An Energy-efficient Task Scheduler for Multi-core Platforms with per-core DVFS Based on Task Characteristics

Ching-Chi Lin, You-Cheng Syu, Chao-Jui Chang, Jan-Jan Wu, Pangfeng Liu, Po-Wen Cheng and Wei-Te Hsu

***Abstract*—Energy-efficient task scheduling is a fundamental issue in many application domains, such as energy conservation for mobile devices and the operation of green computing data centers. Modern processors support dynamic voltage and frequency scaling (DVFS) on a per-core basis, i.e., the CPU can adjust the voltage or frequency of each core. As a result, the core in a processor may have different computing power and energy consumption. To conserve energy in multi-core platforms, we propose task scheduling algorithms that leverage per-core DVFS and achieve a balance between performance and energy consumption. We consider two task execution modes: the *batch* mode, which runs jobs in batches; and the *online* mode in which jobs with different time constraints, arrival times, and computation workloads co-exist in the system.**

For tasks executed in the *batch* mode, we propose an algorithm that finds the optimal scheduling policy; and for the *online* mode, we present a heuristic algorithm that determines the execution order and processing speed of tasks in an online fashion. The heuristic ensures that the total cost is minimal for every time interval during a task's execution.

***Keywords*-Energy-efficient; Task scheduling; Task Characteristics; Multi-core; DVFS**

I. Introduction

Energy-efficient task scheduling is a fundamental issue in many application domains, e.g., energy conservation for mobile devices and the operation of green computing data centers. A number of studies have proposed energy-efficient techniques [1], [2]. One well-known technique is called dynamic voltage and frequency scaling (DVFS), which achieves energy savings by scaling down a core's frequency and thereby reducing the core's dynamic power.

Modern processors support DVFS on a per-core basis, i.e., the CPU can adjust the voltage or frequency of each core. As a result, the cores in a processor may have different computing power and energy consumption. However, for the same core, increased computing power means higher energy consumption. The challenge is to find a good balance between performance and energy consumption.

Many existing works focus on specific application domains, such as real-time systems [3], [4], [5], multimedia applications [6], and mobile devices [7]. In this paper, we consider a broader class of tasks. Specifically, we classify task execution scenarios into two modes: the *batch* mode and the *online* mode. The workload of the former comprises batches of jobs; while that of the latter consists of jobs with different arrival times and time constraints. We divide the jobs in the *online* mode into two categories: *interactive* and *non-interactive* tasks. An *interactive* task is initiated by a user and must be completed as soon as possible; while a *non-interactive* task may not have a strict deadline or response time constraint. In the *online* mode, tasks can arrive at any time.

An example of the *online* mode is an online judging system where users submit their answers or codes to the server. After performing some computations, the server returns the scores or indicates the correctness of the submitted programs. In this scenario, user requests are *interactive* tasks that require short response times. Note that the response time refers to the acknowledgement of receipt of the user's data, not the time taken to return the scores. By contrast, the computation of user data in *non-interactive* tasks is not subject to time constraints.

To conserve energy in multi-core platforms, we propose task scheduling algorithms that leverage per-core DVFS and achieve a balance between performance and energy consumption. In our previous work [8], we have made the following contributions.

- We present a task scheduling strategy that solves three important issues simultaneously: the assignment of tasks to CPU cores, the execution order of tasks, and the CPU processing rate for the execution of each task. To the best of our knowledge, no previous work has tried to solve the three issues simultaneously.
- To formulate the task scheduling problems, we propose a task model, a CPU processing rate model, an energy consumption model, and a cost function. The results of simulations and experiments performed on a multi-core x86 machine demonstrate the accuracy of the models.
- For the execution of tasks in the *batch* mode, we propose an algorithm called *Workload Based Greedy* (WBG), which finds the optimal scheduling policy for both single-core and multi-core processors. Our experiment results show that, in terms of energy consumption, WBG achieves 46% and 27% improvement over two existing algorithms respectively, with only a 4% slowdown and 13% speedup in the execution time.
- To execute tasks in the *online* mode, we propose a heuristic algorithm called *Least Marginal Cost* (LMC), which assigns *interactive* and *non-interactive* tasks to cores. It also determines the processing speeds that minimize the total cost of every time interval during a task's execution in an online fashion.

In this paper, we have improved our work with the following extensions.

- For batch mode, we define the "dominating position set",

which determines the frequency of each batch task according to its position in the task queue. We also derive an algorithm that computes the dominating position ranges efficiently. The algorithm runs in $\Theta(|P|)$, where P is a non-empty set of discrete processing rates on a core.
- For online mode, we propse a method to compute the total cost efficiently with dynamic task insertion and deletion. The time complexity of cost computation is $\Theta(1)$. With this algorithm, we can reduce the overhead of our Least Marginal Cost.

The remainder of this paper is organized as follows. In the next section, we formally define the models for executing tasks, determining the core processing rates, and reducing energy consumption. In Section III, we discuss the proposed *Workload Based Greedy* algorithm, which derives the optimal scheduling policy for the *batch* mode. We also provide a theoretical analysis to prove the optimality of the algorithm. In Section IV, we discuss the proposed heuristic algorithm, *Least Marginal Cost*, for the *online* mode. The experiment results are presented in Section V; Section VI reviews related works; and Section VII contains our concluding remarks.

II. MODELS

A. Task Model

A task comprises a sequence of instructions to be executed by a processor. We define a task j_k in a task set J as a tuple $j_k = (L_k, A_k, D_k)$, where L_k is the number of CPU cycles required to complete j_k, A_k is the arrival time of j_k, and D_k is the deadline of j_k. If j_k has a specific deadline, $D_k > A_k \geq 0$; otherwise, we set D_k to infinity, which indicates that j_k is not subject to a time constraint.

For the *batch* mode, we make two assumptions about the tasks to be executed. First, tasks are non-preemptive, which means that a running task cannot be interrupted by other tasks. In practice, this non-preemptive property reduces the overheads of task switching and migration. Second, we assume that tasks are independent and can be scheduled in an arbitrary order. As the tasks in a batch are not interdependent and can be executed simultaneously , we assume that the arrival time, A_k, of every task is 0. The implication is that the scheduler has the timing information about all the tasks to be processed and it can run the tasks in any order based on the current scheduling policy.

For the *online* mode, we divide tasks into two categories according to their time constraints. *Interactive* tasks are those with early and firm deadlines, so the response time is crucial for such tasks. *Non-interactive* tasks are those with a late deadline or no deadline. We also make the following assumptions about *online* mode tasks. (1) The number of cycles needed to complete a task is known because it can be estimated by profiling. (2) The tasks are independent of each other. (3) A task can preempt other tasks that have a lower priority. In the *online* mode, a task's priority depends on its category. *Interactive* tasks have higher priority than *non-interactive* tasks.

B. Processing Rate

Modern processors support dynamic voltage and frequency scaling (DVFS) on a per-core basis; therefore, each core in a processor may have its own processing rate or frequency. Let $P = \{p_1, p_2, p_3, \ldots\}$ be a non-empty set of discrete processing rates a core can utilize based on the hardware, with $0 < p_1 < p_2 < p_3 < \ldots < p_{|P|}$. We use p_{j_k} from set P to denote the processing rate of a task j_k. Different processors provide different processing rates. For example, the processing speeds offered by the Intel i7-950 processor range from 1.6, 1.73 to 3.06 GHz; while those of the ARM Exynos-4412 CPU range from 0.2, 0.3 to 1.7 GHz.

