공학석사학위논문

Design and Implementation of Platform-Independent Offloading System for a Mobile Web Environment

모바일 웹 환경에서의 플랫폼에 독립적인 오프로딩 시스템 연구

2013년 2월

서울대학교 대학원
컴퓨터공학과
박 세 훈

Design and Implementation of Platform-Independent Offloading System for a Mobile Web Environment

모바일 웹 환경에서의 플랫폼에 독립적인 오프로딩 시스템 연구 및 구현

지도교수 염 헌 영

이 논문을 공학석사학위논문으로 제출함

2013년 2월

서울대학교 대학원

컴퓨터공학과

박 세 훈

박세훈의 석사학위논문을 인준함

2012년 2월

위 원 장 민 상 렬 (인)

부 위 원 장 염 헌 영 (인)

위 원 엄 현 상 (인)

Abstract

Increasingly, smart phones are becoming one of the most popular mobile devices in personal computing environment. As the need for a variety of mobile applications is increasing, the target mobile platform is a primary concern for mobile application developers. To reduce design complexity for different platforms and enhance the compatibility of applications on various mobile OSes, a JavaScript-based web environment became a main target framework for smart phone applications. Computing-intensive and rich graphics-based applications in a smart phone may fully utilize the CPU, and consume a large amount of the battery power accordingly.

In this paper, we propose a platform-independent offloading system, which is a delegated system for a mobile web environment. Our offloading architecture is implemented in a built-in proxy system where the original web resources are selectively divides into a lightweight client code and a computationally heavy code running on the server system. Our evaluation shows that our mobile offloading system increases the response time of the application running in the web browser, and enables a high workload application to run on relatively low-end mobile devices. In addition, this method reduces power consumption of the device. Therefore, our web-based offloading architecture creates a new mobile computing environment, and can be applied various OS platforms of mobile devices.

Keyword: Mobile Device, JavaScript, Offloading, Power Saving, Platform-Independent.

Student number: 2011-20840

Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Motivation: Emergence of Mobile Devices

The emergence of mobile devices has changed many aspects of the current personal computing environment. Various mobile platforms are rapidly introduced to meet our computing needs. The increasing speed of mobile networks, including W-CDMA, LTE, and IEEE 802.11n, accelerates the performance of mobile devices, and it also minimizes the overhead of the client-server model in the network environment. Mobile applications are becoming more elaborated in order to support complex applications, such as games that incorporate augmented reality and high-level mathematical computations. However, along with the complicated and rich graphic-intensive characteristic, these applications are still restricted to the hardware resources in the mobile computing environment.

As a solution to this issue, several research projects [2, 4] propose offloading systems. However, their offloading functions rely on operating systems of mobile devices, and this may lead to non-trivial development overhead in terms of changes or upgrades of operating or runtime systems (detailed in Section 2). Moreover, there are many operating systems such as Android (Google), iOS (Apple), RIM (BlackBerry), Symbian(Nokia), and Bada (Samsung), and this gives another major concern to mobile web application developers. Thus, mobile service providers or mobile manufacturers need a platform-independent offloading systems in this reason. To enhance the compatibility of the mobile platform, many applications are integrated into a

web browser, which is a common environment among the different mobile OS platforms. As a next-generation web standard, JavaScript has been represented as a universal platform language for mobile applications that run on web browsers. The emergence of HTML5 also derives from the need of a common platform for a mobile device, which mainly uses a JavaScript-based platform. We propose a platform-independent JavaScript offloading system to minimize the limitation of hardware resources, and to maximize the compatibility of various mobile OS platforms.

Heavy computational models [7, 11] or gaming applications based on JavaScript require high CPU utilization, which shortens the battery lifetime. These applications usually require high-level hardware specifications, and increase the cost of the devices, which is challenging for low-end devices. Low-cost smart phones are becoming popular in emerging markets, especially in developing countries. These devices reduce the cost and extend the battery lifetime. However, the main problem with them is that their capacity is limited to utilizing complicated web resources, such as a rich UI or highly computational JavaScript applications. To address the issue of the limited resources of low-end devices, we propose a new architecture for application offloading, in which a mobile client out-sources JavaScript functions to an offloading server. With our offloading technique, the low-end devices are able to support complicated UI and high computational web resources.

The basic idea behind our offloading system is to exploit the separation of heavy computation functions specified by the programmer's annotation. The computational parts are moved to the server as a *callee* function, while the client receives the lightweight *caller* functions. Our offloading methodology runs on the application (browser) level; thus it is not limited to the platforms or runtime of the mobile OS. This offloading system reduces the level of CPU

utilization and energy consumption of mobile devices since the main computational works are offloaded to the server side. This client-server model may create an additional network overhead, but the total amount of network traffic received by the client is relatively reduced. It is because the heavy computational functions do not need to be uploaded to the client side. Our offloading system is feasible, as it increases the performance of mobile clients and also enlarges the compatibility of the various mobile platforms.

1.2 Contributions and Overview

The main contributions of this paper are as follows:

- We analyze the advantages of a platform-independent offloading system for a mobile web environment.
- We implemented the mobile offloading system which is a novel offloading methodology that increases the performance of mobile devices, reducing CPU utilization.
- We show that our offloading system minimizes the limitation of hardware resources in the mobile device performance.
- We evaluate the offloading system using a widely known JavaScript application from the Internet, and it increases the response time of the application and extends a battery life of the device.
- We integrate our mobile offloading system into a commercial proxy system as implementing the offloading APIs, and these offloading APIs can be applied in any mobile web framework.

The rest of the paper is organized as follows: Chapter 2 reviews the background and related work on the offloading systems for mobile

environments. Chapter 3 describes the design and implementation of our mobile offloading system. Chapter 4 presents a case study of our mobile offloading system based on a casual game application in JavaScript. An evaluation of the performance using a casual game in our mobile architecture is presented in Chapter 5, and Chapter 6 concludes our work.

Chapter 2
Background & Related Work

Offloading has been widely applied for throughput improvement in many areas such as network [10, 1, 6], database [17, 20], and mobile systems [2, 4]. TCP offloading engine (TOE) in [10, 1] offloads processing of the entire TCP/IP stack to the network controller where processing overhead of the network stack becomes significant in high-speed network interfaces. PacketShader in [6] offloads computation and memory-intensive router applications to GPUs to optimize the packet reception and transmission path, and it shows performance improvement in SSL processing. In [17, 20], FPGAs are used to accelerate query processing in database system, and the FPGA-enabled systems exhibit advantages in terms of parallel stream evaluation. Overall, there are several major distinguished research projects on the issues of computational power and the cost of CPU utilization in mobile computing environments.

2.1 CloneCloud

One is the CloneCloud [2] project, which uses nearby computers or data centers to speed up smart phone applications and reduce CPU utilization on mobile devices by cloning the entire image of a mobile device via cloud computing. This method allows mobile devices to be more power-efficient and reliable, but it requires pre-processing on the client side, which can increase the cost of mobile devices. It can also lead to excessive network traffic from the cloud network to the device. Issues pertaining to security cannot be

neglected either, as this architecture clones the complete image of a mobile system, include the user's private files, in third-party cloud storage over the network.

2.2 MAUI

Another project is MAUI [4], which provides a fine-grained code offload based on the annotation by a programmer to maximize energy savings. This code determines at run-time level which methods should be remotely executed as driven by an optimization engine under current connectivity constraints of the mobile device. However, experimental results have shown that this approach works only on Microsoft Windows platform. Moreover, the level of the code offloading decision is based on .NET framework. Therefore, this method is non-scalable and is not commonly adoptable on various mobile computing OS platforms. However, our approach is reliable, concise and platform-independent, as it operates on JavaScript platform, which runs at the browser level.

2.3 Game Cloud

Recently, LG U-plus, one of Korea's largest telecom companies has launched a cloud platform [12] for games where users can access games that can be played on smart phones and personal computers on the cloud. The platform will store games provided by developers and publishers for customers who do not need to download them, but simply log-in and play via its streaming services. This approach still requires a massive amount of network storages, and it is somewhat limited to widely adapting on all mobile applications

compared to our system. Figure 1 summarizes the various offloading architectures, and emphasizes our offloading approach to be very effective in terms of platform compatibility and design simplicity.

