



國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

Offload Advisory System - 行動雲端之效能監控與排程機制

Offload Advisory System - A Performance Monitoring and Task

Scheduler for Mobile Cloud Computing

吳俊德

Jyun-De Wu

指導教授：洪士灝 博士

Advisor: Shih-Hao Hung, Ph.D.

中華民國 102 年 7 月

July, 2013



國立臺灣大學碩士學位論文  
口試委員會審定書

Offload Advisory System - 行動雲端之效能監控與排程  
機制

Offload Advisory System - A Performance Monitoring and  
Task Scheduler for Mobile Cloud Computing

本論文係吳俊德君（學號 R00922134）在國立臺灣大學資訊工程  
學系完成之碩士學位論文，於民國 102 年 7 月 2 日承下列考試委  
員審查通過及口試及格，特此證明

口試委員：

洪士敏

（指導教授）

陳文敏

何博輝

許永真

許志奇

系主任

許永真



## Acknowledgments

I deeply appreciate everyone who helped me during the last two years, especially my advisor, Prof. Shih-Hao Hung for his kindness, patience and support for different research topics.

Also, thanks to members in my thesis committee, Prof. Pang-Feng Liu, Prof. Wen-Yu Su, Prof. Chi-Sheng Shih and Prof. Chia-Lin Yang, I received many valuable suggestions in my oral defense to improve this thesis.

I am also grateful to every one in the Performance, Application, and Security Lab (PAS Lab), especially JP Shieh, Kuan-Wen Su, I-Chih Lu, Tien-Tzong Tzeng, Shih-Jie Chang, Shuen-Wen Hsiao, Shin-Bo Huang and Kun-Chi Liao. They were great company during the last two years and helped me a lot. Finally, I would like to thank my family and friends for their support and encouragement.



## 中文摘要

近年來智慧型手機普遍流行於大眾, 手機的處理效能和電池壽命是開發者目前遇到的巨大瓶頸。如果能將這些工作傳送到雲端伺服器進行處理, 我們將能改善目前所遇到的困境。對一個行動應用程式來說, 爲了克服自身裝置運算的限制, 就需要更多的雲端資源。流水型編譯模式的應用程式打破了這些限制, 並利用程式的可攜性在不同的行動雲端運算環境下得到分佈式的效能。因此我們提出一個架構, 稱作行動雲端效能監控排程機制系統, 此系統可流水型編譯模式下的應用程式中的元件能充分利用雲端的資源。

我們將介紹所提出的系統如何整合之前實驗室開發的虛擬效能分析工具來找出熱點, 以及整合支持向量機到我們的決策管理者來做動態卸載決定。最後我們將討論我們架構的設計方式以及探討在不同的環境情況下我們得到的實驗數據。

**關鍵詞：**動態決策、行動雲端運算、效能分析、智慧型手機, 安卓, 流水型編譯模式



# Abstract

In light of the growing popularity of smartphones, the processor speed and battery life on such device have been the major bottlenecks for advance mobile computing. If the compute-intensive tasks can be preformed by the cloud server efficiently, we could solve the bottlenecks. For a mobile application, to overcome the resource limitation on its own device, we propose to adopt the Flow-Based Programming (FBP) model to enable the development of portable mobile applications that run across different mobile-cloud computing environments with scalable performance. In this thesis, we integrate the efficiency of offloading task to the cloud and propose a mechanism, called Offload Advisory System (OAS), to guide the offload of components for good performance and proper resource utilization.

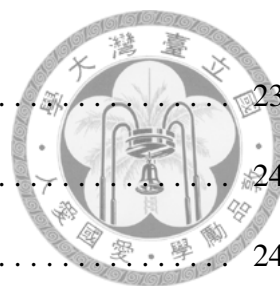
OAS leverages our previous work, Virtual Performance Analysis (VPA), help locate the hotspots in Android applications, and uses *support vector machine* (SVM) into for making dynamic offloading decisions based on application profile and current input data. The thesis discusses the design of the proposed framework and presents several usage scenarios in our experimental studies.

**Index Terms-** *Dynamic offloading, Mobile Cloud Computing, Performance Analysis, Smartphone, Android, flow-based programming*



# Contents

<b>Acknowledgments</b> .....	<b>i</b>
中文摘要.....	ii
<b>Abstract</b> .....	<b>iii</b>
<b>1 Introduction</b> .....	<b>1</b>
<b>2 Background and Related Work</b> .....	<b>3</b>
2.1 Introduction to Android .....	3
2.2 Virtual Performance Analyzer .....	3
2.3 MobileFBP .....	4
2.4 Support Vector Machine .....	5
2.5 Related Work .....	6
<b>3 Framework and Implementation</b> .....	<b>10</b>
3.1 FBP Programming model .....	11
3.2 Perform Analysis .....	14
3.3 Offload Advisory System .....	17
3.3.1 Resource Monitor.....	18
3.3.2 Policy Manager .....	19



3.3.3	Central Control Server Agent .....	23
3.3.4	Relaying Component .....	24
3.4	The procedure of Offload Advisory System .....	24
<b>4</b>	<b>Experimental Results.....</b>	<b>26</b>
4.1	Case Study: Median Filter .....	26
4.1.1	Profiling the Median Filtering Program with Cloud VPA .....	28
4.1.2	Evaluation .....	28
4.2	Case Study: 4-way parallel Median Filtering .....	33
4.2.1	Evaluation .....	34
<b>5</b>	<b>Conclusion and Future work .....</b>	<b>39</b>
5.1	Conclusion .....	39
5.2	Future Work .....	39
	<b>Bibliography.....</b>	<b>41</b>



# List of Figures

Figure 2.1	CloudVPA . . . . .	4
Figure 2.2	An overview of the architecture of the MobileFBP framework . . . . .	6
Figure 3.1	The network declaration of FBP. . . . .	12
Figure 3.2	The component declaration of FBP. . . . .	13
Figure 3.3	The JavaFBP code for a file data sorting application . . . . .	15
Figure 3.4	The profile data from Cloud VPA . . . . .	16
Figure 3.5	The overview of the Offloading Advisory System . . . . .	17
Figure 3.6	Reosource Monitor: ganglia . . . . .	18
Figure 3.7	Reosource Monitor: iperf . . . . .	19
Figure 3.8	The Estimation Model: transfer data matrix and the table of the execution time . . . . .	20
Figure 3.9	The record in our profile database. . . . .	21
Figure 3.10	Define the input parameters. . . . .	21
Figure 3.11	The number of Partition is finite . . . . .	22
Figure 3.12	Function $f$ that takes input <i>Parameters</i> and returns an output <i>Partition</i> . . .	23
Figure 3.13	The flowchart of mechanism for making the migration plan . . . . .	24



Figure 3.14	The flowchart of architecture of Offload Advisory System	25
Figure 4.1	The topology of the median filter program	27
Figure 4.2	The migrating version of Median Filter	27
Figure 4.3	The call graph of the Median Filter program	28
Figure 4.4	The call graph of the Median Filter with MobileFBP API	29
Figure 4.5	Wrapping the MediaFilter function call as a FBP task component.	30
Figure 4.6	Execution time of MediaFilter with static task assignments	31
Figure 4.7	Analysis of network overheads for task offloading	32
Figure 4.8	The topology of the 4-way parallel Median Filtering	33
Figure 4.9	The migrating version of Median Filter	33
Figure 4.10	The input parameters of case study II	35
Figure 4.11	The evaluation of the SVM prediction	36
Figure 4.12	The execution time of different filter radius	37
Figure 4.13	The execution time of different quality of network.	38





# List of Tables

Table 4.1	The experimental environment of case study I . . . . .	29
Table 4.2	The experimental environment of case study II . . . . .	34



# Chapter 1

## Introduction

Mobile cloud computing have rapidly changed human lives, and the convenience of network access have changed the way people utilize computers and networks. However, mobile cloud systems and applications are more complex than those in a traditional client-server environment. The challenges include that selecting a resourceful server among available service providers in cloud, coping with unstable mobile networks, prolonging the battery life time, synchronizing data size for an application developer, and proving a secure system infrastructure. The challenge for execution on a distributed system is that partitioning an application executed locally or remotely is an NP-complete problem [1].

In this thesis, we enhance our previous work to provide an automatic, dynamic application offloading scheme. By extending the programming model on the Android phone with the flow-based programming (FBP) [2] paradigm, our framework can partition an application into processes and provision the requirements of each process. In FBP, an application is defined as a network of black box processes that exchange data across predefined connections via message passing, where the connections are specified externally to the processes. Thus, with this component-oriented feature of FBP, our framework can figure out the control flow and the data flow of an application before executing the application.

In dynamic situations where workloads and environments change over time, a comprehensive performance monitoring and modeling scheme is needed to make the decision of offloading

during the runtime. Thus, we add a profile-based policy manager and profiling service into our framework to make smart decisions based on the available computing and network resources during the runtime and the estimated performance gain from offloading certain processes. Our framework would run an application in advance with our tracing and profiling tools [3] to locate the hotspots in the application and estimate the performance gain with available servers in the cloud.

To minimize the efforts of using our framework, developers would use our tools to find the hotspots in their applications, and convert the hotspots into FBP components. The runtime libraries in our framework will monitor the available resources and decide how to offload the hotspot components. On most systems, our framework aims to reduce the execution time of the application, but either the developer and the user may override the default goal by setting their preferences to reduce the power consumption of the system, or to offload the application to a server which is more cost-effective.

The rest of this thesis discusses the related issues and explains our approach. Section II surveys the related research works. Section III presents the proposed architecture for profile-based dynamic offloading of mobile applications. In Section IV, we describe the experimental evaluation of the prototype. Finally, we conclude the paper in Section V.



## Chapter 2

# Background and Related Work

### 2.1 Introduction to Android

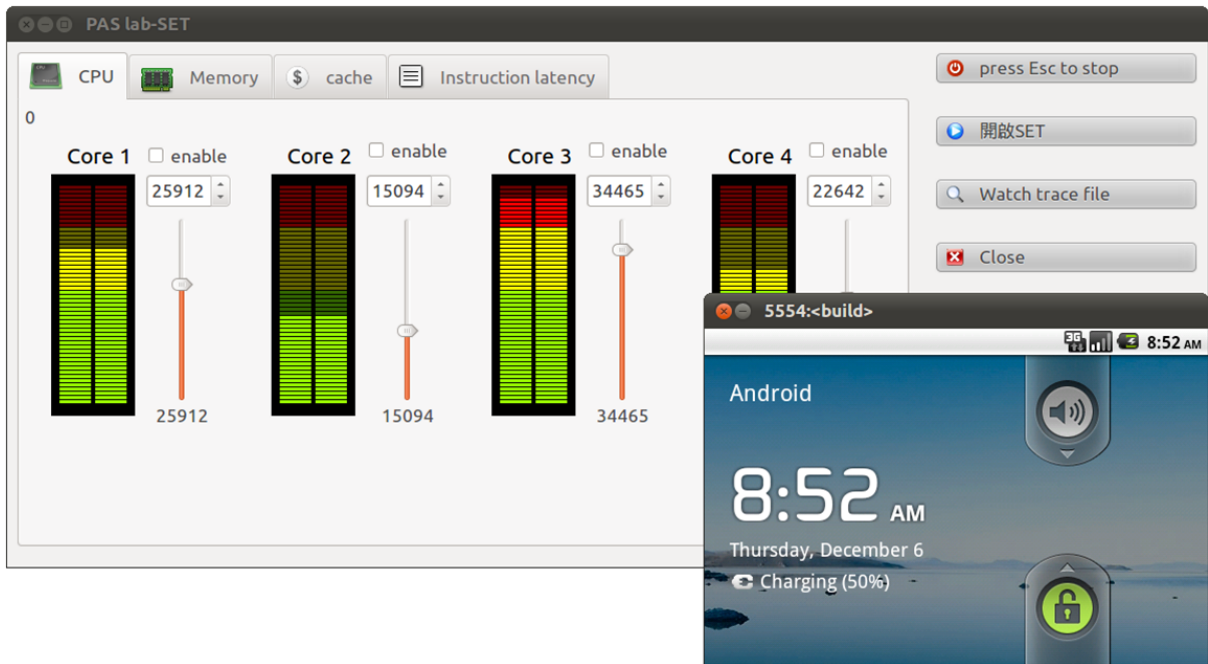
Android is a popular operating system based upon the Linux for smart phone and tablets. Android is a software bunch comprising not only operating system but also middleware and key applications. It is an open source operating system, created by Google, and available to be useful for other platforms and applications.

Android provides software development kit(SDK) for all developers. You can download, build and work on Android in a number of different ways. Developers have full access to the same framework APIs used by the core applications. We integrate our framework on Android and then focus on improving the performance and finding the more better migration strategy for Android applications.

