Energy-Efficient Computation Offloading for Multicore-Based Mobile Devices

Yeli Geng, Yi Yang and Guohong Cao
Department of Computer Science and Engineering
The Pennsylvania State University
E-mail: {yzg5086, yzy123, gcao}@cse.psu.edu

***Abstract*—Modern mobile devices are equipped with multicore-based processors, which introduce new challenges on computation offloading. With the big.LITTLE architecture, instead of only deciding locally or remotely running a task in the traditional architecture, we have to consider how to exploit the new architecture to minimize energy while satisfying application completion time constraints. In this paper, we address the problem of energy-efficient computation offloading on multicore-based mobile devices running multiple applications. We first formalize the problem as a mixed-integer nonlinear programming problem that is NP-hard, and then propose a novel heuristic algorithm to jointly solve the offloading decision and task scheduling problems. The basic idea is to prioritize tasks from different applications to make sure that both application time constraints and task-dependency requirements are satisfied. To find a better schedule while reducing the schedule searching overhead, we propose a critical path based solution which recursively checks the tasks and moves tasks to the right CPU cores to save energy. Simulation and experimental results show that our offloading algorithm can significantly reduce the energy consumption of mobile devices while satisfying the application completion time constraints.**

I. Introduction

Nowadays, computationally intensive applications such as image/video processing, interactive gaming, face recognition and augmented reality are becoming popular on mobile devices. While enjoying the benefits of these applications, users are also frustrated with the quickly draining battery since such complex applications typically consume more energy. To address this problem, many computation offloading techniques [1] have been proposed. Generally, a mobile application is partitioned into a set of tasks that can be executed either on the mobile devices or on the cloud. Computation offloading can reduce the energy consumption of mobile devices by selectively offloading computationally intensive tasks to the cloud.

Although computation offloading has been extensively studied, modern mobile devices equipped with multicore-based processors introduce new challenges. For example, in the ARM big.LITTLE architecture [2], high-performance and high-power big CPU cores are combined with energy-efficient but less-powerful little CPU cores, and thus there is a clear tradeoff between performance and energy for these two types of CPU cores. Existing research on computation offloading [3]–[5] only considers the tradeoff between energy and performance to determine whether the task should be executed locally or offloaded to the cloud. When new architectures (e.g., the big.LITTLE architecture) are introduced, we also need to consider the tradeoff of executing local tasks on which CPU cores.

In this paper, we address the problem of energy-efficient computation offloading on multicore-based mobile devices running multiple computationally intensive applications. To solve this problem, we have the following challenges: (1) The big.LITTLE architecture increases the complexity of the problem. Instead of only deciding locally or remotely running a task in the traditional architecture, we have to consider how to exploit the new architecture to minimize energy while satisfying application completion time constraints. (2) With multiple applications running simultaneously, we have to consider the time constraint of every application. Furthermore, for each application, the task-dependency requirements must be satisfied. (3) As this multicore-based computation offloading problem is NP-hard, our solution should find an energy-efficient schedule with low overhead.

We first formalize the multicore-based computation offloading problem which is NP-hard, and then propose a novel heuristic algorithm to jointly solve the offloading decision and task scheduling problems. We prioritize the tasks from different applications to make sure that both application time constraints and task-dependency requirements are satisfied. To find a better schedule while reducing the schedule searching overhead, we propose a critical path based solution which recursively checks the tasks and moves tasks to the right CPU cores to save energy. Finally, Dynamic Voltage and Frequency Scaling (DVFS) is applied to adjust the CPU frequency to further reduce energy. The main contributions of this paper are as follows.

- We consider the performance and energy tradeoffs of multicore-based computation offloading, and formalize the multicore-based computation offloading problem as a mixed-integer nonlinear programming problem, which is NP-hard.
- We propose a novel heuristic algorithm to minimize the energy consumption while satisfying the completion time constraints of multiple applications.
- We evaluate the proposed offloading algorithm with ex-

This work was supported in part by the National Science Foundation (NSF) under grants CNS-1421578 and CNS-1526425.

tensive simulations and real experiments on the Android phones. The results show that our algorithm can significantly reduce the energy consumption compared to other approaches.

The organization of the paper is as follows: In Section II, we briefly discuss related work. Section III gives the motivation. Section IV introduces the system model and the problem formulation. We present the proposed offloading algorithm in Section V. Section VI presents the evaluation results, and Section VII concludes the paper.

II. Related Work

Most existing research on computation offloading focuses on how to make offloading decisions. In [6], [7], the authors propose techniques to optimize the performance of specific mobile applications. In [3]–[5], techniques have been proposed to use offloading to save energy of the mobile devices. In [8]–[11], the authors propose offloading algorithms to consider both energy and performance (i.e., the application completion time). Other research [12], [13] coordinates the scheduling of offloading requests for multiple applications to to further reduce the wireless energy cost caused by the long tail problem [14], [15]. However, none of them considers how to schedule tasks on multiple local CPU cores for multiple applications to reduce energy or improve performance.

The ARM big.LITTLE architecture has attracted considered attention recently. In [16], [17], the authors demonstrate that the big.LITTLE system can save much power compared to systems with only high performance processors. A power management framework for ARM big.LITTLE architecture was introduced in [18] to minimize the energy consumption. Researchers [19], [20] have also proposed scheduling techniques to leverage the benefits of the big.LITTLE architecture. However, none of them studies how to leverage the big.LITTLE architecture in computation offloading.

Our work is also related to the task scheduling problem in distributed system. In [21], [22], algorithms have been proposed for scheduling tasks on heterogeneous processors to minimize the application completion time. There is a large amount of research on energy-aware task scheduling. For example, the authors in [23], [24] developed scheduling algorithms to reduce energy while guaranteeing hard deadlines for real-time periodic tasks. Lee and Zomaya [25] propose scheduling algorithms which balance the task completion time and energy consumption. However, these solutions cannot be directly applied to computation offloading, which has additional task-dependency requirements for tasks running on the cloud and local cores.

III. Background and Motivation

The ARM big.LITTLE architecture [2] has two kinds of CPU cores. Big cores are designed to provide the best computation performance while little cores are designed for maximum energy efficiency. With big.LITTLE architecture, each task can be dynamically allocated to a big core or little core based on the application requirement. The ARM big.LITTLE architecture is becoming increasingly popular on modern smartphones made by Samsung, LG, Huawei, etc. For example, the LG Nexus 5X smartphone uses Qualcomm Snapdragon 808 with quad-core Cortex-A53 (little) and dual-core Cortex-A57 (big). The little and big cores can operate in a frequency range from 384 MHz to 1.44 GHz, and from 384 MHz to 1.82 GHz, respectively.