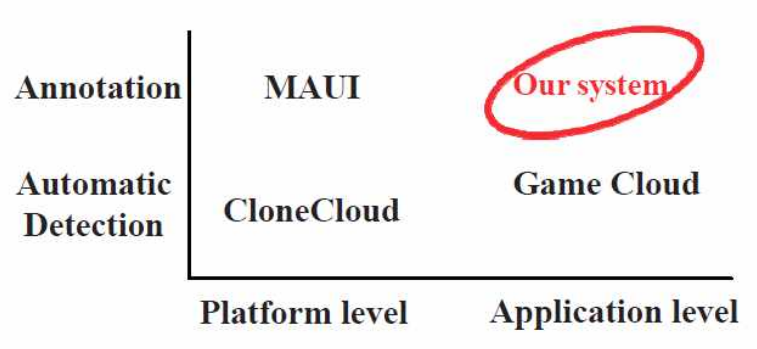

Figure 1. Our Mobile Offloading Approach

2.4 Saving Power on Mobile Device

Moreover, power consumption is a primary concern in mobile and distributed computing communities. Prior work [18, 5] explores saving power via remote execution. Recently, cloud computing is widely known as the potential to save mobile device energy [15], but it causes additional network traffic. This paper provides an analysis of the critical factors affecting the energy consumption of mobile client by the offloading method. The energy trade-offs are mostly triggered by the characteristics of the function of workload, data communication patterns and technologies used. Because our offloading system targets the application on web browsers, the additional node

overhead relatively small compared to the cloud-based architecture. We calculate the additional packets received and sent in our system, and the offloading method almost does not exceed the total amount of the resource received by the client compares to the non-offloading method.

Chapter 3
Design and Implementation

In order to offload the computational function we develop the platform-independent mobile offloading system, a new client-server methodology for mobile environment in which core workloads of the client are executed remotely on the server side. We also apply our offloading system in order to control the offloading function via programmer's annotation to simplify the implementation and minimize the overhead of parsing the web resources.

3.1 Mobile Offloading Architecture

In this section, we describe the offloading system as a mobile computing framework in order to increase the performance and to reduce the CPU utilization of the mobile client. As described in Figure 2, the contents adaptor selectively detects heavy computational function codes from the outbound web servers by programmer's annotation. Then, it generates a server-side process that is not uploaded to the client. Therefore, a client will only receive a stub, which is responsible for carrying out the call method for the offloading object, and its skeleton function will run on server-side.

Our platform-independent offloading system requires the following constraints. i) The architecture requires a built-in proxy, called Contents Adaptor (CA), which basically acts as an intermediary for requests from mobile clients seeking web resources from the outbound servers. A mobile client only connects to the proxy server, which requests web services from outbound servers. This built-in CA facilitates access to the contents on the

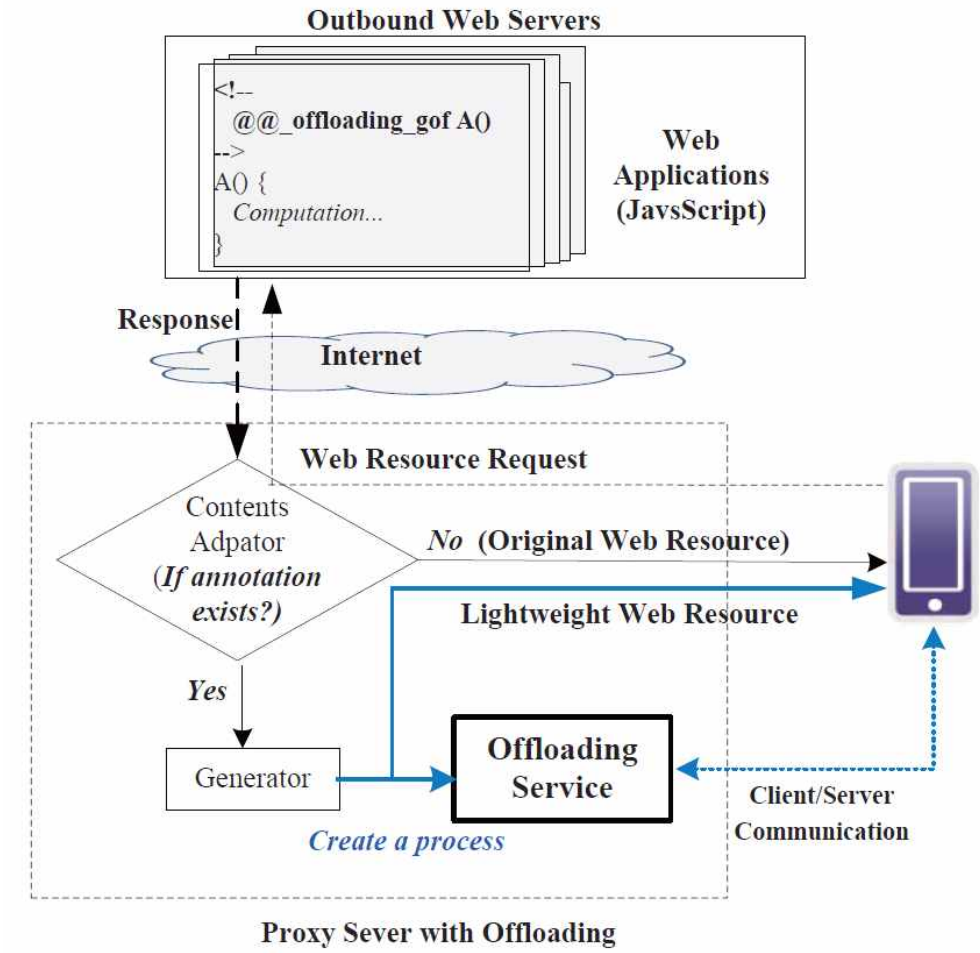

Figure 2. Our Mobile Offloading Architecture

World Wide Web. ii) The codes for offloading should be defined by the programmer with the annotations in the web application code, which runs on JavaScript functions. The CA analyzes the web resources from the outbound server and determines whether or not the web page needs offloading. If the CA does not find any annotation for the offloading on the code, the web page is transferred to the client with regular proxy server functionalities, including compression and caching. In contrast, if it detects the offloading annotations in

the header of a web page, the CA notifies the generator, which creates a skeleton process for the mobile client on the server-side. Concurrently, the CA sends a lightweight modified web page to the client. Accordingly, the performance of the mobile client can be increased, as the client no longer needs to run the heavy computation code.

3.2 Programmer's Annotation

Our mobile offloading process is based on the programmer's annotation, which is located in the header of a web page. To design the prototype of the offloading annotation, we categorize the following three types of offloading entities.

- **General Offloading Function (GOF)**: The functions that are needed for offload generally, mainly computational functions.
- **Nested Offloading Functions (NOF)**: NOF does not need to be directly specified as an offloading function by an annotation, but these functions selectively require to be offloaded along with GOF. The nested functions situated on GOFs should be regarded as an offloading function even though they are not declared on the annotation. However, in this work we also declare NOF in the offloading annotation.
- **Global Offloading Variables (GOV)**: The global variables that need to be shared between the client and the offloading server.

Here, our prototypes of the annotation can be described simply with the following example. Figure 3 shows an example of the annotation.

To minimize latency in parsing a web page, the annotations are located in

the header of the web page. The annotation enclosed by the comment is ignorable to JavaScript, and only recognized by the generator in the proxy. Once the generator detects the annotations in the header for an offloading purpose, the parser starts to search the offloading function in the body of the page. Otherwise, the proxy server passes the webpage to the client directly with only regular proxy functionalities. Detailed offloading implementation is described on Section 4.

```
<!--
    // GOF offloading
        @@_offloading_gof func1_name();
    // NOF offloading
        @@_offloading_nof func2_name();
    // GOV offloading
        @@_offloading_gov var_name;
-->
```

Figure 3. Prototype of Annotation

Inserting these annotations on the top of the web page may generate an additional programmer's effort. However, this way is straightforward and adding an annotation by the author is the most efficient way to select the offloading function. In addition, most of contents' providers (CP) or web service providers are offering the two different layouts which are one for mobile view and the other for the full browser mode. Therefore, this indicates there is already effort to provide the mobile browser mode, so our annotation method will not affect a considerable overwork.

3.3 Code Division in the Client-Server Model

Once the CA receives client's web request, it starts to communicate with the outbound web server to retrieve the web application code. If the CA detects offloading functions in the web application code via the user-specified annotation in the HTML header, it creates a server process that delegates the client work to the remote server. Since each JavaScript program has its own programming pattern, it is difficult to customize for every program. To address this issue, we use the very well-known remote procedure call (RPC) to separate the original JavaScript program and re-generate codes for offloading.

