國立臺灣大學電機資訊學院資訊工程研究所
碩士論文
Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University
Master Thesis

# 在異質多核心平台的省電排程
# A Power Efficient Scheduler for Asymmetric Multi-core Platform

李翔昕
Hsiang-Hsin Li

指導教授：劉邦鋒博士
Advisor: Pangfeng Liu, Ph.D.
共同指導教授：吳真貞博士
Co-Advisor: Jan-Jan Wu, Ph.D.

中華民國 105 年 7 月
July, 2016

doi:10.6342/NTU201601864

# 國立臺灣大學碩士學位論文
# 口試委員會審定書

在異質多核心平台的省電排程

A Power Efficient Scheduler for Asymmetric Multi-core Platform

本論文係李翔昕君（學號 R03922112）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 105 年 7 月 29 日承下列考試委員審查通過及口試及格，特此證明

口試委員： 劉邦鋒
（指導教授）

[illegible]

吳真貞

系 主 任 趙坤茂

# 誌謝

感謝劉邦鋒老師與吳真貞老師，在這兩年的碩士生涯中傳授了我專業上的知識，也給予許多研究方面方法的指導。

感謝林敬棋學長在論文寫作上的幫忙。

感謝莊捷如一直在背後支持著我。

感謝實驗室的同學們陪我渡過這兩年的時間。

doi:10.6342/NTU201601864

# 摘要

在計算平台的最新發展趨勢從同質多核心架構轉移到異構和非對稱多核心架構。因此，新的非對稱多核心平台變成一個重要的議題。然而，大多數現有的排程器著重在如何區分合適的工作負載到那些節能的小核心和性能的大核心上。但是並沒有考慮在非對稱多核心上，以合適的核心頻率下分配工作。

在這篇論文中，在非對稱多核心下，我們提出一個對保證工作產量的省電排程器。我們的排程器不僅能決定每個核心的頻率和指派工作來達成減少電量消耗，而且能保證每個工作的產量。從實作結果中，與常見的完全公平排程器相比，我們提出的排程器省下了 29 % 的電量。

**關鍵字**　省電、節能、排程、非對稱多核心、動態電壓調節、動態時脈調節、保證工作產量

doi:10.6342/NTU201601864

# Abstract

A recent trend in computing platforms is moving from homogeneous multi-core architectures toward heterogeneous and asymmetric multi-core. Therefore, the design of new schedulers for asymmetric multi-core platform has become an important issue. However, most of the existing schedulers focus on how to distinguish workloads suitable for performance “big” cores from those for power-efficient “little” cores, without considering how to distribute tasks to asymmetric cores running at adjustable frequency.

In this paper, we propose an energy-efficient scheduler for throughput guaranteed tasks running on asymmetric multi-core platforms. The proposed scheduler not only determines the frequency of cores and task assignment in order to reduce energy consumption, but also schedules the tasks so that the throughput of all tasks are guaranteed. The implement results indicate that the proposed scheduler consumes 29% less energy than the conventional Compeletely Fair Scheduler with DVFS enabled.

**Keywords** Energy-efficient, Scheduling, Asymmetric multi-core, Throughput Guaranteed Tasks.

doi:10.6342/NTU201601864

# Contents

doi:10.6342/NTU201601864

doi:10.6342/NTU201601864

# List of Figures

doi:10.6342/NTU201601864

# List of Tables

doi:10.6342/NTU201601864

# List of Algorithms

doi:10.6342/NTU201601864

# Chapter 1

# Introduction

A recent trend in computing platform is moving from homogeneous multi-core architectures toward heterogeneous and asymmetric multi-core. An asymmetric multi-core platform consists of cores with the same Instruction Set Architecture but different computing capabilities and power characteristics. An asymmetric multi-core platform seeks to achieve both high performance and low power consumption. Several asymmetric multi-core CPUs, such as ARM big.LITTLE [9], are already commercially available. Asymmetric multi-core architecture has various applications in mobile devices [23], cloud computing [10], multimedia [21], and green data center [5].

The evolution of multi-core platforms affects their task scheduling algorithms. Scheduling goals differ between homogeneous and asymmetric multi-core. The scheduling objective on homogeneous multi-core is *load balancing*, i.e., distributing workload evenly to all cores. An even distribution of workload increases task throughput, minimizes task response times, and avoids overloading individual cores. On the other hand, the scheduling goal on asymmetric multi-core is to maximize *power efficiency* with modest performance sacrifice. Since computing capabilities and power characteristics differ among cores, asymmetric multi-core platforms require new scheduling strategies because traditional load-balancing scheduling strategies may lead to energy waste, due to not being aware of and unable to adapt to core asymmetry.

doi:10.6342/NTU201601864

The design of new schedulers for asymmetric multi-core platform is an important issue. There have been researches and designs, such as the In-Kernel Switcher (IKS) [19] and the Global Task Scheduler (GTS) [12] proposed by Linaro for this purpose. However, most of the existing schedulers focus on how to *distinguish* workloads suitable for performance “big” cores from those for power-efficient “little” cores. These schedulers then distribute workloads to their corresponding type of cores. The execution of workloads is still determined by the underlining scheduler, which may not take energy efficiency into consideration.

In this paper, we design an energy-efficient scheduler for asymmetric multi-core. Specifically, we focus on *throughput guaranteed tasks* such as stream computing applications [24] (e.g., analyzing market information from stock exchanges, processing environmental sensors data, etc.). A throughput guaranteed task must complete a certain amount of workload during every time period in order to meet its expected throughput. The expected throughput of a task may vary from one time period to the next. Our objective is to schedule a set of such tasks so that both the throughput of each task and energy efficiency are guaranteed in every time period.

Our energy-efficient scheduler for asymmetric multi-core performs the following four functions – *task classification*, *frequency selection*, *time assignment*, and *task scheduling*.

**Task classification** The scheduler classifies tasks into two categories – those suitable for big and little cores respectively.

**Frequency selection** The scheduler also selects frequency of each core.

**Time assignment** The scheduler then assigns the percentage of time each task should run on each core.

**Task scheduling** The scheduler ensures that tasks will only run on the cores they are assigned, and tasks will receive the percentage of CPU time they are granted on the cores they are assigned.

doi:10.6342/NTU201601864

**Contributions** The main contributions of this paper are as follows.

- We develop an energy-credit based scheduler that ensures that tasks will run on cores they are assigned, and tasks will receive the percentage of CPU time on the cores they are assigned.

- We implement our scheduler in Linux and evaluate its effectiveness. The experimental results indicate that our scheduler uses 29% less energy than the conventional *Completely Fair Scheduler* with DVFS enabled.

The remainder of this paper is organized as follows. Related works are presented in Chapter 2. In Chapter 3, we describe the components that are related to our work in the Linux. In Chapter 4, we introduce our energy-credit based scheduler. We describe the implementation of our scheduler in detail in Chapter 5. Experimental evidence will be given in Chapter 6. Finally, we summarize our results in Chapter 7.

doi:10.6342/NTU201601864

# Chapter 2

# Related Work

In this chapter, we review existing researches and designs that schedule tasks with consideration of energy-efficiency on asymmetric multi-core platform. We also describe the differences between our work and their works.

Dynamic Voltage and Frequency Scaling (DVFS) is a key technique that reduces CPU power consumption. There have been various studies using DVFS for energy conservation, especially for applications in real-time system domains [18,25]. The common objective of these works is to ensure that such applications can run in real-time systems without violating their deadlines, while minimizing the energy consumption. Yao et al. [25] proposed an optimal off-line algorithm and an online algorithm with a competitive ratio for aperiodic real-time applications. Pillai et al. [18] presented real-time DVFS algorithms that save significant energy while maintaining deadline guarantees for periodic real-time tasks. Saad et al. [22] proposed a software partitioning approach to reduce the energy consumption on heterogeneous embedded systems. The proposed approach aims to minimize the overall energy consumption while avoid deadline violations.

The aforementioned works focus on reducing the energy consumption while keeping the execution time under a threshold. Our scheduler focuses on throughput guaranteed tasks, which do *not* have deadlines. Also, these previous works deploy tasks to different types of cores, and do not consider the possibility of running tasks on cores at adjustable

doi:10.6342/NTU201601864

frequency. In contrast, our proposed scheduler determines the schedule of tasks and sets the frequency of each physical core.

