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

Ying-Dar Lin, *Fellow, IEEE*, Edward T.-H. Chu, *Member, IEEE*, Yuan-Cheng Lai, and Ting-Jun Huang

***Abstract*—Running sophisticated software on smart phones could result in poor performance and shortened battery lifetime because of their limited resources. Recently, offloading computation workload to the cloud has become a promising solution to enhance both performance and battery life of smart phones. However, it also consumes both time and energy to upload data or programs to the cloud and retrieve the results from the cloud. In this paper, we develop an offloading framework, named Ternary Decision Maker (TDM), which aims to shorten response time and reduce energy consumption at the same time. Unlike previous works, our targets of execution include an on-board CPU, an on-board GPU, and a cloud, all of which combined provide a more flexible execution environment for mobile applications. We conducted a real-world application, i.e., matrix multiplication, in order to evaluate the performance of TDM. According to our experimental results, TDM has less false offloading decision rate than existing methods. In addition, by offloading modules, our method can achieve, at most, 75% savings in execution time and 56% in battery usage.**

***Index Terms*—Android, cloud computing, computation offloading, coprocessors.**

I. Introduction

NEARLY 300 million smart phones were sold in 2010, and its number is expected to increase by 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 with 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 technologies, such as instruction-level parallelism, leakage power control, and dynamic voltage scaling, have been proposed to improve processor speed and reduce energy consumption. Although advanced technology can deliver better performance, adopting high-end processors is not always appropriate for budget-limited projects. Recently, cloud computing has become another possible solution to enhance the 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, such as Amazon Elastic Compute Cloud (EC2), Amazon Virtual Private Cloud (VPC), and PacHosting. However, it takes both time and energy to upload data or programs to the cloud and retrieve the results from the cloud. The computation capacity of the cloud can also affect the total execution time. 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 data- or computation-intensive programs from a resource-poor mobile device [2]–[5]. Gu *et al.* [2], Li *et al.* [3], and 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 a significant overhead when partitioning large-size applications. Kumar and Lu [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 of the above works ignored the impact of offloading on execution time, which may result in performance degradation. On the other hand, Ou *et al.* [4] developed an offloading middleware, which provides runtime offloading services to improve the response time of mobile devices. Wolski *et al.* [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 lifetime. In addition, 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 multiobjective optimization problem that simultaneously optimizes these two key performance indexes of smart phones.

In this paper, we develop an offloading framework, named Ternary Decision Maker (TDM), which aims to shorten response time and reduce energy consumption at the same time. Unlike previous works, our targets of execution include an

Manuscript received September 19, 2012; revised May 17, 2013; accepted August 24, 2013. Date of publication November 26, 2013; date of current version May 22, 2015. This work was supported in part by the National Science Council of Taiwan and in part by the Institute for Information Industry.

Y.-D. Lin and T.-J. Huang are with the Department of Computer Science, National Chiao Tung University, Hsinchu 300, Taiwan (e-mail: ydlin@cs.nctu.edu.tw; tjHuang.lec@gmail.com).

E. T.-H. Chu is with the Department of Electronic and Computer Science Information Engineering, National Yunlin University of Science and Technology, Douliou 64002, Taiwan (e-mail: edwardchu@yuntech.edu.tw).

Y.-C. Lai is with the Department of Information Management, National Taiwan University of Science and Technology, Taipei 106, Taiwan (e-mail: laiyc@cs.ntust.edu.tw).

Digital Object Identifier 10.1109/JSYST.2013.2289556

TABLE I
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]	yes	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 & 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

on-board CPU, an on-board GPU, and a 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 end 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, such as network transmission bandwidth, mobile CPU speed, and memory bandwidth, 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.

In order to investigate the applicability and performance of TDM, several experiments were conducted on a popular smart phone: HTC Nexus One. First, we evaluated the overhead of TDM by measuring the execution time and energy consumption of each function. Next, a case study of matrix multiplication was conducted. In this case study, we implemented different versions of the offloaded modules for different offloading targets. In addition, we evaluated the accuracy of our method in approximating the execution time and the energy consumption of offload modules by measurement. Finally, TDM was compared with existing methods in making offloading decisions.

The rest of this paper is organized as follows: Section II introduces related works with computation offloading and some concepts for GPU programming. Section III gives the problem statements and terminologies for the later sections. Section IV proposes the methodologies of offloading decisions, and then, Section V details the implementation. Section VI shows the experimental result and evaluation. Section VII concludes this work and gives important areas of future work.

II. BACKGROUND

Here, we first give a comprehensive comparison between our work and related works. We then introduce OpenGL|ES, which is used in our experiments.

A. Related Works

There have been many research efforts dedicated to offload data- or 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], whereas others targeted 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 I, 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 in four attributes: *adaptability, portability, accuracy*, and *offload target*.

Adaptability indicates the capability of the proposed method to adapt itself efficiently to dynamic workload, resulting from the variance of data input at runtime. If the proposed method can only handle deterministic workload, this shows that its adaptability is poor. *Portability* represents the ability of the proposed method to be ported from one execution environment to another, such as from Linux to Windows. The term language/framework/kernel represents the way we port the method to another platform. For example, if the portability of a work is Language, it implies that some modifications of programming language are required when the work is applied to another platform. On the other hand, if the portability of a work is framework (or kernel), it requires the modification of the framework (or operating system) when the work is applied to another platform. Methods with Language portability are desirable because most of the existing codes can be reused. *Accuracy* is the approximation correctness of the energy model or the execution time model of offloading. Higher accuracy indicates 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 that can process offloaded data or programs. The more targets we have, the more flexible the execution environment will become. According to Table I, 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 the related works in detail.

1) Works on Energy Saving: Maximizing battery lifetime is one of the most crucial design objectives of smart phones because they are usually equipped with limited battery capacity. Some of them adopted profiling–partitioning technology to identify offloaded parts of an application for energy saving [3], [8], [9], [11], [21]. They first profiled the energy consumption of each function of the application. According to the profiling result, they next generated a cost graph, in which each node represented a function to be performed, and each edge indicated the data to be transmitted. The maximum-flow/minimum-cut algorithm or optimization algorithms were then used to partition the cost graph to obtain client and server parts. Finally, the server parts were executed at remote servers for reducing energy consumption of a mobile device. Chen *et al.* designed a similar method to determine whether Java methods and bytecode-to-native code compilation should be executed at remote servers for energy saving [5]. In addition, they assumed that the workload was deterministic, which means that the workload will not vary at runtime. As a result, their methods cannot be applied to dynamic workload, resulting from the variance of data input at runtime.

On the contrary, in order to reduce the profiling overhead, we only profile the energy consumption and execution time of frequently used modules, such as fast Fourier transform (FFT), inverse FFT, 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 runtime.