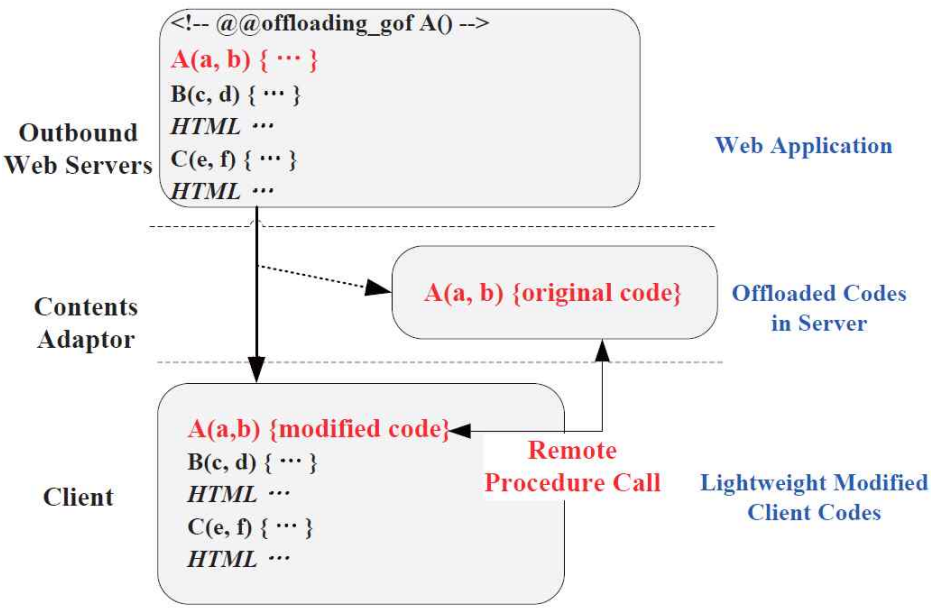

Figure 4. Code Division by Our Mobile Offloading System

Figure 4 describes code manipulation in our system. The CA moves functions for offloading to the server side, and it sends modified web application code to the mobile client. The modified web application code in the client side has the stub code that helps communication for invoking offloaded functions. The server side also has the skeleton code for processing client's requests. We implement remote procedure call mechanism of our system using Node.js [14] that is a platform built on JavaScript runtime. It provides an efficient method to share data on the client-server model. The premised functions support by Node.js is following:

- **node.sendMessage()**: method to call a remote function which is used to do the heavy computation in the original code.
- **node.receiveMessage()**: method to deliver the computing result back to the client.

In addition, since Node.js uses an asynchronous communication, it can be programmed only in an event driven style. Thus, we need a method for the synchronous remote procedure call. To address this problem, we use the Concurrent.Thread [13] library that provides a multi-thread environment in JavaScript1. With the *join* function in this library, we implement the synchronous remote procedure call that does not change program semantics of the original web application code. More information with an example is available in Section 4.

In comparison with the client-based version, the offloading architecture may generate additional data traffic, but it varies by the type of application workload. Our offloading system focuses mostly on the computational functions, which does not require a sizable amount of data exchange. In this

case, the client application only sends a parameter (Integer or Byte) and receives the results. Hence, the network traffic overhead is relatively small in practical situations.

3.4 Fault Tolerance

Since our offloading system bases its architecture on the client-server model, we have to properly manage the situation when the network between mobile devices and the offloading server is unavailable. If a communication channel between the offloading server and the mobile device is not reliable or overloaded, the offloading server process may not respond to the client request in a timely manner; and duplicated requests may arrive. To deal with this problem, the offloading system detects duplicated client requests by checking a unique sequence number for each request. If a proxy server detects a duplicated page request within a certain time period, it treats it as a failure. The contents adaptor converts the pages to the non-offloading model, since it thinks that the offloading system may encounter a critical problem. In this circumstance, a mobile client does not need to change or modify any functionality in our offloading architecture. However, this architecture may require the application reloading if a network failure happens while the application is running.

There is another way to increase the system reliability. The contents adaptor duplicates the actual body of the offloading *callee* function to the client as well. After the client waits for a certain timeout value, the caller function refers to the duplicated body of *callee* function on the client, not from the offloading server.

Chapter 4
Case Study

In this section, we select a web application written in JavaScript which includes heavy computational workloads from public web sites. General web pages do not have the annotation, so we spend much time analyzing the code first, and then figuring out which function should be selected for offloading. For this reason, the offloading method based on the annotation by a programmer has a great chance of utilizing codes in our offloading system.

4.1 Adapting into Web Application

The target program is *Gomoku* [21], a turn-based five-in-a-row board game. The winner is the first player to get an unbroken row of five stones horizontally, vertically, or diagonally. This is one of the popular games written in JavaScript, and we analyze it in order to insert the annotations for the most computational functions. The main computational part used in this program is to evaluate all empty positions from the point at which *Gomoku* pieces have been placed. After a placement is put into a position, the function calculates the highest score to find the next position. As the number of the pieces upon the board grows, the requirement for position computing is increased. In some cases, it even causes the browser to crash, especially on low-end devices because the waiting time becomes intolerably long. To improve a performance of the game with our architecture, we insert our annotation on the head of web pages to divide the original game into a client-server version. Functions on the client side are largely used to generate UI resources, while offloaded

functions to the server side handle the position computing workload, which is the main bottleneck of the program.

```
<!--
    // GOF offloading
        @@_offloading_gof drop();
    // NOF offloading
        @@_offloading_nof judge();
        @@_offloading_nof judgeProcess();
...
    // GOV offloading
        @@_offloading_gov slope_list;
        @@_offloading_gov backslope_list;
...
-->
```

Figure 5. Annotation in *Gomoku*

Figure 5 shows annotation used in this example. The most computationally heavy function, *drop()*, is offloaded to the server. *judge()* and *judgeProcess()* as helper functions for the *drop()* function are also moved to the server. The *drop()* function calculates the next position using *judge()* and *judgeProcess()*. The slope list and backslope list variables need to be shared between the client and the offloading server in this example.

4.2 Mobile Offloading Function Flow

In our system, the massive computing parts of JavaScript codes are migrated to server-side. Thus the client is merely responsible for generating UI and

dealing with user interaction. Figure 6 describes the original function flow for the piece placement. Our system offloads the *drop()* function to the server as shown in Figure 5, and this work does not break the program semantic. The client sends the position of *Go piece* [21] as the input data by the ***node.sendMessage()*** function of the NodeJS framework. The server receives the input data and calls the *drop()* function to calculate the position of next placement. Then, it transmits the result to the client, and the client executes the *addPiece()* function that is responsible for generating the position on the UI.