### 2.2 Virtual Performance Analyzer

The virtual performance analyzer (VPA)[3] is a framework which provides software and hardware developers with the needed facilities, besides timing models, to make a virtual machine useful to performance evaluation with unprecedented profiling/tracing capabilities and effective tools to analyze important hardware and software interactions in the system. The key

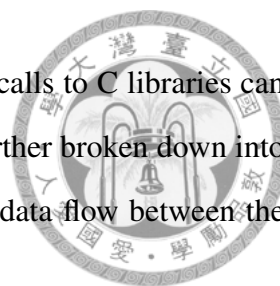
features that VPA framework provided are performance monitoring, power model, speed of data collection, non-intrusive profiling, and tracing. We use VPA to predict the performance gain and the power saving by using our framework. In order to minimize the efforts of using our framework, the application developers can use VPA to find the hotspots in their applications, and convert the hotspots into FBP components. Furthermore, we improve our VPA tool into an online version (as shown in Figure 2.1), developers can profile their application without any complicated installation.



**Figure 2.1** CloudVPA

## 2.3 MobileFBP

MobileFBP [4]’s goal is to enable the programmers to develop dataflow applications that can be executed in mobile-cloud environments. In this section, we present a high-level architecture of MobileFBP’s framework and the implementation of the MobileFBP runtime system. As illustrated in Figure 2.2, MobileFBP provides a programming environment where developers can easily describe the task components of an application which can be offloaded for remote



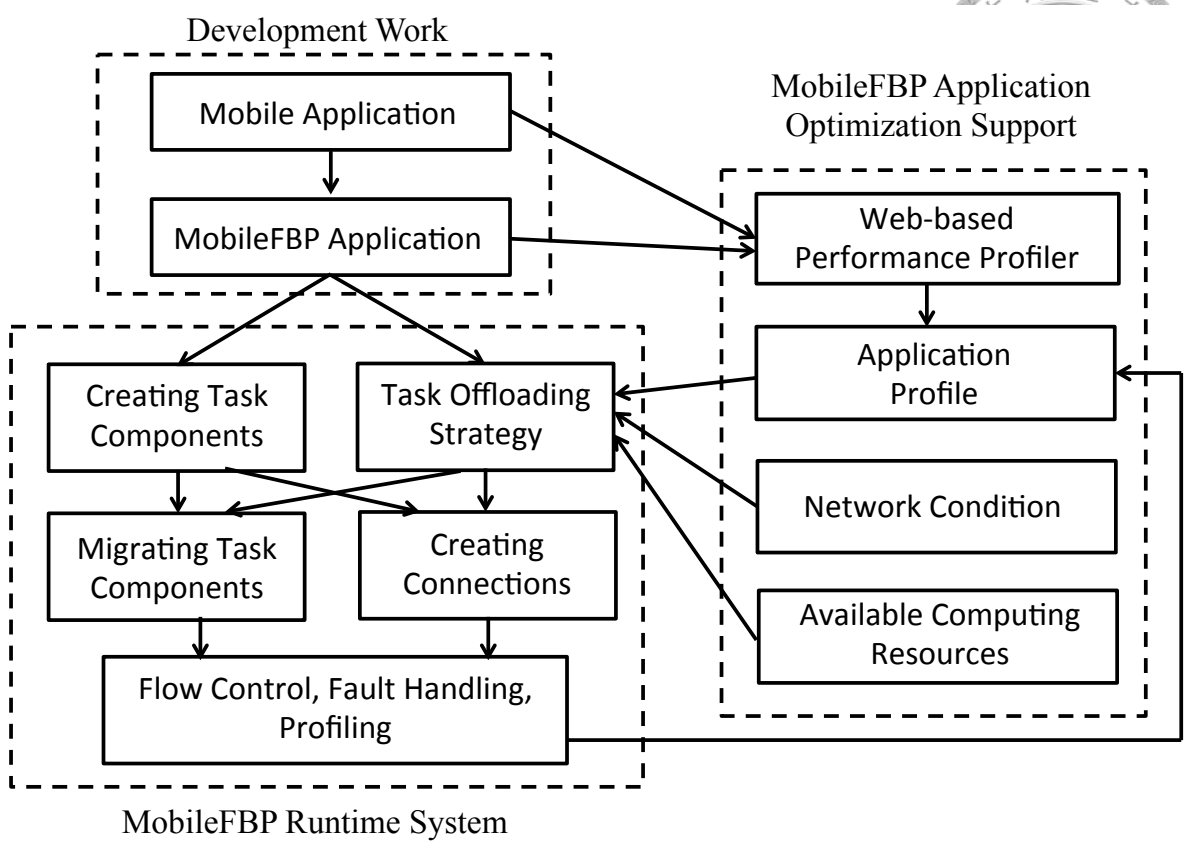
execution. A typical Android application written in Java with function calls to C libraries can be easily converted to FBP as one large task component initially and further broken down into multiple components by declaring the components and expressing the data flow between the components with the assistance from performance profiling tools.

During the runtime, MobileFBP's runtime system optimizes the application execution by offloading selected components to available high-performance processors when the network condition permits. To support performance optimization, profiling information is collected during the development time or the execution time to refine the optimization for future invocations of the task components. The task offloading process is dynamically handled by the runtime system. In case the network slows down or disconnects, or certain remote processors are not responsive, MobileFBP resumes the task components on the local processor. There is a performance penalty associated with the aforementioned situations, and that is a typical risk for any remote task execution.

The dataflow programming paradigm share a similar philosophy and may co-exist with the data parallel or streams paradigms that are widely used today for parallel and distributed processing, e.g. OpenCL , and MapReduce . Thus, it would be straightforward to convert existing data parallel applications into FBP or to utilize programs written in another language within a task component.

## 2.4 Support Vector Machine

Classify data is a common task in machine learning, *support vector machines*(SVM) are supervised learning models with associated learning algorithms that analyze data and recognize patterns, used for classification and regression analysis. Suppose some given data points each belong to one of two classes, and the goal is to decide which class a new data point will be in. In the case of support vector machines, a data point is viewed as a  $p$ -dimensional vector (a list of  $p$  numbers), and we want to know whether we can separate such points with a  $(p-1)$ -dimensional hyperplane. This is called a linear classifier. There are many hyperplanes that



**Figure 2.2** An overview of the architecture of the MobileFBP framework

might classify the data. One reasonable choice as the best hyperplane is the one that represents the largest separation, or margin, between the two classes. So we choose the hyperplane so that the distance from it to the nearest data point on each side is maximized. If such a hyperplane exists, it is known as the maximum-margin hyperplane and the linear classifier it defines is known as a maximum margin classifier; or equivalently, the perception of optimal stability. LIBSVM [5] is a popular library for SVMs and one of the most widely used SVM software. We integrate LIBSVM into our framework.

## 2.5 Related Work

The term *mobile-cloud computing* has been used recently to refer to (1) the collaboration between mobile devices and cloud services for accomplishing specific applications, and (2) a cloud computing environment composed by mobile devices [6, 7]. Our approach aims to ad-



dress the application development issues for both of the above aspects. So far, no simple solutions have been proposed to enable quick deployment of cloud services and dynamic task offloading. For a mobile application to benefit from a mobile-cloud environment by offloading tasks to a nearby server or a server in the cloud, the application needs to have its workload partitioned into tasks and properly select the tasks that to be offloaded, since task offloading does not guarantee any performance improvement. The communication overhead and the saving of execution time depend on the network condition and the configuration of the server, which may not be evaluated until the runtime.

Many previous works researched on the methods of re-designing or partitioning existing applications for task offloading. Some approaches rely on application developers to change the application code so that specific tasks in the application can be offloaded. For example, in Spectra [8], application developers provide *execution plans* and *fidelity metrics* to guide the runtime system on how to schedule the tasks in an application to achieve the quality of service specified by the fidelity metrics. During the runtime, Spectra monitors the available resources in the remote servers and eventually selects the execution plan which maximizes a user-provided utility function. In addition to the execution plans and fidelity metrics, the application developers need to manually identify offloadable tasks in the application and create remote services with API provided by Spectra. Chroma [9] tries to reduce the programming efforts of Spectra by allowing programmers to specify *tactics* with a declarative language.

To further reduce the programmer's burden in partitioning existing applications for task offloading, various automatic program partitioning schemes have been proposed to identify offloadable tasks with source code analysis and perform source-code modifications [10, 11, 12, 13]. While the static analysis perform well for certain high-level application development environment, the approach is language-dependent and has limited success in practice. Some schemes utilize the runtime system to perform coarse-grain task partitioning for applications which are written with high-level program constructs. For example, Coign [10] can automatically partition DCOM applications into coarse-grain client and server components without source-code modification. MAUI[13] proposed a scheme to offload the methods in a .NET

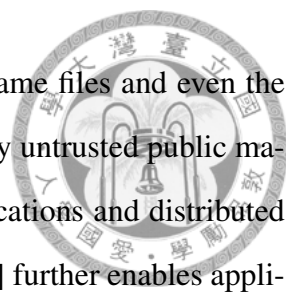


application with code instrumentation by the compiler and .NET's built-in runtime support for code serialization, reflection, and type safety.

Some recent approaches create new languages to aid the compiler and/or the runtime system in partitioning applications for special-purpose systems [14, 15]. For computers with heterogeneous processors, in Hydra [14], developers use a set of special components called Offcodes to express offloadable tasks which can be scheduled to run on the specialized processors such as GPUs and I/O processors. Developers are also responsible for creating communication channels to handle communications between Offcodes, which would be fine for special-purpose systems, but not for our mobile-cloud scenarios. For sensor networks, Wishbone [15] requires that the developer writes a program in WaveScript. The compiler uses a profile-based approach to evaluate the program and create the dataflow graph for the stream operators, and the runtime system performs graph optimizations, generate executable codes for the stream operators, and schedule the stream operators to run on sensor nodes. Our approach with FBP is similar to these approaches, but instead of creating new special-purpose languages, we choose to use a well-established programming paradigm and intend to support general-purpose applications.

Our application framework adopts many of the aforementioned ideas for today's mobile-cloud applications, where various types of computing resources in mobile devices and servers may need to collaborate on a single application in a highly dynamic network environment. The use of FBP allows the developer to express offloadable tasks and data flow explicitly at a coarse-grain or fine-grain level, depending on the target applications and systems. For offloading tasks to a remote server over a wide-area network, coarse-grain tasks with low data transmission overheads are better suited, and such candidates can be identified easily by our FBP runtime system. For offloading tasks to a local processor, the developer may create fine-grain tasks and let the runtime system make scheduling decisions. The FBP paradigm provides a good foundation for application partitioning, which is key to the success of parallel and distributed task processing in our opinion. Section III will further discuss the use of FBP to build mobile-cloud applications.

Finally, for applications to execute remotely, one may replicate or virtualize the application



execution environment so that a mobile application can access to the same files and even the peripheral devices. Cyber foraging [16] and data staging [17] use nearby untrusted public machines, i.e. surrogates, to improve the performance of interactive applications and distributed file systems on mobile clients. Based on a simiar concept, Slingshot [39] further enables applications to deploy *stateful* services dynamically in wireless hotspots. extends this earlier work by adding the capability of dynamically instantiating replicas of “stateful” applications. ThinkAir [18] enables the migration of a smartphone application to the cloud for parallel execution of compute-intensive tasks on multiple virtual machines. Virtual Phone as a Service (VPaaS) [19] is a framework for the end user to create a virtualized execution environment in the cloud which replicates the environment of user’s Android phone. In this paper, we use VPaaS to deploy the remote execution environment in the case studies.



## Chapter 3

# Framework and Implementation

To make a dynamic migration for a given MobileFBP application, the MobileFBP runtime system requires the information of application profile data, server status, user preference, network condition, and available computing resources. To reduce the burden on the mobile devices, we implement the collector and provider of the needed information as a remote service, called *Offload Advisory Service (OAS)*, which may be running on a nearby computer or the network router to reduce the communication latency. In addition, the OAS may also make the task offload decision for the client since the OAS server should outperform the mobile devices in solving performance optimization problems, especially for applications with many task components. Nevertheless, the MobileFBP runtime on the mobile client has its own task offload decision mechanism which can be used when the client is capable of making quick decisions on its own.

Then, we introduce our proposed framework in detail. In Section 3.1, demo how to use our MobileFBP API. In Section 3.2 describes that we integrate Cloud VPA into our framework to minimize the efforts for rewrite the Android application to FBP. Section 3.3 introduction the functionalities of the key components of OAS system. Section 3.4, we illustrates the workflow of OAS system.