We also make some assumptions about the processing rate for the *batch* mode and *online* mode. In the *batch* mode, the processing rate of a CPU core does not change during a task's execution. A core only switches to a new frequency when it starts a new task. By contrast, in the *online* mode, a core can change its processing rate any time based on the scheduling decisions.

C. Energy Consumption

For a task j_k, let e_k denote the energy consumption in joules; t_k denote the execution time in seconds; and p_{j_k} be the processing rate used to execute j_k. Recall that L_k the number of cycles needed to complete task j_k. We define $E(p)$ and $T(p)$ as the energy and the time required to execute one cycle with processing rate p on a CPU core , with the property that $0 < E(p_1) < E(p_2) < E(p_3) < \ldots$ and $0 < \ldots < T(p_3) < T(p_2) < T(p_1)$. Because we assume that a core's processing rate is fixed while running a task in the *batch* mode, we can formulate the energy consumption e_k and execution time t_k of task j_k as shown in Equations 1 and 2.

$$e_k = L_k E(p_{j_k}) \quad (1)$$
$$t_k = L_k T(p_{j_k}) \quad (2)$$

III. TASK SCHEDULING IN THE BATCH MODE

In this section, we discuss our energy-efficient task scheduling approach for *batch* mode execution. We consider running tasks with and without deadlines in both single core and multi-core environments. As a result, we have four combinations of tasks and environments. Below, we formally define the energy-efficient task scheduling problem in each of the four scenarios, and present our theoretical findings for the defined problems.

A. Tasks with Deadlines

We define the problem of scheduling tasks with deadlines on a single core as follows. Given a set of tasks j_k, the number of execution cycles needed L_k, the tasks' deadlines D_k, the possible processing rates of the core P, the energy consumption and time consumption functions E and T, and the total energy budget E^*, the goal is to determine the execution order of the tasks and their processing rates, such that every task can be completed before its deadline and the overall energy consumption is less than E^*.

We call the above problem *Deadline-SingleCore*, and reduce the *Partition* problem to show that *Deadline-SingleCore* is NP-complete. Let $A = \{a_1, \ldots, a_n\}$ be a set of positive integers. The partition problem involves determining if we can partition A into two subsets so that the sums of the numbers in the two subsets are equal. Given a problem instance $A = \{a_1, \ldots, a_n\}$, we construct a problem instance in *Deadline-SingleCore* such that one problem can be solved if and only if the other one has a solution.

Theorem 1: Deadline-SingleCore is NP-complete.

Proof: We construct a problem instance in *Deadline-SingleCore* as follows. There are n tasks $j_1, \ldots, j_n$; and the number of cycles needed for the first n tasks is $L_i = a_i$, as in the given partition problem instance. We use $S = \sum_{i=1}^{n} a_i$ to denote the total number of cycles required to complete n tasks. There are only two processing rates: low speed p_l and high speed p_h; the latter is twice as fast as the former. We also assume that $T(p_l) = 2$, $T(p_h) = 1$, $E(p_h) = 4$ and $E(p_l) = 1$, based on the assumption that the dynamic part of the energy consumption is proportional to the square of the frequency. This assumption follows the classical models in the literature [9]. The time constraint is $1.5S$ and the energy constraint is $2.5S$. The deadline of every task is $1.5S$.

Now we have $1.5S$ time and $2.5S$ energy to run the n tasks. Because $T(p_h)$ and $T(p_l)$ equal 1 and 2 respectively, we need to select the number of tasks whose sum is at least $S/2$ to run at high speed p_h so that all the tasks can be completed in $1.5S$ time. In addition, because $E(p_h)$ and $E(p_l)$ are 4 and 1respectively, we need to select the number of tasks whose sum is at least $S/2$ to run at low speed p_l so that the energy consumption of all the tasks does not exceed $2.5S$.

We conclude that the total number of cycles required for tasks that run at high speed p_h is the same as that for the tasks that run at low speed p_l. Hence, the *Partition* problem can be solved if and only if a solution is found for our *Deadline-SingleCore* problem. The theorem follows. ■

Theorem 1 states that the problem of deciding the processing rate for tasks with deadlines under time and energy constraints on a single core is NP-complete. The single-core results can be extended to multi-cores as follows. We define the *Deadline-MultiCore* problem in a similar way to the *Deadline-SingleCore* problem, and prove that it is NP-complete. We only consider two cores, each of which has the same speed p with $T(p)$. The deadline constraint is set as $S/2$; we do not consider the energy constraint. The problem is exactly the same as partition problem; therefore, *Deadline-MultiCore* is also NP-complete.

Theorem 2: Deadline-MultiCore is NP-complete.

B. Tasks without Deadlines on a Single Core Platform

Next, we consider the problem of scheduling tasks without deadlines. Given a set of tasks and the number of cycles needed to process them, the goal is to find the execution order and the processing rate for each task so that the overall "cost" is minimized.

If we only consider energy consumption, we could run every task at the lowest processing rate in order to minimize the energy used, but it would degrade the performance. On the other hand, running every task at the highest processing rate so as to minimize the total execution time would waste energy. Thus, the cost function must consider both the energy consumption and the execution time. Our cost function converts the two parameters into monetary values. We define the cost of a task j_k as the sum of its energy cost and temporal cost. The energy cost $C_{k,e}$ of task j_k is the amount of money paid for the energy used to execute the task. Recall that $E(p_{j_k})$ is the amount of energy in joules required to execute one cycle with processing rate p_{j_k} on a core. Therefore, the total energy in joules needed to run a task j_k with processing rate p_{j_k} is $L_k E(p_{j_k})$; and the amount of money paid for the energy is $R_e L_k E(p_{j_k})$ as shown in Equation 3, where R_e is a positive constant, which means the cost of a joule of energy. R_e can be regarded as the amount paid for one joule of energy in an electricity bill.

$$C_{k,e} = R_e L_k E(p_{j_k}) \tag{3}$$

Similarly, we define the *temporal cost* $C_{k,t}$ of task j_k as the amount of money paid to compensate a user for waiting for his/her job to be completed. Recall that $T(p_{j_k})$ is the time in seconds required to execute one cycle with processing rate p_{j_k} on a core, so the total time needed to run a task j_k with processing rate p_{j_k} is $L_k T(p_{j_k})$. Without loss of generality, we assume that the execution order of tasks is $j_1, \ldots, j_n$; therefore, the turnaround time for task j_k is comprised of the time waiting for $j_1, \ldots, j_{k-1}$ to be completed and the *execution time* of j_k itself. As a result, the turnaround time of j_k is $\sum_{i=1}^{k} L_i T(p_{j_i})$. In addition, the temporal cost of task j_k is $R_t \sum_{i=1}^{k} L_i T(p_{j_i})$ as shown in Equation 4, where R_t is also a positive constant, which means the amount paid for every second a user has to wait for the execution of his/her task. R_t can be regarded as an *opportunity cost*, which is the amount of money we could earn if we could move the resources elsewhere. Alternatively, R_t can be thought of as the amount a user is willing to pay for a computing service, such as Amazon EC2.

$$C_{k,t} = R_t \sum_{i=1}^{k} L_i T(p_{j_i}) \tag{4}$$

To obtain C_k, the cost of task j_k, we combine the energy cost $C_{k,e}$ and the temporal cost $C_{k,t}$. Using the weighted sum of the energy and flow time as the cost objective function is based on previous works [10], [11]. The total cost of all tasks, denoted as C, is calculated by Equation 8.

$$C_k = C_{k,e} + C_{k,t} \tag{5}$$

$$= R_e L_k E(p_{j_k}) + R_t \sum_{i=1}^{k} L_i T(p_{j_i}) \tag{6}$$

$$C = \sum_{k=1}^{n} C_k \tag{7}$$

$$= \sum_{k=1}^{n} (R_e L_k E(p_{j_k}) + R_t \sum_{i=1}^{k} L_i T(p_{j_i})) \tag{8}$$