Some works built energy models to approximate the energy consumption of offloading [6], [10], [13], [22]–[24]. The energy models can be used to construct the aforementioned 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 experimental results, our energy model ensures higher accuracy than previous works by considering these key parameters. Wang *et al.* demonstrated the possibility of utilizing GPU for offloading [12]. 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 data or programs to GPU may increase the total energy consumption. Our work, on the other hand, achieves higher accuracy by modeling the idle energy consumption. We also provide the ability of offloading data or programs to GPU or Cloud. Recently, Ristanovic *et al.* [25] designed two algorithms, i.e., MixZones and HotZones, to offload bulky socially recommended content from 3G networks. However, their method mainly focused on the need of operators rather than on that of end users. On the contrary, we designed a customizable cost function, which allows end users to adjust the weight of response time and energy consumption. Barbera *et al.* [26] studied the feasibility of both mobile computation offloading and mobile software backups. They gave an evaluation of the feasibility and costs of both off-clones and back-clones in terms of bandwidth and energy consumption. However, how to measure the energy consumption of a smart phone was not discussed in depth. In our work, we developed a software-based energy estimation method, which can be applied to all Android smart phones without extra hardware equipment, such as a DAQ card.

2) 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 data or part of a program to remote servers in order to reduce execution time [2], [7], [14], [17]. Most of them adopted the aforementioned similar profiling–partitioning technology to identify the offloaded parts of an application. Gu *et al.* 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 [2]. Yang *et al.* developed an offloading service that dynamically partitions Java applications and transforms Java classes into a form that can be executed at remote servers [17]. Giurgiu *et al.* developed an exhaustive search algorithm, which is called ALL, to examine all possible partitions in order to find an optimal partition [14]. 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 a 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], and [17], Wolski *et al.* dynamically predicted offloading cost at runtime 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 higher accuracy by modeling these important parameters. According to our experimental results, fewer incorrect offloading decisions are made.

Several works developed offloading mechanisms by integrating existing software packages rather than those started from scratch [15], [16], [18]. Kemp *et al.* used Ibis middleware to offload computationally intensive Java programs to remote servers [15]. Zhang *et al.* adopted a Firefox plug-in framework to transparently offload computations to remote servers [18]. Since these works are closely coupled with specific software packages, it becomes difficult to extend their methods to other execution environments. Lin *et al.* explored the possibility of offloading programs to network processors in order to reduce execution time [16]. They first profiled the IPSec module to identify bottlenecks and then rewrote the 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 the OS kernel and drivers is required, which reduces the portability of the proposed method. In this paper, 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.

3) 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 optimizing the two objectives simultaneously [19], [20]. Wang and Li used similar profiling–partitioning technology to identify offloaded parts and consider energy and time saving at the same time [20]. A similar method was developed by Miettinen and Nurminen to offload the most power-hungry parts in order to reduce energy consumption [19]. 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. Kosta *et al.* designed *ThinkAir*, and Kemp *et al.* developed *Cuckoo* to execute applications [27], [28] on the cloud. However, the models used to estimate the energy consumption and execution time model were not given. In this paper, we provide a higher accuracy energy and execution model by considering important parameters of CPU and offloading targets. Our experimental results indicate that the proposed method can achieve better performance in saving energy and time. Security is also an important issue when designing an offloading mechanism. A secure communication channel is required before we offload private data and programs. A detailed and in-depth discussion on this subject is beyond the scope of this paper. The interested reader can refer to the survey paper [29].

B. OpenGL|ES

OpenGL|ES, which is a Khronos-developed graphics standard of embedded system, is derived from OpenGL. Almost every smart phone now uses OpenGL|ES as a rendering engine. In this paper, we adopt OpenGL|ES 2.0, which is widely supported by many of today's GPUs, in our experiments.

Fig. 1 gives an overview of the graphics processing flow of GPU. The vertex shader provides a programmable method to operate vertices, usually for projection or lighting. To operate vertices, developers need to write a C-like shading program [30] and compile it as a vertex shader. Then, each time we feed vertices into a vertex shader, the shader will calculate the result and then transmit it to primitive assembly blocks. The fragment shader is used to operate fragments, which are produced by previous vertices. Due to the flexibility of the shader program, some works [12], [31], [32] have offloaded general-purpose computation onto GPU. In this paper, we implement a matrix multiplication module by using OpenGL|ES 2.0 shading language.

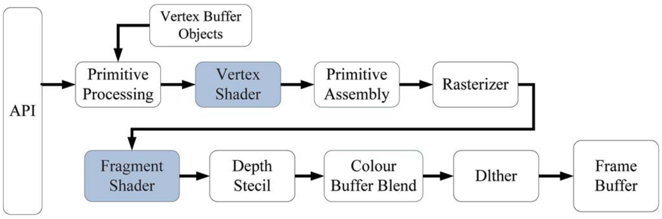


Fig. 1. OpenGL|ES 2.0 programmable pipeline.

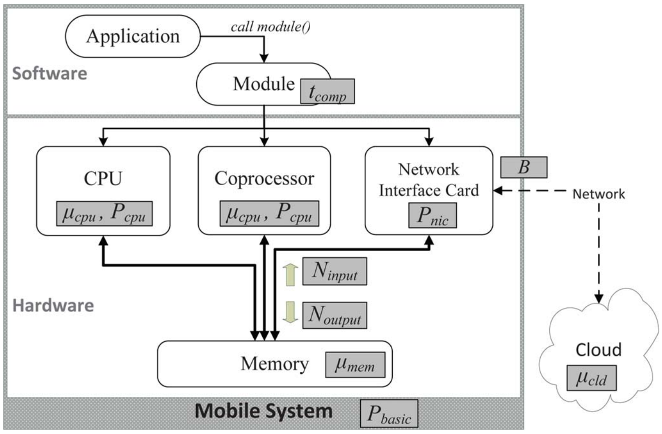


Fig. 2. Flow of application execution.

III. Problem Statement

We design a smart phone offloading framework to determine an execution unit, such as CPU, coprocessor, or cloud, for frequently used modules. Here, we first describe the model of mobile application, CPU, coprocessor, and cloud. We next give the problem statement and describe the challenges.

A. System Model and Problem Statement

Mobile applications usually adopt frequently used modules, such as FFT, convolution, and matrix multiplication, to process data [12], [31], [32]. As Fig. 2 shows, the application first involves a module to process data with the size of N_{input}, which is stored in memory. The data are then processed by CPU, coprocessor, or cloud. Finally, the processed data with the size of N_{output} are written back to memory. Let $\hat{T}_{\text{cpu}}$ denote the execution time if the module is executed on a local CPU. We divide $\hat{T}_{\text{cpu}}$ into two parts. The first part is transmission time t_{trans}, which is used for fetching data from and writing them back to memory. The second part is pure computation time t_{comp}, which is consumed by the CPU to execute codes.