### 3.1 FBP Programming model

Flow-Based Programming is a new/old approach to application development, based on a completely different way of thinking about building applications. Some of its roots can be traced all the way back to the early days of computing, yet it offers solutions to many of the most pressing problems facing application development today. In "Flow-Based Programmng" (FBP), applications are defined as networks of "black box" processes, which exchange data across pre-defined one-way connections. These black box processes can be reconnected endlessly to form different applications without having to be changed internally. It is thus naturally *component-oriented*, and this component-oriented feature is suitably for migration. The main declarations of FBP are *network declaration* and *component declaration*. Network declaration specifies the connectivities between boxes, and component declaration specifies the input/output of each box. In our work, we extend the programming model on Android smartphones with *JavaFBP* [2], a Java implementation of Flow-Based Programming.

To declare a FBP network, JavaFBP provides the following methods:

- *component* : define an instance (an FBP "process") of a component.
- *connect* : define a connection between components.
- *initialize* : define a connection including an initial data.
- *port* : define a port on a process (component).

For example, assume we define two components, Hello and World, which is defined in class a and class b respectively. In the initial, Component Hello receives the message from the port SOURCE. Then component Hello sends a message to component World through the port OUT to the port IN. Note that the work starts by using the method *go()*. The JavaFBP network declaration can be defined as Figure 3.1 :

Fortunately, there is also a diagramming tool (DrawFBP [20]), which can be used to define the network graphically, and which can actually generate the network definitions. Next,

```

public class xxx extends Network {
protected void define() {
    component("Hello", a.class);
    component("World", b.class);
    connect(" Hello.OUT", " World.IN");
    initialize("ok", component("Hello"), port("SOURCE"));
}
public static void main(String[] argv) throws Exception {
    new xxx().go();
}
}

```



**Figure 3.1** The network declaration of FBP.

we introduce that how to define a component, a JavaFBP component basically consists of 5 sections:

- *import statements* : Import the API package, the statement is :
  - *import com.jpmmorrsn.fbp.engine.\*;*
- *metadata* : The FBP components are described by declaring the metadata. Input and output port names will be coded on components using Java 5.0 "attribute" notation. This metadata can be used to do analysis of networks without having to actually execute the components involved. Here is an example of the attributes which describe one output port "OUT", and one input port "IN":
  - *@OutPort(value = "OUT", description = "Generated stream", type = String.class)*
  - *@InPort(value = "IN", description = "Count the packets", type = Integer.class)*
- *declares for ports* : Define the ports, e.g. the input port named "inport", and the output port named "outport", the ports should be declared as :
  - *OutputPort outport;*
  - *InputPort inport;*



- *openPorts()* method : Override the *openPorts()* method to connect the ports and metadata attributes.

- *outport* = *openOutput*("OUT");

- *inport* = *openInput*("IN");

- *execute()* method : Code the component by overriding the *execute()* method. Note that the exchange data between the components are packaged into the *Packet* data structure.

Figure 3.2 is the integral declaration of the component, here we define the component "Hello" as the example.

```
import com.jpMorrsn.fbp.engine.*;

@OutPort(value = "OUT", description = "Generated stream", type = String.class)
@InPort(value = "IN", description = "Count the packets", type = Integer.class)

public class Hello extends Component {
    OutputPort outport;
    InputPort inport;

    @Override
    protected void openPorts() {
        outport = openOutput("OUT");
        inport = openInput("IN");
    }

    @Override
    protected void execute() {
        Packet p = inport.receive();
        ...
        Packet p = create(s);
        outport.send(p);
    }
}
```

**Figure 3.2** The component declaration of FBP.

The application developer converts the java class into FBP components for offloading, and

remains the parts which cannot be offload, like the device specific functions or the user interface, e.g., GPS, GUI, etc.



1. As the instance SF starts, it creates three processes, one for each task component.
2. SF sends a special INIT message to trigger the execution of the ReadFile component, which reads data from `foo.txt`.
3. The ReadFile component reads data and sends the data out via its output port,
4. The QSort receives data from its input port, sort the data, and sends the sorted data out via its output port.
5. The WriteFile receives sorted data from its input port and write the data into a file.


The FBP application can be implemented with JavaFBP as Figure 3.3, where the component method generates an instance (an *FBP process*) of a component and the `connect` method defines a connection between two ports. The `initialize` method sends the data in the argument, i.e. "data.txt" in this example, to an input port of a component to trigger the execution of the component. The `Network` class define a JavaFBP program, which is called from the main program with the method `go()`.

## 3.2 Perform Analysis

To help the runtime system make informed decisions, the developer may run the application with several con

gurations to generate application profiles. The application profiles are automatically collected by the runtime system and will be used to guide the scheduling decisions later. Initially, developer design the android application as usual. The MobileFBP runtime system requires the application profile from the developer via the web-based performance profiler or the user via the MobileFBP runtime. At first, in order to locate the hotspots with the minimal efforts, the





```

Public class SortFile1 extends Network {
    @Override protected void define() {
        // Define the task components:
        component("ReadFile", "ReadFromConsole.class");
        component("QSort", "QSort.class");
        component("WriteFile", "WriteToConsole.class");
        // Define the initial condition:
        initialize("foo.txt", "ReadFile.INIT");
        // Define the connections:
        connect("ReadFile.OUT", "QSort.IN");
        connect("Sort.OUT", "WriteFile.IN");
    }
}

Public class MainActivity extends Activity {
    @Override protected void onCreate(
        Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // Create an instance to sort the file:
        SortFile1 SF = new SortFile1();
        // Start the sort:
        SF.go();
    }
}

```

**Figure 3.3** The JavaFBP code for a file data sorting application

developer use Cloud VPA to profile their applications with specific HW/SW configuration. After profiling, Cloud VPA return the analysis results and a suggestion to help developer rewrite the the heavy part of android application with FBP.

For exmple, we use Cloud VPA profile the Median Filter application, and then we can get emulated execution time, power consumption, function call graph, percentage relative to parent, the Hotspots, as shown in Figure 3.4. Then, we use MobileFBP API to wrap the heavy part as the unit for workload migration. It is an easy way to complete the job with a dozen lines of code. It is shown in Figure 3.1.

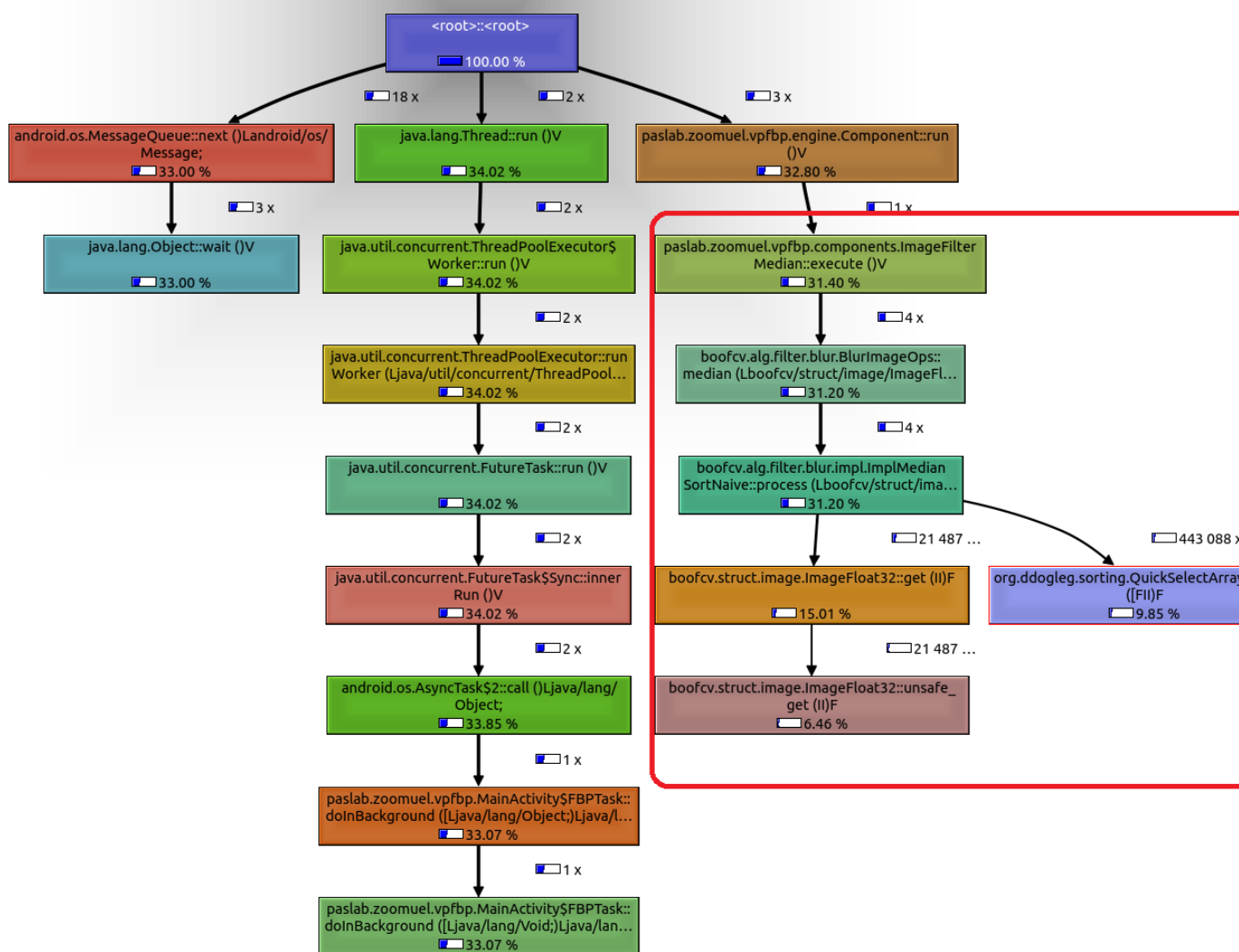


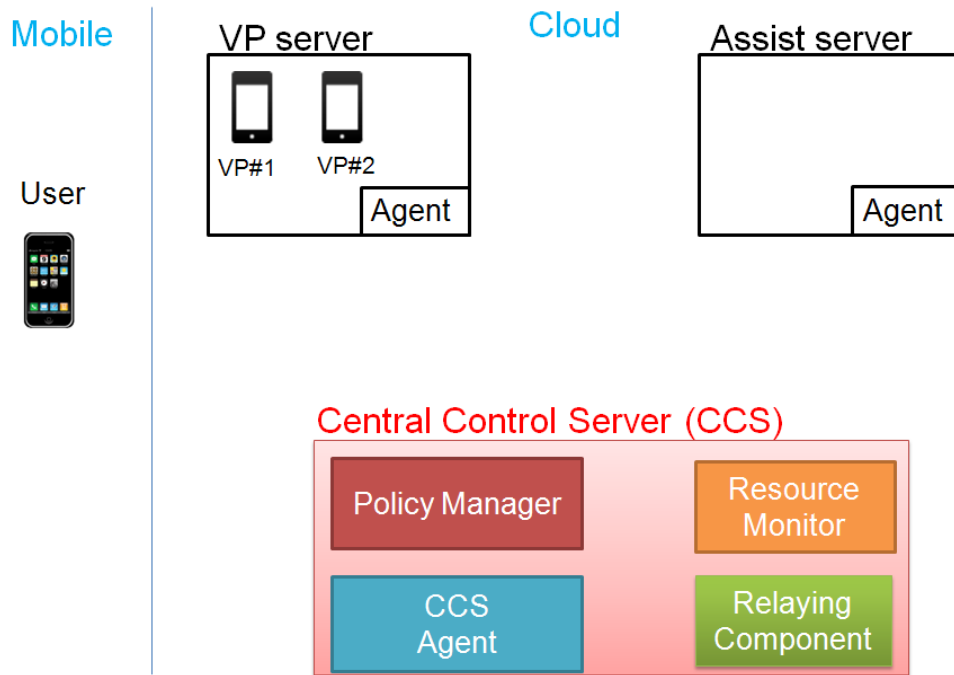
Figure 3.4 The profile data from Cloud VPA

After converting the hotspots into FBP components and declare the network data flow chart, we profile the FBP version of the application to get the performance analysis. During the run time, our proposed framework will issue the application and collect run time information on the Cloud for each FBP component.