Equation 8 is difficult to analyze due to the interaction between a task and all the tasks ahead of it in the execution

sequence. We consider this problem from another perspective. Instead of computing the waiting time caused by other tasks (the second term in Equation 6), we compute the amount of delay that a task causes for other tasks. Consider a task j_k. If j_k runs at processing rate p_{j_k}, the energy cost will be $R_e L_k E(p_{j_k})$ and the time cost will be $(n-k+1)R_t L_k T(p_{j_k})$ for itself and the tasks after it. That is, the temporal cost of j_k will be $R_t L_k T(p_{j_k})$, and that of task j_{k+1} will be $R_t L_k T(p_{j_{k+1}})$, and so on. We can rewrite Equation 8 as follows:

$$C = \sum_{k=1}^{n}(R_e L_k E(p_{j_k}) + (n-k+1)R_t L_k T(p_{j_k})) \quad (9)$$
$$= \sum_{k=1}^{n}(R_e E(p_{j_k}) + (n-k+1)R_t T(p_{j_k}))L_k \quad (10)$$
$$= \sum_{k=1}^{n} C(k, p_{j_k})L_k \quad (11)$$

Note that we define $C(k,p)$ as

$$C(k,p) = R_e E(p) + (n-k+1)R_t T(p) \quad (12)$$

Now we can rewrite Equation 8 as follows:

$$C = \sum_{k=1}^{n} C(k, p_{j_k})L_k \quad (13)$$

Equation 13 shows that when we want to minimize $C(k, p_{j_k})$ in order to minimize C, it is not necessary to consider L_k, the number of cycles needed to execute task j_k. We only need to find the processing rate p_{j_k} that minimizes $C(k, p_{j_k})$ for each k. In other words, the minimum value of $C(k, p_{j_k})$ only involves k, the position of the task in the execution sequence, and it is independent of the task assigned to that position.

Lemma 1: The decision about the processing rate p_{j_k} required to minimize $C(k, p_{j_k})$ only depends on k, the position of the task in the execution sequence.

Lemma 1 implies that we can calculate the minimum $C(k, p_{j_k})$ for each k beforehand if P, E, T, R_e, and R_t are known. As a result, we can find the optimal p_{j_k} that will minimize each $C(k, p_{j_k})$, and then determine the best processing rate without any knowledge of the workload.

Lemma 2: Let $C(k) = \min_{p \in P} C(k,p)$, $C(k)$ is a decreasing function of k, i.e. $C(k+1) < C(k)$.

Proof: Let the optimal processing rate of $C(k)$ be p. If we use p as the processing rate for $C(k+1)$, then we have $C(k+1,p) - C(k,p) = -R_t T(p) < 0$. Therefore, $C(k+1) \leq C(k+1,p) < C(k,p) = C(k)$. ■

Lemma 3: For any four real numbers a, b, x, y where $a \geq b$ and $x \geq y$, then $ax + by \geq ay + bx$.

Proof:

$$(a-b)(x-y) \geq 0 \quad (14)$$
$$\Longrightarrow \quad ax - ay - bx + by \geq 0 \quad (15)$$
$$\Longrightarrow \quad ax + by \geq ay + bx \quad (16)$$

■

Theorem 3: There exists an optimal solution with the minimum cost, where the tasks are in non-decreasing order of the number of cycles.

Proof: Recall that $C(k) = \min_{p \in P} C(k,p)$. From Lemma 1, we know that when the task execution order is $j_1, \ldots, j_n$, the minimum total cost is as follows:

$$C = \sum_{k=1}^{n} C(k)L_k \quad (17)$$

From Lemma 2, we know that $C(k)$ is a decreasing function of k; and from Lemma 3, we know that the cost will not increase if a task with a small number of cycles is switched with a task in front of it that has more cycles. By repeating this process on an optimal solution until there are no tasks to switch, the tasks will be in non-decreasing order of the number of cycles required to complete them, and it is still an optimal solution. ■

To eliminate n (the number of tasks) for generality, we define L_k^B, j_k^B, $C^B(k,p)$, $C^B(k)$ as follow: ("B" means backward)

$$L_k^B = L_{n-k+1} \quad (18)$$

$$j_k^B = j_{n-k+1} \quad (19)$$

$$C^B(k,p) = C(n-k+1,p) = R_e E(p) + kR_t T(p) \quad (20)$$

$$C^B(k) = \min_{p \in P} C^B(k,p) \quad (21)$$

Define $f_i(k) = C^B(k, p_i) = (R_e E(p_i)) + (R_t T(p_i))k$. By definition, p_i is the best choice for j_k^B i.f.f. $f_i(k) = \min_s f_s(k)$. By Inequation 25, we know that for any two different processing rates p_a and p_b with $p_a < p_b$, p_b is no worse than p_a i.f.f. $k \geq \frac{R_e(E(p_b)-E(p_a))}{R_t(T(p_a)-T(p_b))}$.

$$f_a(k) \geq f_b(k) \quad (22)$$
$$\Longleftrightarrow \quad R_e E(p_a) + R_t T(p_a)k \geq R_e E(p_b) + R_t T(p_b)k \quad (23)$$
$$\Longleftrightarrow \quad R_t T(p_a)k - R_t T(p_b)k \geq R_e E(p_b) - R_e E(p_a) \quad (24)$$
$$\Longleftrightarrow \quad k \geq \frac{R_e(E(p_b)-E(p_a))}{R_t(T(p_a)-T(p_b))} \quad (25)$$

Define D_p, the "dominating position set" of p, to be the set of k such that p is the best choice for j_k^B (choose the higher processing rate in case of a tie). (So $D_{p_1}, D_{p_2}, \ldots, D_{p_{|P|}}$ is a partition of natural numbers $\mathbb{N}$) To find D_p for each $p \in P$ is equivalent to find the lower envelope of $f_1, f_2, f_3, \ldots, f_{|P|}$. Due to f_i are linear functions with $y = b + ax$ form, it is equivalent to find the lower (convex) hull in the dual space (transform line $y = b + ax$ to point (a,b)), and the elements in each dominating position set will be consecutive. (So we can call "dominating position set" as "dominating postition range" instead.) In conclusion, we can find the dominating position ranges efficiently via Algorithm 1, which runs in $\Theta(|P|)$.

The pseudo code of the task ordering algorithm is detailed in Algorithm 2, which runs in $O(|J|\log|J|)$. ($\hat{P} = \{p | p \in P \wedge D_p \neq \emptyset\} = \{\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_{|\hat{P}|}\}; \hat{p}_1 < \hat{p}_2 < \ldots < \hat{p}_{|\hat{P}|}$)

Algorithm 1 Finding Dominating Position Ranges

Input: P, E, T, R_e, R_t
Output: D_p ($p \in P$), $\hat{P}$

```
1: function cross(t_0, t_1, t_2)
2:     return (t_1.x − t_0.x)(t_2.y − t_0.y) − (t_2.x − t_0.x)(t_1.y − t_0.y)
3: end function
4: for p ∈ P do
5:     D_p ← ∅
6: end for
7: P̂ ← ∅
8: initialize a stack S; top ← 0
9: for i ← 1 to |P| do
10:    t ← (p = p_i, x = R_t T(p_i), y = R_e E(p_i))
11:    while top ≥ 2 and cross(s[top − 1], s[top], t) ≥ 0 do
12:        top ← top − 1
13:    end while
14:    top ← top + 1
15:    S[top] ← t
16: end for
17: lb ← 1
18: for i ← 1 to top − 1 do
19:    nlb = ⌈(s[i+1].y − s[i].y) / (s[i].x − s[i+1].x)⌉
20:    if lb < nlb then
21:        D_{s[i].p} ← [lb, nlb)
22:        P̂ ← P̂ ∪ {s[i].p}
23:    end if
24:    lb ← nlb
25: end for
26: D_{s[top].p} ← [lb, ∞)
27: P̂ ← P̂ ∪ {s[top].p}
```

