Jade: An Efficient Energy-aware Computation Offloading System with Heterogeneous Network Interface Bonding for Ad-hoc Networked Mobile Devices

Hao Qian, Daniel Andresen
Department of Computing and Information Sciences
Kansas State University
Manhattan, USA
{hqian, dan}@ksu.edu

Abstract—**Mobile device users consistently desire faster results and longer battery life, which is frequently accomplished by offloading computation to cloud-based servers. However, in common cases (Internet connectivity slow/unavailable, privacy/security issues), users have multiple devices with heterogeneous battery and computational capabilities independent of the cloud. Android provides mechanisms for creating mobile code, but lacks a native scheduling mechanism for determining where code should be executed. In this paper we present the results of an investigation into adding sophisticated scheduling capabilities to Android apps, which provides scheduling balancing energy and performance across networked mobile devices. Jade monitors and adapts to workload variation, communication costs, and energy status in a distributed ad-hoc network of Android mobile devices for supporting distributed computation. We show how the two goals can be integrated, and present several algorithms indicating a major advantage (over 75% improvement in energy use) can be achieved through the use of dynamic scheduling information for remote computational devices. We provide a detailed discussion of our system architecture and implementation, and briefly summarize the experimental results which have been achieved.**

Keywords—distributed computing; ad-hoc networking; scheduling; mobile computing; multiple-radio systems

I. Introduction

Mobile devices (e.g., smart phones and tablets) have become a necessity for people because they enable us to perform a wide variety of daily activities (e.g., video calls, emails, gaming, social networking, navigation, etc.). These devices typically are equipped with a relatively powerful mobile processor, a rich set of sensors and a substantial amount of memory. It allows previously unimaginable applications to be developed by integrating many sensors (e.g., motion sensors, position sensors and environmental sensors).

Battery life has become one of the biggest obstacles for future growth of mobile devices. The energy needs of mobile devices are growing fast as processors are getting faster, screens are getting bigger and sharper, and more sensors are installed. Unfortunately, the advances in battery capacity have not kept up with the growing energy needs of mobile devices.

One popular technique to simultaneously reduce the energy consumption and increase the performance of mobile devices is computation offloading: application can reduce energy consumption by delegating code execution to other devices. Traditionally, computations are offloaded to remote servers which are resource-rich.

Nowadays, it is common for people to have more than one mobile device, and they carry those devices at the same time (e.g., smart phone in the pocket and tablet in the briefcase). These devices can be networked directly (e.g., Wi-Fi and Bluetooth) without an intermediate access point. Sometimes offloading computation to these networked devices is preferable than offloading computation to the cloud. One example is when people are using one device while the other device is charging. Offloading computation to the charging device could extend the battery life of the un-charged device. Another example is when the Internet connection is unavailable or not secure, offloading computation to locally networked devices is a good option.

In this paper, we present Jade, a computation offloading system for wireless ad-hoc networked mobile devices. Jade is targeted at mobile devices running the Android operating system. It minimizes energy consumption of applications through fine-grained code offloading while reducing the burden on application developers. Jade also reduces the energy consumption of network interfaces by dynamically choosing the most energy efficient interface (Wi-Fi/Bluetooth) for data transfer. Jade achieves these benefits by providing: 1) the runtime engine which supports computation offloading between wireless ad-hoc connected devices. By gathering the information about application and devices, the runtime engine can decide if code should run locally or remotely (offloaded); and 2) the simple programming model which helps developers to create applications that have the ability to offload computation.

We summarize our contributions here as follows:

- We present a scheduling algorithm which incorporates energy awareness and performance for a heterogeneous local cluster of devices.
- We present the design, implementation and evaluation of a complete system. Jade combines many features of the popular Android platform.
- We show that applications which support computation offloading can be implemented without heavy code modification. Android provides a natural separation between user interactive activities and background computational services. By following the similar concept, the Jade programming model ena-

978-1-4799-5604-3/14/$31.00 copyright 2014 IEEE
SNPD 2014, June 30-July 2, 2014, Las Vegas, USA

bles developers to write offloadable code with little effort.