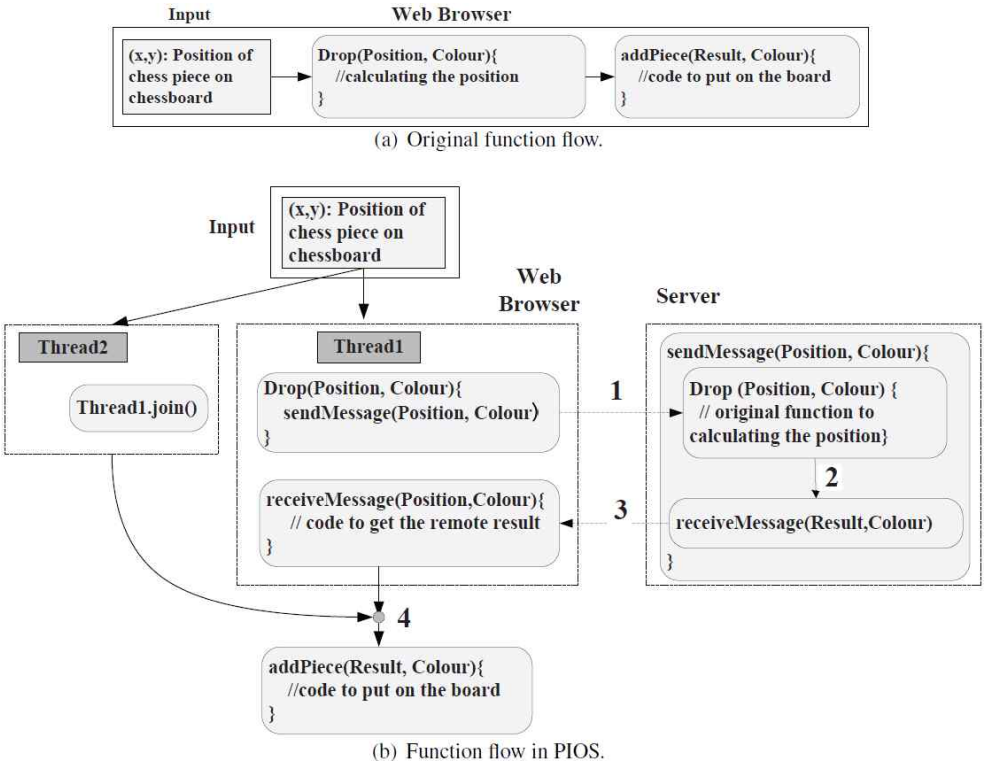

Figure 6. Program semantics of piece placement in *Gomoku.*

As stated in the previous section, the communication in NodeJS is asynchronous. In addition, the web browsers perform all JavaScript processing on a single thread. These features of JavaScript will cause the function to run in asynchronous mode despite of the program sequence. In this example, *addPiece()* may receive an unexpected result because interference may be created by other functions in the code. To solve this problem, we use the *Concurrent.Thread* [13] library. The first thread communicates with the server asynchronously and the second one is used to wait for the first thread until the result comes back. After the join function, the UI is generated according to the result from the first thread.

Chapter 5
Performance Evaluation

To prove the performance gain and effectiveness of our approach, we design several experiment scenarios. The test bed consists of four clients with different specifications and one offloading server.

5.1 Experiment Setup

The mobile client devices connect to the server via HSDPA [8], 802.11b/g [9], and LTE (4G) network. Table 1 and Table 2 describe the detailed hardware specifications and network protocols of the experimental devices. Our experiment compares both cases, executing an original client-based five-in-arow game, *Gomoku* and the client-server offloading version generated by our offloading system. The *Go board* and the *Go pieces* were drawn by the “canvas” element [3] in HTML5, and the game board contained 30x17 positions on which a Go piece can be placed. Client devices communicate with the offloading server using different network interfaces. Because this framework is based on remote computing, the network bandwidth can affect the performance. In all scenarios, the web browser serves the same UI and functions as the user interface regardless of whether the program is offloaded or not. We concentrate on performance and resource consumption in the experiments.

Devices	Configuration	
	Hardware	Software
Offloading Server	Intel Core2 Quad 2.83GHz 8G Memory	Ubuntu 11.10
Apple iPhone3gs	Cortex-A8 600MHz 256M Memory	iOS 5.0.1 Safari for iOS
Apple iPad2	Apple A5 1GHz 1G Memory	iOS 5.0.1 Safari for iOS
Samsung Galaxy Note	Exynos 1.43GHz 1G Memory	Android 2.3.4 Android Browser
Samsung Galaxy Nexus	Ti OMAP 1.2GHz 1G Memeory	Android 4.0.4 Android Browser

Table 1. Device Configuration.

Protocol		Speed
Wi-Fi	802.11G	50Mbps
3G	HSDPA	10Mbps
4G	LTE	50Mbps/up, 100Mbps/down

Table 2. Network Connection.

Experiment	Description
Response Time	Time used to calculate the next position + Time used to generate the UI
CPU Utilization	Percentage of CPU usage during the time to calculate next position
Energy Consumption	Total device power consumption during the application running

Table 3. Performance Metric

5.2 Response Time

In a turn-based game program, the response time (shown in Table 3) is usually a key factor of performance evaluation. Basically, if one user generates his or her user event, we hope that respond time will be short enough to provide a good subjective performance.

5.2.1 Response Time in High-End Device

We first investigate how the offloading system can reduce a response time of application. Figure 7(a) demonstrates the response time of each level of placement (the first ten placements) in the original client-based version, and with our offloading method. This experiment runs in a Wi-Fi environment, and the test device is Samsung Galaxy Nexus. The maximum response time in the client-based version is 28.5 seconds and the average is 13.1 seconds. In contrast, the offloading version shows corresponding values of 8.3 seconds and 3.9 seconds respectively. Therefore, our architecture achieves a speedup of 3.3X on the web application, and the gain becomes even larger as the level of placement gets higher.

5.2.2 Response Time in Low-End Device

Figure 7(b) shows the same kinds of information, but measured by iPhone3GS that is a relatively low-end device compared to Galaxy Nexus. The result shows a similar pattern with Figure 7(a), but iPhone3gs is not able to run the game by more than seventh setup because the response time is too long to be played in the non-offloading method. However, it plays in all

placements with reasonable performance with our offloading method, and the average response time is 5.3 seconds up to tenth placement. Hence, our system reduces the response time, and also allows the game to be played on such low-end devices, which typically is almost impossible due to the limitation of hardware performance of the mobile device.

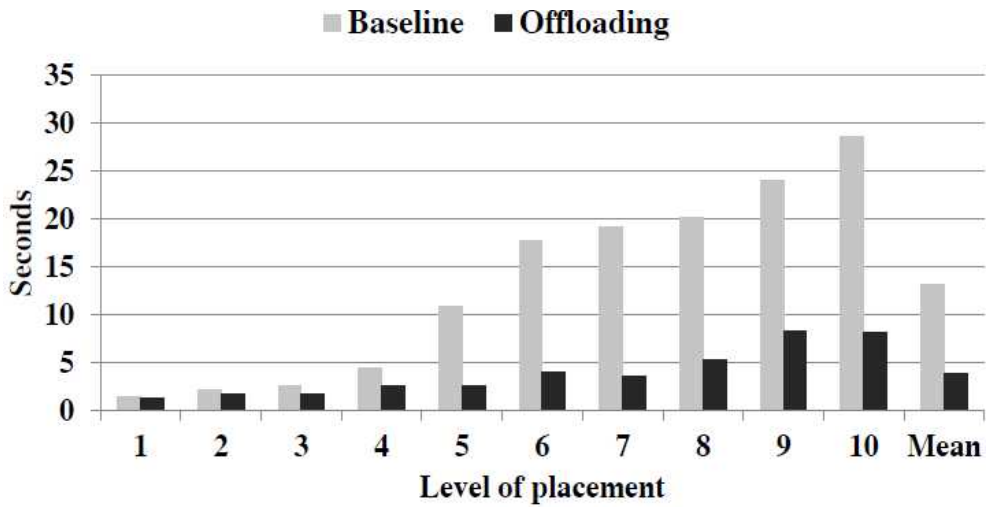

Figure 7(a). Response Time in High-End Device (Galaxy Nexus)

Figure 8 shows the gap of the response time normalized to the value of the first placement on two different devices. We use two different experimental devices, iPad2 and Galaxy Nexus using Wi-Fi connection. In the baseline, the difference in the response time for the two devices increases as the program goes to a higher level. However, the difference is not significant by the offloading system. This demonstrates that the performance without offloading is affected by the hardware specifications of the devices, as all the computing procedures are executed on the client side. However, it also shows that the

differences in specifications have almost no influence on the performance by the offloading method. Hence, devices with low specifications are capable of reaching performance levels nearly identical to those of high-specification devices with our approach.