Algorithm 2 Longest Task Last

Input: $J, \hat{P}, E, T, R_e, R_t$
Output: The execution order O (a list of pairs of tasks and corresponding processing rates that minimize the total cost)

```
1: Find dominating position ranges via Algorithm 1
2: Sort the tasks in J to make L_k^B in non-increasing of k
3: initialize an empty list O
4: for p ∈ P̂ in ascending order do
5:     if D_p ∩ [1, |J|] = ∅ then break
6:     for k ∈ D_p ∩ [1, |J|] in ascending order do
7:         O ← {(j_k^B, p)} + O // concatenate
8:     end for
9: end for
```

C. Scheduling Tasks without Deadlines on Multi-core Platforms

If the cores in a multi-core system are the same type, it is called a *homogeneous* multi-core system. In this subsection, we consider scheduling tasks in *homogeneous* multi-core systems and *heterogeneous* multi-core systems.

The cores in a *homogeneous* multi-core system have the same energy consumption and time consumption functions E and T; hence, they have the same C function, as defined in Equation 12. For ease of discussion, we use an index $k' = n-k+1$ on C' function to describe Equation 12. The index is defined in Equation 26. The rationale is that k starts counting from the beginning of the sequence, while k' starts from the end of the sequence; $k-1$ is the number of tasks in front of the current task, and $k'-1$ is the number of tasks behind it.

$$C'(k', p_{j_{k'}}) = C(n-k+1, p_{j_{n-k+1}}) \tag{26}$$

First, we describe scheduling tasks in a *homogeneous* multi-core system with R cores. From Lemma 2 we know that $C(k)$ is a decreasing function of k; therefore, $C'(k')$ is a non-decreasing function of k'. As a result, we use $k' = 1$ to schedule the R heaviest tasks, i.e., those that require the highest numbers of cycles, on the R cores first. The *last* task on each of the R cores will be one of these tasks, which implies they will be multiplied by the smallest $C'(1)$ so that they contribute the least amount to the total cost. Then, we use $k' = 2$ to schedule the next R heaviest tasks on the R cores in the same manner. We use this round-robin technique to assign tasks so that those with a larger number of cycles are assigned smaller k', and therefore smaller $C'(k')$. Using a similar argument to that in Theorem 3 we derive the following theorem.

Theorem 4: A round-robin scheduling technique that assigns heavier tasks to smaller k' yields the minimum cost in a *homogeneous* multi-core system

Next, we consider a *heterogeneous* multi-core system in which all the cores may have different energy consumption and time consumption functions E and T. We extend $C'(k')$ for a single-core system to $C'_j(k')$ for a *heterogeneous* multi-core system, where j is the index of a core and $1 \leq j \leq R$. Tasks are assigned to cores in a greedy fashion. First, we compute $C'_j(1)$ for $1 \leq j \leq R$. That is, we compute the lowest C' value among all cores if we place a task on that core, and we use j^* to denote the core index where C' is minimized. From the discussion in Theorem 4, we know that the heaviest task should be placed on core j^*. We then find the minimum among $C'_j(1)$ for $j \neq j*$ and $C'_{j*}(2)$, and assign the second largest task to that core. This process is repeated until all the tasks have been assigned to cores. Note that $C'_j(k')$ is independent of the task workload and can be computed in advance for all cores. In practice, we can build a minimum heap to store the $C'_j(k')$ we are considering. In each round, we take the minimum from the heap and add the next $C'(k')$ to the heap from the core that the task is assigned to. Using a similar argument to that in Theorem 4, we can show that this greedy method yields the minimum total cost. We have the following theorem.

Theorem 5: Using a greedy scheduling algorithm to assign heavier tasks to cores with smaller $C'(k')$ yields the minimum cost in a *heterogeneous* multi-core system.

The pseudo code of the greedy algorithm is detailed in Algorithm 3.

IV. Task Scheduling in the Online Mode

In this section, we use an example to describe task scheduling in the *online* mode. Then, we formally define the problem

Algorithm 3 Workload Based Greedy

Input: tasks $j_1, \ldots, j_n$; the number of cycles $L_1, \ldots, L_n$;the set of processing rate P; the energy consumption and time consumption functions E and T for all R cores; R_e, the cost of a joule of energy; and R_t, the amount paid for every second a user has to wait.

Output: An execution sequence S_j for each of the R cores, and the processing rates that minimizes the total cost of each task.

1: Sort the tasks by L_i in decreasing order. Let the new order of tasks be $j'_1, \ldots, j'_n$.
2: Initialize a heap H with $C'_j(1)$, for j between 1 and R.
3: **for** each task j'_i **do**
4: Delete the minimum element $C'_{j^*}(k')$ from heap H.
5: Assign j'_i to the k'-th position in the execution sequence of core j^*, and set its processing rate to the one that minimizes and defines $C'_{j^*}(k')$.
6: Add $C'_{j^*}(k'+1)$ to H.
7: **end for**

of energy efficient task scheduling in the *online* mode and present our heuristic algorithm.

As mentioned in Section I, an online judging system is a good example of scheduling in the *online* mode. In such systems, users submit their answers or codes to the server; and after some computations, the server returns the scores or indicates the correctness of the submitted programs. User requests are *interactive* tasks that require short response times. Recall that the response time refers to the acknowledgment of receipt of the user's data, not the time taken to return the scores. By contrast, the computations of users' data are *non-interactive* tasks that do not have strict deadlines. Because each *non-interactive* task may be submitted by a different user, the performance should be considered in terms of the completion time of each task instead of the makespan of executing all tasks.

We formally define the problem as follows. There are two types of tasks in the *online* mode: *interactive* and *non-interactive*. *Interactive* tasks are submitted by users and have strict deadlines, so they must be completed as soon as possible. By contrast, *non-interactive* tasks may not have strict deadlines or response time constraints. Tasks can arrive at any time. The goal of scheduling is to assign tasks to cores and determine the processing speeds that minimize the total cost for every time interval during the execution of tasks.

We make the following assumptions about the system. (1) The system can be a *homogeneous* or a *heterogeneous* multi-core system. (2) There is an execution queue for each core. The scheduler assigns a task to a core based on our policy. (3) The execution order of tasks in the same queue can be changed. (4) A task can only be preempted by a task with higher priority. (5) *Interactive* tasks have higher priority than *non-interactive* tasks. The cost function in the *online* mode considers both the execution time and the energy consumption.

Note that the *Workload Based Greedy* algorithm can be used to redistribute all tasks to cores when a new task arrives. According to Theorem 5, rearranging the tasks yields the minimum cost. However, because the overhead incurred by the time and energy used to migrate tasks could impact the performance, we need a lightweight strategy without task migration. To this end, we designed a heuristic algorithm, called *Least Marginal Cost*, to schedule both *interactive* and *non-interactive* tasks. The algorithm assigns each newly arrived task to a core, and determines the optimal processing frequency for the task. When a new task arrives, the scheduler finds an appropriate core for it. The most appropriate core is the one that yields the lowest marginal cost if the newly arrived task is executed on that core. We apply different strategies according to the type of task.