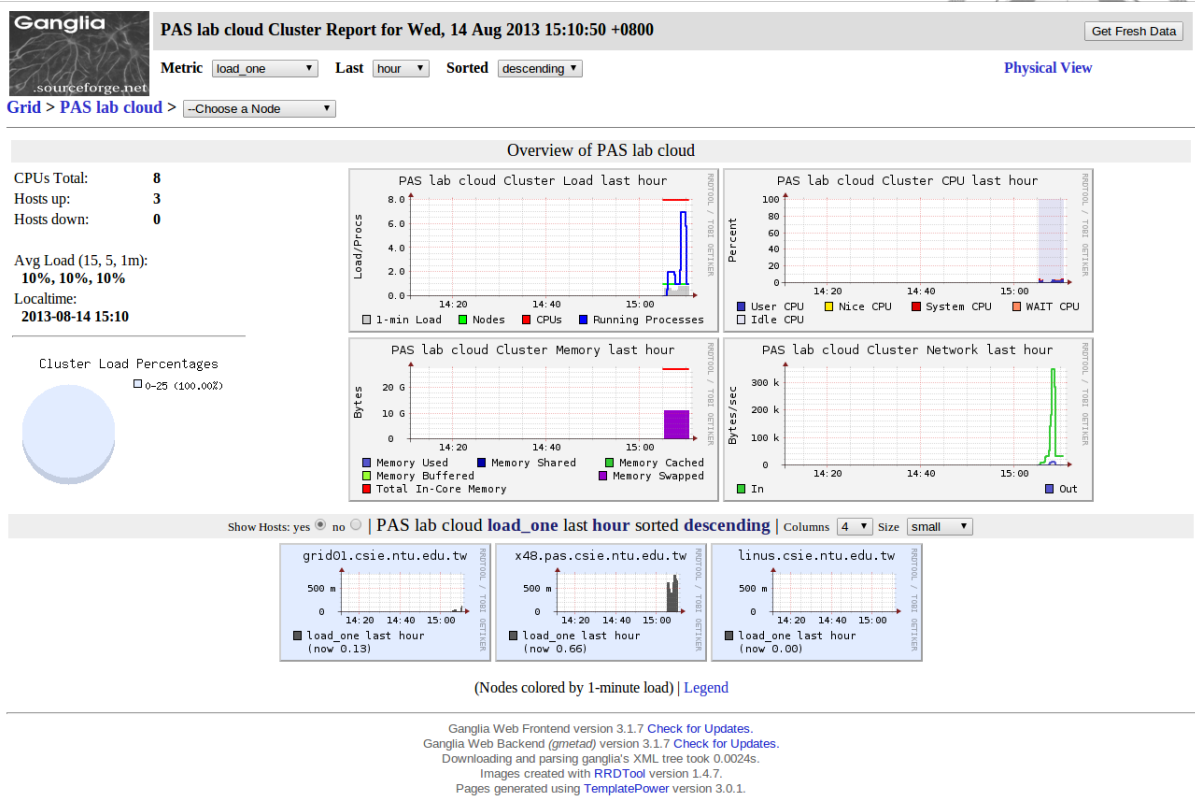
### 3.3 Offload Advisory System

In this chapter, we describe the design and implementation about our framework in detail. The *Offloading Advisory System* (OAS), as depicted in figure 3.5, consists following parts: Physical Phone, Virtual Phone, Virtual server, Assistant Server, and Central Control server.



**Figure 3.5** The overview of the Offloading Advisory System

- *Physical Phone*: User's mobile phone, there is a User Agent take charge of coordinate with policy manager for migration.
- *Virtual Phone*: it is user's virtual environment on the cloud for backup private data, only user can control their virtual phone.
- *Virtual Server*: the virtual phone's host. There is a JVM to process non-private data.
- *Assistant Server*: It's a general server for running JVM , it has powerful GPU or FPGA.
- *Central Control Server*:

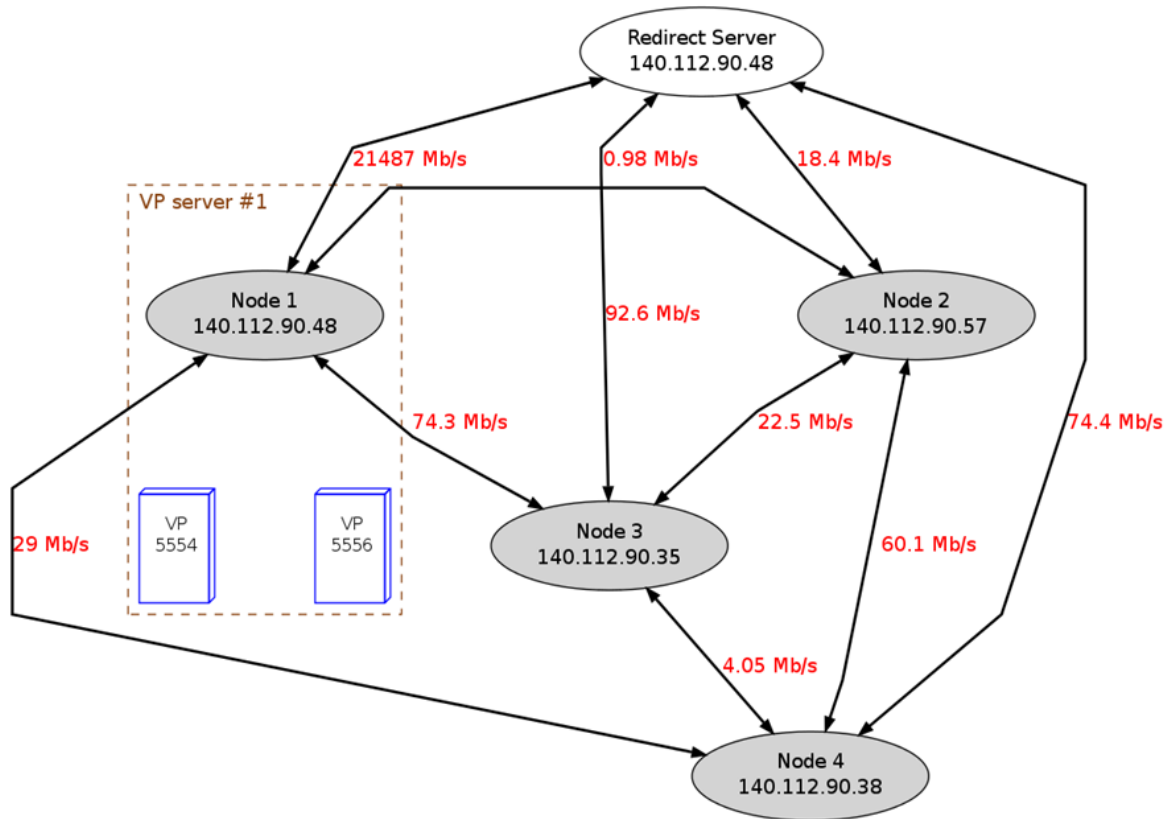


http://140.112.90.48/ganglia/?=hour&c=PAS+lab+cloud&sh=1

**Figure 3.6** Resource Monitor: ganglia

### 3.3.1 Resource Monitor

The OAS maintains a list of available servers for each client. The resource monitor dynamically updates the status of the servers on the list. We use open-source software, *Ganglia*[21] and *iperf*[22], to report the needed information from the servers on the list. Ganglia is a scalable distributed monitoring system used by many cloud service providers. The *iperf* utility is a network testing tool which is popularly used to measure communication latency and network bandwidth. The resource monitor also measure the latency and bandwidth from the OAS to the user's mobile device. However, since the network test consumes extra energy on the mobile device, the network test should not be performed while the mobile device is in a power-saving mode, and it is best be performed while the device was awake. We demonstrate our topology network condition on website ,as shown in Figure 3.6, Figure 3.7.

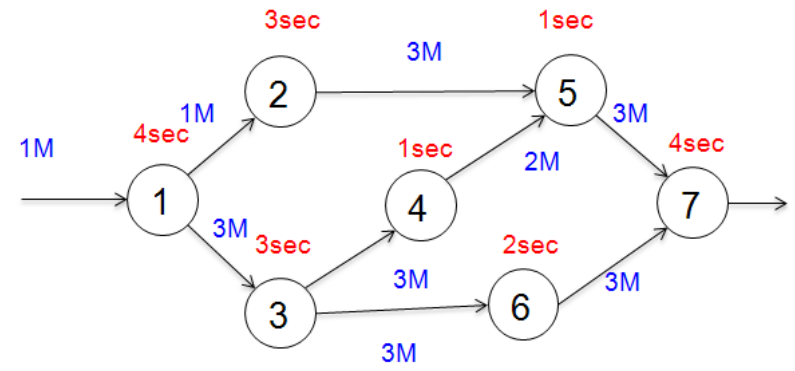
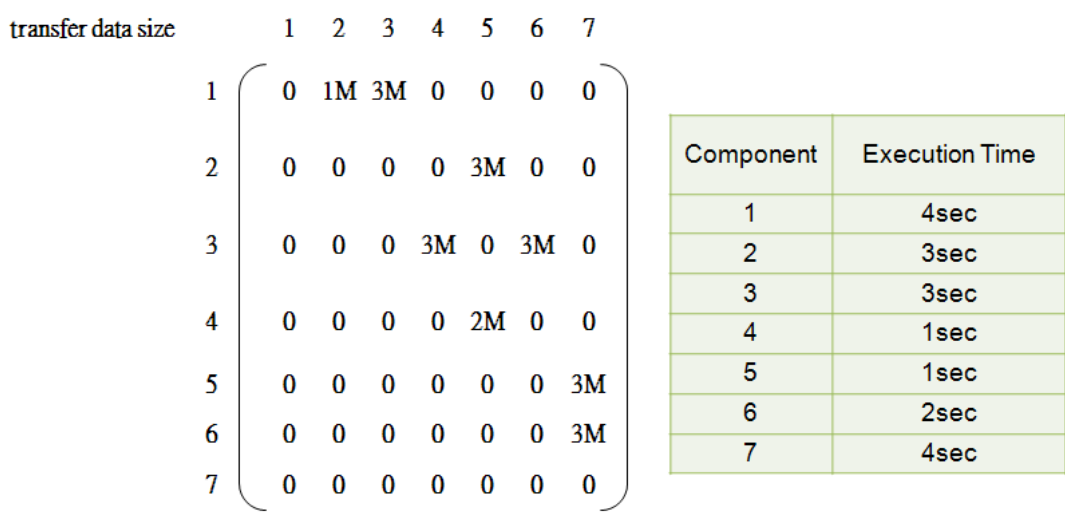


**Figure 3.7** Resource Monitor: iperf

### 3.3.2 Policy Manager

The *Policy Manager* (PM), just like a black box, can generate a migration plan according to the server status, hardware configuration, user preference, and run-time input data for a specific application. We can get an *Estimation Model*, which consists of execution time, power consumption, and transferred data size between components. For instance, suppose that we have an FBP application which can refactor into such a topology graph, and then we can evaluate the Estimation Model: execution time, power consumption, and transferred data size between components. Which can be measured by the MobileFBP framework, as shown in Figure 3.8.

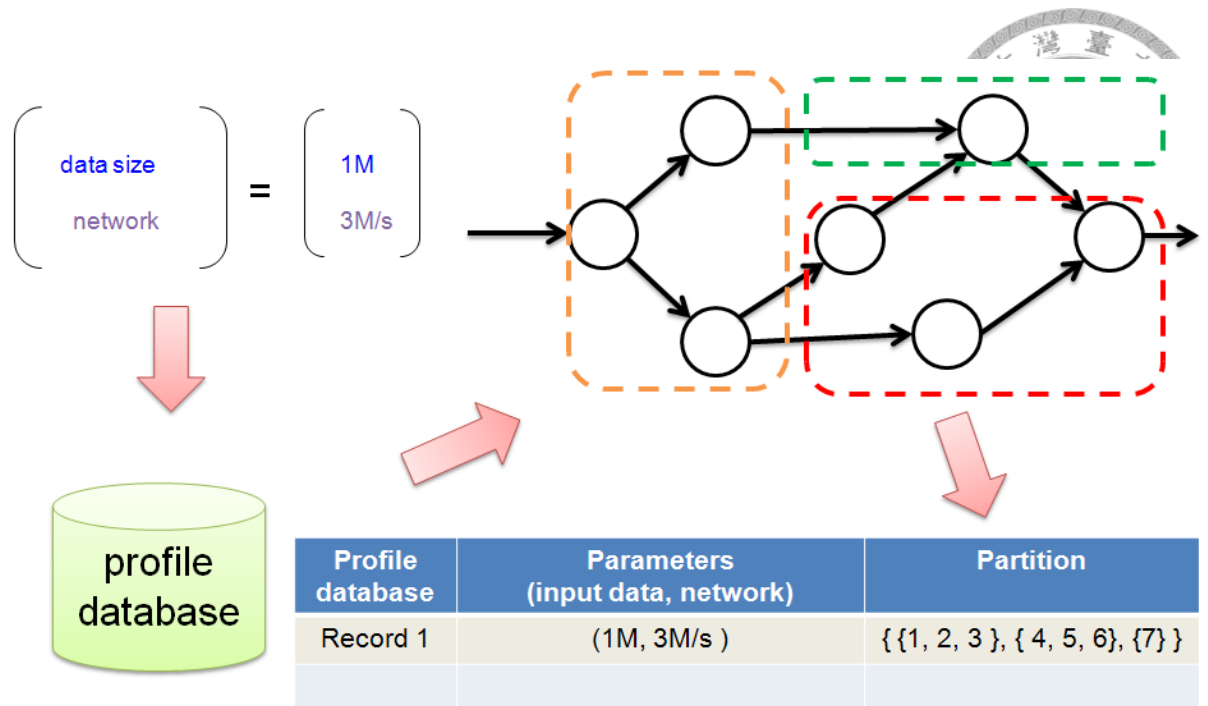
Our policy manager's aim is to find an optimal partition which can accelerate the application. Now we have the Estimation Model, network condition measure in the run-time, so we



**Figure 3.8** The Estimation Model: transfer data matrix and the table of the execution time

calculate the execution time by *Activity on Edge Network* (AOE Network) algorithm for each partition. Using these values, AOE calculates the longest path of planned activities to logical end points or to the end of the project, and the earliest and latest that each activity can start and finish without making the project longer. To get the best partition, we find the minimum execution time heuristically from all partitions, and then we can record the partition method corresponding to the *Parameters* to the profile database. Next when PM is given a set of *Parameters*, it checks the PD. If PD have a precious record which matches the same as the *Parameters*, it will send the migration plan to each server agent, as depicted in Figure 3.9.