To understand the difference between big and little cores in terms of performance and energy, we ran the same program on a little core and a big core of a Nexus 5X smartphone at different frequencies. The program is a loop of a fixed number of iterations which generates 100% CPU load. The CPU frequency is manually controlled by writing files in the */sys/devices/system/cpu/[cpu#]/cpufreq* virtual file system with root privilege. To specify the CPU core, the thread affinity is set by calling `__NR_sched_setaffinity` through the Java native interface (JNI) in Android.

Fig. 1 shows the measured results. The average power of the little core (554 mW) is only one fifth of the power of the big core (2468 mW) at their respective highest frequencies. Compared to the little core at the same CPU frequency, the big core reduces the execution time by one third (Fig. 1(b)), but it also doubles the total energy consumption (Fig. 1(c)). In summary, big cores are much more powerful than little cores, but consume significantly more energy. Therefore, we should carefully decide how to assign local tasks on big and little cores when making offloading decisions. Intuitively, tasks should be assigned to big cores for applications with strict time constraint, and assigned to little cores for applications without strict time constraint to save energy. From Fig. 1(c), we can also see that adjusting the frequency can further save energy.

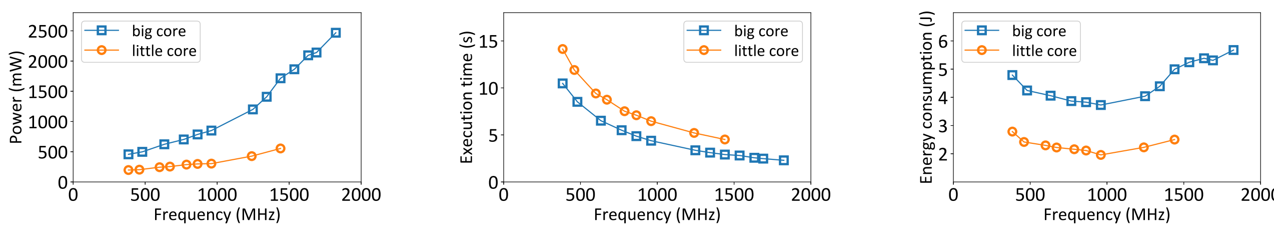
IV. System Model and Problem Formulation

A. Application Model

In our mobile cloud computing model, each application m has a completion time constraint T_m^{max}. A directed acyclic graph is used to represent application m, where each node $X_{m,i}$ represents a task i, and a directed edge $(X_{m,i}, X_{m,j})$ represents the task-dependency requirement that task $X_{m,i}$ should complete its execution before task $X_{m,j}$ starts. If task $X_{m,i}$ is offloaded to the cloud, $d_{m,i}$ and $d'_{m,i}$ denote the amount of data that are required to upload to, or be downloaded from the cloud, respectively. Fig. 2 shows the task graphs of two applications. In a given task graph, the task without any parent is called *entry task*, and the task without any child is called *exit task*. Note that tasks of the same application may have task-dependency requirements, but tasks from different applications are independent of each other.

B. Resource Model

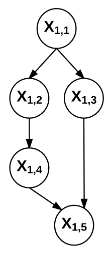
Different commercial implementations of big.LITTLE may have different numbers of big and little cores. Without loss of generality, we assume a mobile device has a total of K cores. The power levels and frequencies are different for big and little cores. Thus, we use f_k to denote the execution frequency of the kth core, and P_k to denote corresponding power level.



(a) Average power (b) Execution time (c) Energy consumption

Fig. 1. Average power, execution time and total energy consumption of a little core and a big core to run the same workload on Nexus 5X.

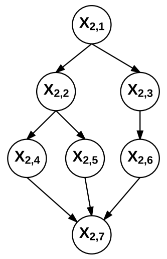
A task can be executed locally on one of the cores or remotely on the cloud. If task $X_{m,i}$ is offloaded to the cloud, there are three steps: sending the input data, executing the task in the cloud, and receiving the output data. We use r^{send} and r^{rcv} to denote the data sending and receiving rate of the wireless channels, respectively, and use P^{send} and P^{rcv} to denote the corresponding power levels.



Execution time

Task	Local		Cloud
	Big	Little	
$X_{1,1}$	4	5	5
$X_{1,2}$	3	4	5
$X_{1,3}$	8	10	5
$X_{1,4}$	7	9	5
$X_{1,5}$	4	6	5

(a) App 1



Execution time

Task	Local		Cloud
	Big	Little	
$X_{2,1}$	3	4	5
$X_{2,2}$	9	12	5
$X_{2,3}$	4	5	5
$X_{2,4}$	6	8	5
$X_{2,5}$	7	9	5
$X_{2,6}$	6	8	5
$X_{2,7}$	5	6	5

(b) App 2

Fig. 2. An example of task graphs for two applications.

C. Computation Model

For a task $X_{m,i}$, let $FT_{m,i}$ denote its finish time on a local core. Let $FT^{send}_{m,i}$, $FT^{exe}_{m,i}$, $FT^{rcv}_{m,i}$ denote its finish time on the wireless sending channel, the cloud, and the wireless receiving channel, respectively. If $X_{m,i}$ is assigned locally, $FT^{send}_{m,i} = FT^{exe}_{m,i} = FT^{rcv}_{m,i} = 0$; if the task is offloaded to the cloud, $FT_{m,i} = 0$.

1) Local Computing: If task $X_{m,i}$ is executed locally on the kth core, we use $T_{m,i,k}$ to denote the local execution time. $T_{m,i,k}$ depends on the computation workload of the task, denoted as $W_{m,i}$, and the operating frequency f_k of the kth core executing the task, which is $T_{m,i,k} = W_{m,i} \cdot f_k^{-1}$. Then, the energy consumption of the task is given by $E_{m,i,k} = P_k \cdot T_{m,i,k}$.

We can only schedule a task when it is ready (i.e., all immediate predecessors of the task must have been finished). Thus, when scheduled locally, the ready time of task $X_{m,i}$ (denoted as $RT_{m,i}$) is defined as follows:

$$RT_{m,i} = \max_{X_{m,j} \in pred(X_{m,i})} max\{FT_{m,j}, FT^{rcv}_{m,j}\}, \tag{1}$$

where $pred(X_{m,i})$ is the set of immediate predecessors of task $X_{m,i}$. If $X_{m,j}$ is scheduled locally, we have $RT_{m,i} \geq FT_{m,j}$, which means $X_{m,i}$ must wait util $X_{m,j}$ finishes its execution. If $X_{m,j}$ is offloaded to the cloud, we have $RT_{m,i} \geq FT^{rcv}_{m,j}$, which means $X_{m,i}$ must wait util $X_{m,j}$'s result is available locally.