If the newly arrived task is *interactive*, it must be completed as soon as possible. Thus, the scheduler chooses a core that is executing a task with lower priority, preempts that task, and executes the new task instead. After finishing the new *interactive* task, the scheduler resumes the pre-empted task. The execution order of other tasks in the queue is unchanged. The increasing cost of C_j^M on core j is calculated as follows:

$$C_j^M = R_e L_i E_j(p_m) + R_t L_i T_j(p_m) + R_t L_i T_j(p_m) N_j \quad (27)$$

In Equation 27, C_j^M denotes the marginal cost incurred if we run an *interactive* task on core j; R_t and R_e are the amounts paid for every second and every joule of energy respectively; L_i is the number of cycles needed to complete the *interactive* task. $E_j(p_m)$ and $T_j(p_m)$ are, respectively, the energy consumption and the time taken for a cycle under the maximum frequency p_m of core j; and N_j is the number of (*non-interactive*) tasks waiting in the queue on core j. The *Least Marginal Cost* algorithm compares the marginal costs and assigns the *interactive* task to core j^*, where $C_{j^*}^M = \min C_j^M$. Note that if the cores are *homogeneous*, we simply choose the core with the least N_j.

If the new task is *non-interactive*, it is added to the execution queue instead of preempting the current task. The marginal cost of adding a *non-interactive* task to a core depends on the position of the new task in queue. According to Theorem 3, the optimal solution with the minimum cost is derived when the tasks are in non-decreasing order of the number of cycles. Because the tasks in a queue are sorted in non-decreasing order, we can perform a binary search to find the position k for the new task. The tasks are still in non-decreasing order after the new task is added.

Least Marginal Cost chooses a core for the new *non-interactive* task in a greedy fashion. First, the scheduler finds k_j for each core j, where k_j is the position that the task will be inserted. It then estimates the marginal cost C_j^M if the new task is inserted in position k_j. The core j^* with the lowest marginal cost will be chosen. The new task is inserted in the k_{j^*}-th position on core j^*. The processing frequency of each task on core j^* is adjusted according to $C(k, p_k)$, where k is the position of the task in the execution sequence.

A. Dynamic Task Insertion and Deletion

In Section III-B we show that how to schedule tasks without deadlines on single-core platform with all tasks known in the

very beginning. In this section, we are going to show how to schedule tasks and compute the total cost efficiently with dynamic task insertion and deletion.

First, we define three functions:

$$\xi([a,b]) = \sum_{k=a}^{b} L_k^B \tag{28}$$

$$\Delta([a,b]) = \sum_{k=a}^{b} (k-a+1) L_k^B \tag{29}$$

$$\gamma([a,b]) = \sum_{k=a}^{b} k L_k^B = \Delta([a,b]) + (a-1)\xi([a,b]) \tag{30}$$

Then, we can rewrite the total cost C:

$$\begin{aligned} C &= \sum_{k=1}^{n} (R_e L_k^B E(p_{j_k^B}) + k R_t L_k^B T(p_{j_k^B})) & (31) \\ &= \sum_{p \in \hat{P}} (R_e E(p)\xi(D_p) + R_t T(p)\gamma(D_p)) & (32) \end{aligned}$$

So if we can build a sorted data structure (sorted by L_k^B in descending order) with efficient insertion, deletion, and ξ, Δ queries, then we can do dynamic scheduling (dynamic tasks insertion and deletion) efficiently.

For any two nearby range $[L, M]$ and $[M+1, R]$ with ξ and Δ known, we can compute the $\xi([L,R])$ and $\Delta([L,R])$ as following. There is no need to know each L_k^B in $[L, R]$ (associative property):

$$\xi([L,R]) = \xi([L,M]) + \xi([M+1,R]) \tag{33}$$

$$\begin{aligned} \Delta([L,R]) = \; & \Delta([L,M]) + \Delta([M+1,R]) + \\ & (M+1-L)\xi([M+1,R]) \end{aligned} \tag{34}$$

A single 1D range tree will be a suitable data structure here. It supports insertion, deletion, and range queries for things with associative property in $O(\log N)$, where N is the number of tasks it contains. (1D range tree is simplest case of range tree, it's basically a balanced binary search tree, with each node keeps (1) the number of nodes, (2) ξ, (3) Δ, of its subtree to meet our requirements.) With this data structure, we can perform insertion and deletion in $O(\log N)$ and compute the total cost in $O(|\hat{P}| \log N)$ (via Equation 32).

This result is still improvable. We can keep (1) two pointers to the lower boundary node and the upper boundary node, (2) ξ, (3) Δ, for each dominating position ranges. Once an insertion or deletion occur, we can maintain these information in $O(|\hat{P}| + \log N)$ due to the fact that the number of different elements of the set of tasks before and after an insert/delete operation for each dominating position ranges is no more than two. So the time complexity of insertion and deletion increase to $O(|\hat{P}| + \log N)$ and the cost computation decrease to $\Theta(1)$ due to C is maintained in insertion or deletion. Notice that $\Theta(1)$ predecessor and successor operation are required to achieve this time complexity, we can do it by keep the predecessor and the successor pointer in each node like in doubly linked list. In addition, dynamic Δ range query is no longer required, While ξ range query (simple range sum) is still needed.

Algorithm 4 Initialize for Single-Core Dynamic Scheduling

Input: $\hat{P}, E, T, R_e, R_t$
Output: $D, Z, \alpha, \beta, a, b, x, d, C$
1: Find dominating position ranges D via Algorithm 1
2: Initialize an empty 1D range tree Z (sorted in descending order)
3: $C \leftarrow 0$
4: **for** $i \leftarrow 1$ to $|\hat{P}|$ **do**
5: $\quad \alpha_i \leftarrow$ NULL; $\beta_i \leftarrow$ NULL
6: $\quad a_i \leftarrow$ lowerbound($D_{\hat{p}_i}$); $b_i \leftarrow a_i - 1$
7: $\quad x_i \leftarrow 0$; $d_i \leftarrow 0$
8: **end for**

Algorithm 5 Insert a task

Input: L: the number of cycles of the incoming task
1: $ptr \leftarrow$ insert(Z, L)
2: $k^B \leftarrow$ rank(ptr)
3: $i \leftarrow \arg_i k^B \in D_{\hat{p}_i}$
4: **if** $k^B = a_i$ **then** $\alpha_i \leftarrow ptr$
5: **if** $k^B > b_i$ **then** $\beta_i \leftarrow ptr$
6: $b_i \leftarrow b_i + 1$
7: $x_i \leftarrow x_i + L$
8: $d_i \leftarrow d_i + (k^B - a_i + 1) \times *ptr +$ range_sum(Z,$[k^B+1, b_i]$)
9: **while** $b_i >$ upperbound($D_{\hat{p}_i}$) **do**
10: $\quad ptr \leftarrow \beta_i$
11: $\quad d_i \leftarrow d_i - (b_i - a_i + 1) \times *ptr$
12: $\quad x_i \leftarrow x_i - *ptr$
13: $\quad b_i \leftarrow b_i - 1$
14: $\quad \beta_i \leftarrow$ predecessor(β_i)
15: $\quad i \leftarrow i + 1$
16: $\quad \alpha_i \leftarrow ptr$
17: $\quad$ **if** $a_i > b_i$ **then** $\beta_i \leftarrow ptr$
18: $\quad b_i \leftarrow b_i + 1$
19: $\quad x_i \leftarrow x_i + *ptr$
20: $\quad d_i \leftarrow d_i + x_i$
21: **end while**
22: $C \leftarrow \sum_{i=1}^{|\hat{P}|} R_e E(D_{\hat{p}_i}) x_i + R_t T(D_{\hat{p}_i})(d_i + (a_i - 1)x_i)$

V. Evaluation

Our experimental platform is a quad-core x86 machine that supports individual core frequency tuning. The CPU model is Intel(R) Core(TM) i7 CPU 950 @ 3.07GHz. Each core has 12 frequency choices. The power consumption is measured by a power meter, model DW-6091. The energy consumption is the integral of the power reading over the execution period. Because other components in the system consume energy, we first measure the power consumption of an idle machine and deduct the idle power reading from our experiment results.