In the development phase, we need developer define the parameters of their application to help our framework train and collect the profile data. Developer have to describe the type, the



**Figure 3.9** The record in our profile database

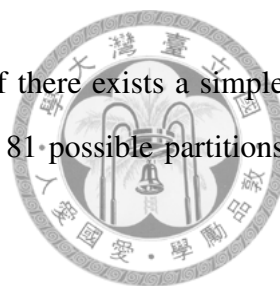
range of the parameters, and define a range of a segment, as shown in figure 3.10. And then our framework have a input generator to help train the profile database.

	segment 1	segment 2	segment 3	segment 4	segment 5
data size	0MB ~ 1MB	1.1MB ~ 2MB	2.1MB~3MB	3.1MB ~ 4MB	4.1MB ~ 5MB
	segment 1	segment 2	segment 3	segment 4	segment 5
network	0M/s ~ 1M/s	1.1M/s ~ 2M/s	2.1M/s ~ 3M/s	3.1M/s ~ 4M/s	4.1M/s ~ 5M/s

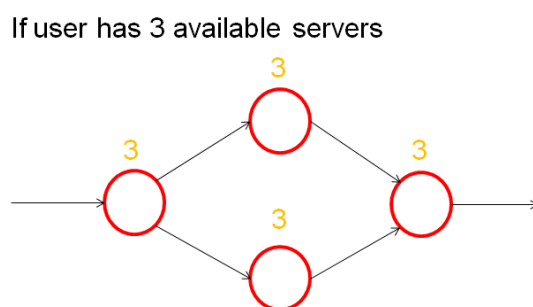
**Figure 3.10** Define the input parameters

However, we encounter some difficulties: First, heuristic search is not a feasible solution for dynamic offloading, because heuristic method is time-consuming and the topology may be a huge graph. (Ex: if the component has 10 component and 10 servers, there are  $10^{10}$  partitions.) Another is that *Parameters* may be a infinite set. The value of the parameters is continuous, so we can not always match the corresponding *Parameters* in our profile database.

Fortunately, the number of partition is a finite number. Supposed that there are many computing resource on the cloud, then we could only allocate 3 powerful servers on the cloud, if



Cloud service provider only offers 3 available servers for each user. If there exists a simple application which can be refactor into 4 components, then it has only 81 possible partitions with 3 available servers, as depicted in Figure 3.11.



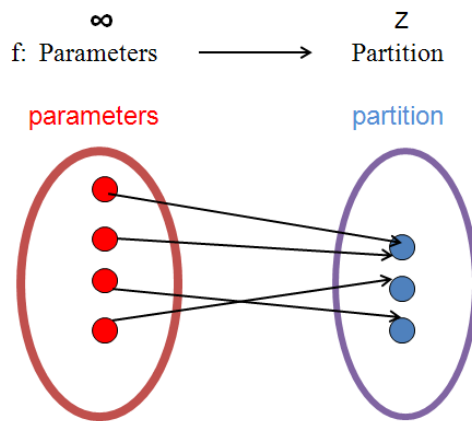
Then we have  $3 * 3 * 3 * 3 = 3^4 = 81$  partitions  
**Figure 3.11** The number of Partition is finite

PM is just like a function  $f$  that takes input *Parameters* and returns an output *Partition*. The set of all combinations of *Parameters* is called the domain, while the set of *Partition* is called the codomain.  $f$  is a function from  $\infty$  to  $\mathbb{Z}$ . The cartesian product of two sets *Parameters* and *Partition* is the set of all ordered pairs, written  $(x,y)$ , where  $x$  is an element of *Parameters* and  $y$  is an element of *Partition*. The  $x$  and the  $y$  are called the components of the ordered pair. The cartesian product of *Parameters* and *Partition* is denoted by  $Parameters \times Partition$ . need a efficient way to train our PM, if we find a new ordered pair  $(x,y)$ .

SVMs are supervised learning models with associated learning algorithms that analyse data and recognize patterns, used for classification and regression analysis. SVMs can efficiently implicitly mapping their inputs into high-dimensional feature spaces. Even the input is out of the domain, SVM would return the approximate output.

In the beginning, if developer only profile the application for a few times, Offload Advisory System will train a SVM which contains only some ordered pair  $(x,y)$  for the specific application, shown in Figure . Given a input parameter which is the domain *Parameters*, SVM return



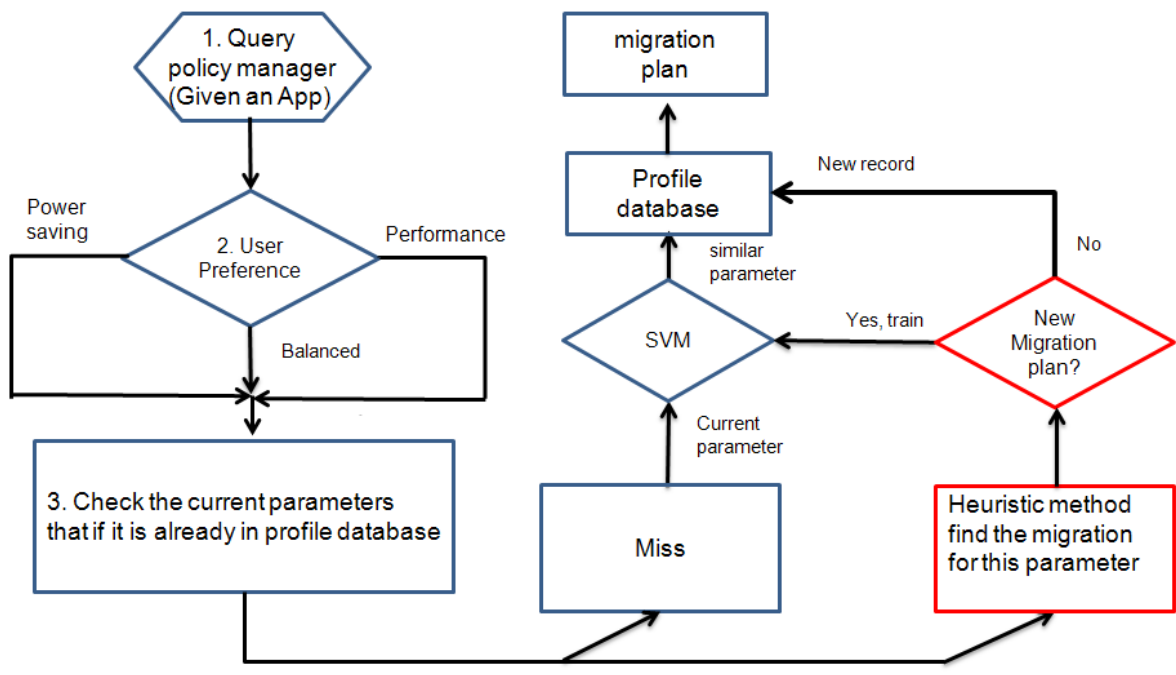


**Figure 3.12** Function  $f$  that takes input *Parameters* and returns an output *Partition*

the corresponding partition that is in the codomain *Partition*. If the input is out of the domain, SVM will return a approximate point  $y$  in the codomain quickly by precomputed calculated. Offload Advisory system will use Cloud VPA and AOE algorithm to find the best migration on the backend server and then trains the new training set to the SVM, so SVM can recognize the new input *parameter* to return right partition next time. With this mechanism, SVM has both learning and evolution ability to adapt the dynamic environments. The Figure 3.13 illustrate the flowchart of mechanism for making the migration plan

### 3.3.3 Central Control Server Agent

*Central Control Server Agent* (CCSA) is responsible for interacting with the User Agent. Mobile user is continuous moving and on a private network, but the cloud server is in the global address realm. It make difficult for two nodes to contact each other directly. Mobile user can send data to cloud server with a global IP, but cloud server cannot send data to the user phone behind a NAT. We reference [23] Hole punching technique, Return server agent works just like Rendezvous server to solve this problem. So even mobile phone is unstable, cloud server still can send data to mobile phone on private network. Besides, CCSA also charge of sending migration plan to mobile phone.



**Figure 3.13** The flowchart of mechanism for making the migration plan

### 3.3.4 Relaying Component

*Relaying Component (RC)* works for a computing result collector on the cloud. Since every component cannot connect user phone and sending the final result directly, Relaying Component will gather all the final computing result and forward it to Central Control Server Agent.

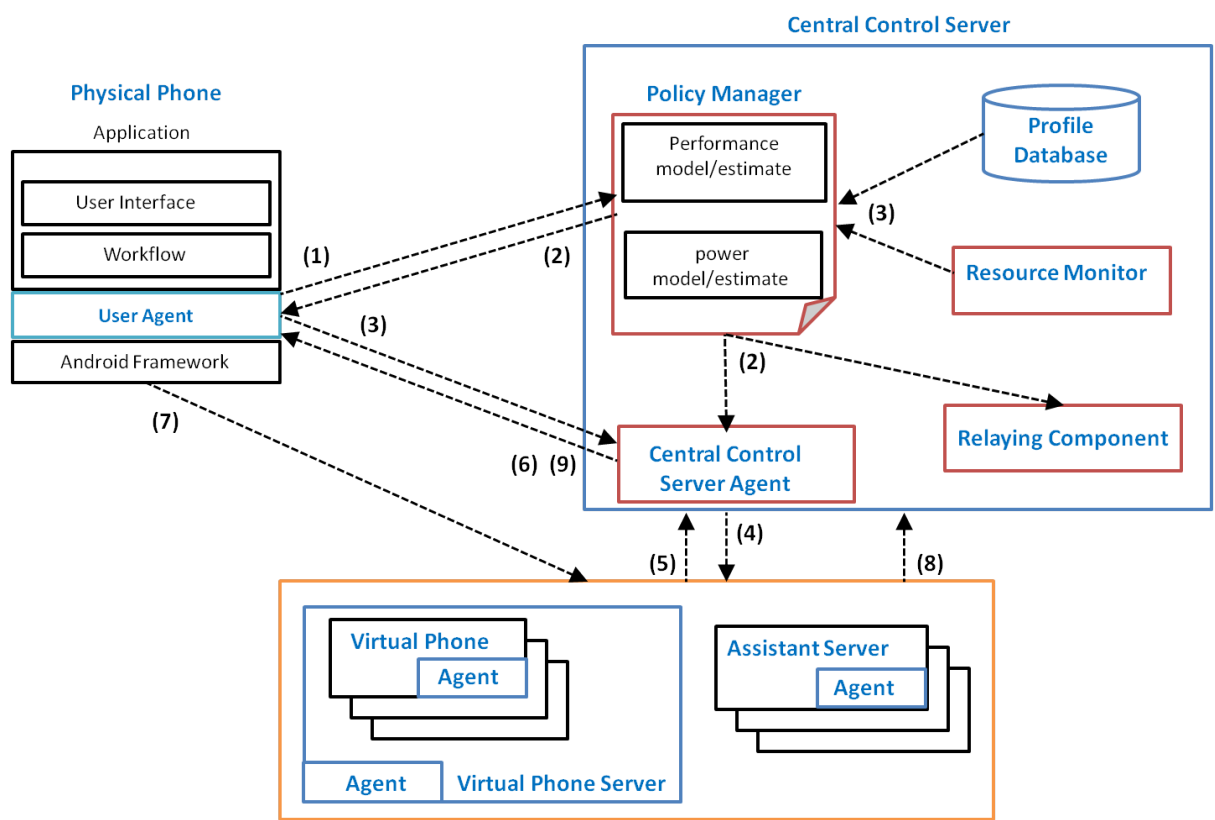
## 3.4 The procedure of Offload Advisory System

The procedure of our framework in Figure 3.14 are described as the following:

1. Notify Policy Manager that the user want to migrate and bind the socket to Central Control server.
2. Create a CCSA and RC for the user and send CCSA port to the user.
3. Collect resource information and generate migration plan. Mobile user uses the port to connect CCSA.