- We evaluate the system with two applications. The result shows that computation offloading is an effective method for wireless ad-hoc networked devices. Jade can reduce up to 86% of energy consumption on mobile devices we tested, while maintaining/improving the performance of applications.

II. Jade Design

In this section, we present the high-level overview of Jade's architecture and its programming model in order to show how they all integrate into one system for distributed execution of mobile applications.

For mobile applications with heavy computation needs, computation offloading is a helpful technique to reduce the energy consumption and enhance the performance. However, it requires additional efforts and skills to develop applications with computation offloading ability, and there is no mature framework or tool for mobile application developers. We have designed Jade to minimize this effort by:

- offering the runtime engine which provides services for wireless communication, device profiling, program profiling and computation offloading. Conceptually, the Jade runtime engine automatically transforms computation on one mobile device into a distributed execution optimized for wireless connection, battery usage and capabilities of devices.
- providing the simple programming model for developing mobile applications which support computation offloading. The programming model includes a set of APIs for applications to interact with the Jade runtime engine.
- integrating with existing development tools that are familiar to developers.

In Jade, mobile applications contain *offloadable code* which can be offloaded to other devices (Figure 1). The device hosting the mobile application is called *the client*. The device which receives and executes the offloaded code is called *the server*. In Jade, if the code is executed on the client (i.e., code is not offloaded), we call it *local execution*. In contrast, if the code is executed on the server (i.e., code is offloaded), we call it *remote execution*.

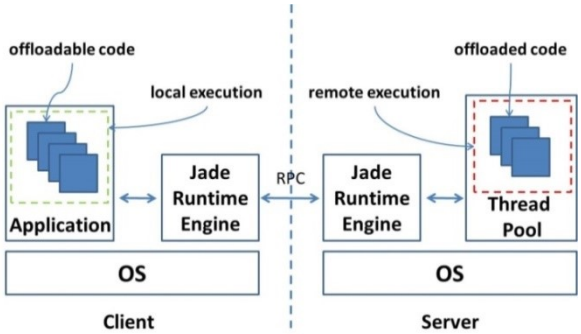

Figure 1: Computation offloading by Jade. In Jade, applications contain offloadable code which can be executed locally or remotely. Jade runtime engine decides where to execute the code and initiates the distributed execution.

A. Jade Runtime Engine

The goal of Jade is to maximize the benefits of computation offloading for mobile devices and minimize the burden on developers to develop such applications. To develop mobile applications which support computation offloading, there are several tasks we need to handle:

- *Communication*. In order to offload code from the client to the server, the applications should have the ability to 1) connect to other devices; 2) send data between devices; 3) coordinate with other devices for distributed execution; 4) track status of remote computation; 5) restore execution if unexpected errors happen (e.g., wireless connection lost, remote execution failure); and 6) exchange and maintain information related to connected devices (e.g. connection speed, CPU usage, battery level).
- *Profiling*. In order to make correct offloading decision, a profiler is needed which performs device and program profiling. Device profiling collects information about device's status, e.g., wireless connection, CPU usage, battery level. Program profiling collects information about applications, e.g., execution time, memory usage, size of data.
- *Optimization*. The purpose of optimization is to decide if computation is suitable for offloading, so as to maximize application's energy savings and performance.

These tasks are common for applications which support computation offloading. But it is time consuming to implement these tasks. Sometimes implementing these tasks is even harder than implementing the application itself. Jade runtime engine handles the above tasks for developers. It consists of components shown in Figure 2. On the mobile device, Jade runtime engine runs in the background as a group of services. Developers can utilize its various functionalities by using the easy-to-use APIs provided by the Jade programming model.