Besides, we can only schedule a task to the kth core when the core is available. We use $AT_{m,i,k}$ to denote the earliest time when core k is available for task $X_{m,i}$. Then, the earliest start time of task $X_{m,i}$ on the kth core is given by:

$$ST_{m,i,k} = max\{RT_{m,i}, AT_{m,i,k}\}, \tag{2}$$

and the earliest finish time on the kth core is given by:

$$FT_{m,i,k} = ST_{m,i,k} + T_{m,i,k}. \tag{3}$$

After a task $X_{m,i}$ is scheduled on a local core, e.g., the kth core, the earliest finish time becomes the actual finish time, i.e., $FT_{m,i} = FT_{m,i,k}$.

2) Cloud Computing: If task $X_{m,i}$ is offloaded to the cloud, we can get the cloud execution time by $T^{exe}_{m,i} = W_{m,i} \cdot f_c^{-1}$, where f_c indicates the operating frequency of the processing unit on the cloud. The time for sending and receiving data is given by $T^{send}_{m,i} = d_{m,i}/r^{send}$ and $T^{rcv}_{m,i} = d'_{m,i}/r^{rcv}$, respectively. Since executing the task on the cloud does not consume energy of the mobile device, the energy consumption to offload task $X_{m,i}$ is given by $E^c_{m,i} = P^{send} \cdot T^{send}_{m,i} + P^{rcv} \cdot T^{rcv}_{m,i}$.

We can only offload a task when it is ready (i.e., all immediate predecessors of the task that are scheduled locally must have been finished). Thus,the ready time of task $X_{m,i}$ on the wireless sending channel (denoted as $RT^{send}_{m,i}$) is defined as follows:

$$RT^{send}_{m,i} = \max_{X_{m,j} \in pred(X_{m,i})} FT_{m,j}. \tag{4}$$

We can only send the input data of $X_{m,i}$ when the sending channel is available. Let $AT^{send}_{m,i}$ denote the time when the sending channel is available. The earliest start time to send input data of task $X_{m,i}$ is given by:

$$ST^{send}_{m,i} = max\{RT^{send}_{m,i}, AT^{send}_{m,i}\}, \tag{5}$$

and the earliest finish time to send input data is given by:

$$FT^{send}_{m,i} = ST^{send}_{m,i} + T^{send}_{m,i}. \tag{6}$$

After the input data is sent to the cloud, the cloud will execute task $X_{m,i}$ if all immediate predecessors of the task that are scheduled on the cloud have been finished. Thus, the ready time of task $X_{m,i}$ on the cloud (denoted as $RT^{exe}_{m,i}$) is defined as follows:

$$RT^{exe}_{m,i} = max\{FT^{send}_{m,i}, \max_{X_{m,j} \in pred(X_{m,i})} FT^{exe}_{m,j}\}. \tag{7}$$

Due to the high processing capability of the cloud, concurrent tasks can be executed in parallel. Thus, the earliest start time to execute task $X_{m,i}$ on the cloud is given by:

$$ST^{exe}_{m,i} = RT^{exe}_{m,i}, \tag{8}$$

and the earliest finish time is given by:

$$FT^{exe}_{m,i} = ST^{exe}_{m,i} + T^{exe}_{m,i}. \tag{9}$$

Since the size of the output data is small, we ignore the queueing time on the receiving channel similar to existing work [9], [12]. So the earliest finish time to receive the output data is given by:

$$FT^{rcv}_{m,i} = FT^{exe}_{m,i} + T^{rcv}_{m,i}. \tag{10}$$

After all tasks are scheduled on a core or cloud, the completion time of application m (denoted as T_m) can be calculated as:

$$T_m = \max_{X_{m,i} \in exit\ tasks} max\{FT_{m,i}, FT^{rcv}_{m,i}\}.$$

The total energy consumption of the mobile device for running m (denoted as E_m) is:

$$E_m = \sum_i E_{m,i},$$

where $E_{m,i}$ equals to $E_{m,i,k}$ if task $X_{m,i}$ is executed locally on the kth core, and equals to $E^c_{m,i}$ if the task is offloaded to the cloud.

D. Problem Formulation

In this subsection, we formalize the multicore-based computation offloading problem. We first define a binary variable $l_{m,i,k}$ for task assignment:

$$l_{m,i,k} = \begin{cases} 1, & \text{if task } X_{m,i} \text{ is assigned to core } k; \\ 0, & \text{otherwise,} \end{cases} \tag{11}$$

where $k = 0$ represents an exception that the task is being offloaded to the cloud. Since each task is assigned to one core, we have the following constraint:

$$\sum_{k=0}^{K} l_{m,i,k} = 1. \tag{12}$$

For task scheduling, we define a binary variable $o_{i,j}$ to specify the scheduling order as follows:

$$o_{i,j} = \begin{cases} 1, & \text{if task } j \text{ is scheduled before task } i; \\ 0, & \text{otherwise,} \end{cases} \tag{13}$$

Based on this definition, if tasks i and j belong to the same application and task j is task i's transient predecessor, then $o_{i,j} = 1$.

Given these two variables, the available time of the kth core for task $X_{m,i}$ should satisfy the following constraint:

$$AT_{m,i,k} \geq l_{m,i,k} \cdot l_{n,j,k} \cdot o_{i,j} \cdot FT_{n,j}.\ for\ \forall X_{m,i}, X_{n,j} \tag{14}$$

That is, if task $X_{m,i}$ is scheduled on core k, all tasks that scheduled before $X_{m,i}$ on the same core should have completed execution.

Similarly, the available time of task $X_{m,i}$ on the wireless sending channel should satisfy the following constraint:

$$AT^{send}_{m,i} \geq l_{m,i,0} \cdot l_{n,j,0} \cdot o_{i,j} \cdot FT^{send}_{n,j}.\ for\ \forall X_{m,i}, X_{n,j} \tag{15}$$

That is, if task $X_{m,i}$ is offloaded to the cloud, all tasks that scheduled to offload before $X_{m,i}$ should have sent their data to the wireless channel.

The multicore-based computation offloading problem can be formulated as a mixed-integer nonlinear programming problem as follows:

$$\begin{aligned} \min \quad & \sum_m \sum_i (\sum_k l_{m,i,k} \cdot E_{m,i,k} + l_{m,i,0} \cdot E^c_{m,i}) \\ \text{subject to} \quad & T_m \leq T^{max}_m,\ for\ \forall m \\ & Eq.\ 1, 2, 4, 5, 7, 11-15. \end{aligned}$$

This problem is in general NP-hard [26]. Therefore, we propose heuristic based algorithm in the next section.