A number of researches [1, 2, 11, 13] are designed for streaming task, especially for the multimedia playing. These works achieve considerable energy savings by leveraging the performance or power consumption characteristic of the specific multimedia applications running on symmetric multi-core platform. There are also researches for energy conservation on mobile platform [20, 26]. These works focus on allocating CPU resource according to resource usage patterns of applications. In contrast, our goal is to achieve energy efficiency while *maintaining the throughput* of tasks on the asymmetric multi-core platform.

Studies of performance and fairness of tasks running in a heterogeneous platform are also in literature. Chen et al. [6] proposed an adaptive Workload-Aware Task Scheduler (WATS) that consists of a history-based task allocator and a preference-based task scheduler. The scheduler uses the historical statistics collected during the execution of a parallel application to schedule jobs. Kwon et al. [15] proposed a hypervisor scheduler that characterizes the efficiency of each virtual core, and maps the virtual cores to the most area-efficient physical core. The scheduler considers performance fairness among virtual machines and performance scalability for changing availability of fast and slow cores. Vishal et al. [10] present HeteroVisor, a heterogeneity-aware hypervisor, that exploits resource heterogeneity to enhance the elasticity of cloud systems. They adding two different types of credits into the CPU credit scheduler of Xen – slow and fast, to achieve virtual core scaling while the workload increases. Kim et al. [14] revised the notion of virtual runtime in the completely fair scheduler(CFS) [17] and proposed their scheduling approach on performance-asymmetric multi-core architecture. They reduce the difference among the maximum virtual runtime of cores than the original CFS. These works focus on distributing workloads among big and little cores on an asymmetric multi-core platform in order to achieve better performance and fairness. On the other hand, our goal is to achieve energy-efficient while maintaining the throughput of tasks.

doi:10.6342/NTU201601864

To summarize, most of the existing scheduling works on asymmetric multi-core platform focus on distinguishing tasks suitable for performance big cores or power-efficient little cores. Our energy-credit based scheduler not only determines the operating frequency of each physical core, but also schedules tasks to physical cores for execution, so that the tasks can achieve their expected throughput while reducing the energy consumption.

doi:10.6342/NTU201601864

# Chapter 3

# Background

We now describe the background information in Linux that are related to our work. There are two key components in Linux scheduling – the *CPU scheduler* and the *power manager*.

## 3.1 CPU Scheduler

The Linux kernel is a multi-core multi-tasking operating system kernel, so the scheduler of the Linux kernel needs to select the task that will run next for each core. For this purpose the scheduler classifies different types of tasks into different scheduling classes. When the scheduler wants to select the next task to run, it will look for tasks in the non-empty class that has the *highest* scheduling priority. Therefore the Linux kernel maintains a *run queue* to store the tasks of *each* scheduling class on *each* core. The scheduler then uses these run queues to determine the next task to run.

Figure 3.1 illustrates the five scheduling classes in Linux. The Linux kernel assigns the priorities of the scheduling classes from high to low: *stop-task scheduling class*, *deadline scheduling class*, *real-time scheduling class*, *fair scheduling class*, and *idle scheduling class*.

**Stop-Task Scheduling Class** The stop-task has the highest priority in the system. A stop-

doi:10.6342/NTU201601864

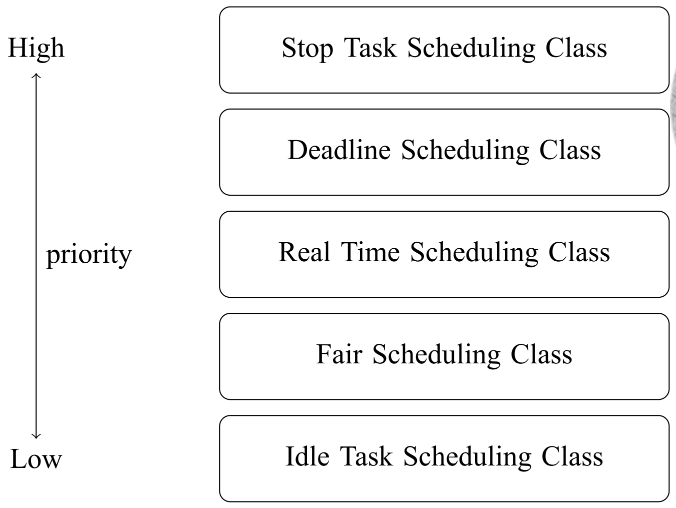

High
priority
Low
Stop Task Scheduling Class
Deadline Scheduling Class
Real Time Scheduling Class
Fair Scheduling Class
Idle Task Scheduling Class

Figure 3.1: Scheduling classes in Linux

task preempts *everything* and will be preempted by *nothing*. Stop-tasks include tasks that kill other tasks and tasks that seriously influence the stability of the system, so the scheduler should run the stop-tasks as soon as possible.

**Deadline Scheduling Class** Two schedulers run tasks in the deadline scheduling class. An *Earliest Deadline First* (EDF) scheduler runs periodic tasks and a *Constant Bandwidth Server* (CBS) runs aperiodic or sporadic tasks. The Earliest deadline first scheduler puts tasks into a priority queue, and a task has a higher priority if its deadline is closer. The goal of a constant bandwidth server is to ensure every task to meet its deadline, not influenced by other tasks, by slowing down aperiodic tasks or tasks trying to execute more than their reserved bandwidth.

**Real-Time Scheduling Class** Linux uses *First-In-First-Out* (FIFO) and *Round-Robin* (RR) policy to run tasks in the real-time scheduling class. The kernel sets a time quantum to to run tasks under these two policies. If a running task does not complete within this time quantum, the scheduler will preempt the task and the task will have to wait for the scheduler to schedule it to run later. The time quantum of the FIFO policy is infinite, so a task runs continuously on a core until it finishes. The time quantum of of the round-robin policy is the same for all tasks so they run in circular order.

doi:10.6342/NTU201601864

**Fair Scheduling Class** Linux uses a *Completely Fair Scheduler* (CFS) to allocate the *same portion* of the computing power to each task in the fair scheduling class. The design principle of CFS is to realize an ”ideal, precise multi-tasking CPU” on real hardware. To achieve this goal, CFS introduces the concept of *virtual runtime*. The virtual runtime of a task is actual runtime of the task multiplied by a *priority ratio*, which depends on the priority of the task. CFS maintains a red-black tree for each run queue and selects the task that has minimal virtual runtime to run next.

**Idle Scheduling Class** When there are no tasks in all the previous four scheduling classes, Linux selects an idle task to run. The priority of this idle task is the lowest, so it waits for any other coming task to preempt it. When a core is in the idle state, the power manager is responsible for power-saving.

A *scheduling domain* is a set of cores that have the same property. Each core is a basic scheduling domain, and the scheduling domains are hierarchical. A scheduling domain spans a set of cores that share the same scheduling policy, and the scheduler will balance the load of tasks among these cores. A tick timer runs periodically on each core and triggers load balancing by raising a software interrupt. The scheduler then balances the load by adjusting the run queues of the cores within the scheduling domain.

## 3.2 Power Manager

The power manager manages resources of the system by one of the low-power models – *system sleep model* and *runtime power management model*.

### 3.2.1 System Sleep Model

When *all* cores are idle, the power manager manages the system-wide resource with a *system sleep model*. The power manager uses the CPU hotplug framework to put all except one core to idle. This is because that core has to wait for a wakeup event that prompts the

doi:10.6342/NTU201601864

system to leave the low-power state. In addition, the power manager suspends memory and disk in system sleep model.

### 3.2.2 Runtime Power Management Model

The runtime power management model manages the resource of the cores dynamically by using a set of *governors*. There are five governors in the power manager, including *performance*, *powersave*, *userspace*, *on-demand*, *conservative*. Different governors have different power consumption characteristics, and they control power consumption by setting the frequency and voltage of CPU. The power manager keeps a table of CPU frequencies and their matching voltages. When a governor selects a CPU frequency, it will inform the power manager to switch CPU voltage to the matching voltage according to the frequency/voltage table. The manager uses *Dynamic voltage and frequency scaling* (DVFS) to change both frequency and voltage of the cores on the fly.

**Performance** The governor sets the CPU frequency to the *highest* frequency level.

**Power-save** The governor sets the CPU frequency to the *lowest* frequency level.

**Userspace** The governor allows users or userspace tasks to assign a specific frequency in the available frequency levels to the core.

