國立交通大學

網路工程研究所

碩 士 論 文

手持式裝置上考慮耗時與耗電的運算量卸載至
協同處理器與雲端處理

Time-and-Energy Aware Computation Offloading in Handheld

Devices to Coprocessors and Clouds

研 究 生：黃霆鈞

指導教授：林盈達　教授

中 華 民 國 一 百 年 六 月

手持式裝置上考慮耗時與耗電的運算量卸載至

協同處理器與雲端處理

Time-and-Energy Aware Computation Offloading in Handheld Devices
to Coprocessors and Clouds

研 究 生：黃霆鈞　　　　　Student：Ting-Jun Huang

指導教授：林盈達　　　　　Advisor：Dr. Ying-Dar Lin

國 立 交 通 大 學

網 路 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Network Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2011

Hsinchu, Taiwan

中華民國一百年六月

手持式裝置上考慮耗時與耗電的運算量卸載

至協同處理器與雲端處理

學生: 黃霆鈞　　　　　　　　　指導教授: 林盈達

國立交通大學網路工程研究所

摘要

　　基於省電與成本考量，手持裝置上的硬體元件通常只適合特定的應用，無法做為一般性的運算使用。但隨著智慧型手持裝置(如手機、平板)的普及，這些裝置上所運行的軟體愈趨多元，導致裝置上的耗時與耗電大幅提高，帶動了手持裝置上運算量卸載的研究，而運算量的卸載並非永遠可以省時與省電。我們實作了一個以環境因子為基礎的卸載決策架構，它以收集而來的環境因子預測在每種運作環境下的耗時與耗電，再依使用者所傾向的使用模式做出決策。我們選定兩種評測程式做為卸載決策成果的評估，分別為矩陣相乘與病毒掃描。在矩陣相乘的運算中，此方法決策錯誤率低於 30%，而且成功的讓裝置在運算過程中節省約 20~300%的運算時間，與 50~130%的電量消耗。在病毒掃描的測試中，採用的是 clamAV 的掃毒程式，此方法決策錯誤率趨近於零。令人意外的是，在測試中發現當掃描較大容量的檔案時(例：大於 2MB)，將檔案傳送到雲端進行掃描的方案完全不適用，反而會導致 160~250%左右的效能損耗。

Time-and-Energy Aware Computation Offloading in Handheld Devices to Coprocessors and Clouds

Student: Ting-Jun Huang   Advisor: Dr. Ying-Dar Lin

Department of Computer and Information Science

National Chiao Tung University

Abstract

The hardware components in handheld devices are usually adapted to specific application and are hard to do general computations. However, with the popular of smart handheld devices, e.g. smart phone and pads, more kinds of software can run on these devices and raising the time and energy consumption. Therefore, some research discussed with computation offloading, but offloading does not promise the time and energy saving can always be achieved. Thus, we implement an offloading decision framework based on environment factors. The framework collects factors for estimating time and energy usage in every computing environment, and then makes decision according to user's preference. We pick two program, matrix multiplication and virus scanning, to evaluate the decision framework. In matrix multiplication computation, the false decision rate is below 30%, and can save 20~300% computing time and 50~130% energy consumption. In virus scanning, we choose the scanner from clamAV, the false decision rate is nearly zero. Surprisingly, we find a large file, e.g. larger than 2MB, is not suitable for offloading to cloud for scanning, which will suffered 160~250% performance decrease.

Keywords: computation offloading, coprocessors, cloud computing, Android

Content

List of Figures

List of Tables

Chapter 1.   Introduction

Nearly 300 million smart phones were sold in 2010, and its number is expected to increase 80% in 2011 [1]. In order to satisfy the needs of billions of users, smart phones feature versatile mobile applications. Examples of the latest functions include multimedia, real-time games, GPS navigation and communication. Most of these mobile applications are user-interactive and data-processing intensive, both of which require quick response and long battery life. However, most commercial off-the-shelf smart phones, compared to desktops, are generally equipped with low speed processors and limited capacity batteries. Running sophisticated software on smart phones can result in poor performance and shorten battery lifetime. Therefore, it becomes a crucial issue in designing smart phones to deliver adequate performance and prolong battery life.

A lot of advanced hardware technology, such as instruction level parallelism, leakage power control and dynamic voltage scaling, have been proposed to improve processor speed and reduce energy consumption. Although the advanced technology can deliver better performance, adopting high-end processors is not always appropriate for budget-limited projects. Recently, cloud computing becomes another possible solution to enhance computing capability of smart phones. The cloud computing vendors provide computing cycles for the registered users to reduce computation and energy consumption of smart phones. Examples include Amazon Elastic Compute Cloud (EC2), Amazon Virtual Private Cloud (VPC) and PacHosting. However, it takes both time and energy to upload programs to the cloud and retrieve the results from the cloud. The computation capacity of the cloud can also affect the response time of the offloaded programs. In order to save both time and energy consumption, there is a clear need for the development of a decision making mechanism before offloading.
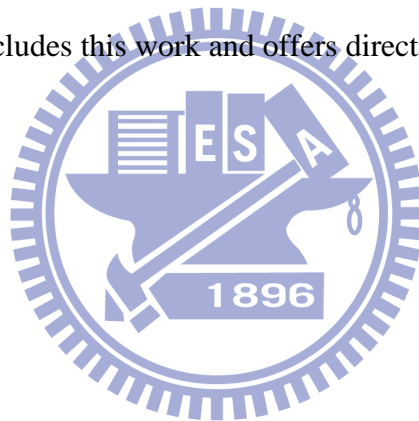
There have been many research efforts dedicated to offload computation intensive programs from a resource-poor mobile device [2-5]. X. Gu *et al.* [2], Z. Li [3] and G. Chen *et al.* [5] partitioned source codes into client/server parts, and then saved energy consumption by running the server parts at remote servers. All these methods perform well for small size applications, but may induce significant overhead when partitioning large size applications. K. Kumar [6] proposed a simplified energy model to quickly estimate the energy saved from cloud services. However, several key power-related parameters were not considered, which may lead to an incorrect offloading decision. In addition, all above works ignored the impact of offloading on execution time, which may result in performance degradation. On the other hand, S. Ou *et al.* [4] developed an offloading middleware, which provides runtime offloading services to improve the response time of mobile devices. Wolski [7] used bandwidth data to estimate the performance improvement through offloading. Both works did not investigate the energy consumption of uploading and retrieving data, and may shorten battery life time. Also, important timing-related factors were not considered, which can result in an incorrect offloading decision. Because it is response time and energy consumption that determine user satisfaction, we address a multi-objective opmization problem that simultaneously optimizes these two key performance indexes of smart phones.

In this work, we develop an offloading framework which aims to shorten response time and reduce energy consumption at the same time. Unlike previous works, our targets of execution include on-board CPU, on-board GPU and cloud, all of which provide a more flexible execution environment for mobile applications. Since response time and energy consumption may be two conflicting objectives, we first design a customizable cost function, which allows users to adjust the weight of response time and energy consumption. We then develop a lightweight profiling

method to estimate the performance improvement and energy consumption from offloading. In order to make correct decisions, several key system factors are considered when constructing cost functions. Finally, an offloading decision is made based on the user-defined cost function, estimated response time and energy consumption.

The rest of this paper is organized as follows. Chapter 2 introduces related works with computation offloading, and some concepts for Android development and GPU programming. Chapter 3 gives the problem statements and terminologies for the later chapters. Chapter 4 proposes the methodologies of offloading decisions, and then Chapter 5 details the implementation. Chapter 6 shows the experiment result and evaluation. Chapter 7 concludes this work and offers directions for future work.

Chapter 2.   Background

In this chapter, we first give a comprehensive comparison between our work and related works. We then introduce Android platform and OpenGL|ES, which are used in our experiments.

2.1 Related Works

There have been many research efforts dedicated to offload computation intensive programs from a resource-poor mobile device [2, 3, 5-18]. Some of them focused on energy saving [3, 5, 6, 8-13] while others targeted at performance improvement. Only few of them considered both energy saving and performance improvement [19, 20]. For ease of reference, all these works are summarized in Table 1, which are classified into three categories: *on energy saving*, *on time saving* and *on energy and time saving*. For each work, we further characterize it by four attributes: *adaptability*, *portability*, *accuracy* and *offload target*.

The *adaptability* indicates the capability of the proposed method to adapt itself efficiently to dynamic workload, resulting from the variance of data input at run-time. If the proposed method can only handle deterministic workload, this shows its adaptability is poor. The *portability* represents the ability of the proposed method to be ported from one execution environment to another, such as Linux to Windows. The term Language/Framework/Kernel represents the way we port the method to another platform. For example, this work is Language level of portability and need to do modifications in programming language when moving to another platform. Methods with Language portability are desirable because most of the existing codes can be reused. The *accuracy* is the approximation error of energy model or execution time model of offloading. The higher accuracy indicates the fewer incorrect offloading decisions. For this attribute, the works that did not develop any execution model or energy model are labeled as "n/a". The *offload target* is the targets, which can execute

offloaded programs. The more targets we have, the more flexible the execution environment will become. According to Table 1, our work is the only one that aims at saving both time and energy while maintaining high adaptability, high portability, high accuracy and multiple offload targets. In the following, we compare our work with each of related works in detail.