TABLE II
NOTATION TABLE

Cost Function			**Definition**
target			Major unit for computation, e.g. CPU/Coprocessor/Cloud
$T_{t\arg et}$			Measured execution time when execute on target
$\hat{T}_{t\arg et}$			Estimated execution time when execute on target
$E_{t\arg et}$			Measured energy consumption when execute on target
$\hat{E}_{t\arg et}$			Estimated energy consumption when execute on target
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	X	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

In addition, the bandwidth of memory access is μ_{mem}, at which the data are read from or written into the memory by CPU, coprocessor, or network interface. If the data processing is offloaded to the cloud, the network transmission speed is B. Let μ_{cpu}, μ_{cop}, and μ_{cld} denote the speed of CPU, coprocessor, and cloud, respectively. 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}. We summarize the notations in Table II, in which we use a circle to indicate a dynamic variable and a cross to indicate a static variable. The static decision factors are deterministic and module independent, whereas the dynamic decision factors are uncertain or module dependent.

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}, we address the problem of selecting an execution environment from CPU, coprocessor, and cloud in order to reduce both execution time and energy consumption.

B. Challenges

Several challenges need to be addressed while we tackle the offloading problem. Some of them are listed as follows.

1) *There may not be positive correlation between execution time and system-wide energy consumption in a heterogeneous computation environment*: As shown in Section II-A, many previous works assumed that short execution time implies low energy consumption. However, this assumption does not always hold, particularly in a heterogeneous computation environment. For example, in some multicore platforms, GPU is faster than CPU but consumes more energy. Hence, if a task is executed on a GPU, its execution time is shorter than that on a CPU. However, its energy consumption is larger than that on a CPU. It becomes challenging when we aim to reduce execution time and energy consumption at the same time (see Section IV).
2) *The unit of execution time is different from the unit of energy consumption*: Due to the different units of execution time and energy consumption, a naive cost function that sums up execution time and energy consumption is not appropriate to our problem. As a result, there is a clear need for developing a new cost function that can reflect both execution time and energy consumption (see Section IV).
3) *A cross-platform power measurement method is desired*: In order to make a correct offloading decision, the information of power consumption is crucial. However, most smart phones are not equipped with power meters or a data acquisition (DAQ) card. The shortage makes it difficult to obtain the necessary information for decision making. As a result, a software-based cross-platform power measurement method is desired in order to improve the applicability of the proposed method (see Section V-E).

IV. TIME-AND-ENERGY-AWARE TDM

Here, we first give an overview of our offloading framework, i.e., TDM, and introduce the mechanism of creating and updating factor tables. We then introduce the equations used

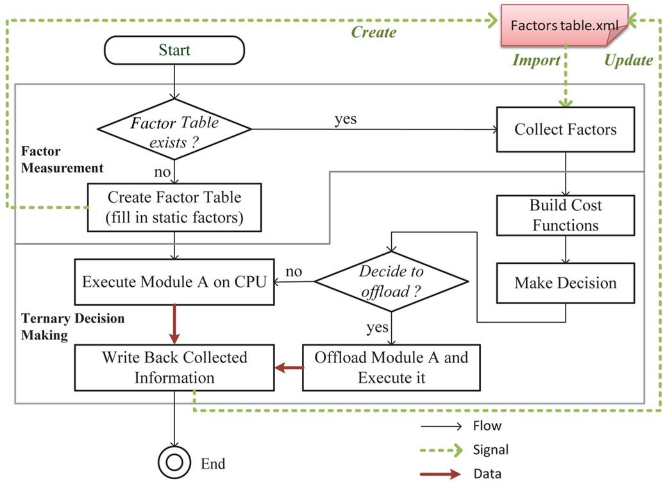


Fig. 3. Flow of TDM.

to calculate the execution time and energy consumption of a module when it is executed on local CPU, local GPU, or cloud. Finally, we describe the methodology of ternary decision making.

A. Overview of the TDM Flow

Our TDM is a daemon that runs as a background process. The TDM includes two parts: factor measurement and ternary decision making. Before a system starts to run, we measure μ_{mem}, P_{basic}, P_{cpu}, P_{cop}, and P_{nic}. We also obtain μ_{cpu} and μ_{cop} from the datasheet. We call these factors static decision factors because they are deterministic and module independent. At runtime, as shown in Fig. 3, when a module is invoked, TDM first checks the existence of the associated factor table of the module, which records necessary parameters to estimate energy consumption and execution time. If the factor table does not exist, the module will be executed on CPU directly, and a corresponding factor table will be created. TDM adds static decision factors, such as μ_{mem}, P_{basic}, P_{cpu}, P_{cop}, P_{nic}, μ_{cpu}, and μ_{cop}, to the factor table. TDM also adds module-dependent decision factors, i.e., N_{input} and N_{output}, provided by the module, to the factor table. After the module is completed, execution time t_{comp} will be added to the factor table for future use. Instead, if the associated factor table exists, TDM will extract the aforementioned decision factors from the decision table. TDM next asks the cloud to provide its execution speed μ_{cld} and estimates wireless bandwidth B. These decision factors are then passed to the cost functions. Based on the results of cost functions, the decision maker determines whether the module should be offloaded or not. In this paper, a module is a frequently used function of mobile applications, such as FFT, convolution, matrix multiplication, and so on. The workloads of these frequently used functions are usually deterministic. If a module is offloaded, it will be executed on the cloud or a local GPU.

B. Create and Update Factor Tables

Each module has its own factor table. A factor table records necessary parameters to estimate energy consumption and execution time of the associated module. We divided these parameters into static and dynamic decision factors. The static decision factors include μ_{cpu}, μ_{cop}, μ_{mem}, P_{basic}, P_{cpu}, P_{cop}, and P_{nic}, which are deterministic and module independent. The dynamic decision factors include B, N_{input}, N_{output}, μ_{cld}, and t_{comp}, which are uncertain or module dependent. The static decision factors are obtained before a system starts to run. When the module is invoked at the first time, a factor table is created to store static decision factors, and the module is executed on CPU. After the module is completed, TDM adds module-dependent decision factors, such as t_{comp}, N_{input}, and N_{output}, to the factor table. If the same module is invoked again, TDM refers to its associated factor table and runtime information, i.e., B and μ_{cld}, to estimate the execution time and energy overhead of offloading. In Section IV-C, we will explain how we used these decision factors to estimate offloading cost and make a decision. In addition, Section V will describe the methods used to obtain decision factors B, μ_{cpu}, μ_{cop}, μ_{mem}, μ_{cld}, t_{comp}, P_{basic}, P_{cpu}, P_{cop}, and P_{nic}.

C. Ternary Decision

This subsection first discusses the execution time and energy consumption of a module when it is executed on different execution environments: local CPU, local GPU, or cloud. It then introduces the algorithm used for decision making.

1) Execution Time and Energy Consumption: First of all, we estimate the execution time $\hat{T}_{\text{cpu}}$ of the case that the module is executed on a local CPU. As (1) shows, $\hat{T}_{\text{cpu}}$ includes two parts, i.e.,

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

The first part is transmission time t_{trans}, which is used for fetching data from and writing them back to memory. The second part is pure computation time t_{comp}, which is consumed by the CPU to execute codes. Since t_{trans} depends on the amount of processed data, we have

$$t_{\text{trans}} = \frac{N_{\text{input}} + N_{\text{output}}}{\mu_{\text{mem}}}. \tag{2}$$

Based on (1), the energy consumption of the local CPU is estimated by

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

Similarly, if the module is executed on a local GPU, also the coprocessor, the estimated execution time $\hat{T}_{\text{cop}}$ is

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

In addition, the estimated energy consumption $\hat{E}_{\text{cop}}$ is

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

After considering the above two cases, we now discuss the execution time and energy consumption in the case of offloading the module to the cloud. We define $\hat{T}_{\text{cld}}$ as the estimated execution time of the module when it is offloaded to the cloud.