4. Send migration plan to each Server Agent (including virtual phone). Each server agent will create the component in the local network and block on Read Socket.
5. After step 4 each server agent (on the cloud) ACK ready flag to CCSA.
6. CCSA inform Mobile device that all components are ready.
7. Mobile device starts computation and offloading.
8. RC return the result to redirect server.
9. CCSA returns the result to mobile device.
10. After all work is done, all agents feedback the component run time information to train the profile database.



**Figure 3.14** The flowchart of architecture of Offload Advisory System



## Chapter 4

# Experimental Results

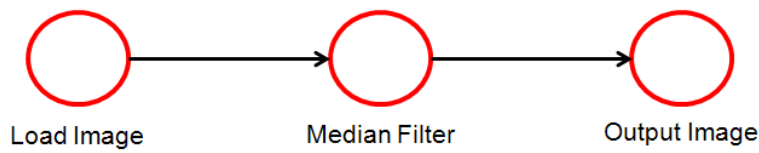
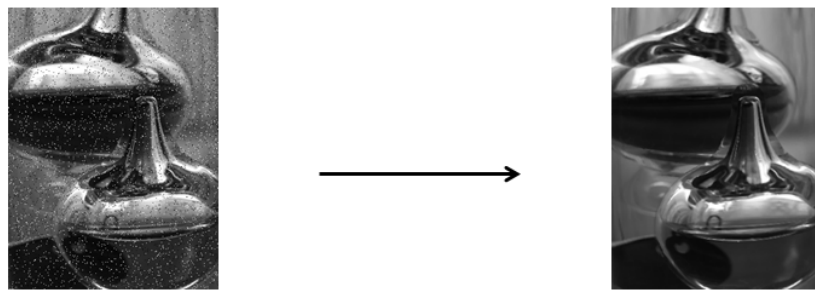
In this chapter, we describe our experiments in some case studies, and the experiment demonstrate how Offload Advisory interact with MobileFBP system. The experiment illustrate how to set up the environment, a tutorial to use our tool, and rewrite the application with MobileFBP API. At last, the experiment data demonstrate the benefit from our Offload Advisory System.

In Section 4.1 introduce the scenario, median filtering which is a proper instance to reveal the performance with our proposed framework. And then illustrate how to profile the application and rewrite with MobileFBP API. In Section 4.2 we evaluate the application ,and we compare the application on different platform or on different environmental condition to demo our SVM can make a better migration plan.

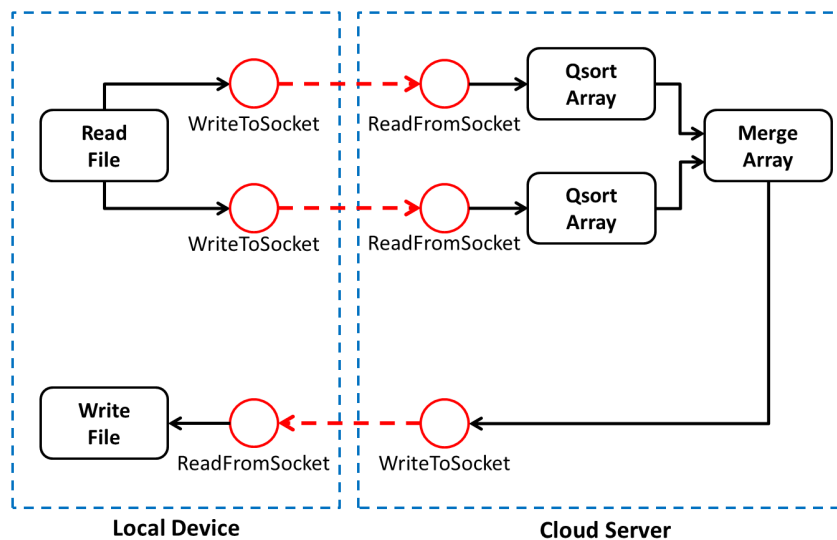
### 4.1 Case Study: Median Filter

The median filter is often used in signal processing, it is often desirable to be able to reduce noise in an image. The median filter is a non-linear digital filtering technique. The median filter considers each pixel in the image in turn and looks at its nearby neighbours to decide whether or not it is representative of its surroundings. The main idea is to run through the signal entry by entry, replacing the pixel value with the median of those values. The median is calculated by first sorting all the pixel values from the surrounding neighbourhood into numerical order

and then replacing the pixel being considered with the middle pixel value.

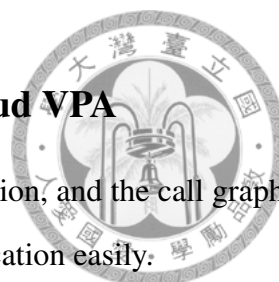


**Figure 4.1** The topology of the median filter program



**Figure 4.2** The migrating version of Median Filter

Our median filter application is based on BoofCV [24] and we rewrite the application to demonstrate the migration with FBP. BoofCV is a pure JAVA computer vision library which is suitable for android application development. The median filter's complexity is  $O(w \times h \times r^2)$ ,  $w$  and  $h$  are the image's width and height and  $r$  is median radius. The network schematic diagram is shown in Figure 4.1 and the migrating version generated automatically by our agents is shown in Figure 4.2. The environment setup is shown in TABLE 4.1



### 4.1.1 Profiling the Median Filtering Program with Cloud VPA

Initially, Cloud VPA helps us find the heavy part of the android application, and the call graph is shown in Figure 4.3. By consulting the data, we can dissect the application easily:

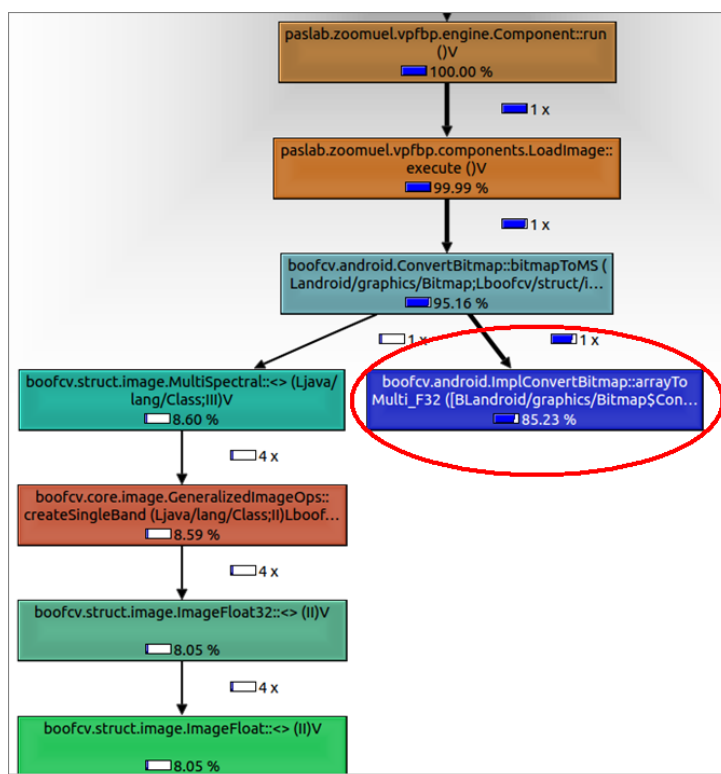


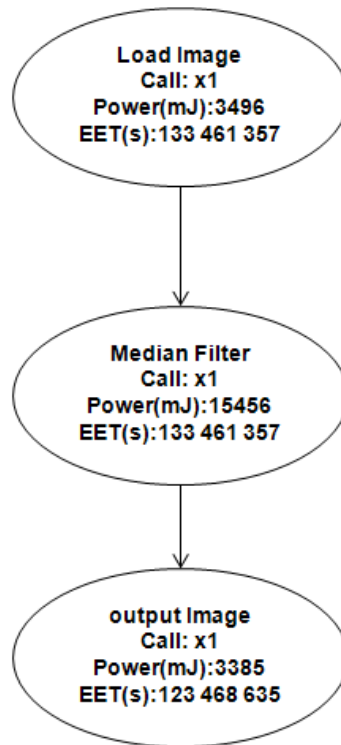
Figure 4.3 The call graph of the Median Filter program

Then, we rewrite the application with FBP API, such as figure 4.5

In addition to Android code, Cloud VPA can get execution time and power consumption of each component in detail, as shown in Figure 4.4.

### 4.1.2 Evaluation

In our experiment, the mobile phone, a *HTC Desire HD*, contains a 1 GHz Qualcomm 8255 single-core processor with 768MB of memory, running Android version 2.3.5. The assistant server contains a 3.4 GHz Intel quad-core Core i7-2600 processor with 12GB memory, running Ubuntu 12.04.2 operating system. The Center Control Server shares the same configuration as the assistant server. A 802.11g network router is used to connect the mobile phone wirelessly



**Figure 4.4** The call graph of the Median Filter with MobileFBP API

to the servers. The servers are connected to the router over wired 1Gbps Ethernet. The size of the image is 640 by 480 for both the input and output of the application for the test case.

**TABLE 4.1** The experimental environment of case study I

<b>Physical Phone: HTC Desire HD</b>	
CPU	1 GHz Qualcomm 8255 single-core
Memory	768MB
OS	Android 2.3.5
Wi-Fi	802.11g
<b>Assistant Server 1</b>	
CPU	3.4GHz Intel quad-core Core i7-2600
Memory	12GB
OS	Ubuntu 12.04

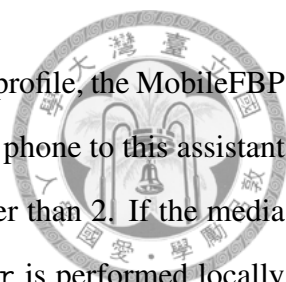
Figure4.6 compares the execution time of Media Filter under static task assignments for various median radiuses. When the task is executed locally on the mobile phone, the execution time increases with  $r$ , and the user has to wait for 2 minutes when the median radius is set to 10. On the other hand, when the task is migrated to the server, it can be completed within 14



```
@InPort(value = "IN",
        description = "BoofCV's MS<ImageFloat32>",
        type = Object.class)
@OutPort(value = "OUT",
         description = "BoofCV's MS<ImageFloat32>",
         type = Object.class)
public class MedianFilter extends Component{
    OutputPort opt;
    InputPort ipt;
    protected void execute() throws Exception{
        Packet p = ipt.receive();
        MS<ImageFloat32> image =
            (MS<ImageFloat32>)p.getContent();
        int height = image.getHeight(),
            width = image.getWidth();
        MS<ImageFloat32> output =
            new MultiSpectral<ImageFloat32>(
                ImageFloat32.class, width, height,
                image.getNumBands());
        for(int i=0;i<image.getNumBands();i++){
            BlurImageOps.median(image.getBand(i),
                                output.getBand(i), 10);
        }
        p = create(output);
        opt.send(p);
    }
}
```

**Figure 4.5** Wrapping the MediaFilter function call as a FBP task component





seconds even when the median radius is set to 10. Given this application profile, the MobileFBP runtime system will decide to offload Media Filter from this mobile phone to this assistant server under a perfect network condition when the media radius is larger than 2. If the media radius is smaller than 2, or the network condition is poor, MediaFilter is performed locally on the mobile phone.