Table 1. Comparison of current offloading works

	Paper Works [Reference #]	Adaptability	Portability	Accuracy	Offload Target
On Energy Saving	Partition Scheme [3]	No	Framework	Low	Cloud
	Study Energy Tradeoffs [5]	No	Framework	n/a	Cloud
	Component Migration & Replication [9]	No	Framework	Medium	Cloud
	Cooperative Dynamic Power Management [11]	No	Framework	n/a	Cloud
	Offload H.264 Encoder [13]	No	Framework	n/a	Cloud
	Content-Based Image Retrieval [10]	Yes	Language	Medium	Cloud
	MAUI Code Offload [8]	No	Framework	n/a	Cloud
	Can Offload Save Energy[6]	Yes	Language	Medium	Cloud
	Face-Recognize with GPU [12]	No	Language	n/a	GPU
On Time Saving	Adaptive Offloading [2]	No	Framework	Low	Cloud
	Effective Offload Service [17]	No	Framework	n/a	Cloud
	Calling the Cloud [14]	No	Framework	n/a	Cloud
	eyeDentify Cyber Foraging [15]	No	Framework	n/a	Cloud
	Heterogeneous Auto-Offload Framework [18]	No	Framework	n/a	Cloud
	Using Bandwidth to Make Offloading Decision [7]	Yes	Language	Medium	Cloud
	VPN Gateway over Network Processors [16]	No	Kernel	n/a	Network Processor
On Energy and Time Saving	Computation Offload Scheme [20]	No	Framework	Medium	Cloud
	Energy Efficiency of Mobile [19]	Yes	Language	n/a	Cloud
	Our Work	Yes	Language	High	GPU, Cloud

Works On Energy Saving

Maximizing battery life time is one of the most crucial design objectives of smart phones because they are usually equipped with limited battery capacity. Z. Li [3], S. Han [9], B. Seshasayee [11], and E. Cuervo *et al.* [8] adopted profiling-partitioning technology to identify offloaded parts of an application for energy saving. They first profiled the energy consumption of each function of the application. According to the profiling result, they then generated a cost graph, in which each node represents a function to be performed and each edge indicates the data to be transmitted. The maximum-flow/minimum-cut algorithm was then used to partition the cost graph to obtain client parts and server parts. Finally, the server parts were executed at remote servers for reducing energy consumption of mobile device. G. Chen *et al.* [5] designed a similar method to determine whether Java methods and bytecode-to-native code compilation should be executed at remote servers for energy saving. In addition, they assumed that the workload was deterministic, which means that the workload will not vary at run-time. As a result, their methods cannot be applied to dynamic workload, resulting from the variance of data input at run-time. On the contrary, in order to reduce profiling overhead, we only profile the energy consumption and execution time of frequently-used modules, such as FFT, IFFT, convolution, matrix multiplication and so on. In addition, we take into account the impact of data size on execution time and energy consumption in order to handle dynamic workload at run-time.

X. Zhao [13], Y. J. Hong [10], and K. Kumar [6] built energy models to approximate the energy consumption of offloading. The energy models can be used to construct the above-mentioned cost graph or make offloading decisions. However, several key parameters, such as workload dynamics, bandwidth variability, and idle mode energy consumption, are not included in their models, which may lead to

inappropriate partitions or incorrect offloading decisions. According to our experiment results, our energy model ensures a higher accuracy than previous works by considering these key parameters. Y. C. Wang [12] demonstrated the possibility of utilizing GPU for offloading. They first identified bottlenecks of programs, and then used OpenGL|ES to rewrite and remove the bottlenecks. However, CPU and GPU are usually integrated on the same chip and cannot be switched off individually. Without considering the idle energy consumption of the chip, offloading programs to GPU may increase the total energy consumption. Our work, on the other hand, achieves a higher accuracy by modeling the idle energy consumption. We also provide the ability of offloading programs to GPU or Cloud.

Works On Time Saving

Responsiveness of mobile applications is important because the mobile applications are usually real-time and user-interactive. Many research efforts have been devoted to offload part of a program to remote servers in order to reduce execution time [2, 7, 14, 17]. Most of them adopted above-mentioned similar profiling-partitioning technology to identify the offloaded parts of an application. Gu *et al.* [2] designed an offloading engine that dynamically partitions an application when the required resources, such as memory and CPU, approach the maximum capacity of the mobile devices. Yang [17] developed an offloading service that dynamically partitions Java applications and transforms offloaded Java classes into a form that can be executed at remote servers. Giurgiu *et al.* [14] developed an exhaustive search algorithm, called ALL, to examine all possible partitions in order to find an optimal partition. They also proposed a heuristic algorithm to partition a program in reasonable time. All these methods perform well on small-size applications, but may induce significant overhead when partitioning large-size applications. On the contrary, we only profile the energy consumption and execution

time of frequently-used modules in order to reduce the overhead of profiling and partition. Unlike [2, 14, 17], R. Wolski dynamically predicted offloading cost at run-time according to the feedback of a resource monitor [7]. However, some important parameters, such as workload dynamics and bandwidth variability, are not included, which may lead to inappropriate predictions and incorrect offloading decisions. Our work, on the other hand, achieves a higher accuracy by modeling these important parameters. According to our experiment results, fewer incorrect offloading decisions are made.

Several works developed offloading mechanisms by integrating exiting software packages rather than started from scratch [15, 16, 18]. R. Kemp *et al.* [15] used Ibis middleware to offload computational intensive Java programs to remote servers. Y. Zhang [18] adopted Firefox plug-in framework to transparently offload computations to remote servers. Since these works are closely coupled with specific software packages, it becomes difficult to extend their methods to other execution environments. Y. N. Lin [16] explored the possibility of offloading programs to network processors in order to reduce execution time. They first profiled the IPSec module to identify bottlenecks, and then rewrote IPSec-related kernel and driver code. Although the performance improvement of network throughput can reach as much as 350%, the energy consumption of network processors may significantly increase. In addition, a modification of OS kernel and drivers is required, which reduces the portability of the proposed method. In this work, we realize our idea of offloading by developing a Linux program at user space in order to increase the portability. We do not rely on any specific software packages. In addition, we do not require any modifications of OS or drivers. Our method can be easily ported to other execution environments, such as Windows Embedded Compact 7.

Works on Energy and Time Saving

Both energy and time saving are crucial design objectives of smart phones. However, few research efforts have been devoted to optimize the two objectives simultaneously [19, 20]. C. Wang [20] used similar profiling-partitioning technology to identify offloaded parts and consider energy and time saving at the same time. A similar method was developed by A. P. Miettinen [19] to offload the most power hungry parts in order to reduce energy consumption. However, both of them use execution time of a program to approximate its energy consumption. The estimated energy consumption, without considering the parameters of CPUs, may be incorrect. In this work, we provide a higher accuracy energy and execution model by considering important parameters of CPU and offloading targets. Our experiment results indicate that the proposed method can achieve better performance in saving energy and time.

2.2 Overview of Android Architecture

Android is a software stack for a mobile device that includes an operating system, middleware and mobile applications. After being acquired by Google, Android has attracted thousands of developers' attention for its open license and flexible architecture. Android OS is released under Apache 2.0 software license, so that anyone can download, modify, and even redistribute the source code. Because of its characteristic of freedom, Android becomes a superb platform adopted by both academe and industry. In addition, Android also has already been ported to numerous hardware platforms. Figure 1 shows the overview of Android multi-layer structure, which includes five layers: Applications, Framework, Libraries, Runtime Environment, and Linux Kernel.
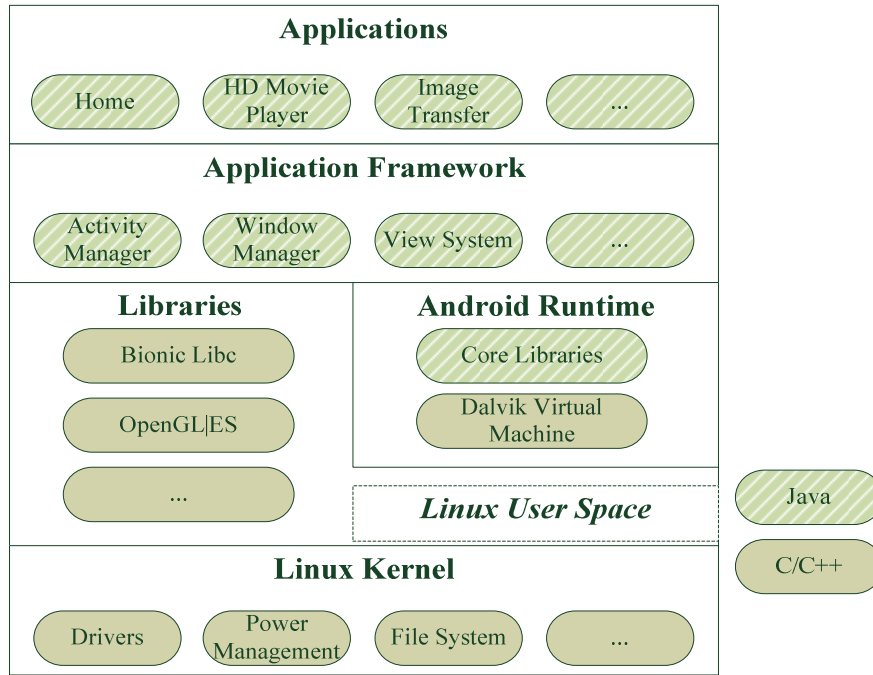
Figure 1. Android Architecture

Android, using a modified Linux kernel, differs in drivers and memory management, but it still keeps the original advantages of robustness and efficiency. Besides kernel, Android adopts Dalvik VM and Bionic C library in order to adapt to resource-limited devices.

