS_time_slot* or both of cores have free time space smaller than $T_{task_i}^{DVS}$, then $task_i$ should use a high voltage to catch up the deadline constraint, the algorithm shows as Fig. 6.

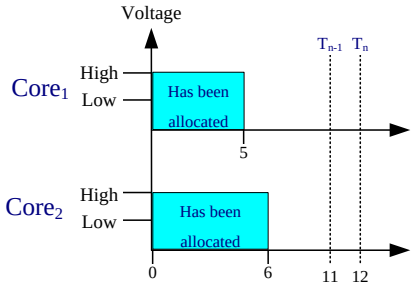


Fig. 5 Example of algorithm

Algorithm DVS_decision
Input: *Remain_time_slot*、*Expected_DVS_time_slot*
Output: $task_i$ execute voltage
Step1: If(*Remain_time_slot* >= *Expected_DVS_time_slot* and has one of cores has enough time space to tolerate $task_i$ DVS execution)
$task_i$ do DVS;
else
$task_i$ can't not do DVS;
Step2:Update *Remain_time_slot* 、 *Expected_DVS_time_slot*.
Fig. 6 DVS decision algorithm

B. *Core_decision*

There are two cases that we have to take care of when we schedule a task on one of cores. Take Fig. 5 for example. Assume $Core_1$ will finish its current task earlier than $Core_2$. We say that $Core_1$ has a "larger" free time space for scheduling, and $Core_2$ has a "smaller" one. If we tend to **s**chedule a task, say $task_i$, on $Core_1$, both of cores may leave too small free time space to accommodate remaining tasks' DVS execution. Conversely, if we tend to schedule $task_i$ on $Core_2$, $Core_1$ will have more free time space left, which can let more other tasks perform their execution by DVS. However, this is not always the best policy for all cases. For example, if we always compact the tasks collectively on one core (e.g., $Core_2$ in Fig. 5), and save more and more free time space on the other (i.e., $Core_1$ in Fig. 5), an obvious problem of load unbalancing will occur. To prevent this problem, we use a variable *Workload_density* to decide which core $task_i$ should be scheduled on.

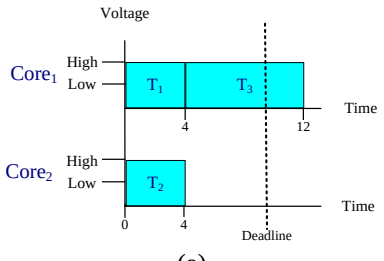
The variable *Workload_density* represents the average workloads per unit time that two cores may suffer during the remaining free time clots. It can be calculated by summarizing the total workloads of all waiting tasks, and dividing it by *Remain_time_slot*, as shown in (Eq. 10):

$$Workload\_density = \frac{\sum_{task_i \in \text{waiting tasks}} T_{task_i}^{Non\_DVS}}{Remain\_time\_slot}, \qquad \text{(Eq. 10)}$$

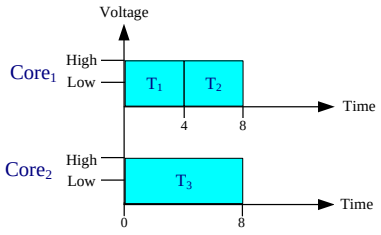
where $T_{task_i}^{Non\_DVS}$ is the minimal requirement of free time space to run $task_i$'s execution on any one core by the highest voltage (i.e., Non_DVS). For example, in Fig. 5, *Remain_time_slot* is 12, and the total required time space for all tasks without DVS is 5 (i.e., (4+2+4)/2); therefore the *Workload_density* is 0.41.

When we select a core for scheduling, we will compare current *Workload_density* and free time space in two cores. We setup a threshold α to identify if *Workload_density* is high or low. If *Workload_density* is greater than α, we will use the policy of which the waiting tasks are scheduled as compact as possible on the core having smaller free time space (e.g., $Core_2$ in Fig. 5). This policy will change reversely if the free time space on the other core (i.e., $Core_1$ in Fig. 5) have been greater than an amount. For example, If we have $task_i$ to be scheduled and now $Core_1$ has free time space of size over than four times of $T_{task_i}^{Non\_DVS}$, then we schedule $task_i$ to $Core_1$ instead. This can make sure that the workloads of two cores after scheduling would not differ too much.