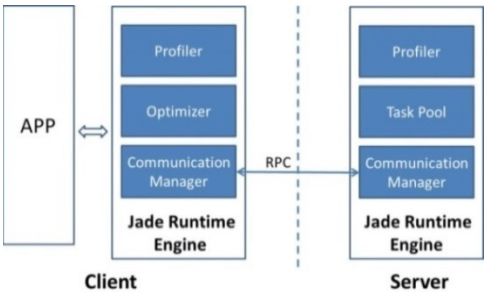

Figure 2: High level view of the Jade runtime engine.

In section III, we describe in detail the mechanisms for the profiler, the optimizer and the communication manager, respectively.

B. Jade Programming Model

In addition to the Jade runtime engine, one of the key contributions of Jade is the programming model. The programming model is offered to developers to help build applications which support computation offloading. It is the interface be-

tween mobile applications and the Jade runtime engine. Any application developed using the programming model can interact with the Jade runtime engine and harness the power of computation offloading.

We provide the details of the Jade programming model in section III.

C. Wi-Fi and Bluetooth Bonding

Today, most mobile devices are equipped with multiple network interfaces (e.g., Wi-Fi, Bluetooth and cellular interface). These interfaces have different characteristics (e.g., range, throughput and power). Each interface has its advantage and disadvantage given different applications. Ideally, we want to combine these interfaces and leverage strength of them in order to improve the application performance and reduce energy consumption.

Wi-Fi and Bluetooth are the most typical network interfaces found in today's mobile devices. Wi-Fi is known for its high throughput, long range and low energy per bit transmission cost. Bluetooth is primarily a cable-replacement technology for battery constrained mobile devices. It provides low bandwidth and covers a shorter range. The downside of Wi-Fi is the high power consumption for wake up and connection maintenance. In active state, the power of Wi-Fi is approximately 890mW, compared to only 120mW for Bluetooth [3]. For mobile devices with limited battery capacity, Wi-Fi has been shown to account for a significant portion (up to 50%) of the total energy consumption [3].

The 802.11 standard defines a Power-Saving Mode (PSM), aimed at reducing the energy consumption of Wi-Fi. In PSM, the typical power consumption of Wi-Fi is effectively reduced to 250mW. In contrast, Bluetooth is optimized to be extremely low-power in idle state, typically consuming on the order of 1mW [3].

Given the different and often complementary characteristics of both network interfaces, it is advantageous to combine Wi-Fi and Bluetooth together, so we can utilize their strength. In Jade, we implement a component in the communication manager called Wi-Fi and Bluetooth Bonding (WBB). The job of WBB is to decide which interface (Wi-Fi/Bluetooth) will be used for data transfer according to different data transfer request of applications. The goal is to reduce the energy consumption of network interfaces.

III. Implementation

In this section we will highlight the important implementation details of Jade. Sections A, B, C and D provide the details of the key components in the Jade runtime engine. Section E shows the details of the Jade programming model.

A. Profiler

At runtime, before the offloadable code is invoked, the Jade optimizer determines whether the code should run locally or remotely. This decision is based on the information provided by the profiler. The profiler collects the following information: 1) the device's status (i.e., charging or not, battery level, CPU load); 2) wireless connection status, such as connected or not, the bandwidth; and 3) characteristics of the offloadable code, such as running time and size. The profiler measures the code characteristics at initialization, and it continuously monitors the device and network characteristics, because for mobile devices, these can often change (e.g., wireless connection lost, battery reaches low level). The optimizer may make wrong decision on whether the code should be offloaded based on a stale measurement. The current implementation of the Jade profiler does not implement automatic program profiling. In the rest of this section, we provide the implementation details of Jade's techniques for device, program, and networking profiling.

1) Program Profiling

We use the DDMS [7] debugging tool provided by the Android Development Tools (ADT) [8] to profile applications. DDMS is a powerful tool which enables us to 1) view heap usage for a process; 2) track memory allocation of objects; and 3) profile methods of object. To measure energy consumption of applications, we use PowerTutor [9] and Trepn Plug-in for Eclipse [10]. PowerTutor is an application for Android phones that displays the power consumed by major system components such as CPU, network interface, display, and GPS receiver and different applications. It uses a power consumption model which provides power consumption estimates within 5% of actual values. Trepn plug-in for Eclipse is a power profiling tool developed by Qualcomm for Android application developers, it is designed to allow developers to easily collect performance data from any application running on a mobile device, analyze the resulting graphs in the Eclipse IDE and modify code to build power-efficient applications.