As Figure 1 shows, Dalvik VM is a light-weight Java virtual machine, which runs applications at the top level and application framework at the second level. There are three differences between Dalvik VM and traditional J2ME VM. First of all, unlike J2ME VM, each application running on Dalvik VM is associated with one VM. Second, Dalvik VM is register-based rather stack-based. Third, Dalvik VM only accepts Dalvik executable format. Similarly, Bionic Libc is a modified version of GNU Libc with some performance optimization, but it still remains the original characteristics of variety and multifunction.

However, although Dalvik VM is a light-weight VM, running programs on a virtual machine can induce performance degradation. As a result, our computation modules are implemented at Linux user space (the dotted block) in order to avoid the

interference caused by the virtual machine.

2.3 OpenGL|ES

OpenGL|ES, a Khronos-developed graphics standard of embedded system, is derived from OpenGL. Nowadays, almost every smart phone supports OpenGL|ES as a rendering engine. However, some of them only support OpenGL|ES 1.x, which can only perform fixed pipeline function. Fortunately, many new GPU chipsets now can support OpenGL|ES 2.0, shown in Figure 2. Since the functionalities of OpenGL|ES 2.0 are more powerful than the OpenGL|ES 1.x, we focus on OpenGL|ES 2.0 in this work.
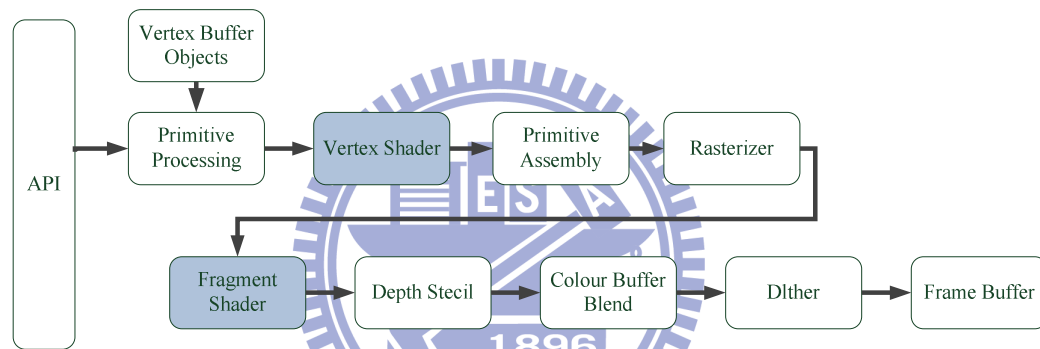

Figure 2. OpenGL|ES 2.0 programmable pipeline

Figure 2 gives an overview of graphics processing flow in GPU. The vertex shader provides a programmable method to operate vertex, usually for projection or lighting. To operate vertex, developers need to write shading program [21], similar to C, and compile the program as vertex shader. Then, each time we feed vertex shader with vertices, the shader will calculate the result and then transmit it to primitive assembly blocks. The fragment shader is to operate fragments, which is produced by previous vertices, usually used for painting. Because of the flexibility of shader program, some works [12] offloaded general-purpose computation onto GPU. In this work, we implement a matrix multiplication module by OpenGL|ES 2.0 shading language.

Chapter 3. Problem Statement

We design an offloading framework for smart phones to determine an execution unit, such as CPU, coprocessor, or cloud, for frequently-used modules. In the following, Section 3.1 first describes each model of mobile application, CPU, coprocessor and cloud. Then, Section 3.2 gives the problem statement.
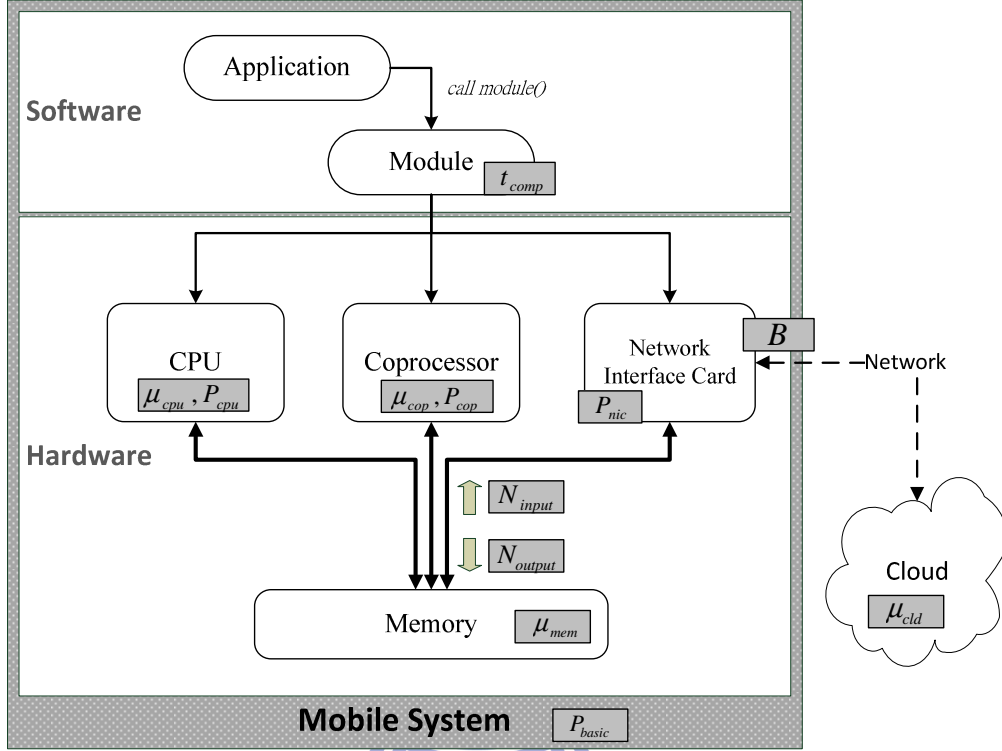
Figure 3. Factors in System

3.1 System Model

Mobile applications usually adopt frequently-used modules, such as FFT, convolution, and matrix multiplication, to process data. As Figure 3 shows, the application first involves a module to process data with input size of N_{input}, which is stored in the memory. The data is then processed by CPU, coprocessor or cloud. After data processing, the output data, with size of N_{output}, is stored in the memory. The bandwidth of memory access is μ_{mem}, at which the data is read from or written into the memory by CPU, coprocessor or network interface. If the data processing is

offloaded to the cloud, the transmission speed is B. We use μ_{cpu} to denote the speed of CPU, μ_{cop} to denote that of coprocessor and μ_{cld} to denote that of cloud. In addition, the power consumption of CPU is P_{cpu}, of coprocessor is P_{cop} and of network interface is P_{nic}. When the system is idle, its power consumption is P_{basic}.

Table 2. Notation Table

| Cost Function			Definition
$target$			Major unit for computation, e.g. CPU/Coprocessor/Cloud
$T_{t\arg et}$			Measured execution time when offload to target
$\hat{T}_{t\arg et}$			Estimated execution time when offload to target
$E_{t\arg et}$			Measured energy consumption when offload to target
$\hat{E}_{t\arg et}$			Estimated energy consumption when offload to target
$F_{t\arg et} = \{\hat{T}_{t\arg et}, \hat{E}_{t\arg et}\}$			Set of time and energy equations on target
f^{α}			Decision function with weight α
Decision Factor	**Unit**	**Variable**	**Definition**
B	Kbps	O	Transmission bandwidth
t_{comp}	Second	O	Module execution time on mobile CPU
N_{input}	KB	O	Amount of processing data into processing unit
N_{output}	KB	O	Amount of resulting data from processing unit
μ_{cpu}	MHz	X	Mobile CPU speed
μ_{cop}	MHz	X	Mobile Coprocessor speed
μ_{cld}	MHz	O	Cloud speed
μ_{mem}	Mbps	O	Memory access bandwidth
P_{basic}	Watt	X	Basic power when idle
P_{cpu}	Watt	X	Mobile CPU running power
P_{cop}	Watt	X	Mobile Coprocessor running power
P_{nic}	Watt	X	Network Interface power consumption

From these decision factors, the cost functions can be calculated. Both time and energy functions have estimation value ($\hat{T}_{t\arg et}$, $\hat{E}_{t\arg et}$) and measurement value ($T_{t\arg et}$, $E_{t\arg et}$) for evaluating the accuracy. Finally, the decision function f^{α} decide where to offload with user's preference α. Table 2 lists all cost functions,

decision factors, and their definitions.