In order to calculate $\hat{T}_{\text{cld}}$, we first determine the amount of data to be transmitted by

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

in which MTU stands for the maximum transmission unit. Moreover, the ACK packets used during the transmission are determined by

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

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

$$\hat{T}_{\text{cld}} = \frac{\sigma + \sigma^{\text{ack}}}{\mu_{\text{mem}}} + \frac{\sigma + \sigma^{\text{ack}}}{B} + \frac{t_{\text{comp}} \times \mu_{\text{cpu}}}{\mu_{\text{cld}}}. \tag{4}$$

In (4), 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 of the cloud is then determined by

$$\hat{E}_{\text{cld}} = P_{\text{basic}} \times \hat{T}_{\text{cld}} + P_{\text{nic}} \times \left(\frac{\sigma + \sigma^{\text{ack}}}{\mu_{\text{mem}}} + \frac{\sigma + \sigma^{\text{ack}}}{B} \right) \tag{5}$$

in which P_{nic} is the power consumption of the network interface. In Section V, we will describe the methodology of measuring $B, t_{\text{comp}}, \mu_{\text{cpu}}, \mu_{\text{cop}}, \mu_{\text{cld}}, \mu_{\text{mem}}, P_{\text{basic}}, P_{\text{cpu}}, P_{\text{cop}}$, and P_{nic}.

2) Decision Making: At runtime, we dynamically measure t_{comp}, N_{input}, N_{output}, and B and then use them to calculate $\hat{T}_{\text{cpu}}$, $\hat{E}_{\text{cpu}}$, $\hat{T}_{\text{cop}}$, $\hat{E}_{\text{cop}}$, $\hat{T}_{\text{cld}}$, and $\hat{E}_{\text{cld}}$. We define β_x as

$$\beta_x = (\hat{T}_{\text{cpu}} - \hat{T}_x)/\hat{T}_{\text{cpu}}$$

in which x can be *cpu*, *cop*, and *cld*. β_{cop} represents the fraction reduction in execution time by offloading the module to the coprocessor. In addition, β_{cld} is the percentage reduction in execution time by offloading the module to the cloud. Obviously, we have $\beta_{\text{cpu}} = 0$ since the module is not offloaded. Similarly, in order to indicate the energy reduction, we define γ_x as

$$\gamma_x = (\hat{E}_{\text{cpu}} - \hat{E}_x)/\hat{E}_{\text{cpu}}$$

in which x can be *cpu*, *cop*, and *cld*. γ_{cop} is the percentage reduction in energy by offloading the module to the coprocessor and γ_{cld} is that to the cloud. To further differentiate the importance of the execution time and the energy consumption, we define a composite cost function as

$$f(\alpha, x) = \alpha \cdot \beta_x + (1 - \alpha) \cdot \gamma_x \tag{6}$$

where α is a user-specified variable in the range [0, 1]. If $\alpha = 0$, it implies that the energy consumption is the only criterion in determining the offload target. On the other hand, if $\alpha = 1$, the execution time is the only criterion.

In order to determine an offload target of the module, we calculate the cost function $f(\alpha, x)$ of each possible offload target and select the one with a minimum value as the offload target. In other words, given the user-specified α, we offload the module to target y, which is determined by

$$y = \arg\min_{x \in N} \{f(\alpha, x)\}$$

where N is the set of $\{cpu, cop, cld\}$.

V. Measurement of Decision Factors

Here, we first introduce the device under test (DUT). We then describe the methods used to measure bandwidth, component speed, execution time, and power consumption individually.

A. DUT

We adopted HTC Nexus One, which is a popular and powerful smart phone, as our DUT. Nexus One is equipped with a Qualcomm QSD8250 1-GHz 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 Nexus One is Android 2.2. We implemented our offloading framework in C language on user space so that it can operate without any privilege restrictions and become more efficient.

B. Wireless Bandwidth: B

As mentioned in the previous section, 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 runtime. Many tools have been developed to measure the transmission bandwidth. Some of them are platform dependent, such as *Iperf* and *ttcp*, whereas others are platform independent. To make our method easily applicable to other Android smart phones, we use *ping*, which is a popular utility, located at the path of /system/bin, to measure the network bandwidth.

For each measurement, the *ping* utility first sends the packets of ICMP-Request from Nexus One to the cloud and then receives the packets sent back by the cloud. In order to have a more accurate approximation of bandwidth, we sent out data that can be fragmented to several packets. In our experiment, we used the command "*ping -c 10 -s 6000 -i 0.1 cloud*" to issue ten 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 s. As Fig. 4 shows, the measurement starts at time t_1 and stops at t_2. For ease of measurement, we set the MTU of NIC to 1500 to avoid extra packet fragmentation. As a result, 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 Nexus One. For example, if $t_2 - t_1$ is 10 ms, the network bandwidth is approximated by

$$B = \frac{(4 \times 1514 + 122) \times 2 \times 10 \times 8(bit)}{10\ \text{(ms)}} = 9884\ \text{Kb/s}.$$

Compared with other bandwidth measurement methods, our method has two advantages. First, it can be applied to all Android smart phones. Second, the overhead of measuring

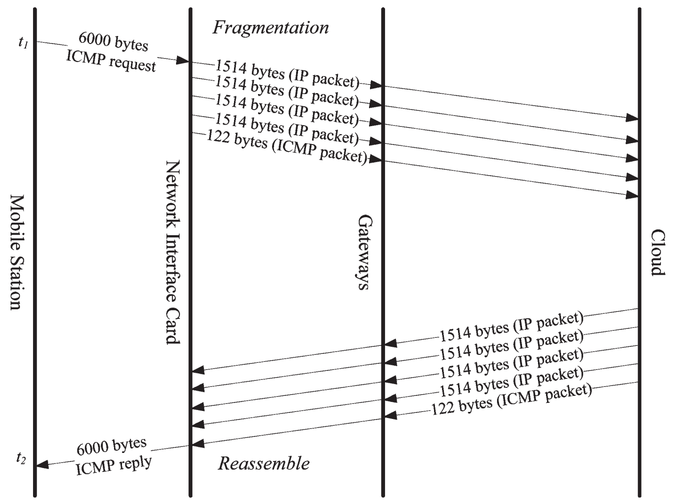


Fig. 4. ICMP request and reply with a payload size of 6000 bytes.

bandwidth is low. In order to reduce the effect of network uncertainty on decision making, we measure the transmission bandwidth whenever a module needs to be offloaded.