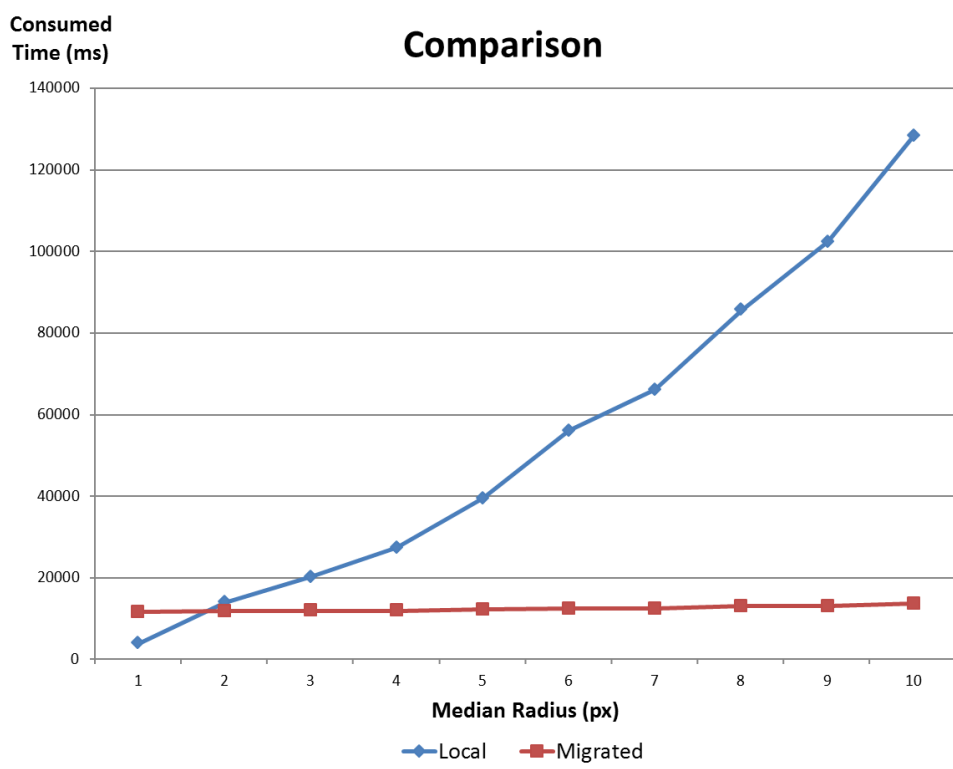
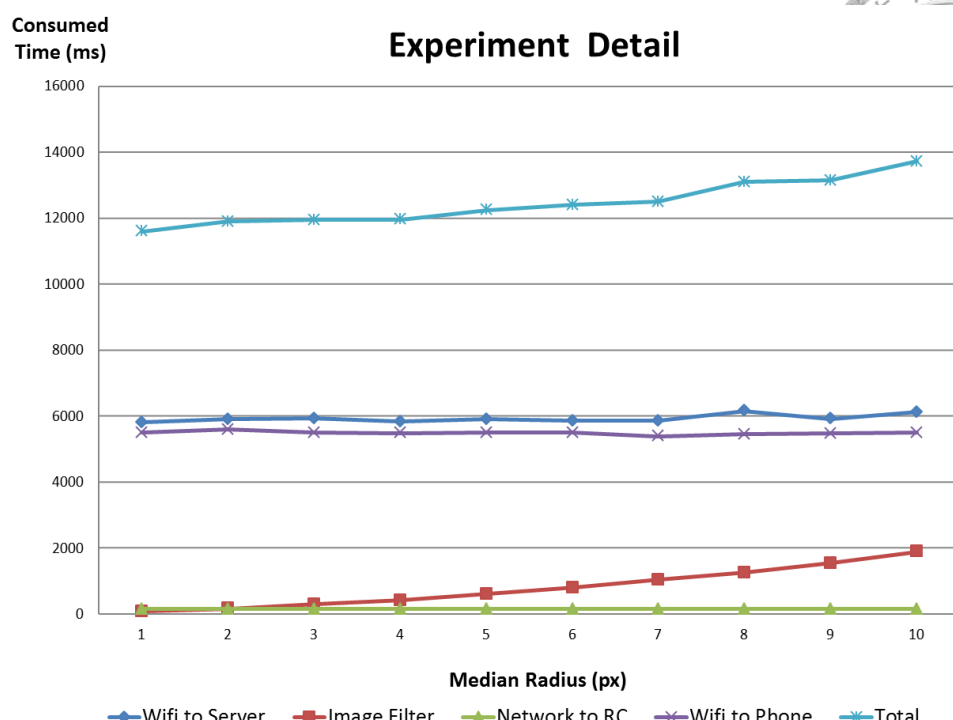
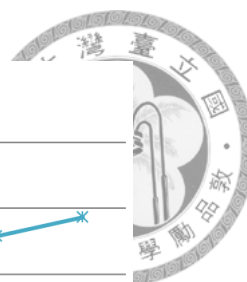


Figure 4.6 Execution time of MediaFilter with static task assignments

In fact, MobileFBP framework profiles the application execution in a greater details. As shown in Figure 4.7, the network overheads are significantly higher than the execution time of MediaFilter on the server. It takes the mobile phone roughly 6 seconds to transfer the image to the assistant server over the 802.11g WiFi network. The computation time for MediaFilter increases with the median radius, but is less than 2 seconds. The overhead for the Center Control Server to relay the results for the assistant server is insignificant since the two servers are connected with high-speed Ethernet. The overhead for the mobile phone to receive the data back is somewhat lower than 6 seconds since the size of the filtered image is smaller.

Obviously, the size of the image affects the task offloading decision. Once the OAS obtains



**Figure 4.7** Analysis of network overheads for task offloading

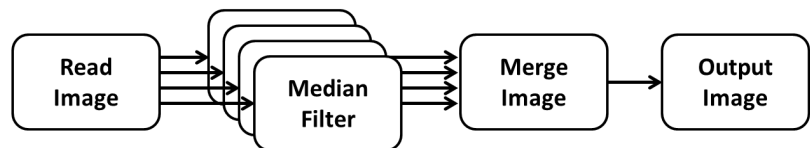
the application profiles for different image sizes, it can make a decision on both image size and the media radius. As we mentioned in Section III, instead of asking the application developers to conduct the sort of performance analysis in this example to detail an execution plan, it is better that the analysis is automated by the runtime system and the OAS on the Center Control Server. The application developer or the user can generate a range of test cases for the OAS to produce a more representative application profile.

Note that the task components which perform peripheral I/O operations, i.e. ReadImage and OutputImage, need to be done locally on the mobile phone; otherwise, they will fail to execute. When the MobileFBP runtime system fail to execute a task component, it records the cause. If causes such as I/O failure are recorded for a task component, the MobileFBP runtime system will not consider to execute the task component remotely.

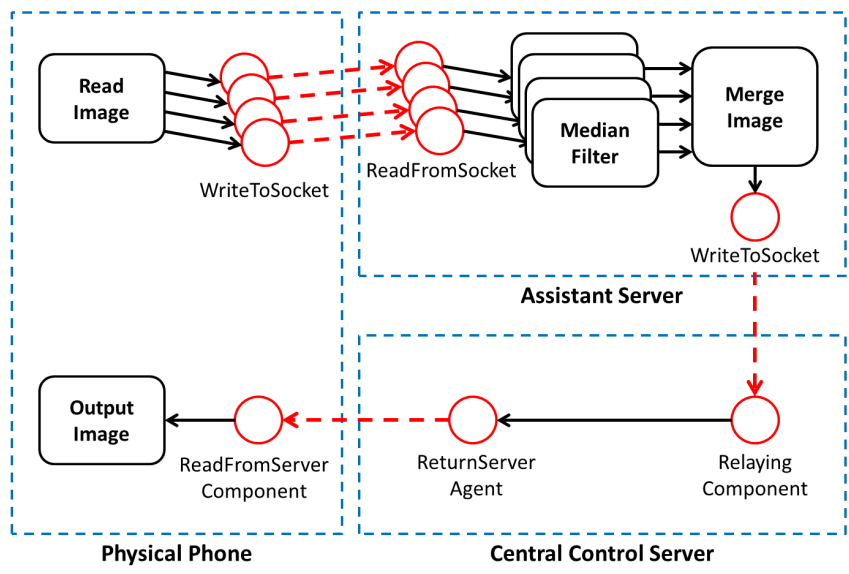


## 4.2 Case Study: 4-way parallel Median Filtering

We modify the program into a parallel 4-way median filtering to demonstrate our FBP framework is conducive to fully utilize cloud system's hardware. The application take the input image into four equal portions then pass them to a filter component separately. The schematic diagram is shown in Figure 4.8.

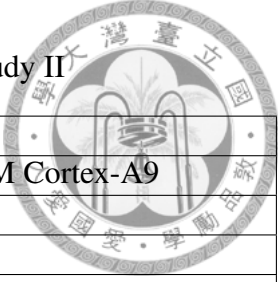


**Figure 4.8** The topology of the 4-way parallel Median Filtering



**Figure 4.9** The migrating version of Median Filter

Our agent will migrate all the four filtering component to cloud as Figure4.9 and as long as the cloud server has enough cores, the migration can also benefit from parallelism. The environment setup is shown in TABLE 4.2

**TABLE 4.2** The experimental environment of case study II


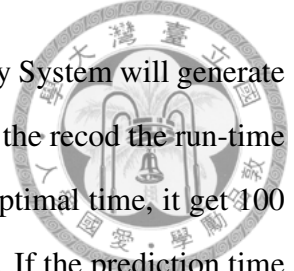
<b>Physical Phone: HTC one X</b>	
CPU	1.5 GHz quad-core ARM Cortex-A9
Memory	1GB
OS	Android 4.1.1
Wi-Fi	802.11g
<b>Assistant Server 1</b>	
CPU	3.4GHz Intel quad-core Core i7-2600
Memory	12GB
OS	Ubuntu 12.04
<b>Assistant Server 2</b>	
CPU	2.40GHz Intel i3 330M processor
Memory	4GB
OS	Ubuntu 12.04

### 4.2.1 Evaluation

In this section, we measure the smart decision of our policy manager based on different environmental parameters. The input is a  $640 \times 480$  image, the executed platform is HTC one X(4-cores) which contains a 1.5 GHz quad-core ARM Cortex-A9 processor with 1GB of memory, running Android version 4.1.1. The assistant server 1 contains a 3.4 GHz Intel quad-core Core i7-2600 processor with 12GB memory, running Ubuntu 12.04.2 operating system. The assistant server 2 contains a 2.40 GHz Intel i3 330M processor with 4GB memory, running Ubuntu 12.04.2 operating system. The Center Control Server shares the same configuration as the assistant server. A 802.11g network router is used to connect the mobile phone wirelessly to the servers. The servers are connected to the router over wired 1Gbps Ethernet.

In order to compare the performance on different cores and discuss the factors(e.g, network, median radius) which may affect the policy manager to migrate the workload to cloud dynamically, we continue use our case study II program, parallel 4-way filtering, to demonstrate the flexibility of our framework.

The first experiment, the input parameters is consist of Data size, Radius, Server1. Figure



4.10 illustrates the segment of the input parameter. The Offload Advisory System will generate the input parameter according the definition made by the developer, and the record the run-time information in the profile database. If the prediction time is equal the optimal time, it get 100 points. If the prediction time is equal the local execution, it get 0 points. If the prediction time is bigger than local execution, it will get negative score.

	segment 1	segment 2	segment 3	segment 4
Data size	1MB ~ 2MB	2.1MB ~ 3MB	3.1MB ~ 4MB	4.11MB ~ 5MB

	segment 1	segment 2	segment 3	...	segment 39
Radius	10	11	12	...	50

	segment 1	segment 2
Server1	Available	Not

**Figure 4.10** The input parameters of case study II

Before we validate the accuracy of the trained-svm, we introduce our formula to calculate the SVM accuracy.