3.2 Problem Description

Problem statement:

Given decision factors B, t_{comp}, N_{input}, N_{output}, μ_{cpu}, μ_{cop}, μ_{cld}, μ_{mem}, P_{basic}, P_{cpu}, P_{cop}, and P_{nic}, design a decision function f^{α}, where α is user's preference, to choose the best one from three offloading cases: F_{cpu}, F_{cop}, or F_{cld}, to execute module, aims to improve performance of module while conserving energy. Assume no queues, i.e., single tasking.
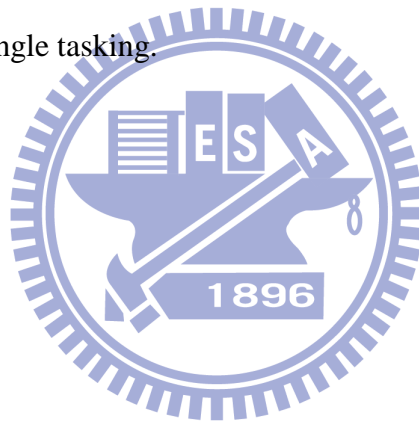
Chapter 4. Time-and-Energy Aware Ternary Decisions

In this chapter, we first give an overview of our offloading framework, which is called time-and-energy aware ternary decision (abbreviated as TETD), and then present factor measurement and ternary decision making respectively.

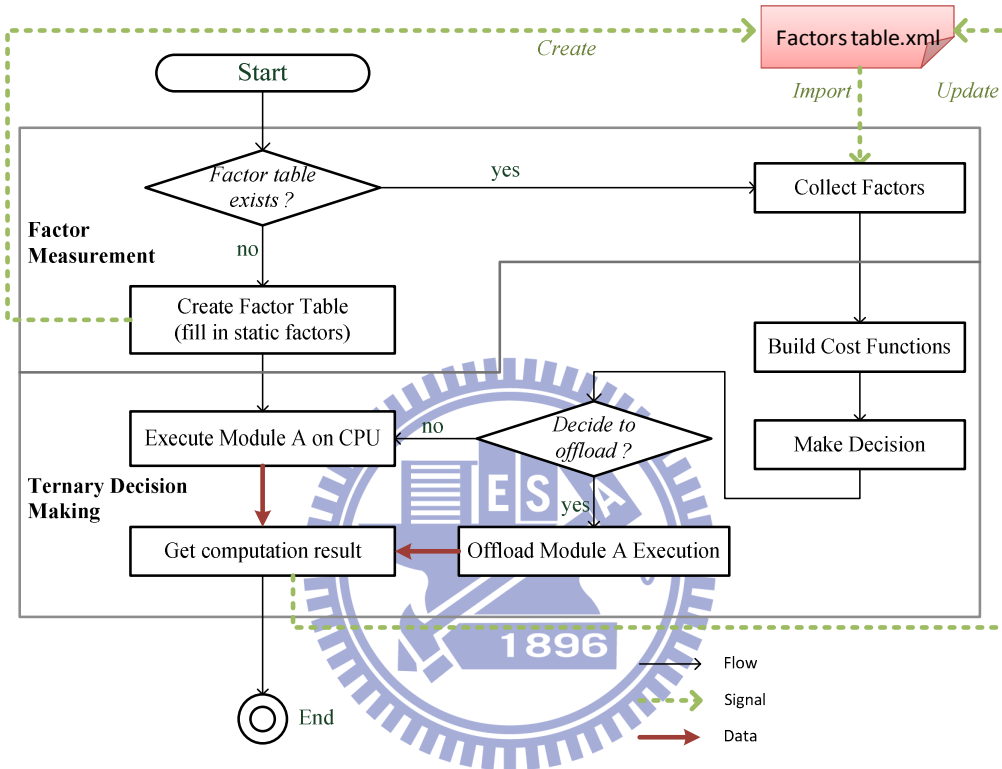## 4.1 Overview of the TETD Flow

Figure 4 Design Flowchart of TETD

Our offloading framework includes two parts: factor measurement and ternary decision making. As Figure 4 shows, when a module is invoked, TETD first check the associated factor table of the module, which stores necessary parameters to estimate energy consumption and execution time. If the factor table does not exist, TETD execute the module on CPU directly and create a corresponding factor table for future need. On the other hand, if the associated factor table is found, TETD extract necessary factors from the table and pass the information to the cost functions. Based on the result of cost functions, the decision maker then determines whether the module should be offloaded or not. After the module is finished, TETD write run-time

15

collected information back to the associated factor table for future use.

4.2 Create and Update Factor Table

In order to correctly estimate the energy consumption and execution time of a module on different execution units, we dynamically create and update a factor table for the module at runtime. A factor table is created when the module is involved at the first time, and is updated when the module is finished. If the same module is invoked again, we refer to its associated factor table to estimate the cost of offloading. The factor table stores both static and dynamic decision factors. The static decision factors include μ_{cpu}, μ_{cop}, and power parameters, which are deterministic and module independent. The dynamic decision factors include B, N_{input}, N_{output}, μ_{cld}, μ_{mem}, and t_{comp}, which are uncertain or module dependent. For dynamic decision factors, we develop a monitor to collect the information at run-time and update the associated factor table when the module is completed.

4.3 Ternary Decision

In this subsection, we first discuss the execution time and energy consumption of a module when it is executed on three difference execution environments: local CPU, local GPU and cloud. We then introduce the algorithm used for making decision.

Cost Functions: Execution Time and Energy Consumption

First of all, we consider the execution time of a module, which is executed on a local CPU. We divide the execution time into two parts. The first part is transmission time t_{trans}, which is used for fetching data from and writing them back to memory. Since t_{trans} depends on the amount of processed data, we have

$$t_{trans} = \frac{N_{input} + N_{output}}{\mu_{mem}}.$$

The second part is pure computation time t_{comp}, which is used by the CPU to execute

codes. Therefore, the execution time is

$$\hat{T}_{cpu} = t_{trans} + t_{comp} . \tag{1}$$

In our experiments, μ_{mem} is set by run-time measuring. In addition, t_{comp} is obtained by $\hat{T}_{cpu} - t_{trans}$. Based on Eq.(1), the energy consumption of the local CPU is calculated by

$$\hat{E}_{cpu} = (P_{basic} + P_{cpu}) \times \hat{T}_{cpu} . \tag{2}$$

Similarly, when the module is executed on a local GPU, the execution time $\hat{T}_{cop}$ is

$$\hat{T}_{cop} = t_{trans} + \frac{t_{comp} \times \mu_{cpu}}{\mu_{cop}} . \tag{3}$$

Also, the energy consumption $\hat{E}_{cop}$ is

$$\hat{E}_{cop} = (P_{basic} + P_{cop}) \times \hat{T}_{cop} . \tag{4}$$

After considering the above two cases, we now discuss the execution time and energy consumption in the case of offloading to the cloud. We defined $\hat{T}_{cld}$ as the execution time of the module when it is offloaded to the cloud. In order to calculate $\hat{T}_{cld}$, we first determine the amount of data to be transmitted by

$$\sigma = \left\lceil \frac{N_{input} + N_{output}}{MTU} \right\rceil \times (DATA\ Packet\ Size), \tag{5}$$

in which *MTU* stands for the maximum transmission unit. Also, the ACK packets used during the transmission is determined by

$$\sigma^{ack} = \left\lceil \frac{N_{input} + N_{output}}{MTU} \right\rceil \times (ACK\ Packet\ Size). \tag{6}$$

Then, $\hat{T}_{cld}$ is calculated by

$$\hat{T}_{cld} = \frac{\sigma + \sigma^{ack}}{\mu_{mem}} + \frac{\sigma + \sigma^{ack}}{B} + \frac{t_{comp} \times \mu_{cpu}}{\mu_{cld}} . \tag{7}$$

In Eq.(7), the first term on the right hand side is the time spent for fetching data from and writing them back to memory. The second term is network transmission time. The third term is the time spent on the cloud. The energy consumption is then determined by

$$\hat{E}_{cld} = (P_{nic} + P_{basic}) \times \left(\frac{\sigma + \sigma^{ack}}{\mu_{mem}} + \frac{\sigma + \sigma^{ack}}{B} \right) + P_{basic} \times \left(\frac{t_{comp} \times \mu_{cpu}}{\mu_{cld}} \right), \quad (8)$$

in which, P_{nic} is the power consumption of network interface.

Decision Making

At run-time, we dynamically measure the value of t_{comp}, N_{input}, N_{output} and B. According to the collected information, we then use the above-mentioned equations to predict the execution time and energy consumption of the module when it is executed in different execution environments. A compound objective function is developed for user to adjust the importance of time and energy. As Figure 5 shows, we calculate the value of the objective function in different cases (line 3, 4 and 5) and take the minimum as the offloading decision.