C. 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 the datasheet. At runtime, we issue a query to the cloud to get cloud speed μ_{cld}. A simple loop program is then executed on the cloud to estimate μ_{cld}. In addition, we measure memory bandwidth μ_{mem} at runtime. 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 20 ms to read 1 000 000 16-bit integers from the memory, memory bandwidth μ_{mem} is estimated by

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

D. Execution Time: t_{comp}, t_{trans}

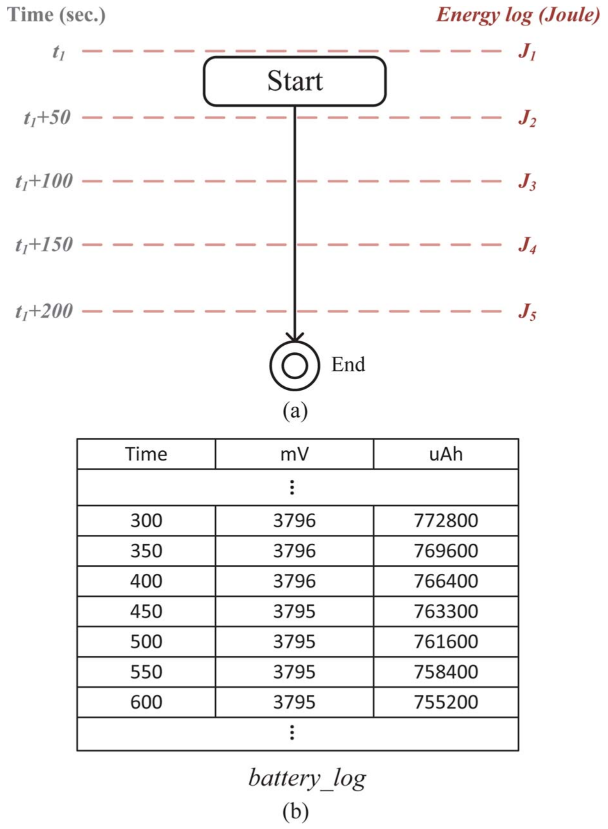
As mentioned in Section IV, t_{comp} represents the pure computation time used by the CPU to execute codes. We calculate t_{comp} by

$$t_{\text{comp}} = T_{\text{cpu}} - t_{\text{trans}}$$

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

E. 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. As a result, we design four different scenarios: 1) idle the system; 2) execute CPU-bound workload; 3) execute GPU-bound workload; and 4) send a large amount of data to the cloud. For each scenario, we use an Android daemon-maintained battery log, located at the path of /sys/kernel/debug/battery_log, to obtain power information. As Fig. 5(a) and (b) shows, an Android daemon updates voltage, current, and power information every 50 s.



(a)

Time	mV	uAh
⋮		
300	3796	772800
350	3796	769600
400	3796	766400
450	3795	763300
500	3795	761600
550	3795	758400
600	3795	755200
⋮		

battery_log

(b)

Fig. 5. (a) Energy measurement in Android. (b) Example of measured values.

TABLE III
POWER PARAMETERS OF NEXUS ONE

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

In scenario 1, we close all unnecessary user programs and keep the system idle for a while. Fig. 5(b) illustrates a battery log in the scenario of idling the system. Based on the log, the energy consumption of the system in [400 s, 450 s] is

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

Therefore, we have

$$P_{\text{basic}} = \frac{12.53, \text{mWh}}{50\ \text{seconds}} = 902\ \text{mW}.$$

A similar approach is used in the 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 a large amount of data to the cloud for a long period. The measurement results are listed in Table III, in which P_{nic} is the average power consumption of the WiFi interface. Compared with other energy

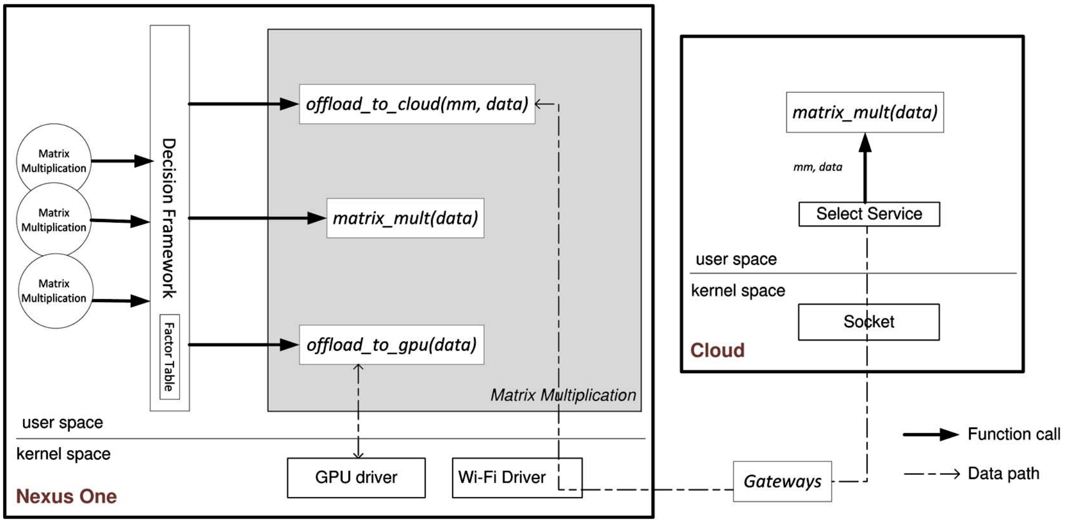


Fig. 6. Experiment environment.

profiling methods, our method has two advantages. First, it can be applied to all Android smart phones. Second, no extra hardware equipment is required, such as a DAQ card.

VI. Experiment and Evaluation

In order to evaluate the effectiveness of our TDM framework, we conducted several experiments based on real-world applications and compared TDM with the methods that consider only execution time or energy consumption. First of all, we introduce our experiment environment in Section VI-A. Next, we discuss the overhead of our TDM framework in Section VI-B. Finally, a case study of matrix multiplication is given in Section VI-C.

A. Testbed

As Fig. 6 shows, the experiment environment includes a Nexus One smart phone and a cloud. In order to eliminate uncertainty and unpredictability, we set the backlight always on and closed unnecessary processes. We implemented our decision framework on Nexus One and installed matrix multiplication applications for experiment. Each application has at least one or more modules to be offloaded. A PC with a 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.

B. Overhead of TDM

In order to understand the overhead of our decision framework TDM, we measured the execution time and energy consumption of each function. As Table IV shows, creating a 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 on the function *Collect Factors*. Since the function *Create Factor Table* is only executed once, the function *Collect Factors* dominates the overhead of our decision framework. We further break down the energy consumption of the function *Collect Factor*. As Fig. 7 shows, collecting the information of bandwidth consumes most time and energy. This overhead is induced by the ping program and the Wi-Fi driver, which can be reduced by decreasing the measurement time.