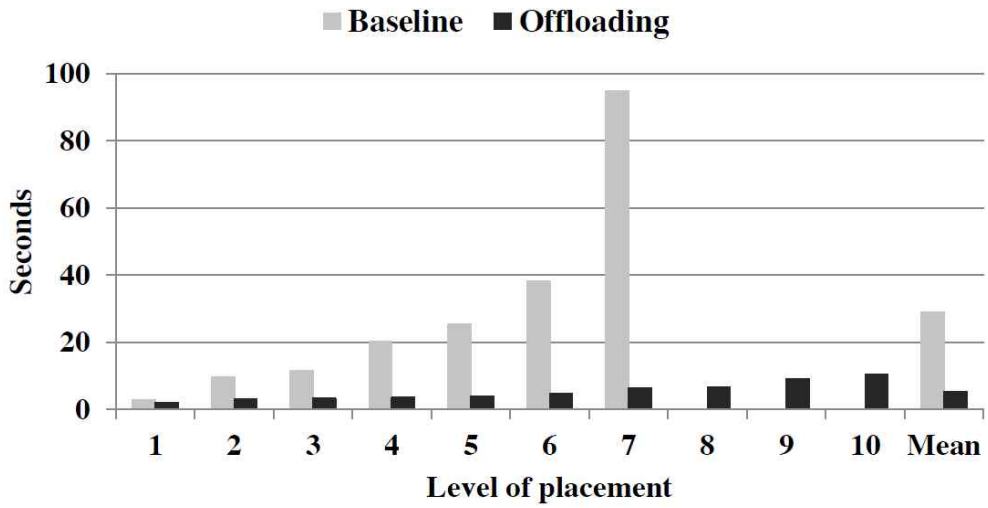

Figure 7(b). Response Time in Low-End Device (iPhone3gs)

5.2.3 Response Time in Different Network Connections

In Figure 9, we compare the response times on different network connections with an offloading version running on Galaxy Note (LTE supported). Since this offloading program is generally based on remote server computing, the network latency should be considered when measuring the response times at each placement. The LTE network gives the best performance with an average response time of 4.3 seconds among all the experiments. The median response time for Wi-Fi and WCDMA connections is 5.1 and 5.9 seconds, respectively. This result shows that low network latency

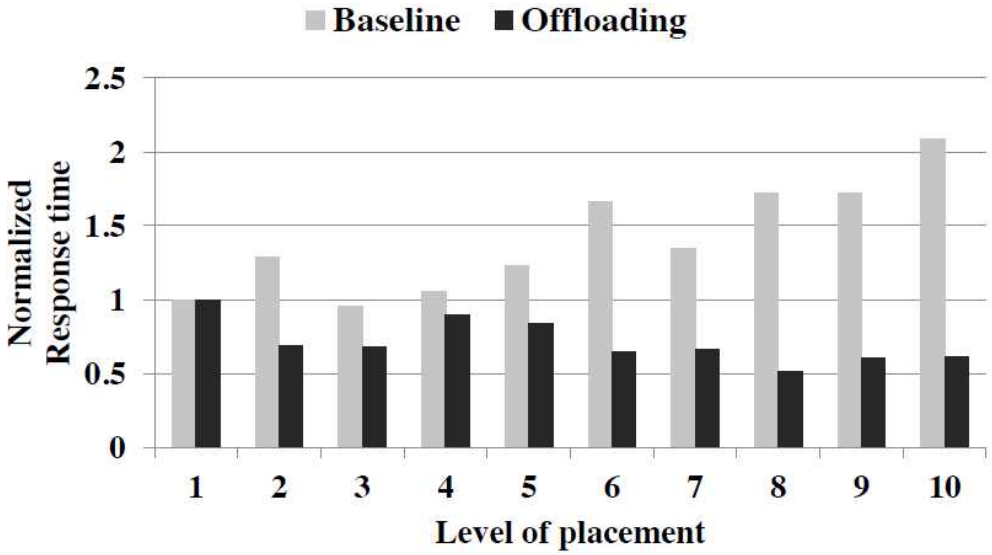

Figure 8. Difference between Response Times Normalized to the First Level

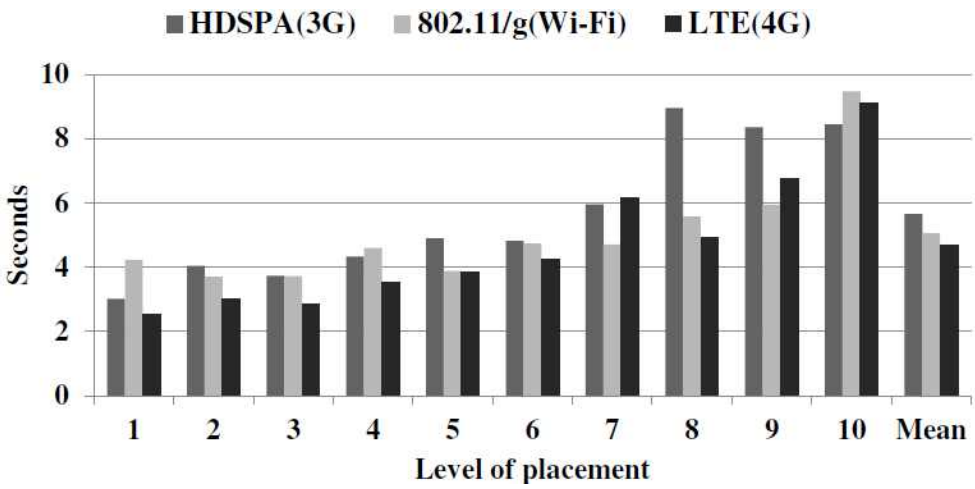

Figure 9. Response Times on Different Network Connections

5.3 Network Usage Overhead

To measure an amount of network traffic caused by the offloading system and CPU utilization during the cpu intensive application running are also a main factor of performance evaluation. Since our offloading system usually generates additional data exchanges in the offloading system, we also collect the total received and sent packets in each experiment to analyze the overhead in the offloading method. Figure 10 measures Tx/Rx packets in bytes by the APIs, TrafficStats(), getUidTxBytes() & TrafficStats.getUidRxBytes() from Android framework in proc file system of the device kernel. While no data traffic is occurred in the client only mode, our mobile offloading system requires additional data traffic only of around 4Kbytes in both Tx and Rx. Since the client usually sends a several parameters for a computation and receives a result only, the additional traffic overhead is relatively small, and this overhead does not affect the device performance.

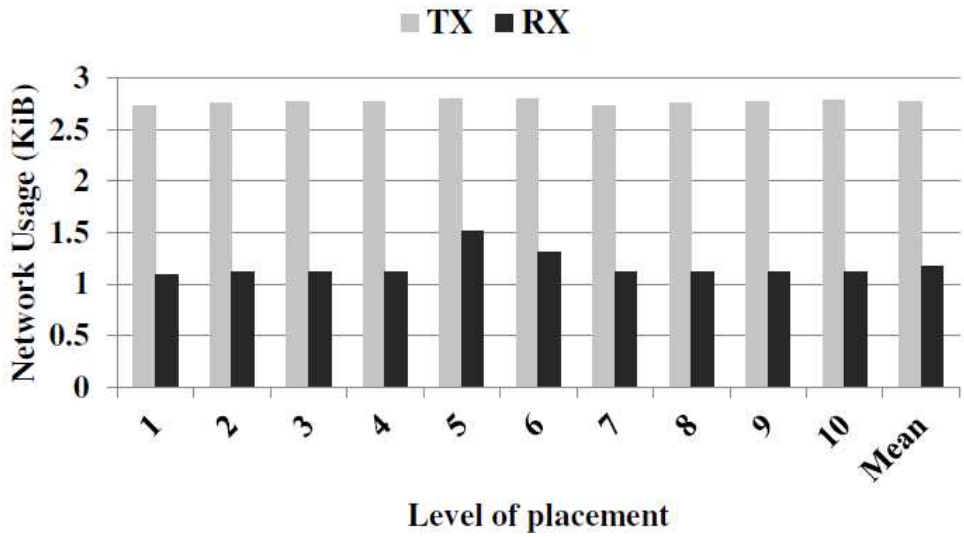

Figure 10. Total Amount of Data Exchange

5.4 CPU Utilization

We compare the peak CPU utilization of the client-based version with the offloading one. Galaxy Nexus with Wi-Fi was selected as the experimental device. We use the Quick System tool [19] to detect the CPU utilization frequency throughout each placement level. Figure 11 describes the CPU utilization for each level of placement on the Y-axis. The CPU utilization is greatly increased as the placement level gets higher, and it exceeds 50% after the fifth level in the client-based version. The peak CPU utilization reaches 55%, which is almost the maximum resource that mobile OS can allocate to the application. By contrast, with the offloading version, the largest CPU utilization value is only 9%, and the average is 6% until the tenth placement level. The result shows that this experimental program is resource-intensive on the client, and more importantly, that our approach is well suitable for those JavaScript programs that require heavy computational work.