V. Multicore-based Computation Offloading

In this section, we propose a multicore-based computation offloading algorithm, which has three phases. First, the initial scheduling generates a schedule with minimum application completion time. Then, we perform task reassignment on the initial scheduling to minimize the energy consumption without violating the completion time constraints. Last, we adjust the operating frequency to further reduce energy.

A. Initial Scheduling Phase

In the initial scheduling phase, we generate a schedule that minimizes the overall completion time for all applications. To guarantee that we can always find a schedule to satisfy the completion time constraints, energy is not considered in this phase; otherwise, we may not be able to satisfy the completion time constraints. Since the goal is to minimize completion time, the high-performance big cores are preferred over little cores. Intuitively, tasks from applications with strict time constraints should have higher priority to be scheduled. Thus, we first introduce how to prioritize tasks based on their application completion time constraints, and then describe how to schedule tasks based on their priorities. Before introducing the details, we define some notations.

1) Notations: The *latest allowable completion time* $(LACT)$ of task $X_{m,i}$ is recursively defined as:

$$LACT_{m,i} = \min_{X_{m,j} \in succ(X_{m,i})} (LACT_{m,j} - T^{min}_{m,j}), \tag{16}$$

where $succ(X_{m,i})$ is the set of immediate successors of task $X_{m,i}$. $T^{min}_{m,j}$ is the minimum execution time of task $X_{m,j}$, that is,

$$T^{min}_{m,j} = min(\min_{1 \leq k \leq K} T_{m,j,k}, T^{send}_{m,j} + T^{exe}_{m,j} + T^{rcv}_{m,j}), \tag{17}$$

where the inner min block is the minimum execution time on K local cores, and the second term is the time for offloading. The $LACT$ of the exit task is T^{max}_m.

Basically, $LACT$ is the tight deadline for a task. If a task cannot complete execution before its $LACT$, it is impossible for the application to satisfy its completion time constraint.

The *latest allowable start time* $(LAST)$ of task $X_{m,i}$ is derived from its $LACT$ as follows:

$$LAST_{m,i} = LACT_{m,i} - T^{min}_{m,i}. \tag{18}$$

It is used to measure the urgency of a task. The task with earlier $LAST$ should start earlier.

If a task finishes exactly at its $LACT$, all remaining tasks will have very tight schedule. Therefore, we define the $LACT'$ of a task $X_{m,i}$ as

$$LACT'_{m,i} = \min_{X_{m,j} \in succ(X_{m,i})} (LACT_{m,j} - T^{max}_{m,j}),$$

where $T^{max}_{m,j}$ is the max execution time of task $X_{m,j}$, which can be calculated by replacing min with max in Eq. 17. The $LACT'$ of the exit task is also T^{max}_{m}.

Compared to $LACT$, $LACT'$ is more conservative. For any task, if all of its immediate predecessors finish before their $LACT'$, the task can always be completed before its $LACT$.

2) Task Priority: Intuitively, a task which is more urgent should have higher priority to be scheduled, and hence $LAST$ is used as the metric of priority. That is, a task with earlier $LAST$ has higher priority to be scheduled. We have the following lemma.

Lemma 1: The increasing order of $LAST$ provides a topological ordering of tasks in the task graph.

Proof: First, we substitute Eq. 16 with Eq. 18 and get:

$$LACT_{m,i} = \min_{X_{m,j} \in succ(X_{m,i})} LAST_{m,j}.$$

By substituting Eq. 18 with the above equation, we have:

$$LAST_{m,i} = \min_{X_{m,j} \in succ(X_{m,i})} LAST_{m,j} - T^{min}_{i}.$$

Thus, $LAST_{m,i} < LAST_{m,j}$, where task $X_{m,i}$ is an immediate predecessor of task $X_{m,j}$. In this way, we have proven that for every directed edge ($X_{m,i}$, $X_{m,j}$) in the task graph of application m, $X_{m,i}$ comes before $X_{m,j}$. ■

With Lemma 1, scheduling tasks based on their priorities can satisfy the task-dependency requirements of all applications.

3) Task Scheduling: The basic idea is to select the highest priority task and assign it to the cloud or a local core that minimizes its execution finish time.

The input of the task scheduling is a priority queue generated in the previous section. If there is only one application, the tasks can be scheduled based on their order in the priority queue. When there are more than one application, a task with lower priority may be ready (i.e., all its immediate predecessors are scheduled) earlier than a task with higher priority from other applications. In this case, we prefer to schedule the ready task first rather than following the order of the priority queue. Therefore, we need an auxiliary waiting list to store tasks that have higher priority than the current task but are not ready for scheduling at the moment.

The task scheduling works as follows:

- Select the task $X_{m,i}$ with the highest priority and calculate its ready time. If $X_{m,i}$ is not ready, it is added to the waiting list. If $X_{m,i}$ is ready, we calculate its finish time on the kth core, $FT_{m,i,k}$, and its finish time when offloaded, $FT^{rcv}_{m,i}$, using Eq. 3 and Eq. 10, respectively. Then, the minimum is chosen as the finish time of $X_{m,i}$.
- Now, we attempt to schedule $X_{m,i}$ to achieve the minimum finish time, which may not be successful if it causes any task in the waiting list to miss its completion time constraint. When a task $X_{m,i}$ is scheduled, we calculate the finish time of every task in the waiting list assuming $X_{m,i}$ is actually scheduled. If a task $X_{n,j}$ cannot finish before $LACT'_{n,j}$, $X_{m,i}$ will not be scheduled, instead, $X_{n,j}$ is scheduled first.
- After a task $X_{m,i}$ is actually scheduled, we delete it from the priority queue, and go back to the first step until all tasks are scheduled.