TABLE IV
Proportions of Time and Energy in Our 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

C. 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 effectiveness of TDM, we implemented three versions of the matrix multiplication for different offloading targets. One is for the execution of local CPU; the other two are for local GPU (i.e., coprocessor) and the cloud. In the version of local CPU, it includes three steps: reading data, processing the multiplication, and storing the results. As mentioned in Section II-B, two programs are implemented in the version of GPU. One is vertex shader, and the other is fragment shader. Because the most popular smart phones are nowadays equipped with QVGA (320 × 240) displays [33], we limited the size of rendering images to 320. Once $offload\_to\_gpu()$ is invoked, it first

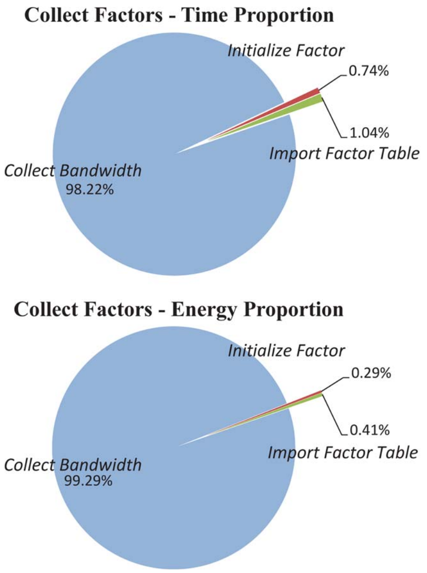


Fig. 7. Proportions of time and energy in the function *Collect Factors*.

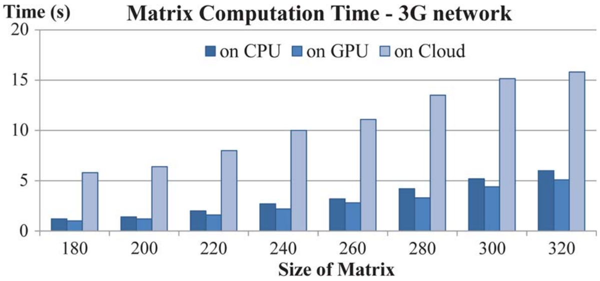


Fig. 8. Matrix multiplication via a 3G network.

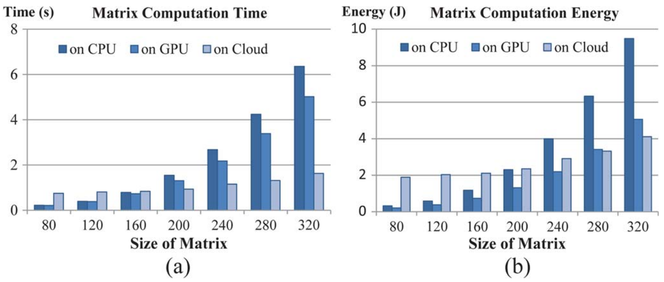
communicates with the GPU driver and compiles shader codes into the executable format of GPU. Next, the data are fed to GPU. Finally, the processed data are written back to the buffer *pBuffer*, and the communication is terminated.

In order to offload the computation to the cloud, on the smart phone, we implemented a function, named $offload\_to\_cloud()$, to send offloaded data to the cloud. On the site of the cloud, a service is deployed to receive the offloaded requests and forward them to a proper function. As shown in Fig. 6, for the matrix multiplication, the module is first forwarded to the function $matrix\_mult()$. The results are then sent back to the smart phone.

1) Estimation Accuracy: This subsection evaluates the accuracy of our method in approximating the execution time of an offloaded module in different execution environments. We first measured the execution time of the offloaded module by varying the matrix size. We then compared our estimated execution time with measurement data.

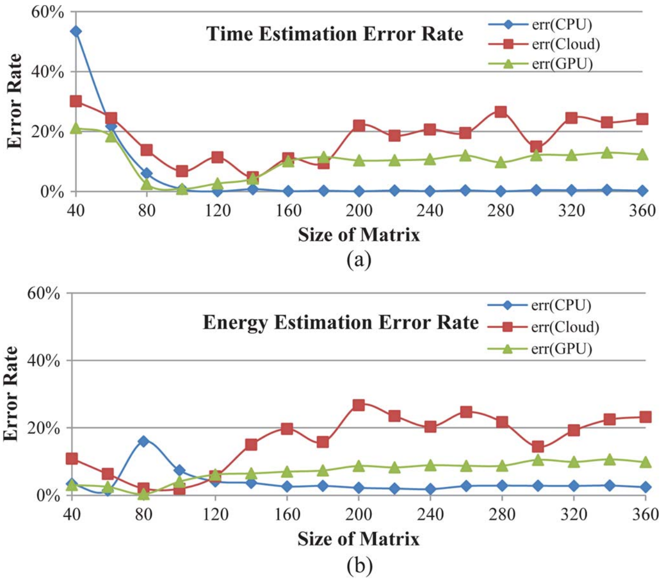
We evaluated our method in both Wi-Fi and 3G networks. According to our experimental results, as shown in Fig. 8, the execution time significantly increases if we offload modules to the cloud through the 3G network. As a result, in the following experiments, we adopt the Wi-Fi network when we offload modules to the cloud.

Fig. 9(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 320 × 320, the speedup ratio is 1.27 if the computation is offloaded to the local GPU and 3.89 if it is offloaded to the cloud. In other words, the percentage reduction in execution time can reach almost $75\%(= (6.20 - 1.55)/6.20)$ in the best case. Similarly, Fig. 9(b) shows the measurement results of energy consumption, in which the x-axis is the size of matrix, and the y-axis is the energy consumption. The percentage reduction in energy consumption can reach almost $56\%(= (9.20 - 4.00)/9.20)$ in the case of size-320 matrix.



(a) (b)

Fig. 9. Matrix multiplication. (a) Execution time. (b) Energy consumption.



(a)

(b)

Fig. 10. Estimation error rate. (a) Execution time. (b) Energy consumption.

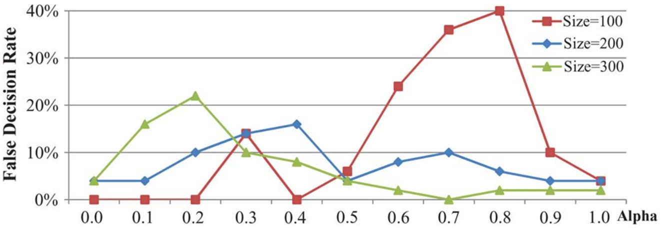


Fig. 11. False decision rate of matrix multiplication.