Procedure Decision_Maker (f^{α})

Input: (1) $\hat{T}_{cpu}$, $\hat{T}_{cop}$, $\hat{T}_{cld}$ (2) $\hat{E}_{cpu}$, $\hat{E}_{cop}$, $\hat{T}_{cld}$ (3) $0 \le \alpha \le 1$

Output: execution unit

Procedure:

(1) Initialize MIN_VALUE to zero

    Initialize TARGET to CPU

(2) While there is offloading target P do

    Calculate $\varepsilon_T^P = \dfrac{\hat{T}_P - \hat{T}_{cpu}}{\hat{T}_{cpu}}$ and $\varepsilon_E^P = \dfrac{\hat{E}_P - \hat{E}_{cpu}}{\hat{E}_{cpu}}$

    If MIN_VALUE < $\alpha \cdot \varepsilon_T^P + (1-\alpha) \cdot \varepsilon_E^P$

        Set MIN_VALUE to $\alpha \cdot \varepsilon_T^P + (1-\alpha) \cdot \varepsilon_E^P$

        Set $TARGET$ to P

    End If

    End While

(3) Return TARGET

Figure 5 Pseudo Code of Decision Maker

Chapter 5.   Implementations

In this chapter, we first introduce the device under test (DUT). We then describe the methods used to measure bandwidth, component speed, computation time and power consumption respectively.

5.1 Device Under Test

We adopt HTC Nexus One, a popular and powerful smart phone, as our DUT. The Nexus One quips with a Qualcomm QSD8250™ 1GHz processor, a 512 MB Flash ROM, a 512 MB RAM and a Wi-Fi IEEE 802.11 b/g interface. The operating system used in the Nexus One is Android 2.2. We implement our offloading framework in C language on user space so that it can operate without any privilege-restrictions and becomes more efficient. Our user-level implementation can be easily ported to other operating systems, such as Windows Embedded Compact 7.

5.2 Factor Measurement

Wireless Bandwidth: B

As mentioned in the previous chapter, the wireless bandwidth B is a crucial decision factor in our offloading framework. In order to adapt to environment changes, we dynamically measure the transmission bandwidth at run-time. Many tools have been developed to measure the transmission bandwidth. Some of them are platform dependent, such as *Iperf* and *ttcp*, while others are platform independent. To make our method easily applicable to all Android smart phones, we use the popular utility *ping*, located at the folder /system/bin, to measure the network bandwidth. For each measurement, the *ping* utility first sends the packets of ICMP-Request from the Nexus One to the cloud and then receives the packets sent back by the cloud. In our experiment, we use the command "ping -c 10 -s 6000 -i 0.1 cloud" to issue 10 ICMP requests, in which the amount of data to be sent is 6000 bytes and the wait interval between sending each packet is 0.1 sec. As Figure 6 shows, the measurement starts at

time t_1 and stops at t_2. For each ICMP request, it needs four IP packets and one ICMP packet because the maximal size of the data in the Ethernet-frame is 1500 bytes. Similarly, four IP packets and one ICMP packet are required to send an ICMP reply back to the Nexus One. If t_{2-t_1} is 100ms, the network bandwidth is approximated by

$$B = \frac{(4 \times 1514 + 1 \times 122) \times 2 \times 10 \times 8}{t_2 - t_1} = 9894 \; Kbps.$$
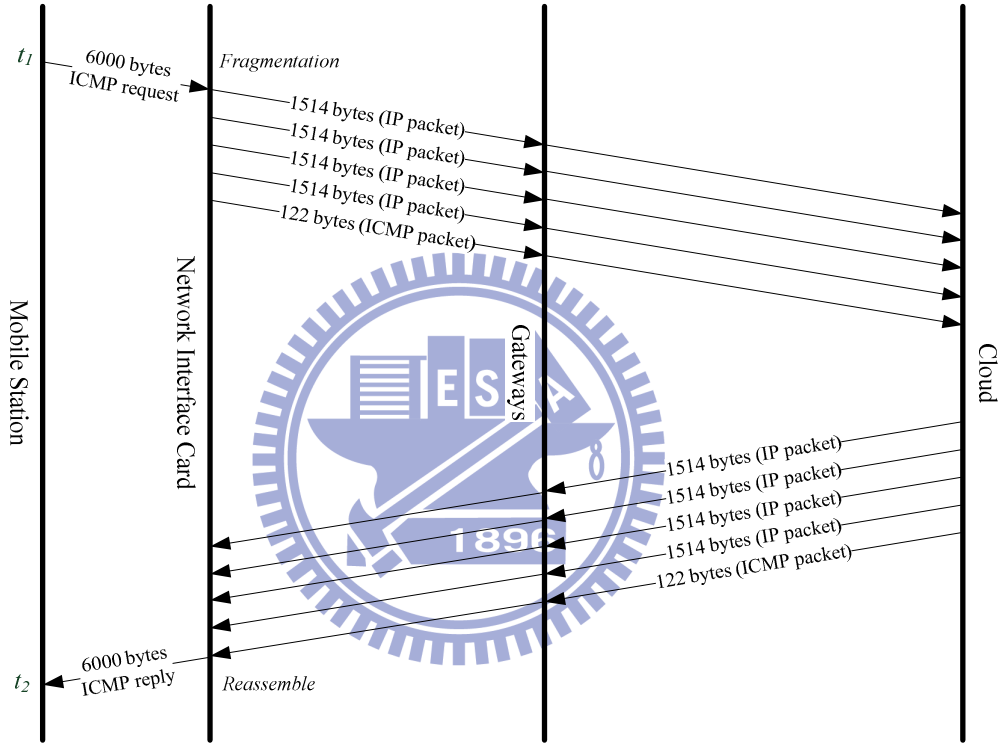

Figure 6. ICMP Request & Reply with Payload Size 6000 bytes

Component Speed: μ_{cpu} , μ_{cop} , μ_{cld} , μ_{mem}

In our experiments, we obtain the local CPU speed μ_{cpu} and the local GPU speed μ_{cop} by referring to datasheet. On the other hand, we measure the cloud speed μ_{cld} and memory bandwidth μ_{mem} at run-time. In order to estimate μ_{cld}, we ask the cloud to measure the time t_{cld} spent in executing the offloaded program. Then, we calculate μ_{cld} by $\mu_{cld} = (t_{comp} / t_{cld}) \mu_{cpu}$. The memory bandwidth is estimated by measuring the time of accessing a large amount of data stored in the memory. For example, if it takes 20ms to read 1,000,000 16-bit integers from the memory, the memory bandwidth μ_{mem} is

estimated by

$$\frac{1{,}000{,}000 \times 16 \text{ (bits)}}{20 \text{ (ms)}} = 800\text{Mbps}.$$

Computation time: t_{comp}

As mentioned in Chapter 4, t_{comp} represents the pure computation time used by the CPU to execute codes. We calculate t_{comp} by

$$t_{comp} = T_{cpu} - t_{trans},$$

in which T_{cpu} is the total execution time and t_{trans} is the memory transmission time. In order to obtain T_{cpu}, we insert the Android-supported function *clock_gettime()* at the beginning and the end of the program, and then calculate the difference. The value of t_{trans} is obtained by

$$t_{trans} = \frac{N_{input} + N_{output}}{\mu_{mem}}.$$

The definition of t_{comp} is similar to "worst case execution time" and only those regular computations are worthy to be offloaded.

Power: P_{basic}, P_{cpu}, P_{cop}, P_{nic}

Due to the hardware limitation, we are not able to measure the energy consumption of each component directly. Instead, we design four different scenarios: 1) idle system, 2) execute CPU-bound workload, 3) execute GPU-bound workload, 4) send a large amount of data to the cloud, and use an Android daemon-maintained battery log, located at /sys/kernel/debug/battery_log, to obtain power information. As Figure 7(a)(b) shows, the Andriod daemon updates voltage, current and power information every 50 sec. Compared to other profiling methods, our method has two advantages. First, it is available on all Android smart phones. Second, no extra hardware equipment is required, such as data acquisition (DAQ) card.