**On-demand** The governor sets the CPU frequency according to the current CPU usage. If the CPU usage is more than a *up threshold*, the governor increases the CPU frequency to the *highest* frequency level. On the other hand, if the CPU usage is less than a *down threshold*, the governor decreases the CPU frequency by *one* frequency level.

**Conservative** The conservative governor also sets the CPU frequency according to on the current CPU usage. However, the governor tries to keep the CPU usage within a range. The governor increases the CPU frequency up *one* frequency level when

doi:10.6342/NTU201601864

the CPU usage exceeds the up threshold. On the other hand, the governor reduces the CPU frequency by one level when the CPU usage is below the down threshold.

doi:10.6342/NTU201601864

# Chapter 4

# Energy-credit Based Scheduler

A schedule in our system determines frequency of each core and allocates time percentage for each task to run on each core. The operating frequency of a core determines the number of CPU cycles it can provide in a unit of time. Therefore the amount of CPU cycles provided by a core is the product of the frequency and the length of time period. Also note that a task can *migrate* among cores, therefore a schedule must indicate the amount of time each task should run on each core.

We describe the operations of our *energy-credit based scheduler*, including *task classification*, *frequency selection*, *time assignment*, and *task scheduling*.

## 4.1 Task Classification

A throughput guaranteed task requires a *minimum* number of CPU cycles in order to meet the expected throughput in a time period. For ease of notation we will use *workload* to indicate this number of CPU cycles in this paper. Formally we denote the workload of a throughput guaranteed task $j$ by $w(j)$. In practice we can estimate the workload of a task by profiling.

Performance big core and power-efficient little core differ in architectures, so the amounts of work they can do per CPU cycle are different. As a result, the same task may cause different workloads on different cores. For example, a task on a big core may

doi:10.6342/NTU201601864

cause a smaller workload to meet its expected throughput than the workload on a little core. Therefore, we denote the ratio between the computing resources required running on little core and on big core to be $\alpha \geq 1$. That is, if a task $j$ requires workload $w(j)$ on a big core, then it requires $\alpha(j)w(j)$ on a little core. We also assume that a task will only run on big cores, or on little cores, but not both in a time period. This is due to the high overhead of switching between the big and little cores during execution.

Before we decide how to assign tasks to big and little cores, we need to observe their difference in performance and power consumption. We first compare the *dynamic power consumption per cycle*, between big and little cores. The dynamic power consumption is proportional to the product of the working voltage $V$ and the frequency $f$. In practice when we increase the operating frequency $f$, the working voltage $V$ will also increase, therefore the power consumption per cycle is an *increasing* function of the frequency $f$. We also observe that in the current design of big.Little core architectures, the dynamic power consumption per cycle of a big core is always *larger* than that of a little core under similar frequency [8]. Therefore, we prefer assigning tasks to little cores than to big cores in order to reduce power consumption. We only assign tasks to big cores under the following two conditions.

- The required number of CPU cycles in a unit of time of a task is larger than the highest available operating frequency of a little core. Recall that the operating frequency of a core determines the number of CPU cycles it can provide in a unit of time. If we assign the task to little cores, the schedule will not be able to provide enough number of CPU cycles.
- The little cores cannot provide sufficient number of CPU cycles to all tasks. If we assign all tasks to little cores, the tasks cannot receive enough number of CPU cycles to meet their expected throughput.

The scheduler determines which cores each task should be assigned to before generating schedules for the two types of cores. The scheduler classifies the tasks into two groups,

doi:10.6342/NTU201601864

$G_B$ for big cores and $G_L$ for little cores. Initially all tasks are in $G_L$. The scheduler first compares the required number of CPU cycles of each task with the highest operating frequency of a little core, and move tasks with requirement larger than the highest frequency to $G_B$.

After we classify the tasks, if the sum of the required CPU cycles in $G_L$ is still larger than the number of cycles little cores can provide, the scheduler moves tasks to $G_B$ according to their *cycle ratio* $\alpha$. The scheduler repeatedly moves the task with the highest $\alpha$ to $G_B$ until the required CPU cycles of the remaining tasks in $G_L$ can be satisfied by the little cores. The idea is that we have to move a fixed number of CPU cycles from $G_L$ to $G_B$ anyway. By choosing tasks with higher $\alpha$, we can decrease the number of CPU cycles being moved into $G_B$.

## 4.2 Frequency Selection

We make two assumptions on the available frequency set $F$ on each type of cores. It is easy to see that the two conditions are *necessary* to select a feasible frequency level.

- The highest available frequency of a core is no less than the number of CPU cycles required by the heaviest task assigned to that type of core. This is to ensure that a core can provide sufficient number of CPU cycles to all tasks assigned to it.

$$\max_{f \in F} f \geq \max_{j} \alpha(j) w(j), \;\; j \in G_L \tag{4.1}$$

$$\max_{f \in F} f \geq \max_{j} w(j), \;\; j \in G_B \tag{4.2}$$

- The cores can provide sufficient number of CPU cycles to all tasks assigned to this type of cores. That is, there must be a frequency fast enough to run all tasks on all

doi:10.6342/NTU201601864

cores if the workload is evenly distributed among all cores.

$$\max_{f \in F} f \geq \frac{\sum_{j=1}^{m} \alpha(j) w(j)}{m}, \; j \in G_L \tag{4.3}$$

$$\max_{f \in F} f \geq \frac{\sum_{j=1}^{m} w(j)}{m}, \; j \in G_B \tag{4.4}$$

Our scheduler repeatedly selects a frequency for a core, then assigns tasks to it. The process repeats on the remaining cores and tasks until all tasks are assigned to cores. We decide the frequency of cores one at a time. Suppose we now want to set the frequency for core $c$. We will select the *smallest* frequency $f$ for $c$ that satisfies all the following constraints.

1. $f$ must be large enough so that $c$ can accommodate the heaviest remaining task.

2. $f$ must be large enough so that all the remaining cores, if run at $f$, can accommodate all the remaining tasks.

3. If the scheduler has already assigned a part of a task $j$ to another core $c'$ in the previous task assignment step, and $j$ causes a CPU load of $x$ on core $c'$. A core can provide only up to 100% of its computing power and therefore we must set $f$ large enough so that the CPU load of task $j$ on core $c$ is no more than $1 - x$, since the total CPU load on all cores must be no more than 1.

## 4.3 Time Assignment

After we determine the frequency $f$ of a core $c$, we then assigned tasks to it. Let $r$ denote the available CPU cycles of core $c$, which is initialize to the frequency $f$ in the previous step. We go through the remaining tasks in *decreasing* workload order, and assign tasks to core $c$ if it can accommodate the whole task. The technical reason for assigning tasks in decreasing workload order will be explained later. We repeat this until core $c$ cannot accommodate the next task *completely*. Let $j$ denote this task that $c$ cannot

doi:10.6342/NTU201601864

accommodate. We then assign *part* of task $j$ to use up *all* the remaining CPU cycles of core $c$, and update the required number of CPU cycles of $j$ accordingly. The remaining part of task $j$ will be assigned to the next core in the next round.

Algorithm 1 shows the pseudo code of our algorithm that selects the frequency set and determines to the time percentage each task should run on each core.

**Algorithm 1** Frequency Selection and Time Assignment

```
Input: $n$ cores, a set of available frequency $F$, $m$ throughput guaranteed tasks, each task
    $j$ has a workload $w(j)$.
Output: The frequency $f(p)$ of each core $p$, a set of $a_{j,p}$ indicating the time percentage
    of task $j$ on core $p$.
 1: Compute the total amount of workloads $u = \sum_{j=1}^{m} w(j)$.
 2: Set the frequency of every cores to “undecided”.
 3: for each core $p$ do
 4:     if $u = 0$ then
 5:         Set $f(p)$ to 0.
 6:     else
 7:         Select the smallest frequency $f$ from $F$ for $c$ that satisfies all the constraints.
 8:         repeat
 9:             Assign the next task $j$ in decreasing workload order to core $p$.
10:             Update $u$ and $a_{j,p}$.
11:         until $p$ cannot completely accommodate the next task $j$.
12:         Assign part of task $j$ to use up all the remaining CPU cycles of core $p$.
13:         Update $u$, the required number of CPU cycles of task $j$, and $a_{j,p}$ accordingly.
14:     end if
15: end for
```

We assign tasks in *decreasing* workload order so that by our scheduler, the frequencies of cores will be *non-increasing* in their selected order. That is, if Algorithm 1 selects $f_i$ in the $i$-th round, then $f_i \geq f_{i+1}$ for $i$ between 0 and $n-1$.