Algorithm 1: Initial scheduling phase

```
Data: Task graphs, a priority queue $PQ$ of all tasks
Result: Schedule sequence $\mathcal{S}$
Initialization:
Schedule sequence $\mathcal{S} \leftarrow \emptyset$ ;
Waiting list $waitList \leftarrow \emptyset$ ;
Fuction MainScheduleFunction
    Current selected task $X_{m,i} \leftarrow PQ.head$ ;
    while $PQ \neq \emptyset$ do
        compute $RT_{m,i} \leftarrow$ Eq. (1), $RT^{send}_{m,i} \leftarrow$ Eq. (4) ;
        if $X_{m,i}$ is not ready then
            add $X_{m,i}$ into $waitList$ ;
        else
            AttemptToSchedule ($X_{m,i}$) ;
        update $X_{m,i}$ ;
Fuction AttemptToSchedule($X_{m,i}$)
    GetTaskFinishTime ($X_{m,i}$) ;
    if $FT^{rcv}_{m,i} < T_{m,i,k}, for \ \forall \ k$ then
        candidate resource $c_k \leftarrow$ cloud ;
    else
        candidate resource $c_k \leftarrow \min_{1 \leq k \leq K} T_{m,i,k}$ ;
    if $waitList = \emptyset$ or $X_{m,i}$ is the first task in $waitList$ then
        ScheduleTask ($X_{m,i}, c_k$) ;
    for each task $X_{n,j}$ on $waitList$ do
        GetTaskFinishTime ($X_{n,j}$) ;
        if $\forall \ k, FT_{n,j,k} > LACT'_{n,j}$ and $FT^{rcv}_{n,j} > LACT'_{n,j}$ then
            AttemptToSchedule ($X_{n,j}$) ;
    ScheduleTask ($X_{m,i}, c_k$) ;
Fuction GetTaskFinishTime($X_{m,i}$)
    for $k = 1$ to $K$ do
        compute $ST_{m,i,k} \leftarrow$ Eq. (2), $FT_{m,i,k} \leftarrow$ Eq. (3) ;
    compute $ST^{send}_{m,i} \leftarrow$ Eq. (5) ;
    compute $FT^{send}_{m,i}, RT^{exe}_{m,i}, ST^{exe}_{m,i}, FT^{exe}_{m,i} \leftarrow$ Eq.(6)-(9);
    compute $FT^{rcv}_{m,i} \leftarrow$ Eq. (10) ;
    return $FT_{m,i,k}$, $FT^{rcv}_{m,i}$ ;
Fuction ScheduleTask($X_{m,i}$, $c_k$)
    schedule $X_{m,i}$ onto the resource $c_k$ ;
    update $\mathcal{S}$ ;
    delete $X_{m,i}$ from $PQ$ ;
```

The formal description of the initial scheduling phase is shown in Algorithm 1. As an example, we perform the initial scheduling on the task graphs shown in Fig. 2. Since the execution time on the cloud is very short in this example, we set $T^{exe}_{m,i} = 1$, $T^{send}_{m,i} = 3$ and $T^{rcv}_{m,i} = 1$ for all tasks for simplicity. The completion time constraints of the two applications are $T^{max}_{1} = 20$s and $T^{max}_{2} = 30$s. The power level of big/little cores and the wireless channels is set using the values measured in Fig. 1(a) and Table. I. For simplicity, we only use one big core and two little cores in this example.

Fig. 3 presents the initial scheduling. Based on the $LAST$ value of the tasks, the priority queue is ($X_{1,1}$, $X_{1,2}$, $X_{1,3}$, $X_{1,4}$, $X_{2,1}$, $X_{2,2}$, $X_{1,5}$, $X_{2.3}$, $X_{2,4}$, $X_{2,5}$, $X_{2,6}$, $X_{2,7}$). For example, after task $X_{1,1}$ is scheduled to the big core, $X_{1,2}$

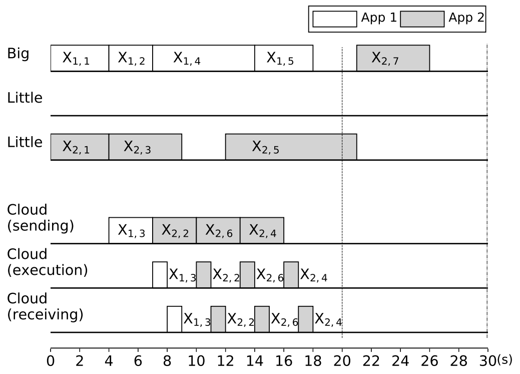


Fig. 3. The schedule generated by the initial scheduling phase when the completion time constraints of two applications are $T_1^{max}=20$s, $T_2^{max}=30$s. (total energy $E=92J$, completion time $T_1=18s$, $T_2=26s$)

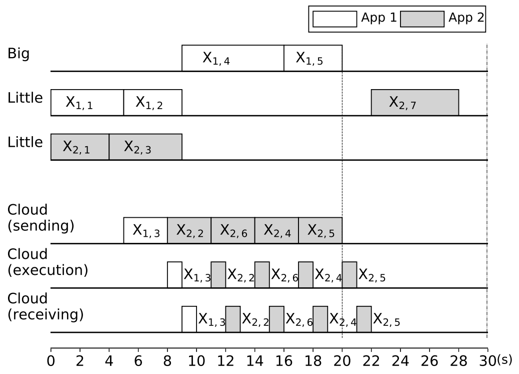


Fig. 4. The schedule generated by the task reassignment phase when the completion time constraints of two applications are $T_1^{max}=20$s, $T_2^{max}=30$s. (total energy E=68J, completion time $T_1=20s$, $T_2=28s$)

and $X_{1,3}$ have to wait until $X_{1,1}$ finishes. However, $X_{2,1}$ is ready and can be scheduled to the little cores directly. Since the goal is performance, big core is preferred over little core. When the big core is busy, some tasks are scheduled to little cores to satisfy the completion time constraints.

B. Task Reassignment Phase

In the second phase, tasks from the initial schedule are reassigned to save energy. The basic idea is to move tasks from high-power big cores to energy-efficient little cores under the completion time constraints. For a given schedule, a task can be moved to another CPU core or to the cloud. Since there are many tasks and cores, there are many reassignment choices, and will generate many new schedules. Among these new schedules, the one with the lowest energy consumption is chosen. For this chosen schedule, another task reassignment may still change the energy consumption. Thus, the entire process is repeated until no better schedule is found.

To improve the efficiency of searching among different choices for task reassignment, we introduce the idea of *critical path*. For a path p from the entry task to the exit task in application m's task graph, the slack time of p is defined as

$$slack_p = T_m^{max} - \sum_{X_{m,i}\in p} T_{m,i}, \tag{19}$$

where T_m^{max} is the completion time constraint, and $T_{m,i}$ is the execution time of task $X_{m,i}$ in a schedule. The critical path is the path with the highest $slack_p$. When a task is reassigned, its execution time may be increased and violate the completion time constraint. Since the critical path has the most free time, tasks on the critical path should be considered first for reassignment.

Given a schedule, we identify the critical path p first. Assume there are N tasks on p, each of them can be moved to $K-1$ other local cores or to the cloud. We have a total of $N\times K$ reassignment choices. Then, for each reassignment, the given schedule is adjusted to get a new schedule.

For the new schedule, the energy consumption is $E=\sum_m E_m$, and the difference between the application completion time constraint and the actual completion time is: $T=\sum_m (T_m^{max}-T_m)$. Among all the new schedules, the one with the largest $\Delta E/\Delta T$ will be chosen. Note that the schedules that violate the application time constraints will not be considered.