$T_{local}$ : The local execution time on physical phone

$T_{predict}$ : The execution time predicted with SVM

$T_{optimal}$ : Optimal execution time

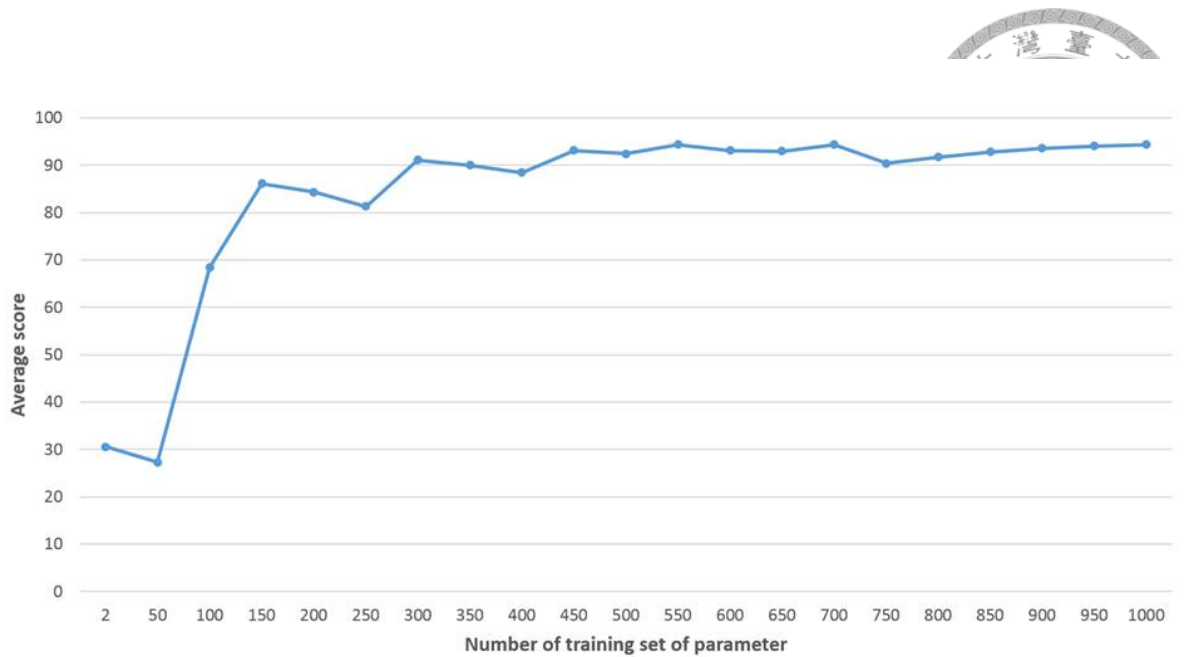
$$scores = \frac{T_{local} - T_{predict}}{T_{local} - T_{optimal}}$$

If  $T_{predict} = T_{optimal}$ ,  $scores = 100$

If  $T_{predict} = T_{local}$ ,  $scores = 0$

If  $T_{predict} > T_{local}$ ,  $scores < 0$

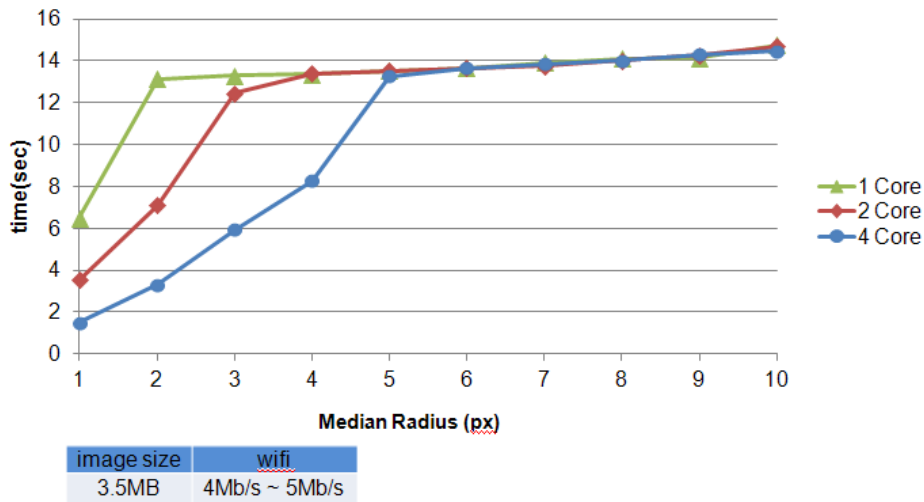
Figure 4.11 illustrates the the evaluation of the SVM prediction. The parameter are following: image (4MB 5MB), wifi (4Mb/s 5Mb/s ), radius (10 50), Server1 (Available or not). In



**Figure 4.11** The evaluation of the SVM prediction

the beginning, the SVM trained only 2 record, so it got only 30 points. If SVN trained more 48 training sets, it got lower scores. The SVM got lower accuracy, because the prediction time is longer than local execution time. After that, we trained more training sets to SVM, the accuracy was getting better. When the training sets is more than 300, it converges at a smooth line, it show that we can utilize the SVM as our policy manager.

Figure 4.12 is the evaluation result, we are able to find that when increasing the radius of filter, the excution time of different cores grows up exponentially and then converge at a rising slowly line. When the excution time of local site is longer than remote excution time, our policy manger will maigrate the components to cloud. For 1-core, limited by single core, it processes each filter component sequentially, all components are excuted in remote site when the radius  $> 2$ . For 2-core, it offloaded all components to cloud when the radius is bigger than 3. For 4-core, all componets can be processed on physical phone parallely, it can afford all components when radius is smaller than 7. After increasing the radius, policy manager will migrate all workloads to cloud.



**Figure 4.12** The execution time of different filter radius

Figure 4.13 depicts the execution time on each network condition when the filter radius is fixed at 6 px . We slow down the bandwidth to simulate different network condition. It shows that, for 1-core, is spends 97s on physical phone, even bandwidth is very low (2Mb), the policy manager will migrate all components to cloud, because of its weak processing capability. For 2-core it migrates all components to cloud when the bandwidth (4Mb ~ 10Mb). For 4-core, it utilizes 4 cores to running the program within 18 seconds. It runs the application on local site, because the 4 cores can process the 4 components parallely. However, it also can be benefit from offloading when bandwidth raise up to 8Mb.

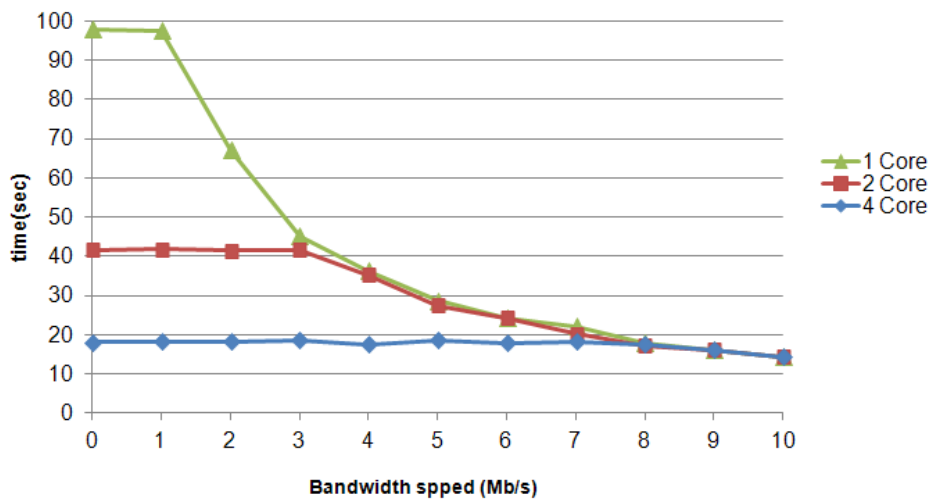


image size  
4.3MB

**Figure 4.13** The execution time of different quality of network





## Chapter 5

# Conclusion and Future work

### 5.1 Conclusion

In this thesis, we build the Offload Advisory System by improving our previous work on Virtual Performance Analyzer to minimize the efforts for profiling applications and locating the hotspots and helps developers rewrite Android apps with FBP for remote execution. Our resource monitor collects environmental information during runtime and enables the OAS to make dynamic offloading decisions. A well-trained SVM is capable of giving user quick, near optimal migration plan in our case studies by considering the server status, environmental parameters, hardware configuration, user preference, and run-time input data.

### 5.2 Future Work

Performance profiling is very time-consuming. For the Cloud VPA to profile a application which spends about 1 minute, it takes about 1 hour as the emulator needs to estimate execution time with timing models, which can be speeded up if the profiling can be done. Based on AOE network algorithm, our framework can estimate the execution time for the components which are single-threaded. However, if a component is multi-threaded or there are multi-components which run on a single core, then our current profiling mechanism may not report the execution

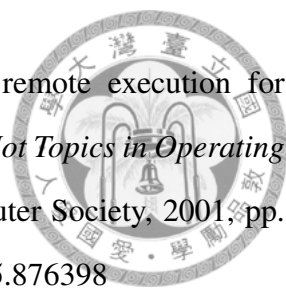
time actually. To fix this issue, the instrumentation for FBP component needs to be enhanced.

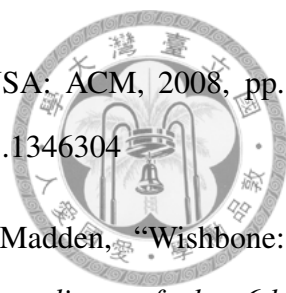




## Bibliography

- [1] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some simplified np-complete problems," in *Proceedings of the sixth annual ACM symposium on Theory of computing*, ser. STOC '74. New York, NY, USA: ACM, 1974, pp. 47–63. [Online]. Available: <http://doi.acm.org/10.1145/800119.803884>
- [2] "JavaFBP." [Online]. Available: <http://www.jpaulmorrison.com/fbp/index.shtml#JavaFBP>
- [3] S.-H. Hung, T.-W. Kuo, C.-S. Shih, and C.-H. Tu, "System-wide profiling and optimization with virtual machines," in *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, 30 2012-feb. 2 2012, pp. 395 –400.
- [4] T.-T. Tzeng, "MobileFBP – A Dynamic Migration Framework for Android Applications."
- [5] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- [6] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mob. Netw. Appl.*, vol. 18, no. 1, pp. 129–140, Feb. 2013. [Online]. Available: <http://dx.doi.org/10.1007/s11036-012-0368-0>
- [7] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey," *Future Gener. Comput. Syst.*, vol. 29, no. 1, pp. 84–106, Jan. 2013. [Online]. Available: <http://dx.doi.org/10.1016/j.future.2012.05.023>

- 
- [8] J. Flinn, D. Narayanan, and M. Satyanarayanan, “Self-tuned remote execution for pervasive computing,” in *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, ser. HOTOS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 61–. [Online]. Available: <http://dl.acm.org/citation.cfm?id=874075.876398>
- [9] R. K. Balan, M. Satyanarayanan, S. Y. Park, and T. Okoshi, “Tactics-based remote execution for mobile computing,” in *Proceedings of the 1st international conference on Mobile systems, applications and services*, ser. MobiSys '03. New York, NY, USA: ACM, 2003, pp. 273–286. [Online]. Available: <http://doi.acm.org/10.1145/1066116.1066125>
- [10] G. C. Hunt and M. L. Scott, “The coign automatic distributed partitioning system,” in *Proceedings of the third symposium on Operating systems design and implementation*, ser. OSDI '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 187–200. [Online]. Available: <http://dl.acm.org/citation.cfm?id=296806.296826>
- [11] U. Kremer, J. Hicks, and J. M. Rehg, “Compiler-directed remote task execution for power management,” in *Workshop on Compilers and Operating Systems for Low Power*, 2000.
- [12] M. Neubauer and P. Thiemann, “From sequential programs to multi-tier applications by program transformation,” in *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '05. New York, NY, USA: ACM, 2005, pp. 221–232. [Online]. Available: <http://doi.acm.org/10.1145/1040305.1040324>
- [13] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “MAUI: Making smartphones last longer with code offload,” in *Proceedings of ACM MobiSys*, 2010, pp. 49–62.
- [14] Y. Weinsberg, D. Dolev, T. Anker, M. Ben-Yehuda, and P. Wyckoff, “Tapping into the fountain of cpus: on operating system support for programmable devices,” in *Proceedings of the 13th international conference on Architectural support for programming languages*

- 
- and operating systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 179–188. [Online]. Available: <http://doi.acm.org/10.1145/1346281.1346304>
- [15] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden, “Wishbone: profile-based partitioning for sensornet applications,” in *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, ser. NSDI’09. Berkeley, CA, USA: USENIX Association, 2009, pp. 395–408. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1558977.1559004>
- [16] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang, “The case for cyber foraging,” in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, ser. EW 10. New York, NY, USA: ACM, 2002, pp. 87–92. [Online]. Available: <http://doi.acm.org/10.1145/1133373.1133390>
- [17] J. Flinn, S. Sinnamohideen, N. Tolia, and M. Satyanarayanan, “Data staging on untrusted surrogates,” in *Proceedings of the 2nd USENIX conference on File and storage technologies*, ser. FAST’03. Berkeley, CA, USA: USENIX Association, 2003, pp. 2–2. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1973355.1973357>
- [18] 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 *INFOCOM, 2012 Proceedings IEEE*, march 2012, pp. 945–953.
- [19] S.-H. Hung, C.-S. Shih, J.-P. Shieh, C.-P. Lee, and Y.-H. Huang, “Executing mobile applications on the cloud: Framework and issues,” in *Computers; Mathematics with Applications*, vol. 63, no. 2, Jan 2012, pp. 573 –587.
- [20] T.-T. Tzeng, “MobileFBP – A Dynamic Migration Framework for Android Applications.”
- [21] The Ganglia team, “Ganglia is a scalable distributed monitoring system for high-performance computing systems such as clusters and grids.” 2000. [Online]. Available: <http://ganglia.sourceforge.net/>

[22] The Iperf team, “Iperf was developed by nlanr/dast as a modern alternative for measuring maximum tcp and udp bandwidth performance.” 2010. [Online]. Available: <http://sourceforge.net/projects/iperf/>



[23] B. Ford, P. Srisuresh, and D. Kegel, “Peer-to-peer communication across network address translators,” in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 13–13. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1247360.1247373>

[24] “boofcv.” [Online]. Available: <http://http://boofcv.org>