Fig. 7 shows such an example. Assume that we have $task_1$, $task_2$, and $task_3$ to be scheduled in order and the deadlines of them are 5, 9, 10, respectively. If we have scheduled $task_1$ on $Core_1$ from time 0 to time 4, then we have two options to schedule $task_2$: the first is to schedule it on $Core_2$ like Fig. 7 (a), and the second is to schedule it on $Core_1$ like Fig. 7 (b). It is clear to know that only saving more free time space on one core, as shown in Fig.7 (b), can let $task_3$ work without missing its deadline. If now we assume the deadline of $task_3$ is far from time 10 and we continue to schedule $task_3$ on $Core_1$, it is obvious that the workloads of two cores are very unbalanced, which may cause many side-effects on utilization, and energy-saving etc.



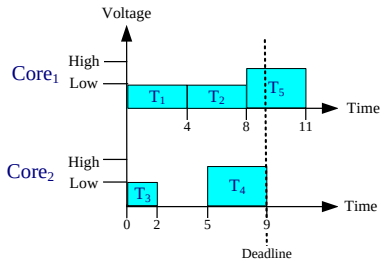
(a)



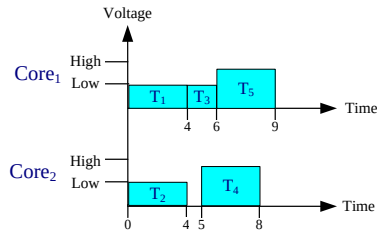
(b)

Fig. 7 Example of *Workload_density* is greater than α

On the hand, if *Workload_density* is lower than α, we will consider the core having larger free time space first for scheduling. Fig. 8 shows such an example. Assume at first we have $task_1$, $task_2$, and $task_3$ to be scheduled in order and their deadlines are 5, 8, and 8 respectively. If we tend to compact the tasks on one core and leave more free time space on the other, then we have the scheduling as shown in Fig. 8(a). Otherwise we have another scheduling as shown in Fig. 8(b). Now assume that we have $task_4$ and $task_5$ arriving at time 5, and their deadlines are the same at time 9. For Fig. 8(a), it is obvious that $task_4$ can be scheduled to $Core_2$, but there are no free time space left for $task_5$ any more. But for Fig. 8(b), both of cores have enough free time space to accommodate $task_4$ and $task_5$ at the same time. This example shows that Fig. 8(a)'s policy is not suitable for the tasks of low *Workload_density* because may free time space saved from previous tasks will become unused or say a waste for the later tasks. In this case, it is better to let previous tasks (i.e., $task_1$ to $task_3$) finish their execution as early as possible and let later tasks (i.e., $task_4$ and $task_5$) start their execution immediately.



(a)



(b)

Fig. 8 Example of *Workload_density* is lower than α

The detail algorithm of *Core_decision* is shown in Fig. 9. In our experiments, we use different workloads of tasks to evaluate the value of α. The experimental results shows the better value of α would be 0.4 for most cases.

Algorithm Core_decision
Input: $task_i$、α
Output: $task_i$ execute at which core
If(*workload_density* > α && both of cores has enough time space to tolerate $task_i$'s execution && no free time slot >= 4* $task_i$ execution time)
  Choose the core with small free time space;
else
  Choose the core with large free time space;

Fig. 9 Core decision algorithm

VI. Experimental Result

We implement a task generator to the generate tasks aperiodically for evaluation. We first determine the workload of each task, say WL_{task_i}, by a normal distribution model with mean ρ. The WL_{task_i} is defined as:

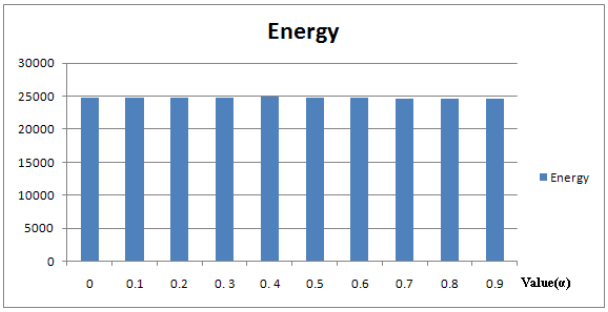
$$WL_{task_i} = \frac{T_{task_i}^{Non\_DVS}}{D_i - R_i}, \qquad \text{(Eq. 11)}$$

where D_i and R_i are the deadline and the release time of $task_i$. In later experiments, we will assign the value of ρ from 0.1 to 0.9 to represent different workloads of tasks. Once the release time (R_i) and the workload of each task (WL_{task_i}) are determined, its minimal requirement of free time space for high-voltage execution ($T_{task_i}^{Non\_DVS}$) and the deadline (D_i) can be obtained. We let each task's release time (R_i) follow the Poisson distribution.