## 4.4 Task Scheduling

There are two key concepts in the scheduler – *energy credits* and *run queues*. The energy-credit based scheduler gives energy credits to tasks according to the time percentage generated by Algorithm 1. The amount of the energy credit of a task $j$ on a core $p$ is proportional to the *time percentage* $a_{j,p}$ computed from Algorithm 1. After giving energy

doi:10.6342/NTU201601864

credits to tasks, the scheduler repeatedly selects a task to run on a core when the core is ready. A running task consumes a credit per unit of time. The goal is to schedule tasks so that *every* task uses up all its credits for this time period, thus the throughput is met.

Each core maintains a *run queue* to manage tasks. A task will only exist in the run queue of *one* core. This ensures that a task will not run on two cores simultaneously. In addition, a task can only appear in the queue of a core where it still has credits. The scheduler will schedule a new task to run when the current running task yields the core voluntarily (e.g., due to I/O), or uses up all its credits on that core. Then the scheduler selects the next task from the run queue according to high, medium, and low priority defined below. Tasks with the same priority run in a First-In-First-Out (FIFO) order. Algorithm 2 shows the pseudo code of the energy-credit scheduler.

**High priority** A task is a *split* task if the task has credits on more than one core. A split task should have a high priority to run, otherwise it would run late in this core, and even later in the other cores in which the task also has credits. As a result the task may not meet its throughput.

**Median priority** A non-split tasks with available credits will have median priority.

**Low priority** A task has a low priority in a core if it is *not* in the run queue of this core, but does have credit on this core.

Migrating tasks frequently between cores incurs significant overheads, so the energy-credit scheduler only migrates task under two conditions. First, if a core is ready but its run queue is empty, the scheduler will search for a task with credits on this core from the run queues of other cores. If such a task is found, the scheduler migrates the task to this core. This is the standard *task stealing* technique in the literature [4].

Second, if a task uses up its credits on a core, the schedule will migrate it to a core that it still has credits. This is called *out-of-credit migration*. The task will join the run queue where it still has credit. If no such core can be found, then the task must have met its throughput, and will wait for its execution in the next time period.

doi:10.6342/NTU201601864

**Algorithm 2** Energy-credit Based Scheduler

```
1:  for each time interval do
2:      Generates a schedule by algorithm 1.
3:      for every core do
4:          Set the operating frequency according to the schedule.
5:      end for
6:      while core $p$ has tasks. do
7:          repeat
8:              Executes the workload of a throughput guaranteed task and consumes the cor-
                responding credits.
9:          until The core becomes “available”.
10:         if The task being executed consumes all the credit on this core. then
11:             Migrate the task to another core according to its credit.
12:         else
13:             Move the task to the end of the run queue.
14:         end if
15:         Pick the next task from the run queue for execution.
16:         if No executable task in the run queue. then
17:             Steal tasks from the other cores.
18:         end if
19:     end while
20: end for
```

It is essential to reduce the overhead of out-of-credit migration. From the analysis of Algorithm 1 it is easy to see that Algorithm 1 will assign a task to at most *two* cores. We observe, also guaranteed by the analysis of Algorithm 1, only up to two entries in the same row are non-zero. As a result, the scheduler only needs to check the other non-zero entry, instead of the entire row, while deciding which core the task should migrate to during out-of-credit migration.

The fact that Algorithm 1 will assign a task to at most *two* cores also helps reduce the overheads of task stealing. It is easy to see that there will be at most *two* partial tasks assigned to a core – one from the previous round and one that continues to the next round. Consequently, if a run queue is out of tasks, that means all the tasks that were completely assigned to this core have finished, so the core only needs to steal from *at most* two cores that the partial tasks were also assigned. This helps reduce the effort in locating tasks to steal.

For example, Table 4.1 illustrates the credits each task receives on all cores by running

doi:10.6342/NTU201601864

Algorithm 1. Each row indicates the credits a task has on every core.

Table 4.1: An example of energy credits each task receives on each core.

| | $Core_1$ | $Core_2$ | $Core_3$ | $Core_4$ |
|---|---|---|---|---|
| $Task_1$ | 90 | 0 | 0 | 0 |
| $Task_2$ | 10 | 70 | 0 | 0 |
| $Task_3$ | 0 | 30 | 40 | 0 |
| $Task_4$ | 0 | 0 | 50 | 0 |
| $Task_5$ | 0 | 0 | 10 | 30 |

doi:10.6342/NTU201601864

# Chapter 5

# Implementation

In this chapter, we describe the implementation of energy-credit based scheduler. This scheduler add a new *throughput guaranteed task scheduling*. We will describe the execution flow of task in this scheduling class.

## 5.1 Throughput Guaranteed Task Scheduling Class

We propose to add a new *throughput Guaranteed Task Scheduling Class*. The goal of this scheduling class is to maintain the throughput of tasks and achieves energy efficiency for tasks in this class. The scheduler will provides a throughput guaranteed task the CPU resources that the task needs.

The priority of the throughput guaranteed task scheduling class is between stop-task class and deadline scheduling class. A throughput guaranteed task is no more critical than the stop-tasks, which seriously influence the stability of the system. On the other hand, the deadline tasks only need to meet their deadlines, but the throughput guaranteed task need meet their expected throughput in *every* time period. As a result, the deadline task is no more critical than the throughput guaranteed tasks.

doi:10.6342/NTU201601864

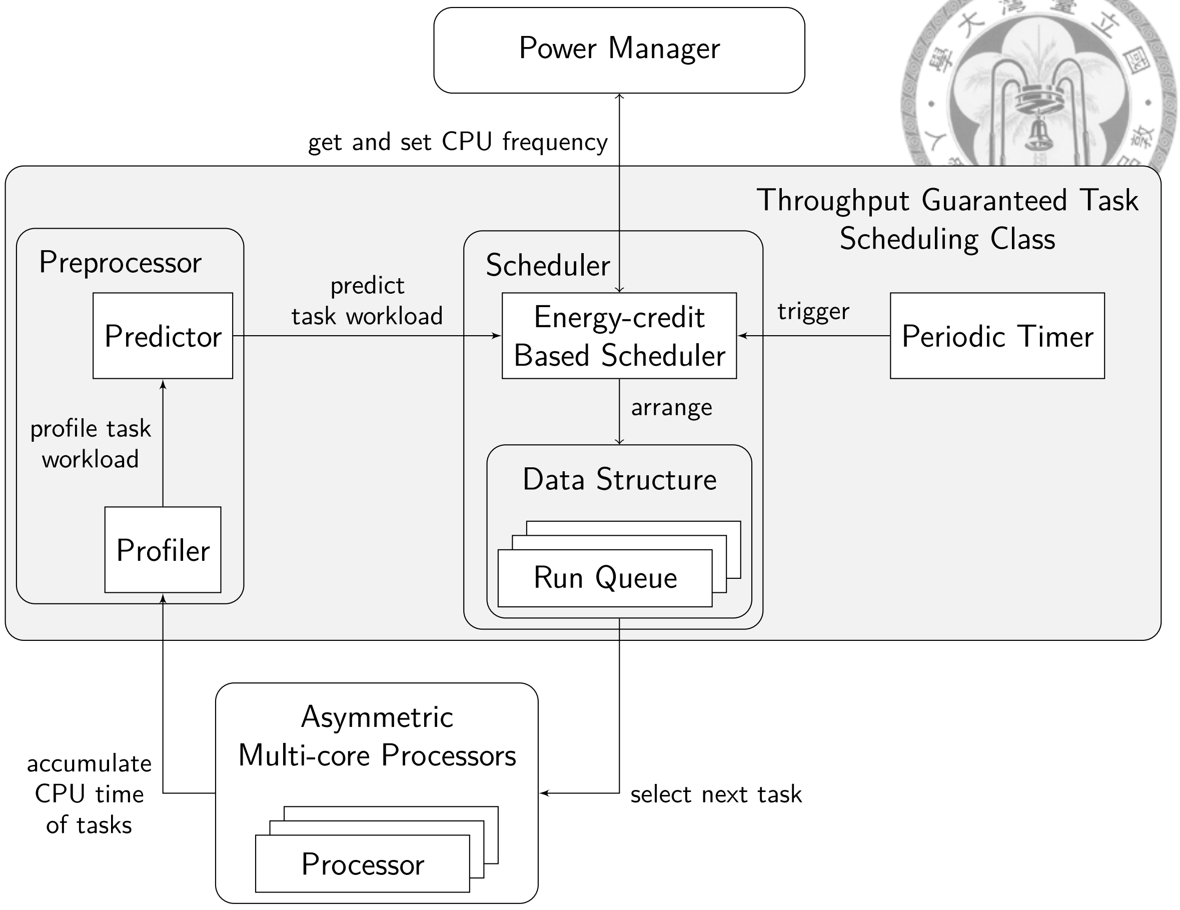


Figure 5.1: Execution flow of Throughput Guaranteed Task Scheduling Class for Asymmetric Multi-core Processors

## 5.2 Execution Flow

Figure 5.1 illustrates the execution flow of tasks of throughput guaranteed scheduling class on asymmetric multi-core processors. The execution flow requires *periodic timers*, a *preprocessor*, and a *scheduler*, which are explain in details in later sections.