The offloadable code is invoked multiple times, each time with a randomly chosen input. For each execution, we measure: 1) runtime duration; 2) the size of data needs to be sent to the server (i.e., the size of the code, the size of data referenced by the code and the size of data required to be returned to the client once execution is complete); and 3) energy consumption of running the offloadable code. The final result is the average of multiple invocations.

2) Device Profiling

At runtime, the profiler keeps monitoring status of the device. Android uses broadcasts to notify applications if device status changes. Applications can register broadcast listeners to receive these broadcasts. The profiler registers broadcast listeners to receive broadcasts related to 1) battery (i.e., battery level, charging or not); and 2) wireless connection (i.e., Wi-Fi turned on/off).

Due to the nature of mobile devices, the status of wireless connection could change frequently (e.g., user moves to other location). Fresh information about wireless connection is critical for the optimizer to make correct offloading decision. Similar to MAUI [1], we use a simple method to measure the wireless link: Each time the Jade runtime engine offloads code, the profiler measures the transfer duration to obtain a more recent average throughput. This simple approach allows the profiler to take into account both the latency and bandwidth characteristics of the network. We also build a simple

energy cost model of wireless transfer using this approach: we send some synthetic data from the client to the server, varying the size of the data, and we measure the energy consumption of each transfer. This model lets us predict the energy consumption of transferring data as a function of the size of the data.

B. Optimizer

The purpose of the Jade optimizer is to pick which offloadable code to offload to the server, so as to find an offloading strategy that minimizes the client's energy consumption. The optimizer makes the offloading decision by solving an optimization problem using information collected by the profiler as input.

It requires a global view of the application and devices to decide where to execute the offloadable code. The optimizer can make adaptive decision based on the status of devices. For example: 1) if one device (client or server) is charging, the optimizer will try to offload as many computation as possible to that device in which case it is advantageous in terms of saving battery; 2) if the battery level of the server is low, the optimizer will send less code to the server (similarly if the wireless connection is bad); and 3) if the battery level of the client is low, more code will be offloaded to the server. The goal is to reduce as much energy consumption as possible on the client without unduly burdening the battery on the server.

The characteristics of offloadable code also determine where it should be executed. The code should be offloaded only if the energy consumed to execute it locally is greater than the energy consumed to transfer it. Code which performs heavy computation on small data falls into this category. For some code, the cost of transfer outpaces the cost of local execution (e.g., light weight computation on big data), they should not be offloaded.

Based on the information provided by the profiler, the optimizer finds the best offloading strategy by solving the following problem. I represents the set of offloadable code in the application. For each offloadable code $i \in I$, E_i^e is the energy consumed to execute it locally, E_i^t is the energy consumed to transfer i between the client and the server. T_i^l is the execution time of i on the client, T_i^r is the execution time of i on the server, T_i^t is the transfer time of i. I_i is the indicator variable: $I_i = 0$ if i is executed locally, $I_i = 1$ if i is executed remotely. The optimizer needs to find the assignment for I_i such that:

- *maximizes* $\sum_{i \in I} I_i \times (E_i^e - E_i^t)$
- *guarantees* $\sum_{i \in I} (1 - I_i) \times T_i^l + I_i \times T_i^r + I_i \times T_i^t \leq l$

The first formula is the total energy savings. The second constraint stipulates that the total execution time is within l. l can be configured by developers according to different requirements.