Then, we use the chosen schedule to update the given schedule and repeat the previous steps until the energy consumption cannot be further reduced under the completion time constraints.

Fig. 4 shows how task reassignment works. Comparing with the initial schedule (Fig. 3), $X_{1,1}$, $X_{1,2}$ and $X_{2,7}$ are reassigned from big core to little core, and $X_{2,5}$ is reassigned from little core to cloud to save energy. Note that to save energy, App 1 delays its completion time from 18s to 20s which meets the time constraint exactly, and App 2 delays its completion time from 26s to 28s. App 1 has a strict time constraint, and its tasks $X_{1,4}$ and $X_{1,5}$ have to run on the big core. For App 2, energy-efficient little cores are used.

C. Frequency Scaling Phase

In the previous phases, the maximum operating frequency is used for each core. For modern smartphones, the CPU frequency and the voltage can be adjusted at run-time, which is called the Dynamic Voltage and Frequency Scaling (DVFS). In this phase, DVFS is applied to adjust the CPU frequency to further reduce energy.

We employ a similar approach as the task reassignment phase. Instead of searching among different choices for task reassignment, different choices for frequency scaling are searched to find a schedule with the lowest energy consumption. Assume each core can operate at Q different frequency levels. There are many adjustment choices, and the critical path is also used to improve the efficiency.

Given a schedule, we identify the critical path p first. Assume there are N tasks on p, each of them can use one of the other $Q-1$ frequencies. We have a total of $N\times(Q-1)$ frequency scaling choices. Then, for each scaling choice, the given schedule is adjusted to get a new schedule. Among all the new schedules, the one with the largest $\Delta E/\Delta T$ will be chosen. Then, we use the chosen schedule to update the given schedule and repeat the previous steps util the energy consumption cannot be further reduced under the completion time constraints.

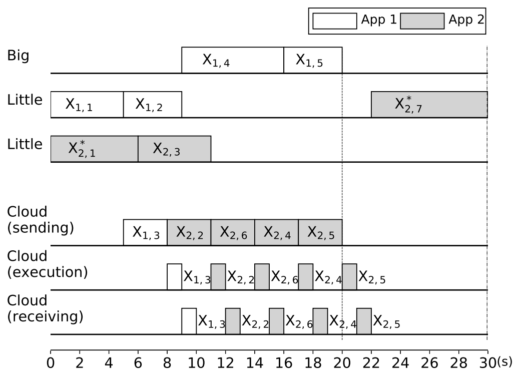


Fig. 5. The schedule generated by the frequency scaling phase when the completion time constraints of two applications are $T_1^{max} = 20$s, $T_2^{max} = 30$s. The frequency of the tasks marked with * is adjusted. (total energy E=66J, completion time $T_1 = 20s$, $T_2 = 30s$)

Fig. 5 shows how frequency scaling works. After the task reassignment shown in Fig. 4, App 1 meets its completion time constraint with no free time left, while App 2 still has 2s free time. Therefore, the frequency of task $X_{2,1}$ and $X_{2,7}$ can be further reduced to save energy. Comparing Fig. 5 with Fig. 4, the execution time of $X_{2,1}$ increases from 4s to 6s, and from 6s to 8s for $X_{2,7}$. We cannot further reduce the frequency of any task without violating the completion time constraints.

VI. Performance Evaluations

In this section, we evaluate the performance of the proposed multicore-based offloading algorithm. In the evaluation, we consider two sets of task graphs: randomly generated and real world applications. We first compare our algorithm with other approaches using randomly generated task graphs and then evaluate the performance of our algorithm using real world applications. Before presenting the evaluation results, we first present the evaluation setup.

A. Evaluation Setup

Our evaluation uses a Nexus 5X smartphone and a Dell desktop with Intel i7-3770@3.4GHz CPU and 16GB RAM as the cloud server. The smartphone is connected to the server via LTE. The power level under a given frequency of different CPU cores is shown in Fig. 1(a). The power level and the throughput of the wireless interface are shown in Table I.

To measure the power consumption, we modify the battery connection and use an external power monitor to provide power supply for the smartphone. Different from old phone models, modern smartphones like Nexus 5X have very tiny battery connectors, making it very challenging to connect a power monitor to them. To solve this problem, we design a battery interceptor based on Flex PCB, which is a very thin circuit board that uses corresponding battery connectors (Hirose BM22-4 for Nexus 5X) and a customized circuit to modify the battery connection. As shown in Fig. 6, we use this interceptor to connect the Nexus 5X smartphone with the Monsoon Power Monitor to measure the power consumption.

Since there is no existing multicore-based computation offloading algorithm, we modify some existing algorithms and

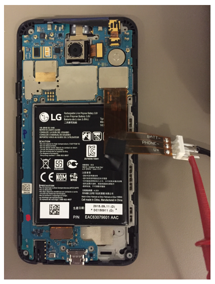


Fig. 6. Battery interceptor for the Nexus 5X smartphone to be connected with the Monsoon Power Monitor.

TABLE I
Power level and throughput of the wireless network

State	Power (mW)	Throughput (Mbps)
Sending	1936.4±51.6	0.98±0.3
Receiving	1598±33.8	15.2±3.9

make them applicable for multicore-based offloading. Specifically, we compare the proposed multicore-based computation offloading algorithm (denoted as "MCO") with the following approaches.

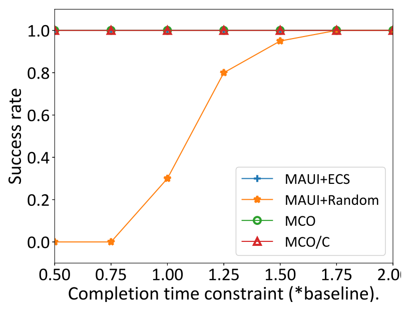
- **MAUI+Random**: it uses MAUI [3] to partition tasks into local tasks and cloud tasks, and then schedules local tasks to a randomly selected local core.
- **MAUI+ECS**: it uses MAUI to partition tasks into local tasks and cloud tasks, and then uses the algorithm in [25] to schedule tasks among multiple local cores.
- **MCO/C**: It is based on MCO. Instead of performing task reassignment and frequency scaling in separate phases, it determines the core and frequency at the same time.

The comparisons of the algorithms are based on the following metrics:

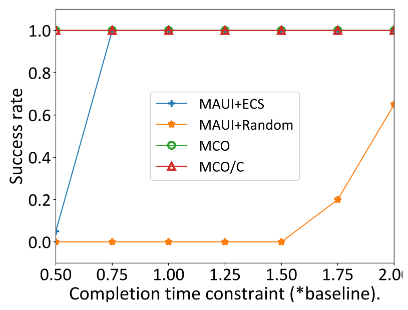
- **Success rate**, the percentage of schedules ensuring that all applications are completed within their specified completion time constraints.
- **Energy consumption**, the total amount of energy consumed by the mobile device.