The frequencies of cores are computed according to our algorithm before each experiment. To prevent interference from the Linux OS frequency governor, we disable the automatic core frequency scaling of Linux. The DVFS mechanism can be disabled by setting the content in "/sys/devices/system/cpu/cpuX/cpufreq/scaling_governor" to *userspace*, where "X" is the CPU number. Since the

Algorithm 6 Delete a task

Input: ptr

1: $k^B \leftarrow$ rank(ptr)
2: let i be the maximum integer s.t. $a_i \leq b_i$
3: **while** $a_i > k$ **do**
4: $tptr \leftarrow \alpha_i$
5: $d_i \leftarrow d_i - x_i$
6: $x_i \leftarrow x_i - *tptr$
7: $b_i \leftarrow b_i - 1$
8: **if** $a_i \leq b_i$ **then**
9: $\alpha_i \leftarrow$ successor(α_i)
10: **else**
11: $\alpha_i \leftarrow$ NULL
12: $\beta_i \leftarrow$ NULL
13: **end if**
14: $i \leftarrow i - 1$
15: $\beta_i \leftarrow tptr$
16: $b_i \leftarrow b_i + 1$
17: $x_i \leftarrow x_i + *tptr$
18: $d_i \leftarrow d_i + (b_i - a_i + 1) \times *tptr$
19: **end while**
20: $d_i \leftarrow d_i - (k^B - a_i + 1) \times *ptr +$ range_sum(Z,$[k^B+1, b_i]$)
21: $x_i \leftarrow x_i - *ptr$
22: $b_i \leftarrow b_i - 1$
23: **if** $a_i > b_i$ **then**
24: $\alpha_i \leftarrow$ NULL
25: $\beta_i \leftarrow$ NULL
26: **else if** $\alpha_i = ptr$ **then**
27: $\alpha_i \leftarrow$ successor(α_i)
28: **else if** $\beta_i = ptr$ **then**
29: $\beta_i \leftarrow$ predecessor(β_i)
30: **end if**
31: delete(ptr)
32: $C \leftarrow \sum_{i=1}^{|\hat{P}|} R_e E(D_{\hat{p}_i}) x_i + R_t T(D_{\hat{p}_i})(d_i + (a_i - 1)x_i)$

DVFS mechanism is invalid, we can set the frequency of an individual core by changing the content in “/sys/devices/system/cpu/cpuX/cpufreq/scaling_setspeed”. However, the frequency choices are limited to those in “/sys/devices/system/cpu/cpuX/cpufreq/scaling_available_frequencies”. After setting the frequency of core “X”, we can verify the change from “/sys/devices/system/cpu/cpuX/cpufreq/scaling_cur_freq”.

The power consumption of a core is related to its frequency and voltage. In our experiments, we assume that each frequency has a corresponding voltage. By adjusting the frequency, a core will automatically change its operating voltage, thus results in different power consumptions.

A. Experiment Results for the Batch Mode

In this sub-section, we first verify the accuracy of the energy and cost models. Then, we evaluate the effectiveness of the *Workload Based Greedy* algorithm by comparing it with two existing algorithms.

TABLE I
AVERAGE EXECUTION TIMES OF THE WORKLOADS (SECONDS)

Benchmark	train input	ref. input
perlbench	43.516	749.624
bzip	98.683	1297.587
gcc	1.63	552.611
mcf	17.568	397.782
gobmk	189.218	993.54
hmmer	109.44	1106.88
sjeng	224.398	1074.126
libquantum	5.146	1092.185
h264ref	218.285	1549.734
omnetpp	108.661	439.393
astar	191.073	880.951
xalancbmk	142.344	453.463

TABLE II
PARAMETERS IN BATCH MODE

p_k	1.6	2.0	2.4	2.8	3.0
$E(p_k)$	3.375	4.22	5.0	6.0	7.1
$T(p_k)$	0.625	0.5	0.42	0.36	0.33

1) Experiment Settings: In the *batch* mode, we use *SPEC2006int*, which contains 12 benchmarks, each with *train* and *ref* inputs. As a result, we have 24 different workloads. The computation cycles needed by each workload are computed as follows. First, we measure the execution time by running each workload ten times on a core with the lowest frequency (1.6 GHz) and compute the average execution time. Then, we estimate the cycles needed by multiplying the average execution time by the core frequency, which is the number of cycles per second. Table I shows the average execution times of the workloads.

Table II shows the parameters used in the *batch* mode. Recall that $E(p_k)$ is the amount of energy in joules required to execute one cycle on a core with processing rate p_k; and $T(p_k)$ is time in seconds needed to execute one cycle with processing rate p_k. To obtain the values of $E(p_k)$, we measure the power consumption of a core with 100% loading using different p_k, and divide the result by p_k. Note that $T(p_k)$ is equal to $1/p_k$.

We set R_e and R_t accordingly. R_e is the amount paid for a joule of energy, and R_t is the amount paid to a user for every second he/she has to wait for a task's execution.

2) Model Verification: To evaluate the accuracy of the energy model and cost model, we conduct simulations and experiments. We use two frequencies, 1.6 GHz and 3.0 GHz; R_e is set at 0.1 cent per joule and R_t is set at 0.4 cents per second. We take the 24 workloads in Table I as input tasks. Each workload is executed once. The *Workload Based Greedy* algorithm is used to generate an optimal scheduling plan, including execution order and frequencies, for the input workloads.

The simulator takes the average execution time of the tasks and the parameters in Table II as inputs, simulates the execution of the tasks based on the scheduling plan, and generates the energy cost, time cost, and total cost.

In the experiment, we execute the 24 workloads on the quad-core x86 machine according to the scheduling plan generated by the *Workload Based Greedy algorithm*, and collect the time and energy data with a power meter. Then, we convert the

collected data into the cost and compare it with the simulation result.

Figure 1 shows the comparison results between simulation and experiment. The actual cost of executing the workloads on the x86 machine is about 8% higher than the simulation result. There are two possible reasons. The first one is that even if workloads are running simultaneously on different cores, they can still affect each other, e.g., by competing for last-level cache or memory. Such resource contention may cause longer execution times, and result in higher energy consumption and overall cost. The second reason is that running different tasks require different resources such as cache or memory operations. Doubling the processing speed of a task does not guarantee exactly half of the execution time. However, we deem the 8% discrepancy acceptable because considering all the system-level overheads would make the scheduling problem too complicated to analyze.