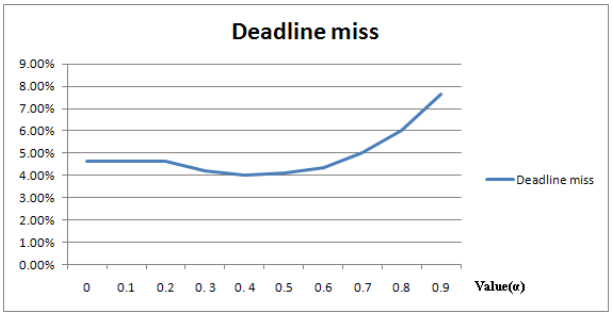
To evaluate the optimal scheduling, we implement the ILP model in Lingo 11.0 [18], which is an optimization modeling software for linear Programming. The results found by Lingo can be considered as a bound for conventional approaches. On the other hand, we implement a runtime scheduler to run the heuristic algorithm. The outputs of the task generator discussed above will be fed into the scheduler. The evaluation metrics collected from the scheduler include the energy consumption, deadline miss rate, and each core's utilization. Here we use the same power model as used in [7] to calculate the energy consumption. The deadline miss rate is the ratio of tasks missing their deadlines by their executions, and each core's utilization is the ratio of total working time over the total time for each core.

At first, we vary the threshold α in algorithm *Core_decision* to determine which core selection policy should be used for high or low *Workload_density* of tasks. Fig. 10 shows the impacts of different α on energy consumption and deadline miss rate. We can find that α has

few impacts on the former but has significant impacts on the later. This is because when the α is increased, the most tasks will allocate on core with large free time slots, thus causes the deadline miss (i.e., *Core*$_1$ in Fig. 7(a).) But some of tasks will allocate on another core (i.e., *Core*$_2$ in Fig. 7(a).), which means that the core has enough time slots for those tasks execute with DVS, thus reduce energy consumption. Because the minimal deadline miss rate happens at α=0.4, we apply this value in the following experiments.



(a)



(b)

Fig. 10 The Different α value in algorithm *Core_decision*

Fig. 11 shows the energy savings obtained by our scheduling algorithm for different workloads of tasks (i.e., varied by ρ). We compare the energy consumption of the heuristic algorithm with the ILP algorithm. Both of results are normalized to the energy consumption without any DVS execution. When ρ equals to 0.1, the energy consumption of the heuristic algorithm can be saved about 38% while the bound is about 42%. The energy savings reduce when ρ increases. This is because when the workloads of tasks are very heavy, both of cores almost have no free time space to let the tasks perform their execution by DVS. Fig. 11 also shows that the differences between the heuristic algorithm's results and the bounds of energy saving can be limited in 5%.

Fig. 12 further shows the deadline miss rates of the heuristic algorithm for different workloads of tasks. We can find that only when ρ=0.9 the deadline miss rate has a significant increase, about 10%. For most cases, the deadline miss rates are less than 10%.

Fig. 13 further shows the core utilization of the heuristic algorithm for different workloads of tasks. For each cores utilization only has about 4.8% difference average.