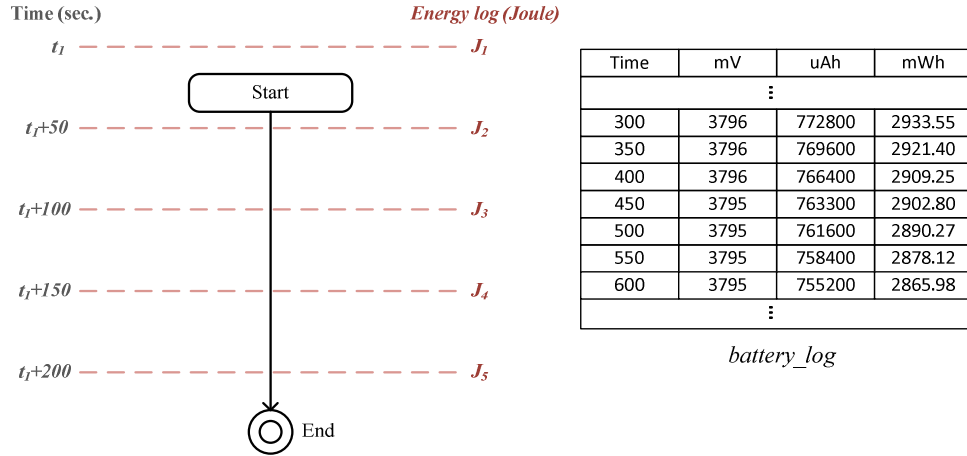| Time	mV	uAh	mWh
⋮			
300	3796	772800	2933.55
350	3796	769600	2921.40
400	3796	766400	2909.25
450	3795	763300	2902.80
500	3795	761600	2890.27
550	3795	758400	2878.12
600	3795	755200	2865.98
⋮			

battery_log

Figure 7. (a) Energy Measurement in Android   (b) Example of Measured Value

In scenario 1, we close all unnecessary user programs and keep the system idle for a while. Figure 7(b) illustrates a battery log of system idle. Based on the log, the energy consumption of the system in [400s, 450s] is

$$\frac{3796 \times 766400 - 3795 \times 763300}{1000 \times 1000} = 12.53 \text{ mWh} (= 45.1 \text{ joule}).$$

Therefore, we have P_{basic}

$$P_{basic} = \frac{45.1 \text{ joules}}{50 \text{ seconds}} = 902 \text{ mW}.$$

A similar approach is used in other scenarios. In scenario 2, we execute a CPU-bound program to make the CPU busy. In scenario 3, we execute an OpenGL|ES 2.0 program on the GPU and keep the CPU idle. In scenario 4, we send data to the cloud for a long period. The measurement results are listed in Table 3.

Table 3. Power Values in Nexus One

	P_{basic}	P_{cpu}	P_{cop}	P_{nic}
power (W)	0.886	1.539	1.056	2.262

Chapter 6. Experiment and Evaluation

In this chapter, we first introduce the experiment environment. Next, Section 6.2 measures the overhead of the proposed decision framework. Finally, Section 6.3 and 6.4 adopt two case studies, matrix multiplication and virus scanning, to evaluate the proposed method.
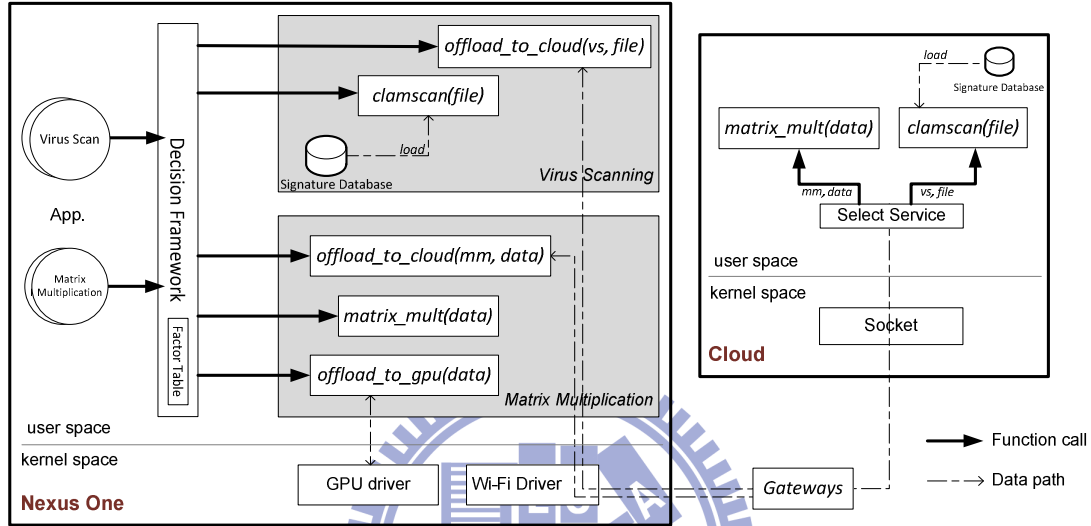

Figure 8. Experiment Environment

6.1 Testbed

Figure 8 illustrates the experiment environment, which includes a Nexus One smart phone and a cloud. In order to eliminate uncertainty and unpredictability, we set backlight always on and close unnecessary processes. We implement our decision framework on the Nexus One and install virus scanning and matrix multiplication applications for experiment. Each application has one or more functions, which can be offloaded to the cloud. A PC with 2.4-GHz Intel processor and 4-GB RAM is used to simulate the execution environment of cloud, and the operating system of the cloud is Linux 2.6.35.

6.2 Evaluation of Decision Framework

In order to understand the overhead of our decision framework, we measure the execution time and energy consumption of each function respectively. As Table 4

23

shows, creating factor table and collecting factors consume significant energy if the factor table does not exist. For example, the function *Create Factor Table* consumes 43.03% of total execution time and 33.79% of total energy consumption. On the other hand, if the factor table is already created, most of time (98.54%) and energy (98.99%) are spent in the *Collect Factor*. Since the function *Create Factor Table is* only executed once, the function *Collect Factor* dominates the overhead of our decision framework. We further breakdown the energy consumption of the function *Collect Factor.* As figure 9 shows, collecting the information of bandwidth consumes most of time and energy. This overhead is induced by the *ping* program and Wi-Fi driver, which can be further optimized in the future.

Table 4. Proportions of Time and Energy in Decision Framework

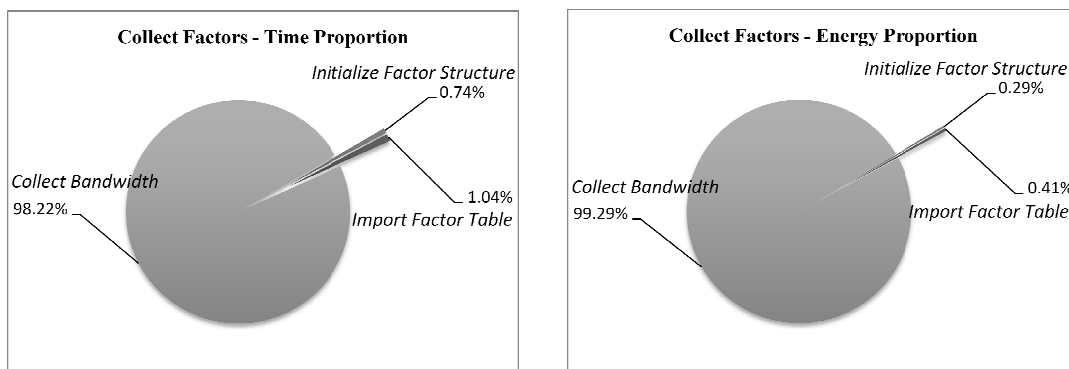| Factor Table Exist? Functions	No		Yes	
	Time	Energy	Time	Energy
Create Factor Table	43.03%	33.79%		
Collect Factors	56.14%	65.53%	98.54%	98.99%
Build Cost Functions	0.32%	0.25%	0.55%	0.38%
Make Decision	0.20%	0.16%	0.34%	0.24%
Update Factor Table	0.32%	0.27%	0.56%	0.39%
Total	1641 ms	3135 mJ	935 ms	2075 mJ

Figure 9. Proportions of Time and Energy in *Collect Factors*

6.3 Case Study: Matrix Multiplication

Matrix multiplication is a CPU-intensive module, which has been widely used in the applications of encoding, decoding, image compression and rotation. In order to evaluate the energy consumption, we implement three versions of the matrix multiplication for different execution environments. One is for the execution of local CPU, another two are for local GPU and Cloud. In the version of local CPU, it includes three steps: reading data, processing the multiplication and storing the results. As mentioned in Section 2.3, we implement two programs in the version of GPU. One is vertex shader and another is fragment shader. Whenever the function *offload_to_gpu()* is invoked, this function first communicates with the GPU driver and compiles shader codes in to the executable format of GPU. The data are then feed to GPU and written back to the buffer *pBuffer* after they are processed. The function *offload_to_gpu()* finally terminates the communication.

In order to offload computation to the cloud, on the smart phone, we implement a function, named *offload_to_cloud()*, to send offloaded data to the cloud. On the site of cloud, another service is deployed to receive the offloaded module and forward them to a proper function. As shown in figure 8, for the matrix multiplication, the module is first forwarded to the function *matrix_mult()*. Then, the results are sent back to the smart phone.

Estimation Accuracy

This subsection evaluates the accuracy of our method in approximating the execution time of an offloaded module in difference execution environments. We first measure the execution time of the offload module by varying the matrix size. We then compare our estimated execution time with measurement data.