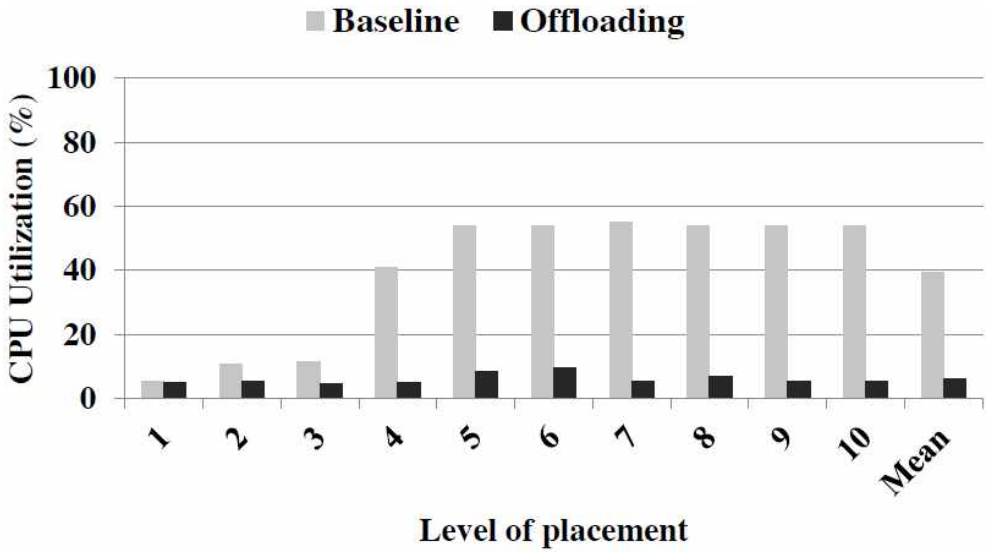

Figure 11. CPU Utilization in *Gomoku*

5.5 Stress Test

In the multi tasking mobile environment, we run *Gomoku* application with another CPU-intensive application simultaneously. We implement a program that use a loop calculation generating more than 50% cpu utilization, and check how the *Gomoku* game can be affected by the CPU-intensive application. After third level placement the game is unresponsive and returns an error message shown in Figure 12 in the client-based version. It is because the operating system protects the device to not exceed the certain level of CPU-utilization. On the other way, using our mobile offloading method, the gaming application still works as normal. Our mobile offloading system minimizes the application overhead so that our method is compatible in any circumstance.

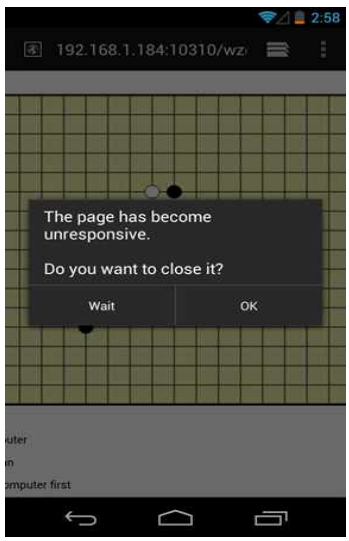

Figure 12. Unresponsive application in the stress test

5.6 Power Consumption

In mobile device environment, battery consumption is a main concern. We analyze the total battery consumption with the different methods (non-offloading & offloading) in order to prove that our offloading system reduces the battery consumption by the device over the client-based method. At First, we utilize the hidden API which is limited to a user level in Android platform called BatteryStats [22]. This API allows generating the energy consumption of the related components on the device with the offloading system. With Galaxy Nexus, open source reference Google phone, we are able to modify the source codes of Setting Menu as displaying the battery consumption of each component in mAs (Milli-ampere-second) using BatteryStats API. The analysis of the power consumption of application A can be derived as the following equations in this experiment.

- ***Total Power consumption of Application A =***

  (CPU + Wakelock + Traffic + Wi-Fi/3G + sensors) power consumption.

where

CPU Power consumption =

$\sum$(CPU active time * Average current for CPU)

(Average current is a constant value provided by a manufacture of hardware)

Wakelock power consumption =

wakelock on time * average current for wakelock on time

Traffic power consumption =

(tcpBytesReceived + tcpBytesSent) *average cost per byte

Average cost per byte =

Total cost / Total traffic =

(3g cost per byte * totalTrafficBy3g + wifi cost per byte * totalTrafficByWIFI)

/ (totalTrafficBy3g + totalTrafficByWIFI)

3G cost per byte =

3G active state average power consumption per second

/ 3g average transferring bytes per second

Wi-Fi cost per byte =

Wi-Fi active state average power consumption per second

/ Wi-Fi average transferring bytes per second

Wi-Fi on state power consumption =

Wi-Fi running time * average current for wifi on state

Sensors power consumption =

$\sum$(sensorActiveTime * sensor average current when sensor is active) for all sensors

Figure 13 shows the total power consumption in each step as the game running on 3G modes with the non-offloading and offloading method. The offloading method reduces the power consumption compared to the

non-offloading way, and the gap increases as the program goes the higher step level. In the first and second step, the level of battery consumption between non-offloading and offloading method is similar since the offloading requires an additional 3G network power, which does not have on the non-offloading one. Comprehensively, our offloading architecture provides the power saving and reduces energy consumption by an average of three times against the non-offloading method.

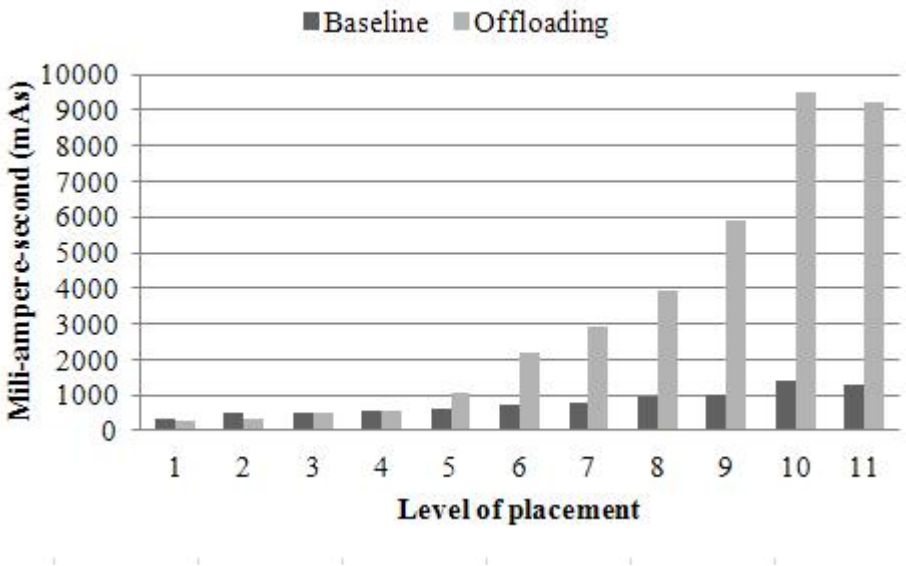

Figure 13. Comparison of the Total battery consumption in each method

Secondly, we use a Monsoon power monitor FTA22D [16] for power measurement. The power monitor supplies the voltage of a lithium battery set to 3.69 V to Galaxy Nexus device, and provides a robust power measurement solution. We run the five-in-a-row game by the client-based version and the offloading method in a Wi-Fi connection, and compare the energy consumption

of the device in each method respectively. Figure 14 (a) and (b) describe the average current (mA) of client-based version and the offloading method respectively. Each graph represents total eleven peaks, which start when the placement of the game launches, and stop when the computation is completed. In Figure 14 (a), once each placement has been set, the current is increased to the average of 600 mA (maximum 800 mA) and drops back to the standby state (around 230 mA). During 4 minutes with the entire eleven placements, the median current of the client-based method is 460.62mA. Our offloading system, on the other hand, shows the current energy efficiency. As shown in Figure 14 (b), eleven placements are completed within 1.8 minutes, and then the device stays for 2.2 minutes in standby mode, which has the average current of 230 mA. Since each calculation time of the offloading method is relatively short, and also the maximum peak current is not more than 600 mA, our offloading system explicitly reduces the total device energy. With the offloading system, the application spends in the average current of 254.67 mA, which is 55% lower than the client-based version.

Table 4 summarizes and calculates the total consumed energy as the game runs for 4 minutes on Wi-Fi mode with the client-based version and our offloading system. The mobile offloading system reduces the average current by 45%, and also saves the average power by 45% compared to the client-based version. We calculate the total consumed energy (mAh) in 4 minutes gaming, the client-based version requires more power consumption compared to our offloading system by an average of 45%. However, the game of total eleven placements by the offloading method is completed much earlier than the client-based one because the response time is almost 3X higher. In this case, the offloading method reduces the total consumed energy by 74% when we measure based on the completion time of eleven placements (1.8

minutes). Therefore, our mobile offloading system reduces not only the CPU utilization of the device, but it is also effective for the energy saving of a mobile device.