### 5.2.1 Periodic Timers

The execution of throughput guaranteed tasks uses two timers – a *global timer* and a *local timer*. A global timer triggers the scheduler to reschedule all run queues in order to present the time interval in the Algorithm 2. The global timer triggers the scheduler to retrieve the CPU frequency and the workload of each task from *all* cores so that the power

doi:10.6342/NTU201601864

manager can schedule the tasks in the later preprocessing stage.

A local timer on each core triggers the scheduler to select a new task to run for time sharing. The local timer goes through three steps. First, the local timer suspend the running task so that it relinquishes the core. Second, the local timer triggers the scheduler to move the suspended task to the end of it run queue. Finally, the scheduler selects a new task to run according to the priority of the tasks.

### 5.2.2 Preprocessor

A preprocessor measures the execution time and predicts the workload for each task in the next time period. The preprocessor consists of a *profiler* and a *predictor*. The profiler accumulates the execution time of each task as the workload of each task. Note that we cannot estimate the workload of each task simply by accumulating the number of CPU clock cycles, which is not be available on *every* devices. As a result we use the *software clock* of the Linux kernel to measure CPU time. After profiling the tasks, we covert the execution time $t$ to the workload $w$ by Equation 5.1. The execution time percentages of a task is simply its execution time $t$ divided by the length of a time period $T$. The workload $W$ (as the number of CPU cycles) of a task is the time percentage, i.e., the workload, multiplied by the CPU frequency $F$.

$$w = \frac{t}{T} \times F \tag{5.1}$$

A *predictor* predicts the workload of each task according to historical data. Let $d(j)$ denote the *highest* workload of task $j$. We implement a *system call* and tasks can use the system call to inform the predictor their highest workloads ($d(j)$). If task $j$ does not inform the predictor its $d(j)$, the predictor sets $d(j)$ to the highest CPU frequency on big core.

After receiving workload from the tasks, the predictor now predicts the workload of each task. Let $w(j, t)$ denote the profiled workload of task $j$ in the time period $t$, and $p(j, t)$ be the predicted workload of task $j$ by the predictor. We predict the $p(j, t + 1)$

doi:10.6342/NTU201601864

according to Equation 5.2. If the resource we predicted ($p(j, t)$) is equal to the resource actually needed ($w(j, t)$), that means that right now the recourse is not enough or barely enough. As a result we set the resource to the highest possible from the history ($(d(j))$. When $p(j, t)$ is more than $w(j, t)$, it means we have give task $j$ excessive resource, and should now reduce it.

$$p(j, t+1) = d(j), \quad w(j, t) = p(j, t) \tag{5.2}$$

$$p(j, t+1) = \frac{w(j, t) + p(j, t)}{2}, \quad \text{otherwise} \tag{5.3}$$

### 5.2.3 Scheduler

We create a new *governor* to set CPU frequency because in Linux the power manager and the CPU scheduler are independent, so the scheduler cannot set CPU frequency directly. Since the scheduler cannot interrupt the power manager to notify that it wishes to set CPU frequency, instead the governor will periodically *poll* the scheduler to know if the scheduler decides to set the frequency or not. The scheduler cannot interrupt the power manager because it may be blocked, and the scheduler are not allowed to be blocked by anything in Linux. Also note that we cannot schedule the throughput guaranteed tasks when the Linux boots up because Linux loads the power manager *after* loading the CPU scheduler.

The scheduler distributes the split task evenly among the core for performance. Since the split tasks have high priority, having many split tasks in a core may delay their execution so that they may not meet their throughput.

In order to manage the run queues the scheduler first locks all run queues. This can prevent the timers from changing the data structure of the run queses, causing inconsistency in the data structures of tasks in the run queues. Then the scheduler assigns tasks to cores, and move the tasks to the assigned run queues. After that the scheduler notifies the governor to adjust CPU frequency, then it unlocks all run queues. Finally, the scheduler

doi:10.6342/NTU201601864

select a new task to run on each core.

Note that the management of run queue does not wait for the current running tasks to finish. If the scheduler waits for all running tasks to finish or migrate, the overhead will be tremendous. As a result we simply manage the queues without waiting for running task to complete or migrate.

Due to the "no-wait" policy of the scheduler, there could be the cases that one task is running on a core, but its data structure as a task has been moved to the run queue of another core. We define this task as in an *inconsistent* state, meaning that it is running on one core, and in fact it is in the run queue of another core.

After adjusting tasks among run queues of all cores, the scheduler will select a task to run for a core from its run queue. The scheduler may select a task to run on a core $c$ if all four following conditions are satisfied. Note that the scheduler will only select a task from three run queues – the run queue of $c$, and up to two run queues of $c$'s neighboring cores. Here neighboring cores mean they have adjacent indices. As a result the scheduler locks at most three run queues to select a task from them.

1. The task has available credits on this core.
2. The task is runnable.
3. The scheduler has not assigned this task to a core.
4. The task is *not* in the inconsistent state, or the task is in the inconsistent state, and the run queue of core $c$ has this task. Note that if the scheduler selects a task in the inconsistent state (running on core $c'$) to run on a core $c$, then the scheduler must wait for the task to give up core $c'$ before migrating the task to core $c$.

When a task wakes up after sleeping for several time periods, the scheduler puts the task into the run queue of an available core. If no cores are available, the scheduler puts the task into a big core, then it adjusts the CPU frequency of the big core to satisfy throughput of all tasks in this big core. Note that we do not reschedule *all* tasks because of its huge overhead.

doi:10.6342/NTU201601864

# Chapter 6

# Experiment

This chapter describes metrics that evaluate the effectiveness of our scheduler and the experimental performance results.

## 6.1 Methodology

Our target platform is a Juno ARM development board. The Juno ARM development board is an asymmetric multi-core platform consists of performance big cores (two Cortex-A57) and energy-efficient little cores (four Cortex-A53).

The platform supports per-cluster DVFS, i.e., clusters can work under different operating frequencies, so that we can adjust the frequency of each cluster according to the schedule. Table 6.1 details the specifications of the related hardware and software. Both Cortex-A57 and core Cortex-A53 can run on five frequency levels. Table 6.2 shows the available CPU frequency level for each core type in Linaro release version 15.07.

We implement our energy-credit based scheduler on the Juno ARM development board. The scheduler periodically generates schedules for both types of cores. A schedule consists of the frequencies of cores and the energy credits of tasks. The scheduler then assigns throughput guaranteed tasks to cores for execution according to their energy credits. We set a scheduling period to one second in our experiments.

We use *VLC* [16], a free and open source cross-platform multimedia player, as our

doi:10.6342/NTU201601864

| Hardware | |
|---|---|
| Processor | Dual Cluster, ARMv8 big.LITTLE configuration<br>Dual-Core Cortex-A57 (2MB L2 cache)<br>Quad-Core Cortex-A53 (1MB L2 cache) |
| Memory | RAM: DDR3 8GB |
| Storage | micro SD card lots<br>configuration EEPROM |
| Software | |
| Linaro Release Version | 15.07 |
| Linux Kernel Version | 3.10.80 |
| Root File System | Ubuntu 14.10 |

Table 6.1: Specifications of Juno ARM Development board

| Core type | CPU frequency levels | | | | |
|---|---|---|---|---|---|
| Cortex-A57 | 450 | 625 | 800 | 950 | 1100 |
| Cortex-A53 | 450 | 575 | 700 | 775 | 850 |

Table 6.2: Available CPU frequency levels (MHz)

benchmark. A VLC process consists of fifteen threads and each thread does different work. We consider each thread of the VLC as a throughput guaranteed task.