In this work, the network system for experiment is Wi-Fi instead of 3G network. This can be explained from figure 10 that the speed of 3G network is too slow, and

cannot express the power of cloud computing. Then, we choose Wi-Fi as network media.
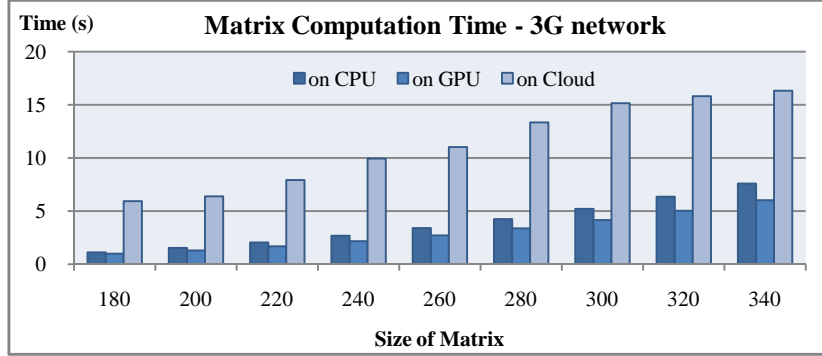

Figure 10. Matrix Multiplication via 3G network

Figure 11(a) shows the measurement results of execution time, in which the x-axis is the size of matrix and the y-axis is the execution time. For example, in the case of 320x320, the speedup can achieve 1.27 and 3.89 when the computation is offloaded to local GPU and the cloud. Figure 11(b), similarly, shows the measurement results of execution energy, in which the x-axis is the size of matrix and the y-axis is the energy value.
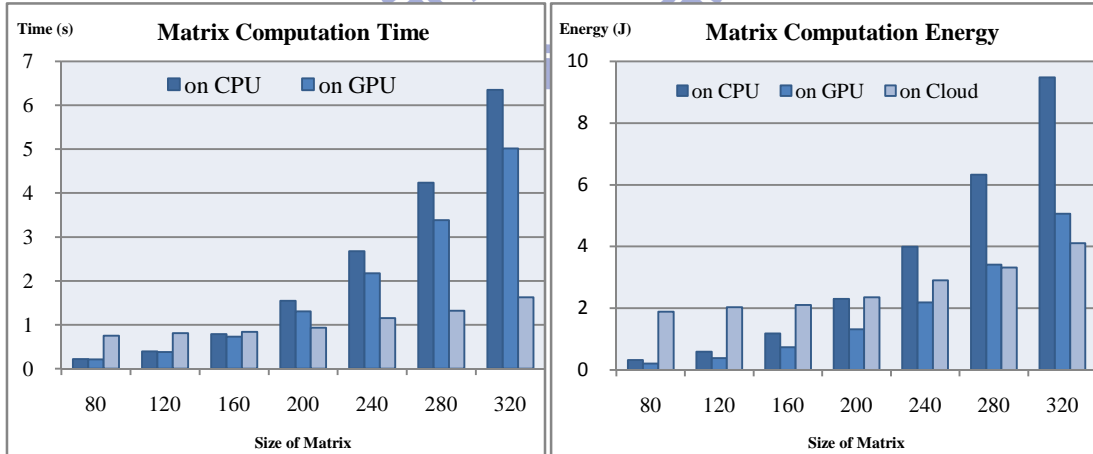

Figure 11. Matrix Multiplication (a) Time (b) Energy Cost

We define the *error rate* of execution time as

$$err(target) = \left| \frac{\hat{T}_{target} - T_{target}}{T_{target}} \right|, target \in \{CPU, GPU, Cloud\},$$

in which $\hat{T}_{target}$ is the estimated execution time, which is obtained by the

26

equations mentioned in Chapter 4. As figure 12(a) shows, the error rate becomes larger when the size of matrix is smaller. This is because the overhead of OS context switch cannot be ignored when the execution time of matrix multiplication is short. The same method can be used in energy evaluation and the result is shown in figure 12(b). According to our experiment results, when the size of matrix is larger then 80, the error rate is less than 20%.
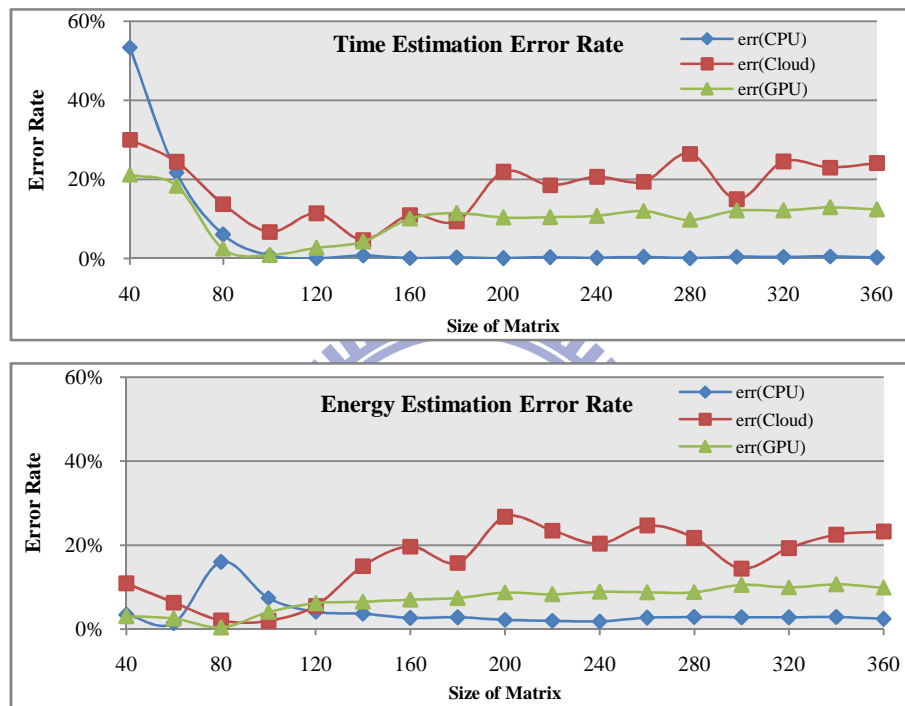

Figure 12. Matrix Multiplication (a) Time (b) Energy Estimation Error Rate

Decision Accuracy

Since the error rate can result in an incorrect offloading decision, we define the *false decision rate* as

$$false\ decision\ rate = 1 - \frac{\#\ of\ correct\ decisions}{\#\ of\ decisions}$$

In our experiment, we vary the value of alpha in 0.0 to 1.0 and fix the matrix size in 100, 200, and 300. As figure 13 shows, when the matrix size is 100, false decision rate is nearly 40% when alpha is 0.8. This is induced by closed estimation values of processing units. In other words, when matrix size becomes larger, the false decision

27

rate is smaller. For example, when alpha is 0.6, the false decision rate of size-300 matrix and size-200 matrix are smaller than that of size-100 matrix. Although our decision function cannot always deliver the optimal decision, it ensures the execution time and energy consumption can be reduced after offloading modules.
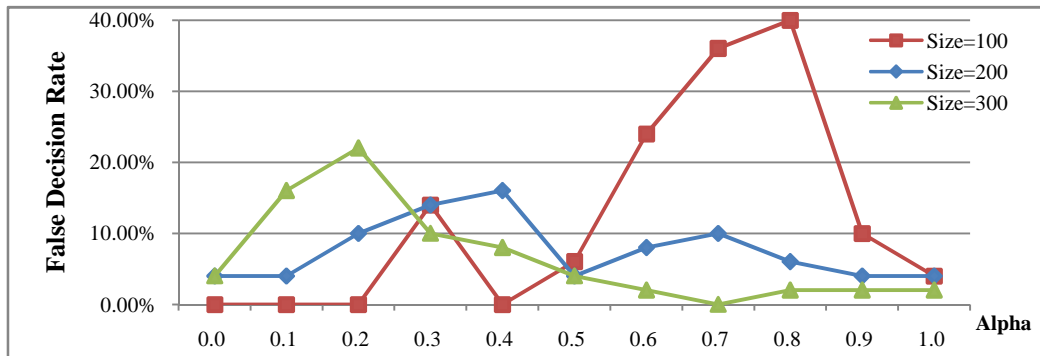

Figure 13. Matrix Multiplication False Decision Rate

Compared with other works, our decision method also delivers the best performance among all works. For instance, in the case of size-300 matrix, the previous works focusing on saving execution time will always offload the computation to the cloud. In addition, the previous works focusing on reducing energy consumption will always offload the computation to the GPU. All these methods cannot satisfy user's expectation, and can result in high false decision rate because of selecting the worst module.
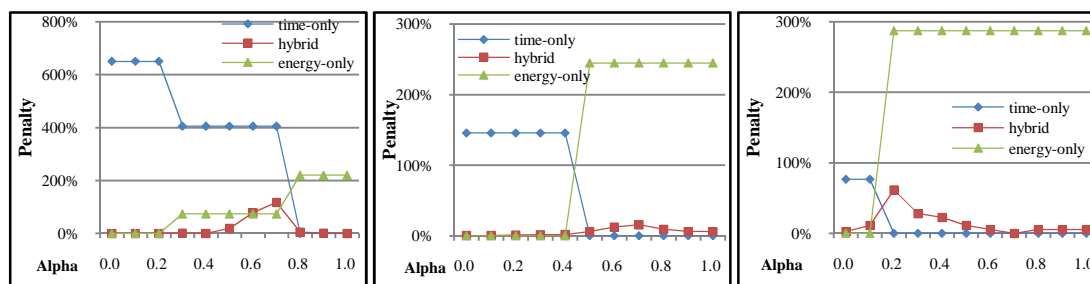

Figure 14. False Decision Penalty of (a) size-100 (b) size-200 (c) size-300

The penalty bring from false decision are shown in figure 14, in which the x-axis is the alpha value and y-axis is the penalty, i.e. the sum of time error rate and energy error rate, from wrong decision. The term "hybrid" is TETD and works very well in

28

different sizes with low penalty. The other two methods, i.e. time-only and energy-only, are usually suffered from high penalty since they only considered the partial of user's preference.