B. Comparisons with Other Approaches

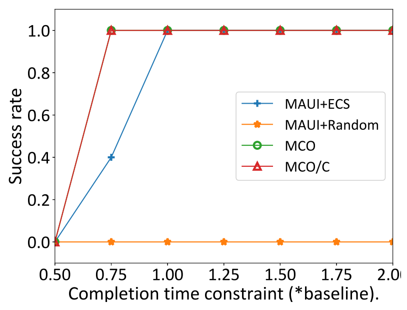
For the experiments in this section, the number of applications running simultaneously on a mobile device ranges from one to three. For each application, 10 different values (i.e., 5, 10, ..., 45, 50) are used as the task graph size. For each graph size, 100 task graphs are randomly generated using *TGFF* [27], which has been extensively used to generate random task graphs for scheduling and allocation research. The input and output data size of tasks in a task graph follows Gaussian distribution $\mathcal{N}(\mu_1, \sigma_1^2)$ (μ_1 = 200KB and σ_1 = 50) and $\mathcal{N}(\mu_2, \sigma_2^2)$ (μ_2 = 20KB and σ_2 = 5), respectively. The total amount of computation workload also follows the Gaussian distribution $\mathcal{N}(\mu_3, \sigma_3^2)$ (μ_3 = 2000M CPU cycles and σ_3= 200). The completion time constraint of an application depends on its task graph, whose default value is set to the sum of the execution time on a big core of all tasks.



(a) Number of apps = 1

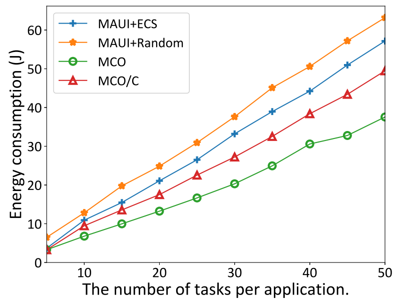


(b) Number of apps = 2

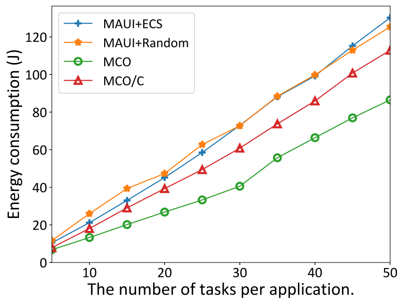


(c) Number of apps = 3

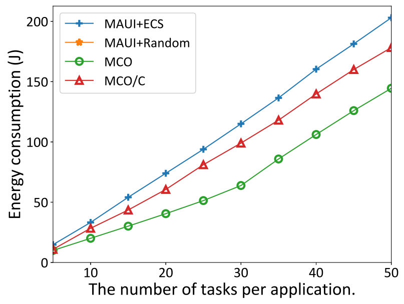
Fig. 7. Success rate of offloading algorithms with respect to different number of applications.



(a) Number of apps = 1



(b) Number of apps = 2



(c) Number of apps = 3

Fig. 8. Energy consumption of offloading algorithms with respect to different number of applications.

To evaluate the success rate of different algorithms, we change the completion time of an application by multiplying a factor (i.e., [0.5, 2.0]) to the default value. Fig. 7 shows the success rate with different numbers of applications when task graph size is 30. When only one application is running (Fig. 7(a)), the success rate reaches 100% for all approaches except MAUI+Random. This is because MAUI+Random randomly schedules tasks on local cores without considering application time constraints, resulting in the lowest success rate. As shown in Fig. 7(a), when the time constraint increases (i.e., the factor increases from 0.5 to 2.0), it is easier to find a schedule to meet all time constraints, and then the success rate of MAUI+Random also increases from 0 to 100%. Note that other algorithms schedule tasks considering the application time constraint, and thus have much higher success rate.

When the number of applications increases, multiple applications must compete for the computation resource to meet their respective time constraints, resulting in lower success rate. For the same completion time constraint, as shown in Fig. 7, when the number of applications increases from one (Fig. 7(a)) to three (Fig. 7(c)), the success rate of MAUI+Random decreases significantly. Similar trends can also be found in other algorithms, and our algorithm always has the highest success rate.

Fig. 8 shows the energy consumption when the number of tasks in each application changes from 5 to 50. Here, the application completion time constraint is set to be two times of the default value. As can be seen from the figure, MCO can significantly reduce the energy consumption compared to other approaches. For example, as shown in Fig. 8(a), with 50 tasks, the energy consumption is reduced by 34.3%, 40.6% and 24.1% comparing to MAUI+ECS, MAUI+Random, and MCO/C, respectively. MAUI+Random has the worst performance since it ignores the energy differences between big and little cores. Note that MAUI+Random is not shown in Fig. 8(c) because its success rate is 0 as explained in Fig. 7 (c).

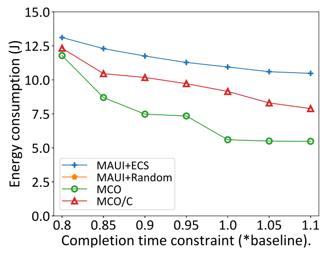
As for MAUI+ECS and MCO/C, they determine the CPU core and CPU frequency of each task at the same time which makes them less effective. This is due to the fact that moving tasks from higher frequency to lower frequency is less energy efficient than from big core to little core, as shown in Fig. 1(c).

C. Real World Applications

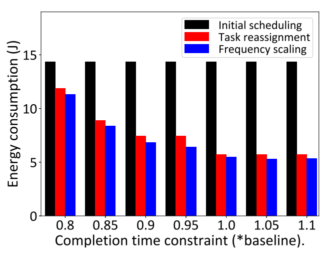
In addition to randomly generated task graphs, we also consider two real world applications: a face recognition application and a Fast Fourier Transformation (FFT) application. For the face recognition application, the input image is randomly selected from 100 images. For the FFT application, the input audios are generated by human voice ranging from 10s to 30s. The completion time constraint is set similar to previous section and the experiments are conducted 100 times with different input data.

Fig. 9(a) shows the energy consumption of different algorithms. Since MAUI+Random cannot find a schedule satisfying the application time constraints, its energy consumption is not shown here. Our algorithm tries to save energy while satisfying the application time constraints. When the application time constraint increases, our algorithm can save more energy. Compared with MAUI+ECS, MCO/C, our algorithm reduces the energy up to 48.9% and 38.8%, respectively.