We can adjust the *bit rate* of the media player to control the its workload. We use a video converter *FFmpeg* [7] to encode the video with a *constant bit rate (CBR)*, so the video player will consumes output from a decoder at a constant rate. As a result we can control the workload of the media player by adjusting its bit rate.

The Linux *Priority System* controls the priority of a thread with a *nice value* from -20 to 19. A niceness of -20 is the highest priority and 19 is the lowest priority. The default nice value of a task is 0, and we set nice value of the VLC to 0 in our experiment.

We use by *ARM Energy Probe* to measure the power consumption of the Juno ARM development board. The ARM Energy Probe can only measure the energy of big cluster and little cluster, so our data exclude the power consumption of the other devices.

We use *DS-5 Streaming* [3] to analyze the power consumption reported by ARM Energy Probe. DS-5 Streaming starts a background task *gator* to collect the data from ARM Energy Probe. A gator is a high priority profiling task, so we set its nice value to -19.

doi:10.6342/NTU201601864

We test three workload scenarios. The first case is a *light-weight workload*, where we run one VLC that plays a video encoded with a constant bit rate of 400 $kb/s$. In the second *Median-weight workload case* we run eight VLCs simultaneously, and each of them plays a video with the same bit rate as in the first case. In the third *heavy-weight workload case* we run one VLC that plays a high quality video without compression.

We compare the power consumption of our Energy-credit Based Scheduler (denoted as ECS) with Completely Fair Scheduler (denoted as CFS) under different workloads. In the comparison CFS uses two existing governors – *on-demand* and *conservative*. The On-demand governor is the default governor for power manager in Linux, and the conservative governor is a power saving governor.

Recall that the on-demand and the conservative governor have a up threshold and a down threshold. The governors increase the CPU frequency up frequency level when the CPU usage exceeds the up threshold. On the other hand, the governors reduce the CPU frequency when the CPU usage is below the down threshold. We set the up threshold to 80% and the down threshold to 20% respectively.

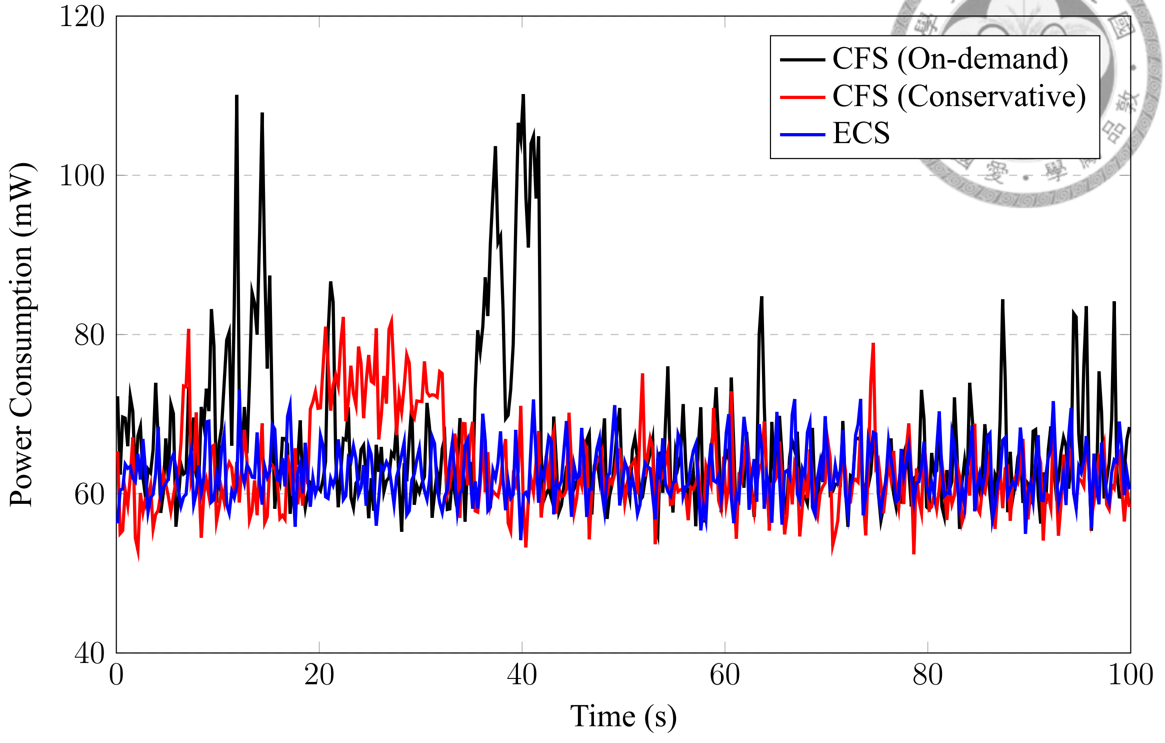
We evaluate the power consumption of two configurations. In the first configuration, there is only one CFS scheduler, which schedules *all* processes, including VLC, gator, and other system services. In the second configuration, there are two schedulers – CFS and ECS. ECS schedules the throughput guaranteed task scheduling class of VLC, and CFS schedules gator and other system services. The reason we do not use ECS to schedule *everything* is that we cannot control the workload of these system services. Also these services do not need to guarantee their throughput.

## 6.2 Experiment Results

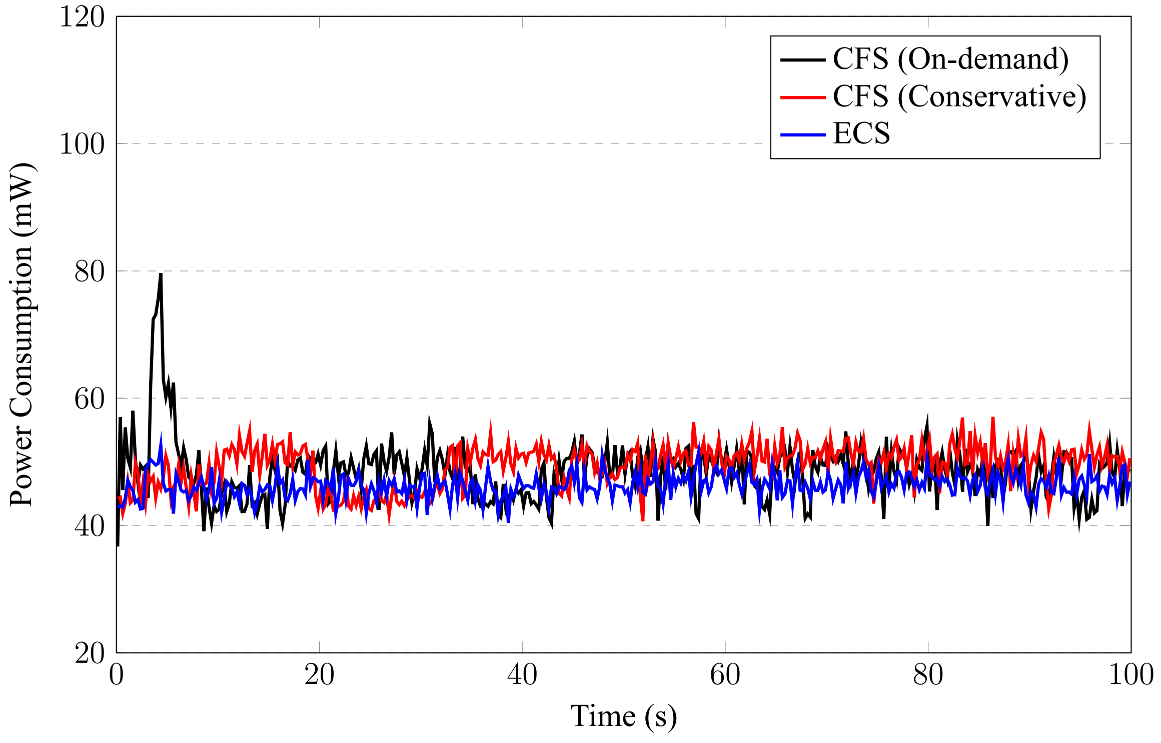
### 6.2.1 Light-weight Workload

Figure 6.1 shows power consumption of two CPU clusters under light-weight workload. CFS ignores the workload of tasks and evenly distribute them to cores, so it will as-

doi:10.6342/NTU201601864



(a) Big core cluster



(b) Little core cluster

Figure 6.1: Power consumption required by CFS (On-demand), CFS (Conservative), and ECS with light-weight workload on different CPU cluster.

doi:10.6342/NTU201601864

sign some throughput guaranteed tasks to the big core cluster. This assignment increases the total power consumption no matter which governor it uses.