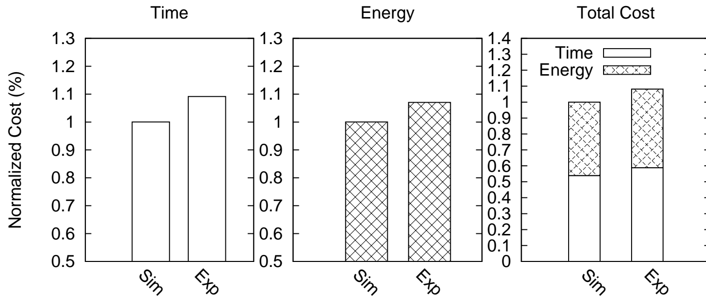

Fig. 1. Comparison of The Simulation And Experimental Results

3) Comparison with Other Scheduling Methods: We conducted experiments to compare the cost of our scheduling scheme with that of *Opportunistic Load Balancing* (OLB) [12] and *Power Saving*. OLB schedules a task on the core with the earliest ready-to-execute time. The main objective of OLB is to ensure the cores are fully utilized and finish the tasks in the shortest possible time. *Power Saving*, which restricts the frequency of a core to conserve energy, is widely used in the power-saving mode of mobile devices. For dynamic frequency scaling, we set the Linux frequency governor to "On-demand" for both OLB and *Power Saving*. If a core's loading is higher than 85%, the frequency governor increases the core's frequency to the largest available selection. On the other hand, if the loading is lower than the threshold, the frequency governor reduces the processing frequency by one level. The loading of a core is measured every second.

First, we generate the scheduling plans with *Workload Based Greedy*, *Opportunistic Load Balancing*, and *Power Saving*. Then, we execute the plans on the experimental platform and measure their costs. In this experiment, we limit the available frequencies in *Power Saving* to the lower half of the CPU frequency range, i.e., 1.6, 2.0, and 2.4 GHz. As in the previous section, we set R_e at 0.1 cent per joule and R_t at 0.4 cents per second.

Figure 2 shows the cost of the three scheduling plans. *Workload Based Greedy* consumes 46% less energy than *Opportunistic Load Balancing* with only a 4% slowdown in the execution time. The total cost reduction is about 27%. Compared with *Power Saving*, *Workload Based Greedy* consumes 27% less energy and improves the execution time by 13%. The main reason is that our Workload Based Greedy schedules the shortest tasks first with a high processing rate, thus reduce the waiting time of other tasks as little as possible. Also we apply slower processing rate to larger tasks, hence reduce the energy consumption.

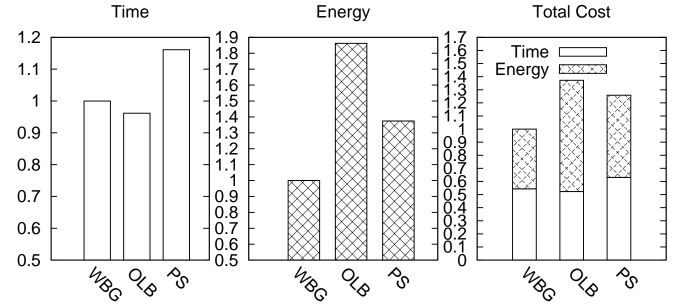

Fig. 2. Cost Comparison of Different Scheduling Methods

B. Experiment Results for the Online Mode

We conduct a trace-based simulation to verify the accuracy of the *Least Marginal Cost* algorithm. The experimental environment is the same as that in Section V-A. The workload is a piece of the trace from Judgegirl [13], which is an online judging system in National Taiwan University. Students submit their codes to the system in order to solve different problems. The length of the trace is half hours during the final exam, which includes five problems. We treat the score querying requests from students as *interactive* tasks, and the codes they submit as *non-interactive* tasks. There are 768 *non-interactive* tasks and 50525 *interactive* tasks in the trace.

In our trace-based simulation, the number of CPU cycles required of each task can be retrieved from the trace. In practice, we can estimate the resource requirement by profiling or historical data. The *interactive* tasks in an online judging system are mostly problem choosing and score querying. We can profile the CPU cycles required to complete these kinds of tasks while building the system. On the other hand, the resource requirement of a *non-interactive* task strongly depends on the code submitted by users. However, we can still predict the resource requirement of a newly arrival *non-interactive* task by taking average of the previous completed submissions.

We build an event-driven simulator. The simulator takes the workload trace as input. An event can be either a *task arrival* or a *task completion*. For *task arrival*, the simulator decides the core and frequency for the new task according to the scheduling algorithm. On the other hand, the simulator calculates the cost of time and energy of the task for a *task completion* event. The total cost is the cost summation of every tasks in the trace.

We compare the cost of the following scheduling strategies: *Opportunistic Load Balancing*, *On-demand*, and *Least Marginal Cost*. *Opportunistic Load Balancing*(OLB) [12] schedules a task on the core with the earliest ready-to-execute

time. The objective of OLB is to ensure the cores are fully utilized and finish the tasks in the shortest possible time. OLB keeps the processing frequency of each core at the highest level. *On-demand* [14] is a strategy in Linux that decides the processing frequency according to the current core loading. Once a core's loading reaches a predefined threshold, *On-demand* scales to the highest processing frequency of that core. On the other hand, if the loading is lower than threshold, *On-demand* reduces the processing frequency by one level. Since *On-demand* does not schedule tasks to core, we assign the arriving tasks to core in a round-robin fashion. In OLB and *On-demand*, *interactive* tasks have higher priority than *non-interactive* tasks. Tasks on a core with the same priority will be executed in a FIFO fashion. *Least Marginal Cost* is the strategies we propose. R_e and R_t are set to 0.4 cents per joule and 0.1 cent per second, respectively.

Figure 3 shows the cost comparison among scheduling methods. *Least Marginal Cost* method consumes 11% less energy and spends 31% less time than *Opportunistic Load Balancing*, and has 17% less total cost. Similarly, *Least Marginal Cost* method consumes 11% less energy, spends 46% less time than the *On-demand* method, and has 24% less total cost. The results indicate that *Least Marginal Cost* heuristic saves energy and reduces task waiting time than existing algorithms.

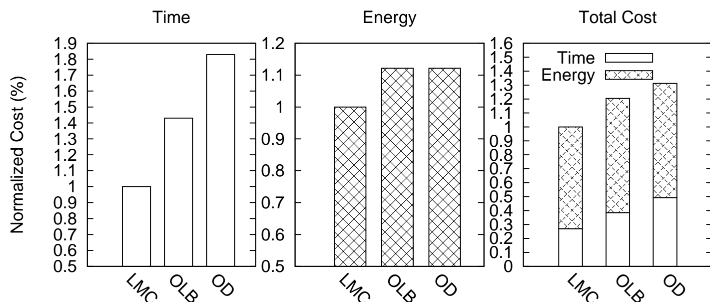

Fig. 3. Cost Comparison of Different Scheduling Mthods

VI. Related Work

Dynamic Voltage and Frequency Scaling (DVFS) is a key technique that reduces a CPU's power consumption. There have been several studies of using DVFS, especially for applications in real-time system domains. The objective is to ensure that such applications can be executed in real-time systems without violating their deadline requirements, while minimizing the energy consumption. Yao et al. [4] proposed an offline optimal algorithm as well as an online algorithm with a competitive ratio for aperiodic real-time applications. Pillai et al. [15] presented a class of novel real-time DVS (RT-DVS) algorithms, including Cycle-conserving RT-DVS and Look-Ahead RT-DVS. Aydin et al. [5] developed an efficient solution for periodic real-time tasks with (potentially) different power consumption characteristics. However, the above works only consider single-core real-time systems, so they do not deal with the assignment of tasks to cores.

A number of recent studies of DVFS scheduling in real-time systems focused on multi-core processors. Kim et al. [3] proposed a dynamic scheduling method that incorporates a DVFS scheme for a hard real-time heterogeneous multi-core environment. The objective is to complete as many tasks as possible while using the energy efficiently. Yang et al. [16] designed a 2.371 approximation algorithm that reduces the amount of energy used to process a set of real-time tasks that have common arrival times and deadlines on a chip multi-processor. Lee [17] introduced an energy-efficient heuristic that schedules real-time tasks on a lightly loaded multi-core platform. The above works focus on real-time systems in which tasks behave in a periodic or aperiodic manner. By contrast, the tasks considered in this paper are general computation tasks with or without deadlines.