We define the *error rate* of execution time as

$$\mathrm{err}_t(\mathrm{target}) = \left| \frac{\hat{T}_{\mathrm{target}} - T_{\mathrm{target}}}{T_{\mathrm{target}}} \right|$$
$$\mathrm{target} \in \{CPU, GPU, Cloud\} \tag{7}$$

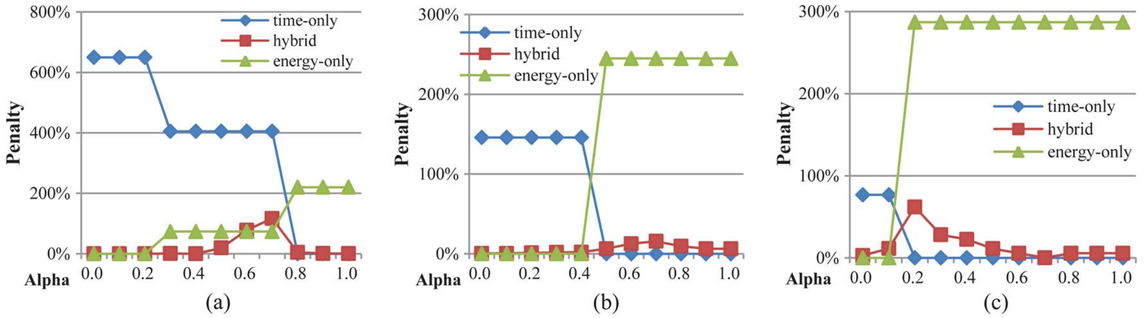


Fig. 12. False decision penalty of (a) size-100, (b) size-200, and (c) size-300.

in which $\hat{T}_{\text{target}}$ is the estimated execution time, which is obtained by (1), (3), or (4). As Fig. 10(a) shows, the error rate becomes larger when the size of the matrix is smaller. This is because the overhead of the OS context switch cannot be ignored for small-size matrix multiplication.

Similarly, the *error rate* of energy consumption is defined as

$$\text{err}_e(\text{target}) = \left|\frac{\hat{E}_{\text{target}} - E_{\text{target}}}{E_{\text{target}}}\right|$$
$$\text{target} \in \{CPU, GPU, Cloud\}. \quad (8)$$

As Fig. 10(b) shows, if the size of the matrix is larger than 80, the error rate of energy consumption is around 20%.

2) Decision Accuracy: Since the error rate can result in an incorrect offloading decision, we define the false decision rate, i.e., ρ, as

$$\rho = 1 - \frac{\#\ of\ correct\ decisions}{\#\ of\ decisions}.$$

In our experiment, we varied the value of α in the range [0,1] and fixed the matrix size in 100, 200, and 300. As Fig. 11 shows, the false decision rate is less than 15% in most cases. When matrix size becomes larger, the false decision rate is smaller. This is because the overhead of the OS context switch cannot be ignored in the case of small matrices. For example, when α is 0.6, the false decision rates of size-300 matrix and size-200 matrix are smaller than that of size-100 matrix. If α is set at 0 or 1, the false decision rate is almost zero since only the energy consumption or execution time needs to be predicted.

Let E_{opt} and T_{opt} denote the energy consumption and execution time of the optimal offloading decision. Moreover, for an offloading decision, we use $\hat{E}$ and $\hat{T}$ to represent the energy consumption and execution time. We define penalty as

$$\frac{\hat{E} - E_{\text{opt}}}{E_{\text{opt}}} + \frac{\hat{T} - T_{\text{opt}}}{T_{\text{opt}}}.$$

Obviously, if an offloading decision is optimal, the penalty is zero. As shown in Fig. 12, the x-axis is the value of alpha (α), and the y-axis is the penalty. For each α, we measure the associated E_{opt} and T_{opt}. We also measure $\hat{E}$ and $\hat{T}$ for different offloading methods. The term "hybrid" is TDM, which has a low penalty in each size. However, the other two methods, i.e., time-only and energy-only, suffer from a high penalty because they did not consider time and energy at the same time.

As Fig. 12 shows, by varying the value of α, we have 11 test cases in each subgraph. There are, in total, 33 test cases in three subgraphs. Our method makes near-optimal decisions (i.e., penalty is close to 0) in 24 cases. This is because the penalty of the second best decision is almost the same as that of the optimal decision. On the contrary, the time-only method makes 18 near-optimal decisions, and the energy-only method makes only ten near-optimal decisions. As a result, our improvement in making the number of near-optimal offloading decision can reach, at most, twice as that in the energy-only method.

3) Evaluate Decision Overhead: We have analyzed the overhead of our decision framework in Section VI-B. Now, we are going to evaluate the impact of the overhead on energy and time reduction. According to our experimental results, for the case where the matrix size is 100, the execution time is 197 ms if the module is executed on the local CPU. On the other hand, the execution time becomes 118 ms if it is uploaded to the cloud. Since it takes extra 935 ms to complete the execution of the proposed method, the total execution time increases to 1053 ms. In this case, performing local computation is much better than offloading the computation to the cloud. According to our experimental results, in order to save energy and time by offloading, the matrix size should be larger than 250.

VII. Conclusion and Future Work

In this paper, we have designed and implemented a decision framework for computation offloading. The decision is based on estimated execution time and energy consumption. 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 presented a case study to validate the applicability in 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, at most, 75% savings in execution time and 56% in battery usage. Our results also demonstrate high accuracy and low false decision rates of the proposed decision framework. Generally speaking, the false decision rate is less than 15% in most cases. Our future work includes three aspects. First, we will implement a lightweight *ping* function in order to further reduce the overhead in collecting the parameter of bandwidth. Second, we plan to investigate effects of different smart phones, mobile applications, and network environment on the accuracy of TDM. Finally, we will extend this work by considering different wireless technologies, such as LTE and WiMAX, and security issues.

REFERENCES

[1] Lavendershih, "Booming popularity of smartphone helps to increase nand flash demand," DRAMeXchange, Tech. Rep., 2011.

[2] X. Gu, A. Messer, I. Greenberg, D. Milojicic, and K. Nahrstedt, "Adaptive offloading for pervasive computing," *IEEE Pervasive Comput.*, vol. 3, no. 3, pp. 66–73, Jul./Sep. 2004.

[3] Z. Li, C. Wang, and R. Xu, "Computation offloading to save energy on handheld devices: A partition scheme," in *Proc. 2001 Int. CASES*, 2001, pp. 238–246.

[4] S. Ou, K. Yang, and J. Zhang, "An effective offloading middleware for pervasive services on mobile devices," *Pervasive Mobile Comput.*, vol. 3, no. 4, pp. 362–385, Aug. 2007.

[5] G. Chen, B.-T. Kang, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and R. Chandramouli, "Studying energy trade offs in offloading computation/compilation in java-enabled mobile devices," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 9, pp. 795–809, Sep. 2004.

[6] K. Kumar and Y. Lu, "Cloud computing for mobile users: Can offloading computation save energy?" *Computer*, vol. 43, no. 4, pp. 51–56, Apr. 2010.

[7] R. Wolski, S. Gurun, C. Krintz, and D. Nurmi, "Using bandwidth data to make computation offloading decisions," in *Proc. IEEE IPDPS*, Apr. 2008, pp. 1–8.

[8] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: Making smartphones last longer with code offload," in *Proc. 8th Int. Conf. MobiSys*, Jun. 2010, pp. 49–62.

[9] S. Han, S. Zhang, and Y. Zhang, "Energy saving of mobile devices based on component migration and replication in pervasive computing," in *Proc. Ubiquitous Intell. Comput.*, Aug. 2006, pp. 637–647.