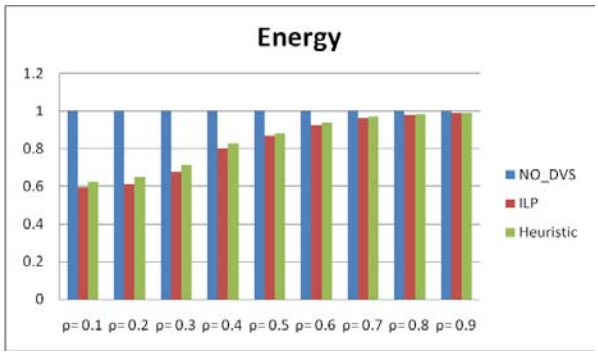


Fig. 11 Energy saving bound

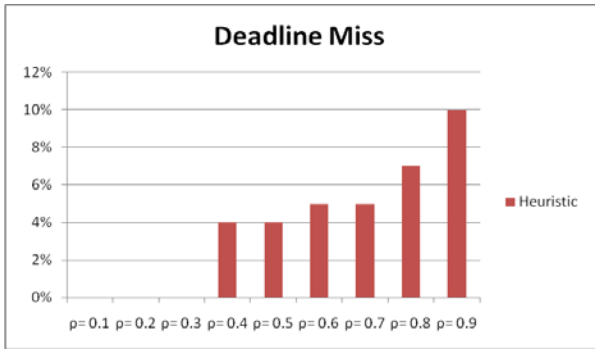


Fig. 12 Deadline miss of heuristic algorithm

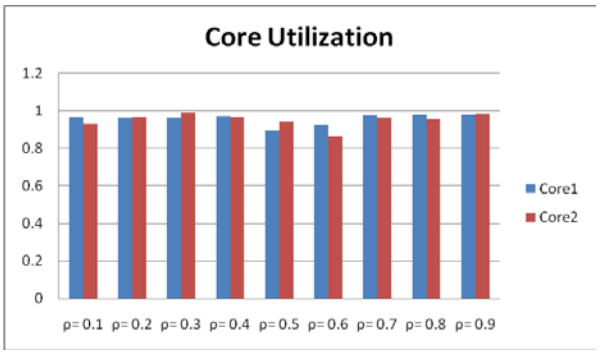


Fig. 13 Core utilization of heuristic algorithm

VII. CONCLUSION

In this paper, we proposed an energy-aware task scheduling algorithm for aperiodic tasks running on a dual-core system. We developed the ILP models for the off-line approach to find optimal solutions, and designed a heuristic algorithm for the on-line approach.

In the proposed heuristic algorithm, we use two decision functions to reduce core's energy consumption and maintain task's performance. The *Core_decision* will leave more time space in one core during scheduling tasks; therefore the following tasks can get larger time space to execute with low voltage to reduce energy consumption. The *DVS_decision* will let as most as waiting tasks perform their execution by DVS but satisfying their deadline constraints;

thus performance can be maintained. The experimental results show the energy consumption can reduce effectively by our heuristic algorithm, and is close to the optimal bounds obtained by the ILP model.

REFERENCES

[1] AYDIN, H et al., "Dynamic and aggressive scheduling techniques for power-aware real-time systems," IEEE Real-Time Systems Symposium (RTSS), 95-105, 2001

[2] BURD, T. et al., "Design issues for dynamic voltage scaling," In Proceedings of International Symposium on Low Power Electronics and Design (ISLPED-00), 2000

[3] X. FEN et al., "Intraprogram Dynamic Voltage Scaling:Bounding Opportunities with Analytic Modeling," ACM Transactions on Architecture and Code Optimization, Vol. 1, No. 3, Pages 323–367, September 2004

[4] Euiseong Seo et al., "Energy Efficient Scheduling of Real-Time Tasks on Multicore Processors," IEEE Transactions on Parallel and Distributed System, Vol.19, No. 11, 2008

[5] G. Magklis et al., "Dynamic Frequency and Voltage Scaling for a Multiple-Clock-Domain Microprocessor," IEEE Micro., Vol.23, 62-68, 2003

[6] G. QUAN et al., "Energy Efficient DVS Scheduling for Fixed-Priority Real-Time Systems," ACM Transactions on Embedded Computing System, Vol 6, No. 4, Article 29, 2007

[7] J. Chen et al., "Energy-Efficient Real-Time Task Scheduling in Multiprocessor DVS System," Asia and South Pacific Design Automation Conference, 2005

[8] J. ZHUO et al., "Energy-Efficient Dynamic Task Scheduling Algorithms for DVS System," ACM Transactions on Embedded Computing System, Vol 7, Article 17, 2000

[9] J. ZHUO et al., "System-level energy-efficient dynamic task scheduling ," IEEE Design Automation Conference, page 628-631, 2005

[10] J.M. Lopez et al., "Worst-Case Utilization Bound for EDF Scheduling on Real-Time Multiprocessor Systems," Proc. 12th Euromicro Conf. Real-Time Systems, 25-33, 2000

[11] P. Tan, "Task scheduling of Real-time System Systems on Multi-Core Architectures," IEEE Computer Society, 2009

[12] P. Mohanty, "Peak Power Minimization Through Datapath Scheduling," IEEE Computer Society Annual Symposium on VLSI, 2003

[13] P. Mohanty, "Energy-Efficient Datapath Scheduling Using Multiple Voltages and Dynamic Clocking," ACM Transactions on Embedded Computing System, Vol10, No. 2, Pages 330-353, 2005

[14] R. Jerjurikar et al., "Leakage Aware Dynamic Voltage Scaling for Real-Time Embedded Systems," Proc. 41st Ann. Technical Conf. Design Automation, page. 275-280, 2004

[15] S.K. Baruah, "Optimal Utilization Bounds for the Fixed-Priority Scheduling of Periodic Task Systems on Identical Multiprocessors," IEEE Trans. Computers,Vol.53, 781-784, 2004

[16] W. Shiue, "Low-Power Scheduling with Resource Operating at Multiple Voltages," IEEE Transactions on Circuits and System, Vol. 47, No 6, 2000

[17] W. KWON et al., "Optimal Voltage Allocation Techniques for Dynamically Variable Voltage Processors," ACM Transactions on Embedded Computing System, Vol 4, Pages211-230, 2005

[18] http://www.lindo.com