In addition to real-time systems, some studies have investigated using DVFS for other purposes. To minimize energy consumption, Bansal et al. [18] performed a comprehensive analysis and proposed an online algorithm to determine the processing speed of tasks with deadlines on a processor that has arbitrary speeds. Pruhs et al. [19] investigated a problem setting where a fixed energy volume E is given and the goal is to minimize the total flow time of the tasks. They considered the offline scenario where all the tasks are known in advance and showed that optimal schedules can be computed in polynomial time. Albers et al. [10] developed an approach that uses a weighted combination of the energy and flow time costs as the objective function and exploits dynamic programming to minimize it in an offline fashion. This is similar to our approach, except that Albers et al. consider unit-size tasks, whereas our tasks can be any arbitrary size. Lam et al. [11] also studied scheduling to minimize the flow time and energy usage in a dynamic speed scaling model. They devised new speed scaling functions that depend on the number of active jobs, and proposed an online scheduling algorithm for batched jobs based on the new functions. The above works focus on single processors with discrete speeds. However, we consider both single and multi-core architectures in a per-core DVFS fashion.

Other energy-efficient algorithms have been proposed for multi-core platforms. Bunde [20] investigated flow time minimization in multi-processor environments with a fixed amount of energy. Aupy et al. [21] performed a comprehensive analysis of executing a task graph on a set of processors. The goal is to minimize the energy consumption while enforcing a prescribed bound on the execution time. They considered different task graphs and energy models. In contrast to our approach, their method assumes that the mapping of tasks to cores is given, which is different to our approach.

VII. Conclusion

In this paper, we propose effective energy-efficient scheduling algorithms for multi-core systems with DVFS features. We consider two task execution modes: the *batch* mode for batches of jobs; and the *online* mode for a more general execution scenario where *interactive* tasks with different time constraints or deadlines and *non-interactive* tasks may co-exist in the system. For each execution model, we propose scheduling algorithms and prove that they are effective both analytically

and empirically. The algorithms solve three problems simultaneously: the assignment of tasks to CPU cores, the execution order of tasks, and the CPU core frequency for executing each task.

For the *batch* mode, we prove that (1) the decision about the processing rate p_k used to minimize the cost $C(k, p_k)$ only depends on k, the position of the task in the execution sequence for a CPU core; and (2) the decision is independent of the execution workload of the task. We also show that there exists a polynomial-time optimal solution with the minimum cost in which the tasks are assigned in a greedy fashion in non-decreasing order of the number of cycles to the cores. Based on our theoretical findings, we propose a scheduling algorithm called *Workload Based Greedy*. For the *online* mode, we propose a heuristic called *Least Marginal Cost*, which assigns *interactive* and *non-interactive* tasks to cores. It also determines the processing speeds that will minimize the total cost of every time interval during a task's execution.

Our experiment results show that, for the *batch* mode, the *Workload Based Greedy* algorithm consumes 46% less energy than the *Opportunistic Load Balancing* algorithm, with only a 4% slowdown in the execution time. It also achieves a 27% improvement in energy consumption and a 13% improvement in the execution time over the widely used *Power Saving* method. For the *online* mode, the *Least Marginal Cost* algorithm yields a 17% and 24% improvement in the total cost compared with two existing algorithms in a trace-based simulation.

References

[1] P. Grosse, Y. Durand, and P. Feautrier, "Methods for power optimization in soc-based data flow systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, no. 3, pp. 38:1–38:20, Jun. 2009.

[2] S. Lee and T. Sakurai, "Run-time voltage hopping for low-power real-time systems," in *Proceedings of the 37th Annual Design Automation Conference*, ser. DAC '00. New York, NY, USA: ACM, 2000, pp. 806–809.

[3] S. I. Kim, H. T. Kim, G. S. Kang, and J.-K. Kim, "Using dvfs and task scheduling algorithms for a hard real-time heterogeneous multicore processor environment," in *Proceedings of the 2013 Workshop on Energy Efficient High Performance Parallel and Distributed Computing*, ser. EEHPDC '13. New York, NY, USA: ACM, 2013, pp. 23–30.

[4] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced cpu energy," in *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, ser. FOCS '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 374–.

[5] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez, "Determining optimal processor speeds for periodic real-time tasks with different power characteristics," in *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, ser. ECRTS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 225–.

[6] K. Choi, K. Dantu, W.-C. Cheng, and M. Pedram, "Frame-based dynamic voltage and frequency scaling for a mpeg decoder," in *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD '02. New York, NY, USA: ACM, 2002, pp. 732–737.

[7] Y.-M. Chang, P.-C. Hsiu, Y.-H. Chang, and C.-W. Chang, "A resource-driven dvfs scheme for smart handheld devices," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 3, pp. 53:1–53:22, Dec. 2013.

[8] C.-C. Lin, C.-J. Chang, Y.-C. Syu, J.-J. Wu, P. Liu, P.-W. Cheng, and W.-T. Hsu, "An engergy-efficient task scheduler for multicore platforms with per-core dvfs based on task characteristics," in *Proceedings of the 2014 IEEE International Conference on Parallel Processing*, ser. ICPP '14, 2014.

[9] J.-J. Chen, "Multiprocessor energy-efficient scheduling for real-time tasks with different power characteristics," in *Proceedings of the 2005 International Conference on Parallel Processing*, ser. ICPP '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 13–20.

[10] S. Albers and H. Fujiwara, "Energy-efficient algorithms for flow time minimization," *ACM Trans. Algorithms*, vol. 3, no. 4, Nov. 2007.

[11] T.-W. Lam, L.-K. Lee, I. K. To, and P. W. Wong, "Speed scaling functions for flow time scheduling based on active job count," in *Proceedings of the 16th Annual European Symposium on Algorithms*, ser. ESA '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 647–659.

[12] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *J. Parallel Distrib. Comput.*, vol. 61, no. 6, pp. 810–837, Jun. 2001.

[13] "Judgegirl," https://github.com/ntuparallellab/judgegirl.

[14] V. Pallipadi and A. Starikovskiy, "The ondemand governor: past, present and future," in *Proceedings of Linux Symposium, vol. 2, pp. 223-238*, 2006.

[15] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, ser. SOSP '01. New York, NY, USA: ACM, 2001, pp. 89–102.

[16] C.-Y. Yang, J.-J. Chen, and T.-W. Kuo, "An approximation algorithm for energy-efficient scheduling on a chip multiprocessor," in *Design, Automation and Test in Europe, 2005. Proceedings*, 2005, pp. 468–473 Vol. 1.

[17] W. Y. Lee, "Energy-saving dvfs scheduling of multiple periodic real-time tasks on multi-core processors," in *Distributed Simulation and Real Time Applications, 2009. DS-RT '09. 13th IEEE/ACM International Symposium on*, 2009, pp. 216–223.

[18] N. Bansal, T. Kimbrel, and K. Pruhs, "Speed scaling to manage energy and temperature," *J. ACM*, vol. 54, no. 1, pp. 3:1–3:39, Mar. 2007.

[19] S. Irani and K. R. Pruhs, "Algorithmic problems in power management," *SIGACT News*, vol. 36, no. 2, pp. 63–76, Jun. 2005.

[20] D. P. Bunde, "Power-aware scheduling for makespan and flow," in *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '06. New York, NY, USA: ACM, 2006, pp. 190–196.

[21] G. Aupy, A. Benoit, F. Dufossé, and Y. Robert, "Reclaiming the energy of a schedule: models and algorithms," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 11, pp. 1505–1523, 2013.