Fig. 9(b) shows the energy consumption of our algorithm at difference phases. The initial scheduling phase finds the schedule with the minimum completion time, thus, it does not change with the completion time constraint. Our offloading



(a) Energy comparison



(b) Energy of different phases

Fig. 9. Effectiveness of multicore-based computation offloading.

algorithm mainly depends on the task reassignment phase to reduce the energy consumption. When the time constraint increases, more tasks can be moved to energy-efficient little cores, which can significantly reduce the energy consumption. For example, the energy consumption after the task assignment phase reduces from 12.1J to 6.2J when the time constraint increases from 0.8 to 1.1. When the time constraint is large enough that most tasks are already scheduled to little cores with low frequencies, there is little space to further reduce the energy consumption.

The energy reduction of running tasks from big core to little core is much larger than that from high frequency to low frequency on the same core, as shown in Fig. 1(c). This explains why the frequency scaling phase is less significant than the task reassignment phase on energy saving. Note that our algorithm completes after the frequency scaling phase, and hence the energy consumption of the frequency scaling phase in Fig. 9(b) matches that of MCO in Fig. 9(a).

VII. Conclusions

In this paper, we identified new research challenges in computation offloading introduced by the big.LITTLE architecture. Instead of only deciding locally or remotely running a task in the traditional architecture, we have to consider how to exploit the new architecture to minimize energy while satisfying application completion time constraints. We addressed the problem of energy-efficient computation offloading on multicore-based mobile devices running multiple applications. We first formalized the multicore-based computation offloading problem, and then proposed heuristic based solutions to jointly solve the offloading decision and task scheduling problems. To find a better task schedule to satisfy both application time constraints and task-dependency requirements, while reducing the schedule searching overhead, we proposed a critical path based solution which recursively checks the tasks and moves tasks to the right CPU cores to save energy. We have evaluated the proposed algorithm through simulations and real applications on Android. The simulation and experimental results demonstrate that our algorithm can significantly reduce the energy consumption of mobile devices while satisfying the application time constraints.

References

[1] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, "A Survey of Computation Offloading for Mobile Systems," *Mobile Networks and Applications*, vol. 18, no. 1, pp. 129–140, 2013.

[2] "ARM big.LITTLE," https://developer.arm.com/technologies/big-little.

[3] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making Smartphones Last Longer with Code Offload," in *ACM MobiSys*, 2010.

[4] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic Execution between Mobile Device and Cloud," in *ACM EuroSys*, 2011.

[5] Y. Geng, W. Hu, Y. Yang, W. Gao, and G. Cao, "Energy-Efficient Computation Offloading in Cellular Networks," in *IEEE ICNP*, 2015.

[6] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan, "A Framework for Partitioning and Execution of Data Stream Applications in Mobile Cloud Computing," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 4, pp. 23–32, 2013.

[7] L. Yang, J. Cao, H. Cheng, and Y. Ji, "Multi-User Computation Partitioning for Latency Sensitive Mobile Cloud Applications," *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2253–2266, 2015.

[8] W. Zhang, Y. Wen, and D. O. Wu, "Energy-Efficient Scheduling Policy for Collaborative Execution in Mobile Cloud Computing," in *IEEE INFOCOM*, 2013.

[9] S. Guo, B. Xiao, Y. Yang, and Y. Yang, "Energy-Efficient Dynamic Offloading and Resource Scheduling in Mobile Cloud Computing," in *IEEE INFOCOM*, 2016.

[10] M.-H. Chen, B. Liang, and M. Dong, "Joint Offloading and Resource Allocation for Computation and Communication in Mobile Cloud with Computing Access Point," in *IEEE INFOCOM*, 2017.

[11] X. Lin, Y. Wang, Q. Xie, and M. Pedram, "Energy and Performance-Aware Task Scheduling in a Mobile Cloud Computing Environment," in *IEEE CLOUD*, 2014.

[12] L. Xiang, S. Ye, Y. Feng, B. Li, and B. Li, "Ready, Set, Go: Coalesced Offloading from Mobile Devices to the Cloud," in *IEEE INFOCOM*, 2014.

[13] L. Tong and W. Gao, "Application-Aware Traffic Scheduling for Workload Offloading in Mobile Clouds," in *IEEE INFOCOM*, 2016.

[14] W. Hu and G. Cao, "Energy Optimization Through Traffic Aggregation in Wireless Networks," in *IEEE INFOCOM*, 2014.

[15] ——, "Quality-Aware Traffic Offloading in Wireless Networks," in *ACM MobiHoc*, 2014.

[16] ARM, "big.LITTLE Technology: The Future of Mobile," *ARM White paper*.

[17] E. L. Padoin, L. L. Pilla, M. Castro, F. Z. Boito, P. O. A. Navaux, and J.-F. Méhaut, "Performance/Energy Trade-Off in Scientific Computing: The Case of Arm big.LITTLE and Intel Sandy Bridge," *IET Computers & Digital Techniques*, vol. 9, no. 1, pp. 27–35, 2014.

[18] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, "Hierarchical Power Management for Asymmetric Multi-Core in Dark Silicon Era," in *IEEE DAC*, 2013.

[19] Y. Zhu and V. J. Reddi, "High-Performance and Energy-Efficient Mobile Web Browsing on Big/Little systems," in *IEEE HPCA*, 2013.

[20] D. H. Bui, Y. Liu, H. Kim, I. Shin, and F. Zhao, "Rethinking Energy-Performance Trade-Off in Mobile Web Page Loading," in *ACM MobiCom*, 2015.

[21] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing," *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002.

[22] D. Bozdag, U. Catalyurek, and F. Ozguner, "A Task Duplication Based Bottom-Up Scheduling Algorithm for Heterogeneous Environments," in *IEEE IPDPS*, 2006.

[23] J. Luo and N. Jha, "Static and Dynamic Variable Voltage Scheduling Algorithms for Real-Time Heterogeneous Distributed Embedded Systems," in *IEEE VLSID*, 2002.

[24] P. Rong and M. Pedram, "Power-Aware Scheduling and Dynamic Voltage Setting for Tasks Running on a Hard Real-Time System," in *IEEE ASP-DAC*, 2006.

[25] Y. C. Lee and A. Y. Zomaya, "Energy Conscious Scheduling for Distributed Computing Systems under Different Operating Conditions," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 8, pp. 1374–1381, 2011.

[26] P. Belotti, C. Kirches, S. Leyffer, J. Linderoth, J. Luedtke, and A. Mahajan, "Mixed-Integer Nonlinear Optimization," *Acta Numerica*, vol. 22, pp. 1–131, 2013.

[27] "Task Graphs For Free," http://ziyang.eecs.umich.edu/ dickrp/tgff/.