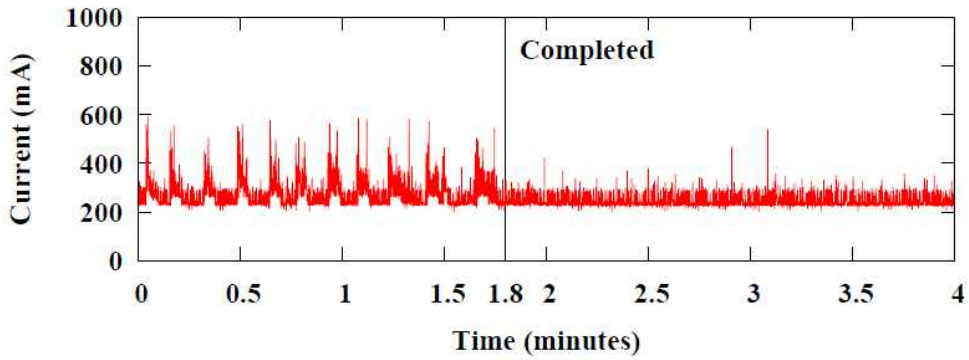

Figure 14(b). Average Current by the offloading method

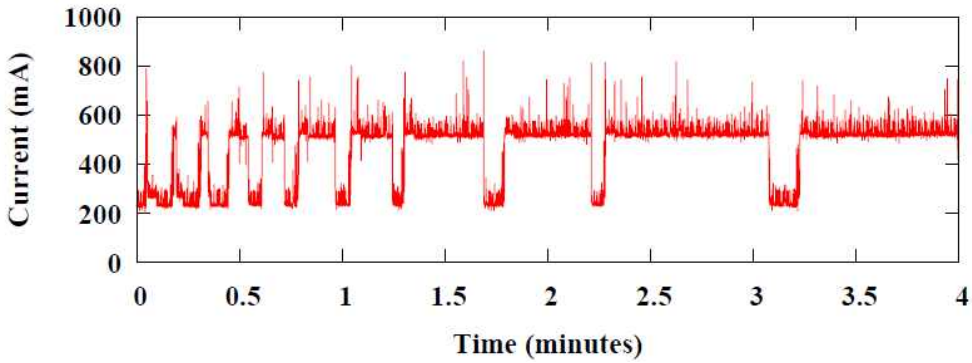

Figure 14(a). Average Current by the Client-based version

	Client-based	Offloading (4 mins)	Offloading (1.8 mins)
Ave. Current(mA)	460.62	254.67	270.57
Ave. Power(mW)	1,702.32	941.35	1000.03
Total Energy(mAh)	31.02	17.10	8.12

Table 4. Average Current and Total Consumed Energy

Chapter 6

Integration into Proxy system with offloading APIs

We also implement the offloading APIs written in Java, so that our mobile offloading system can be integrated into the commercial proxy server system. Our mobile offloading APIs mainly consist of two parts; the annotation detecting class and the offloading service generating class. The flow of the mobile offloading APIs in the proxy server is described in Figure 15. The first interface named *IsAnnotation{}* provides features for checking whether the page needs the offloading given by the requested URL as a parameter, then decides whether to create an offloading service or just to bypass the web page to the proxy server without any modification if it does not detect any annotation for an offloading. Once *IsAnnotation{}* class detects offloading functions, the other class named *JsParser{}* extracts the body of the offloading functions and creates the client page which the offloading function body is replaced to the stub function code and then generates the offloading process on the server. After that, the modified client page is returned to the proxy server which actually sends the mobile device. Finally, the mobile device directly communicates to the offloading server via Node.js service.

Table 5 describes the offloading API classes with each their methods. We integrates the mobile offloading system into the commercial proxy server using the listed APIs. The proxy server is based on Netty [23] which is an asynchronous event-driven network application framework, and has ease of development and flexibility with Java-based APIs. Also, these offloading APIs can be applied in any kind of mobile web framework.

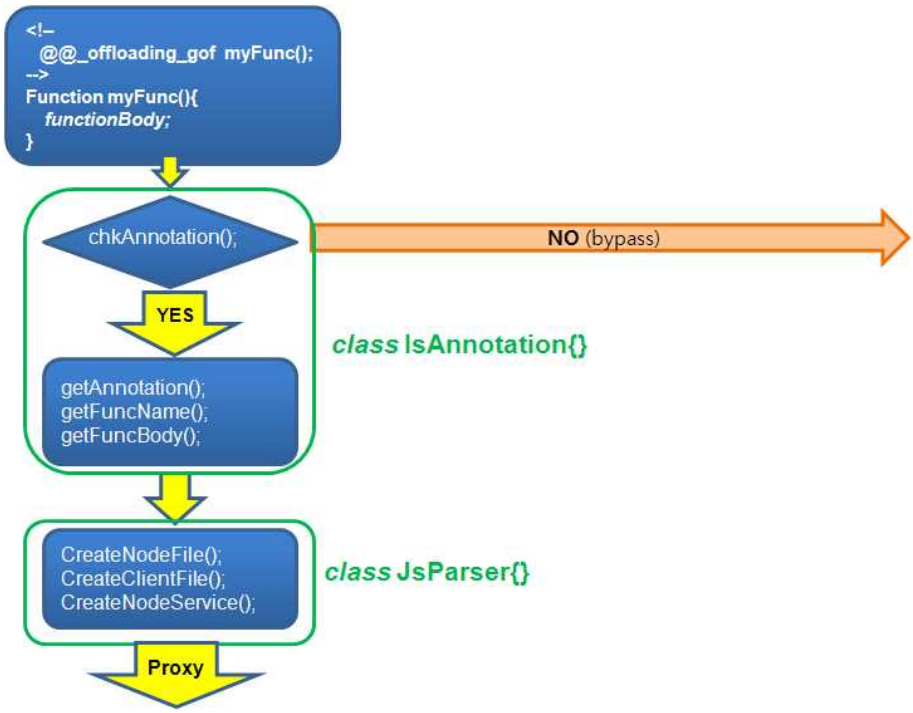

Figure 15. The Offloading Process in Proxy server

Class	Methods	Description
IsAnnotation{}	int chkAnnotation()	checks whether the given URL contains an offloading annotation (returns 0 if no offloading)
	String getAnnotation()	Returns the detected annotation
	String getFuncName()	Returns the offloading function name
	String getFuncBody()	Returns the function body
JsParser{}	String parseToNode()	Parsing the offloading function
	int createNodeFile()	Create Node.js file
	int createNodeService()	Run the offloading service
	int createClientFile()	Create a file for the client

Table 5. Offloading API classes and methods

Chapter 7
Conclusion

Our offloading architecture provides a highly efficient way of increasing the performance of mobile devices. The built-in proxy system selectively detects heavy computational function codes from the outbound web servers by programmer's annotation. Then, it generates a server-side process that is not uploaded to the client. Therefore, a client will only receive a stub, which is responsible for carrying out the call method for the offloading object, and its skeleton function runs on server-side.

Our mobile offloading system reduces the response time of JavaScript based web applications, and minimizes CPU utilization of mobile devices. This approach is even more useful for low-cost devices. It eliminates the limitation generated by the hardware performance of low-end devices since the heavy-loaded computational applications on the client side will be offloaded to the server side. In addition, our architecture proves that the offloading system is very effective regarding energy saving as proven in the evaluation.

Our experimental result shows a considerable performance gain. Our framework reduced the response time from 49.8 to 5.3 seconds on average in the turn-based gaming application. It provided a nearly constant performance irrespective of the network bandwidth environment. We also show that the CPU utilization by mobile devices when running an intensive application can be reduced greatly by our offloading architecture. By allowing CPU utilization to be significantly lowered, it reduces energy consumption by 45-74% compared to the non-offloading baseline. We also apply this architecture on the web-based scientific calculator, and still seek web applications to maximize

the efficiency of our offloading system. Overall, our approach provides a new methodology to increase a performance in mobile environment.

참고 문헌

[1] 10 Gigabit Ethernet Alliance, 2002. TCP/IP offload Engine (TOE).

[2] Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., Patti, A., 2011. CloneCloud: elastic execution between mobile device and cloud. In: Proceedings of the sixth conference on Computer systems (EuroSys ’11).