[10] Y. Hong, K. Kumar, and Y. Lu, "Energy efficient content-based image retrieval for mobile systems," in *Proc. IEEE ISCAS*, May 2009, pp. 1673–1676.

[11] B. Seshasayee, R. Nathuji, and K. Schwan, "Energy-aware mobile service overlays: Cooperative dynamic power management in distributed mobile systems," in *Proc. 4th ICAC*, Jun. 2007, p. 6.

[12] Y. Wang, B. Donyanavard, and K. Cheng, "Energy-aware real-time face recognition system on mobile CPU–GPU platform," in *Proc. 11th ECCV*, Sep. 2010, pp. 411–422.

[13] X. Zhao, P. Tao, S. Yang, and F. Kong, "Computation offloading for H.264 video encoder on mobile devices," in *Proc. IMACS*, Oct. 2006, pp. 1426–1430.

[14] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso, "Calling the cloud: Enabling mobile phones as interfaces to cloud applications," in *Proc. 10th ACM/IFIP/USENIX Int. Conf. Middleware*, Dec. 2009, pp. 1–20.

[15] R. Kemp, N. Palmer, T. Kielmann, F. Seinstra, N. Drost, J. Maassen, and H. Bal, "eyedentify: Multimedia cyber foraging from a smartphone," in *Proc. 11th IEEE ISM*, Dec. 2009, pp. 392–399.

[16] Y. Lin, Y. Lin, Y. Lai, and C. Lin, "VPN gateways over network processors: Implementation and evaluation," *J. Internet Technol.*, vol. 11, no. 4, pp. 457–463, Jul. 2010.

[17] K. Yang, S. Ou, and H. Chen, "On effective offloading services for resource-constrained mobile devices running heavier mobile Internet applications," *IEEE Commun. Mag.*, vol. 46, no. 1, pp. 56–63, Jan. 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 *Proc. 4th ICIW*, May 2009, pp. 193–199.

[19] A. Miettinen and J. Nurminen, "Energy efficiency of mobile clients in cloud computing," in *Proc. 2nd USENIX Conf. HotCloud*, Jun. 2010, p. 4-4.

[20] C. Wang and Z. Li, "A computation offloading scheme on handheld devices," *J. Parallel Distrib. Comput.*, vol. 64, no. 6, pp. 740–746, Jun. 2004.

[21] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *Proc. 6th ACM SIGOPS/EuroSys*, 2011, pp. 301–314.

[22] C. Cai, L. Wang, S. U. Khan, and J. Tao, "Energy-aware high performance computing: A taxonomy study," in *Proc. 17th IEEE ICPADS*, 2011, pp. 953–958.

[23] M. Guzek, J. E. Pecero, B. Dorrosoro, P. Bouvry, and S. U. Khan, "A cellular genetic algorithm for scheduling applications and energy-aware communication optimization," in *Proc. ACM/IEEE/IFIP Int. Conf. HPCS*, 2010, pp. 241–248.

[24] S. U. Khan, "A goal programming approach for the joint optimization of energy consumption and response time in computational grids," in *Proc. 28th IEEE IPCCC*, Dec. 2009, pp. 410–417.

[25] N. Ristanovic, J.-Y. Le Boudec, A. Chaintreau, and V. Erramilli, "Energy efficient offloading of 3G networks," in *Proc. IEEE 8th Int. Conf. MASS*, 2011, pp. 202–211.

[26] M. V. Barbera, S. Kosta, A. Mei, and J. Stefa, "To offload or not to offload? The bandwidth and energy costs of mobile cloud computing," in *Proc. IEEE INFOCOM*, 2013, pp. 1285–1293.

[27] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proc. 31st Annu. IEEE (INFOCOM) Int. Conf. Comput. Commun.*, 2012.

[28] R. Kemp, N. Palmer, and T. Kielmann, "Cuckoo: A computation offloading framework for smartphones," presented at the Proc. 2nd Int. Conf. Mobile Comput., Appl., Serv., Santa Clara, CA, USA, 2010.

[29] A. N. Khan, M. L. Mat Kiah, S. U. Khan, and S. A. Madani, "Towards secure mobile cloud computing: A survey," *Future Generation Computer Systems*, vol. 29, no. 5, pp. 1278–1299, Jul. 2013.

[30] A. Munshi, D. Ginsburg, and D. Shreiner, *OpenGL ES 2.0 Programming Guide*. Reading, MA, USA: Addison-Wesley, 2008.

[31] L. Marziale, G. Richard, III, and V. Roussev, "Massive threading: Using GPUs to increase the performance of digital forensics tools," *Digit. Investigation*, vol. 4, pp. 73–81, Sep. 2007.

[32] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: A GPU-accelerated software router," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 195–206, Oct. 2010.

[33] IMSDroid. [Online]. Available: https://code.google.com/p/imsdroid/

Ying-Dar Lin (F'13) received the Ph.D. degree in computer science from the University of California, Los Angeles, CA, USA, in 1993.

He is a Professor of computer science with National Chiao Tung University, Hsinchu, Taiwan. Since 2002, he has been the Founding Director of Network Benchmarking Lab (www.nbl.org.tw), which reviews network products with real traffic. In 2002, he cofounded L7 Networks Inc., which was later acquired by D-Link Corporation. He published a textbook titled "Computer Networks: An Open Source Approach" (McGraw-Hill, 2011). His research interests include network security, wireless communications, and embedded systems.

He is an IEEE Fellow and serves on the editorial board of several IEEE journals and magazines.

Edward T.-H. Chu (M'11) received the Ph.D. degree in computer science from National Tsing Hua University, Hsinchu, Taiwan, in 2010.

He has more than four years of work experience in the industry, where he worked on embedded software and owns a Chinese patent. In 2009, he was a Visiting Scholar with Purdue University, West Lafayette, IN, USA. In 2010, he joined the Department of Electronic and Computer Science Information Engineering, National Yunlin University of Science and Technology, Douliou, Taiwan, as an Assistant Professor. His research interests include low-power embedded systems and real-time operating systems.

Dr. Chu received the Best Paper Award at the 2012 IEEE International Symposium on Computer, Consumer and Control (IS3C) in Taiwan.

Yuan-Cheng Lai received the Ph.D. degree from National Chiao Tung University, Hsinchu, Taiwan, in 1997.

In August 1998, he joined the faculty of the Department of Computer Science and Information Science, National Cheng Kung University, Tainan, Taiwan. In August 2001, he joined the faculty of the Department of Information Management, National Taiwan University of Science and Technology, Taipei, Taiwan, where he has been a Professor since February 2008. His research interests include performance analysis, protocol design, wireless networks, and web-based applications.

Ting-Jun Huang received the B.S. degree in computer science from National Tsing Hua University, Hsinchu, Taiwan, in 2009 and the M.S. degree in computer science from National Chiao Tung University, Hsinchu, in 2011.

He is currently with Lionic, Inc., Taiwan. His research interests include cloud computing and embedded systems.