Figure 6.1(a) indicates that the power consumption of the CFS with on-demand governor will sometimes increase dramatically. The reason is that when the CPU usage of the big core exceeds the up threshold, the on-demand governor sets the CPU frequency to the *highest* frequency. On the other hand, the conservative governor increases the frequency of the core only by one level, so its power consumption is much smaller than that of the on-demand governor.

Figure 6.1(a) indicates that our scheduler consumes much less power than CFS. The reason is that our ECS can classify the workload of each task, so all light-weight tasks stay will stay on the little cores. Our scheduler also knows the total workload of each core, so it always sets the little cores to the lowest CPU frequency.

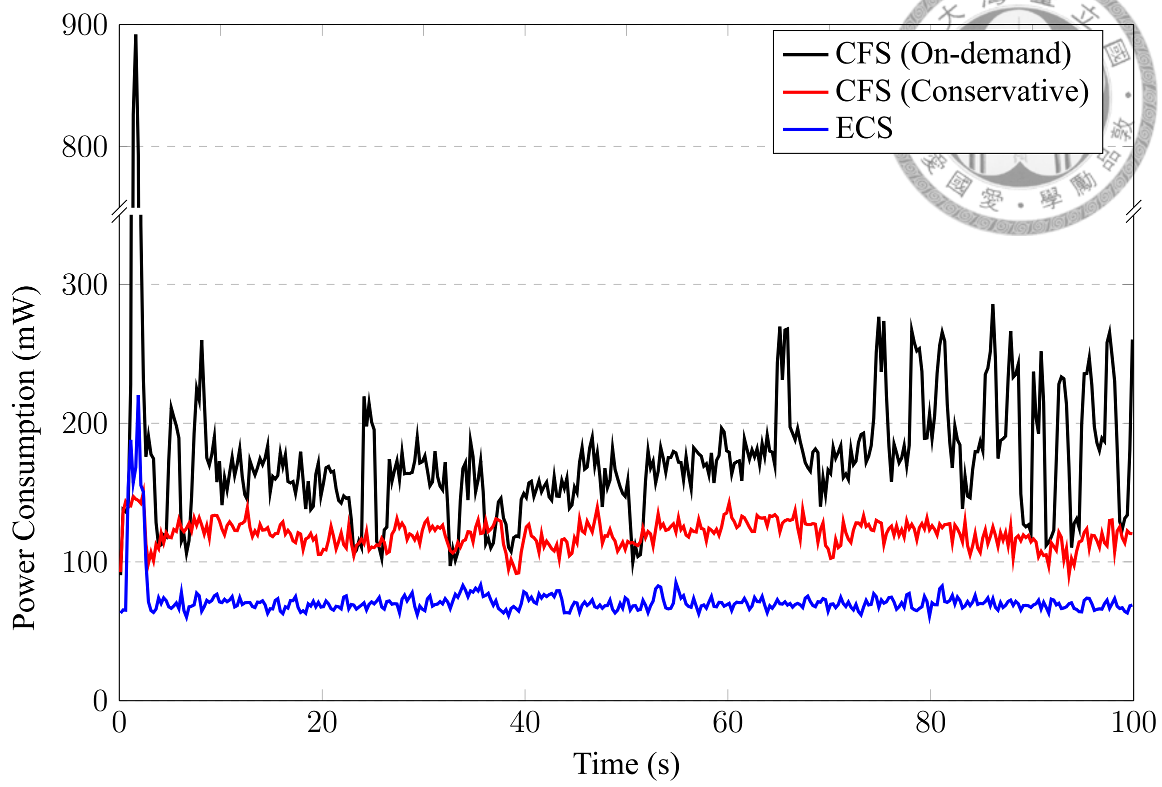
We observe that the power consumption under all scheduling methods will oscillate due to background system service tasks. Even though ECS does not assign tasks to big core cluster, the background service tasks will still run on the big core cluster because they are scheduled by CFS. This causes power consumption oscillation we observed.

### 6.2.2 Median-weight Workload

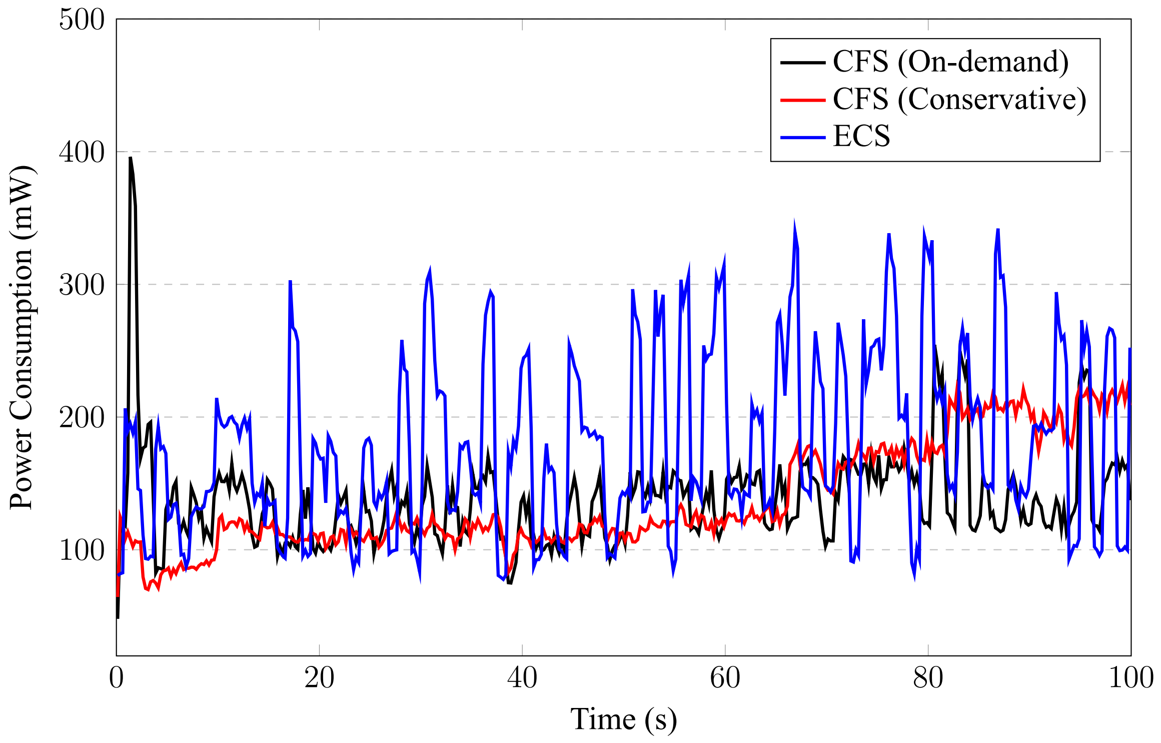
Figure 6.2 shows the power consumption under the median-weight workload on two CPU cluster. When many VLCs start to run they create many threads, and each of them is a throughput guaranteed task. Both CFS and ECS assign some tasks to big core cluster because the number of tasks is too many. CFS, with the on-demand governor detects the high CPU usage, so it scales up to the highest frequency. In contrast ECS and CFS with the conservative governor scale up only one frequency level. After that, ECS realizes that the total workload decreases, so it scales down CPU frequency immediately.