[3] Consortium, W. W. W., 2010. HTML5: A vocabulary and associated APIs for HTML and XHTML.

[4] Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R., Bahl, P., 2010. MAUI: making smartphones last longer with code offload. In: Proceedings of the 8th international conference on Mobile systems, applications, and services (Mobisys ’10).

[5] Flinn, J., Satyanarayanan, M., 1999. Energy-aware adaptation for mobile applications. In: Proceedings of the seventeenth ACM symposium on Operating systems principles (SOSP 99).

[6] Han, S., Jang, K., Park, K., Moon, S., 2010. PacketShader: a GPU-accelerated software router. In: Proceedings of the ACM SIGCOMM 2010 conference (SIGCOMM 10).

[7] Hill, A., MacIntyre, B., Gandy, M., Davidson, B., Rouzati, H., 2010. Kharma: An open kml/html architecture for mobile augmented reality applications. In: Proceedings of the 9th IEEE International Symposium on Mixed and Augmented Reality (ISMAR).

[8] Holma, H., Toskala, A., 2006. HSDPA/HSUPA for UMTS: High Speed Radio Access for Mobile Communications. WILEY.

[9] IEEE, 2003. IEEE 802.11g-2003: Further Higher Data Rate Extension in the 2.4 GHz Band.

[10] Kim, H.-y., Rixner, S., 2006. Connection handoff policies for tcp offload network interfaces. In: Proceedings of the 7th symposium on Operating systems design and implementation (OSDI 06).

[11] Lee, D.-Y., Saha, R., Yusufi, F. N. K., Park, W., Karimi, I. A., 2009. Web-based applications for building, managing and analysing kinetic models of biological systems. Briefings in Bioinformatics 10 (1).

[12] LG U+, 2012. Cgames (Cloud Game). http://www.uplus.co.kr.

[13] Maki, D., 2012. Concurrent.Thread. http://sourceforge.net/apps/mediawiki/jsthread.

[14] Maki, D., Iwasaki, H., 2007. JavaScript Multithread Framework for Asynchronous Processing. IPSJ Transactions on Programming 48 (12).

[15] Miettinen, A. P., Nurminen, J. K., 2010. Energy efficiency of mobile clients in cloud computing. In: Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '10).

[16] Monsoon Solutions, Inc, 2012. Monsoon power monitor. http://www.msoon.com/LabEquipment/PowerMonitor.

[17] Mueller, R., Teubner, J., Alonso, G., 2009. Data processing on FPGAs. Proc. VLDB Endow. 2 (1).

[18] Rudenko, A., Reiher, P., Popek, G. J., Kuenning, G. H., 1998. Saving portable computer battery power through remote process execution. SIGMOBILE Mob. Comput. Commun. Rev. 2 (1).

[19] Shawn Q., 2012. Quick System Info. https://play.google.com/store/apps/details?id=org.uguess.android.sysinfo.

[20] Sukhwani, B., Min, H., Thoennes, M., Dube, P., Iyer, B., Brezzo, B., Dillenberger, D., Asaad, S., 2012. Database analytics acceleration using FPGAs. In: Proceedings of the 21st international conference on Parallel architectures and compilation techniques (PACT 12).

[21] Wikipedia, 2012. Gomoku. http://en.wikipedia.org/wiki/gomoku.

[22] Batterystats API, Refer to Device/frameworks/base/core/java/android/os/BatteryStats.java

[23] Netty Project, 2012. https://netty.io/.

초 록

모바일 디바이스의 사용이 폭발적으로 증가하고 있고, 특히 스마트폰을 위한 많은 어플리케이션이 개발되면서 점차 기존 PC 의 환경을 대체하고 있다. 하지만, 여전히 스마트폰이 가지는 상대적으로 낮은 하드웨어 성능과 제한적인 배터리 용량은 기존의 모든 PC 환경을 대체하기에는 한계가 있다. 특히, High-Computational 어플리케이션이나 인코딩을 위한 이미지 프로세싱을 위한 워크로드에서는 모바일 단말의 많은 CPU 자원과 전력을 소비하게 된다. 따라서 본 논문에서는 이러한 워크로드를 서버에 위임하여 처리하고 그 결과만 받아서 표현하는 모바일 오프로딩 기법을 제안한다.

다양한 모바일 운영체제와 어플리케이션들을 모두 통합 할 수 있는 플랫폼으로 본 논문에서는 웹 기반의 자바스크립트 오프로딩 기법을 제시하였다. 오프로딩 시스템은 해당 웹 리소스에서 오프로딩이 필요한 함수를 파싱하여 서버에 함수 서비스를 할 수 있는 프로세스를 생성하고, 단말은 callee 함수만 가지는 light-weight 페이지만 가지게 된다. 그 결과 어플리케이션의 응답시간을 크게 개선할 수 있었고, 하드웨어 스펙이 낮아 실행이 어려웠던 low-end 스마트폰에서도 정상적으로 동작하는 것을 확인할 수 있었다. 또한 해당 어플리케이션 실행 시 단말의 부화를 줄임으로서 모바일 디바이스의 CPU-Utilization을 낮추고 소비전력도 크게 낮출 수 있었다. 웹 기반의 모바일 오프로딩 방식은 다양한 플랫폼에 적용될 수 있으며, 향후 새로운 모바일 컴퓨팅 환경을 제시하고 있다.

주요어: 오프로딩, 모바일 디바이스, 자바스크립트, 파워세이빙, 독립 플랫폼
학 번: 2011-20840

감사의 글

모든 일은 시작할 때와 끝날 때의 느낌이 많이 다른 것 같습니다. 석사과정을 시작할 때 막연했던 느낌이 지금은 오히려 아쉬움으로 되돌아옵니다. 2년이라는 시간을 위해 준비도 많이 하고 계획도 많이 세웠지만 지금 생각해 보니 정말 금방 지나버린 것 같습니다. 실험을 하고 연구 결과를 분석하면서 때로는 절망하기도 하였지만 문제를 해결에 나가면서 크고 작은 희열과 재미를 느껴왔던 것 같습니다. 지난 2년 동안 새로운 곳에서 새로운 경험을 느끼게 해주신 염헌영 교수님 및 엄현상 교수님께 진심으로 깊은 감사의 말씀을 드리고 싶습니다. 연구 활동 이외에도 많은 조언과 격려를 해주신 말씀들을 다시금 새겨들어 앞으로 더 많이 깨달을 수 있도록 하겠습니다.

제가 학교생활 및 연구에만 전념할 수 있게 도와주신 사랑하는 아내와 가끔은 저보다 더 의젓함을 보이는 네 살배기 아들 현준에게 고맙다고 말하고 싶습니다. 또한, 무엇보다 지금의 저를 있게 하시고 항상 용기와 격려를 보내주신 부모님께 감사드립니다.

같이 연구실 생활을 해온 소중한 인연에 대해서도 감사의 말씀을 드립니다. 이미 졸업하여 멋지게 자신의 꿈을 펼치고 계시는 정임영 박사님, 김형석 박사님, 신동인 박사님, 유영진 박사님, 조인순 박사님 그리고 무엇보다 제 논문과 연구에 많은 도움을 주신 한혁 박사님께 다시 한 번 진심으로 감사의 말씀을 드립니다. 앞으로 랩을 잘 이끌어 가실 신웅님, 언제나 랩의 구성원들을 올바른 길로 이끌어 주시는 영원한 랩장 신규님, 그리고 연구실의 궂은일을 항상 도맡아 하는 내영님과 용석에게도 다시 한 번 감사의 말씀을 드립니다. 많은 도움을 주신 1년차 선배들 용경, 완희, 승미, 승민, 운태, 그리고 함께 고생한 소중한 동기인 국태형, 재우, 찬호, 윤희에게도 졸업을 축하하면서 아울러 감사의 말씀도 함께 드립니다. 앞으로 분산시스템 랩을 잘 이끌어 갈 것이라고 믿어 의심치 않는 소중한 후배님들이신 웅, 동유, 민영, 성재, 계신, 문봉, 시봉, 지웅, 한울, 다혜에게도 감사의 말씀을 드립니다. 더 많은 추억을 일일이 글로 옮기지는 못하겠지만 그동안 많은 분들과 나누었던 소중한 추억들을 소

중히 간직하겠습니다. 앞으로 앞날에 무한한 희망과 축복이 함께 하기를 진심으로 희망합니다. 감사합니다.