Evaluate Decision Overhead

We have analyzed the overhead of our decision framework in Section 6.2. Now we are going to evaluate the impact of the overhead on energy and time reduction. According to our experiment results, when matrix size is 100, the execution time is 197ms if the module is executed on the local CPU. On the other hand, the execution time becomes 118ms if it is uploaded to the cloud. Since it takes extra 953ms to complete the execution of the proposed method, the total execution time becomes 1053 when the module is offloaded to the cloud. In this case, performing local computation is much better than offloading the computation to the cloud. According to our experiment results, in order to save energy and time by offloading, the size of matrix size should be larger than 250.

6.4 Case Study: Virus Scanning

Virus scanning becomes more and more important for mobile phone. A typical virus scanning process on the mobile device includes three steps. First, the anti-virus program loads a signature database from flash ROM or remote server. Second, it reads the scanned file. Third, it compares the content of the scanned file and the signature. Unlike the matrix multiplication, scanning two files with the same size may consume different time and energy because of the file contents are not the same.

In our experiment, we ported the well-known anti-virus program ClamAV to Android, and install the same version of ClamAV in the cloud. Similar to the previous case study, we implement one native module *clamscan()* and one offloading module *offload_to_cloud()* on the mobile phone. The service name is *vs* (virus scanning). Also, cloud provides a service *clamscan() to* receive the file and return the result.

Estimation & Decision Accuracy

To validate the availability of our decision framework on virus scanning, we use three files for testing. One is *mediaserver*, which is an 5KB-size executable file in Android system. Others are two linkable libraries *libffmpegsumo.so* and *libpdf.so*, which are used by Google Chrome browser. Table 5 lists the experiment results, including the execution time and estimation error rates for each program. According to our experiment result, our decision framework performs well, even in the case of two targets only (CPU and cloud). Therefore, the proposed method is flexible and can be applied to multiple offloading targets.

Table 5. Virus Scanning Execution Time and Error Rate

File	Size	T_{cpu} (ms)	T_{cld} (ms)	err(CPU)	err(Cloud)
/system/bin/mediaserver	5 KB	400	90	0.17%	10.31%
libffmpegsumo.so	2 MB	1,274	3,320	0.07%	9.66%
libpdf.so	15MB	6,029	22,035	0.02%	11.58%

In particular, the estimated execution time in local CPU is highly accurate since the error rate is less than 1%. In addition, the error rate in estimating cloud execution time is low, which is 10% in average. These lower errors rates deliver lower false decision rates, which are almost 0% among different alpha values.

Benefits from Offloading

As table 5 shows, virus scanning can benefit from offloading only if the file size is small. In the case of offloading a large file, such as libpdf.so is 15MB, both the execution time and energy consumption are increased. This is because file transmission consumes significant time and energy consumption. The situation becomes worse in a low speed 3G network. Refer to Table 4, the TETD overhead of *mediaserver* is 935 ms, which is not worthy to offload. Hence, for mobile devices, computation offloading may not be always suitable for virus scanning applications.

Chapter 7.   Conclusions and Future Work

In this work, we design and implement a decision framework for computation offloading. The decision is based on estimated execution time and energy values. We aim to save both execution time and energy consumption at the same time. Unlike previous works, which consider only binary decisions, our ternary decision is suitable for multiple offloading targets.

In our experiment, we present two case studies to validate the applicability of different situations. Based on our decision framework, the matrix multiplication module tends to be offloaded to more powerful processors, such as local GPU or cloud. By offloading modules, we can achieve about 20~300% saving in execution time and 50~130% in battery usage. For the case of virus scanning, offloading either small or large files cannot reduce energy and time. As a result, the virus scanning program should not be offloaded to cloud. Our results also demonstrate high accuracy and false decision rates of the proposed decision framework. Generally speaking, the error rate is less than 20%, and false decision rate is less than 30% in most cases.

In the future, we plan to implement a light-weight *ping* function in order to reduce the overhead in collecting bandwidth. Moreover, we will adopt more wireless technologies, such as LTE or WiMAX, and more applications to evaluate the proposed offloading decision. Since our method assumes there is single tasking in handheld devices, if there are more tasks running on devices simultaneously, our method might be invalid.

References

[1]   DRAMeXchange, "Booming Popularity of Smartphone Helps to Increase NAND Flash Demand," 2011.

[2]   X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic, "Adaptive Offloading for Pervasive Computing," *IEEE Pervasive Computing, vol. 3, pp. 66-73, September 2004.*

[3]   Z. Li, C. Wang, and R. Xu, "Computation Offloading to Save Energy on Handheld Devices: A Partition Scheme," *in International Conference on Compilers, Architecture and Synthesis for Embedded System, pp. 238-246, November 2001.*

[4]   S. Ou, K. Yang, and J. Zhang, "An effective offloading middleware for pervasive services on mobile devices," *Pervasive and Mobile Computing, vol. 3, pp. 362-385, August 2007.*

[5]   G. Chen, B. T. Kang, M. Kandemir, N. Vijaykrishnan, M. J. Irwin*, et al.*, "Studying Energy Trade Offs in Offloading Computation/Compilation in Java-Enabled Mobile Devices," *IEEE Transactions on Parallel and Distributed Systems, vol. 15, pp. 795-809, September 2004.*

[6]   K. Kumar and Y. H. Lu, "Cloud Computing for Mobile Users: Can Offloading Computation Save Energy?," *Computer, vol. 43, pp. 51-56, April 2010.*

[7]   R. Wolski, S. Gurun, C. Krintz, and D. Nurmi, "Using Bandwidth Data to Make Computation Offloading Decisions," *in the 22nd IEEE International Parallel and Distributed Processing Symposium, pp. 1-8, April 2008.*

[8]   E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu*, et al.*, "MAUI: Making Smartphones Last Longer with Code Offload," *in the 8th International Conference on Mobile Systems, Applications, and Services, pp. 49-62, June 2010.*

[9]   S. Han, S. Zhang, and Y. Zhang, "Energy Saving of Mobile Devices Based on Component Migration and Replication in Pervasive Computing," *Ubiquitous Intelligence and Computing, vol. 4159, pp. 637-647, August 2006.*

[10]   Y. J. Hong, K. Kumar, and Y. H. Lu, "Energy Efficient Content-Based Image Retrieval for Mobile Systems," *in International Symposium on Circuits and Systems, pp. 1673-1676, May 2009.*

[11]   B. Seshasayee, R. Nathuji, and K. Schwan, "Energy-Aware Mobile Service Overlays: Cooperative Dynamic Power Management in Distributed Mobile Systems," *in the 4th International Conference on Autonomic Computing, pp. 6-6, June 2007.*

[12]   Y. C. Wang, B. Donyanavard, and K. T. Cheng, "Energy-Aware Real-Time

Face Recognition System on Mobile CPU-GPU Platform," *in the 11th European Conference on Computer Vision, September 2010.*

[13] X. Zhao, P. Tao, S. Yang, and F. Kong, "Computation Offloading for H.264 Video Encoder on Mobile Devices," *in Computational Engineering in Systems Applications, pp. 1426-1430, October 2007.*

[14] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso, "Calling the Cloud: Enabling Mobile Phones as Interfaces to Cloud Applications," *in the 10th ACM/IFIP/USENIX International Conference on Middleware, pp. 1-20, December 2009.*

[15] R. Kemp, N. Palmer, T. Kielmann, F. Seinstra, N. Drost*, et al.*, "eyeDentify: Multimedia Cyber Foraging from a Smartphone," *in the 11th IEEE International Symposium on Multimedia, pp. 392-399, December 2009.*

[16] Y. N. Lin, C. H. Lin, Y. D. Lin, and Y. C. Lai, "VPN Gateways over Network Processors: Implementation and Evaluation," *Journal of Internet Technology, vol. 11, pp. 457-463, 2010.*

[17] K. Yang, S. Ou, and H. H. Chen, "On Effective Offloading Services for Resource-Constrained Mobile Devices Running Heavier Mobile Internet Applications," *IEEE Communications Magazine, vol. 46, pp. 56-63, January 2008.*

[18] Y. Zhang, X. Guan, T. Huang, and X. Cheng, "A Heterogeneous Auto-Offloading Framework Based on Web Browser for Resource-Constrained devices," *in the 4th International Conference on Internet and Web Applications and Services, pp. 193-199, May 2009.*

[19] A. P. Miettinen and J. K. Nurminen, "Energy efficiency of mobile clients in cloud computing," *in the 2nd USENIX Conference on Hot topics in Cloud Computing, pp. 4-4, June 2010.*

[20] C. Wang and Z. Li, "A Computation Offloading Scheme on Handheld Devices," *Journal of Parallel and Distributed Computing, vol. 64, pp. 740-746, February 2004.*

[21] A. Munshi, D. Ginsburg, and D. Shreiner, "OpenGL® ES 2.0 programming guide," Addison-Wesley, 2008.