VLC will preload video that it is play, and preloading affects its workload. When the VLC is preloading the workloads of the tasks increase. When the VLC is playing the video without preloading, the workloads of the tasks decreases. When the VLC preloads

doi:10.6342/NTU201601864



(a) Big core cluster



(b) Little core cluster

Figure 6.2: Power consumption required by CFS (On-demand), CFS (Conservative), and ECS with median-weight workload on different CPU cluster.

doi:10.6342/NTU201601864

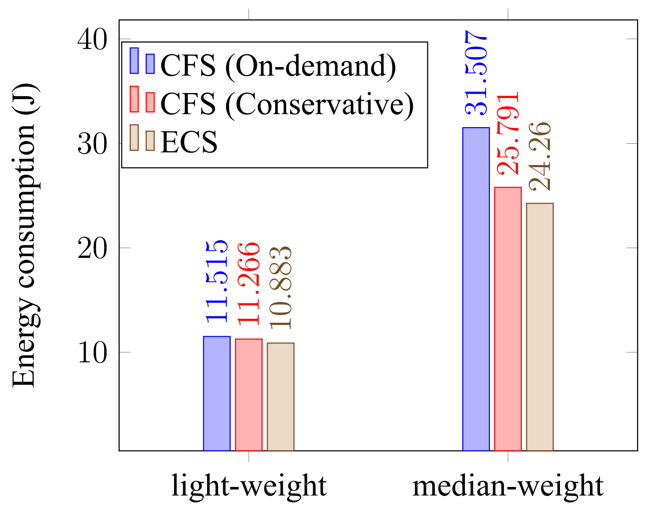


Figure 6.3: Totally energy consumption required by CFS (On-demand), CFS (Conservative), and ECS with different workload of the system.

the video, our predictor accurately predicts that the workload of the task will rise to the highest of the demand, so it adjusts the frequency accordingly.

### 6.2.3 Heavy-weight Workload

VLC plays video smoothly under ECS, but not under CFS, neither with on-demand nor conservative governor. This is because CFS assigns the heavy tasks to the little core cluster, so they cannot have the resource to meet their throughput, even through the frequency of the little core has been set to the highest level. In contrast, ECS knows that the tasks are heavy, so it assigns the tasks to the big cores. After this assignment, ECS increases frequency of the big core cluster for performance, and decreases the frequency of the little core cluster to save power.

### 6.2.4 Summary

We observe from Figure 6.3 that our energy-credit based scheduler consumes 5.5% and 3.4% less energy than CFS with the on-demand governor and CFS with the conservative governor respectively under the light-weight workload. CFS consumes more energy because CFS assigns light-weight tasks to the big core cluster. Also our scheduler consumes

doi:10.6342/NTU201601864

29.8% and 6.3% less energy than CFS with the on-demand governor and CFS with the conservative governor respectively under the median-weight workload. ECS consumes less energy in this case because it can classify the task and accurately adjust CPU frequency according to the total workload for each core.

VLC can smoothly play the video with ECS under heavy-weight workload because ECS assigns all of heavy-weight tasks to the big core cluster. VLC cannot smoothly play the video on CFS with either on-demand or conservative governor because CFS assigns some of heavy tasks to the little core cluster.

doi:10.6342/NTU201601864

# Chapter 7

# Conclusion

In this paper, we design an energy-credit based scheduler for throughput guaranteed tasks on an asymmetric multi-core platform. The proposed scheduler consists of four key components - *task classification*, *frequency selection*, *time assignment*, and *task scheduling*. The system classifies tasks suitable for big and little cores respectively, determines the frequency of each core, assigns the percentage of time each task should run on each core, and ensures that tasks will only run on cores they are assigned, and tasks will receive the percentage of CPU time they are granted on the cores they are assigned. We implement our energy credit-based scheduler by adding a throughput guaranteed task scheduling class within Linux. The experiment results indicate that our proposed scheduler consumes 29.8% and 6.3% less energy than the Completely Fair Scheduler with on-demand and conservative frequency governors respectively.

doi:10.6342/NTU201601864

# Bibliography

[1] A. Das A. K. Singh and A. Kumar. Energy optimization by exploiting execution slacks in streaming applications on multiprocessor systems. Design Automation Conference, 2013.

[2] Akash Kumar Anup Das and Bharadwaj Veeravalli. Reliability and energy-aware mapping and scheduling of multimedia applications on multiprocessor systems. IEEE Transactions on Parallel and Distributed Systems, 2016.

[3] ARM. Streamline performance analyzer, ds-5 development studio. https://developer.arm.com/products/software-development-tools/ds-5-development-studio/streamline/overview.

[4] Robert D. Blumofe and Dionisios Papadopoulos. The performance of work stealing in multiprogrammed environments (extended abstract. In *In Proceedings of the 1998 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Poster Session*, pages 266–267. ACM Press, 1998.

[5] P. Bogdan, S. Garg, and U.Y. Ogras. Energy-efficient computing from systems-on-chip to micro-server and data centers. In *Green Computing Conference and Sustainable Computing Conference (IGSC), 2015 Sixth International*, pages 1–6, Dec 2015.

[6] Quan Chen and Minyi Guo. Adaptive workload-aware task scheduling for single-isa asymmetric multicore architectures. *ACM Trans. Archit. Code Optim.*, 11(1): 8:1–8:25, February 2014.

[7] FFmpeg. https://ffmpeg.org/.

[8] Andrei Frumusanu. The samsung exynos 7420 deep dive - inside a modern 14nm soc, 2015.

[9] Peter Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White Paper*, 2011.

[10] Vishal Gupta, Min Lee, and Karsten Schwan. Heterovisor: Exploiting resource heterogeneity to enhance the elasticity of cloud platforms. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, pages 79–92, New York, NY, USA, 2015. ACM.

[11] JUAN HAMERS and LIEVEN EECKHOUT. Exploiting media stream similarity for energy-efficient decoding and resource prediction. ACM Transactions on Embedded Computing Systems, 2012.

[12] Brian Jeff. big.little technology moves towards fully heterogeneous global task scheduling. November 2013.

[13] David Atienza Karim Kanoun, Nicholas Mastronarde and Mihaela van der Schaar. Online energy-efficient task-graph scheduling for multicore platforms. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2014.

[14] Myungsun Kim, Soonhyun Noh, Sungju Huh, and Seongsoo Hong. Fair-share scheduling for performance-asymmetric multicore architecture via scaled virtual runtime. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2015 IEEE 21st International Conference on*, pages 60–69, Aug 2015.

[15] Youngjin Kwon, Changdae Kim, Seungryoul Maeng, and Jaehyuk Huh. Virtualizing performance asymmetric multi-core systems. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 45–56, New York, NY, USA, 2011. ACM.

doi:10.6342/NTU201601864

[16] VLC media player. http://www.videolan.org/vlc/.

[17] Chandandeep Singh Pabla. Completely fair scheduler. *Linux J.*, 2009(184), August 2009.

[18] Padmanabhan Pillai and Kang G Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 89–102. ACM, 2001.

[19] Nicolas Pitre. Linux support for arm big.little. http://lwn.net/Articles/481055/, 2012.

[20] Chin-Chiang Pan Po-Hsien Tseng, Pi-Cheng Hsiu and Tei-Wei Kuo. User-centric energy-efficient scheduling on multi-core mobile devices. Design Automation Conference, 2014.

[21] Rafael Rodríguez-Sánchez and Enrique S Quintana-Ortí. Architecture-aware optimization of an hevc decoder on asymmetric multicore processors. *arXiv preprint arXiv:1601.05313*, 2016.

[22] Elsayed Saad, Medhat Awadalla, Mohamed Shalan, and Abdullah Elewi. Energy-aware task partitioning on heterogeneous multiprocessor platforms. *arXiv preprint arXiv:1206.0396*, 2012.

[23] Wonik Seo, Daegil Im, Jeongim Choi, and Jaehyuk Huh. Big or little: A study of mobile interactive applications on an asymmetric multi-core platform. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 1–11, Oct 2015.

[24] William Thies, Michal Karczmarek, and Saman Amarasinghe. *Compiler Construction*, chapter StreamIt: A Language for Streaming Applications, pages 179–196. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

doi:10.6342/NTU201601864

[25] Frances Yao, Alan Demers, and Scott Shenker. A scheduling model for reduced cpu energy. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 374–382. IEEE, 1995.

[26] Yuan-Hao Chang Yu-Ming Chang, Pi-Cheng Hsiu and Che-Wei Chang. A resource-driven dvfs scheme for smart handheld devices. ACM Transactions on Embedded Computing Systems, 2013.

doi:10.6342/NTU201601864