As explained in section E, the Jade programming model requires that every offloadable code i is independent of each other. This further simplifies the above problem. For each offloadable code i, the optimizer only needs to solve the following inequation:

$$E_i^e - E_i^t > 0$$

$$T_i^r + T_i^t - T_i^l \leq l$$

$E_i^e - E_i^t$ is the energy saving if i is executed remotely. i should be considered for offloading only when $E_i^e > E_i^t$. Similarly, the second inequation guarantees that time difference between remote execution and local execution is not greater than l.

C. Communication Manager

Computation offloading is handled by the communication manager. The communication manager is responsible for: 1) code manipulation (i.e., code transfer, code invocation); and 2) device coordination.

After the optimizer decides an offloadable code should be offloaded, the communication manager will perform the following tasks:

1. looks up the server table for available server (if no server available, then code is executed locally).
2. records information of the code in the offloaded code table, the purpose of the table is to track the status of the offloaded code.
3. offloads the code to the server.
4. the communication manager of the server receives the code, and executes it in a new thread in the task pool.
5. The communication manager of the server sends the result back to the client after the execution is complete.
6. the communication manager of the client receives the code and updates its information in the offloaded code table.

In case of remote execution failure, the communication manager has mechanisms to guarantee the correctness of the application. For example, if wireless connection is lost during the remote execution, the communication manager of the client will re-execute the code locally, and the communication manager of the server will abandon the failed execution. If the returned code shows its result as failed, it will also be re-executed on the client as well.

D. Wi-Fi and Bluetooth Bonding

Different applications have different patterns of data transfer. For example, video streaming applications need to keep receiving a lot of data in high frequency from the server. Weather applications transfer a small amount of data with low frequency (e.g., 10 minutes, 30 minutes or 1 hour). For big data transfers (e.g., image, video), Wi-Fi is the best choice, because it provides high throughput and energy efficiency. But for small and infrequent data transfer (e.g., transfer 1KB data every 10 seconds), Wi-Fi may not be the best choice. As shown in previous section, with the implementation of PSM, the power of Wi-Fi can be reduced effectively in idle state. But Wi-Fi has high power consumption for wake up and connection maintenance. For small and infrequent data transfer, the wake up cost for Wi-Fi is high, so Bluetooth is a better choice than Wi-Fi.

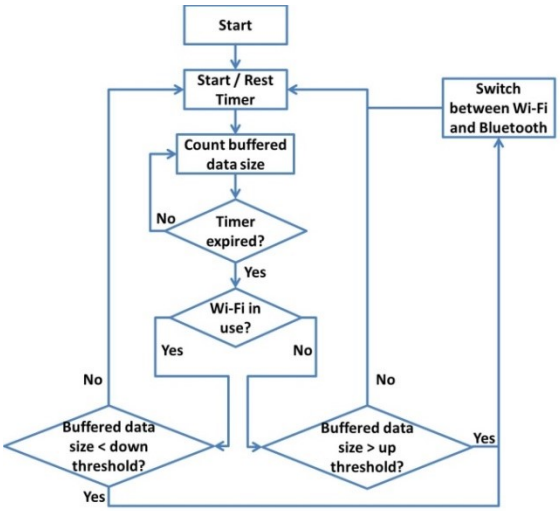

Figure 3: Wi-Fi and Bluetooth Bonding implementation

In Jade, we implement Wi-Fi and Bluetooth Bonding (WBB). WBB is adaptive to different data transfer pattern of applications. It can dynamically choose the suitable network interface (Wi-Fi/Bluetooth) for different data transfer request (Figure 3).

In WBB, the data to be transferred is put into a buffer. The size of the buffer is represented as *S*, down threshold is *D* and up threshold is *U*. The constraint is $0 \leq D < U \leq S$.

S, *D* and *U* can be configured by developers, for example, *S* is 500KB, *D* is 50KB and *U* is 350KB. At runtime, the buffer acts like a queue. The data to be transferred is put into the buffer from one end, then retrieved from the other end and finally sent by Wi-Fi/Bluetooth. WBB keeps monitoring the buffer and dynamically switches between Wi-Fi and Bluetooth according to the size of data in the buffer (Figure 4).

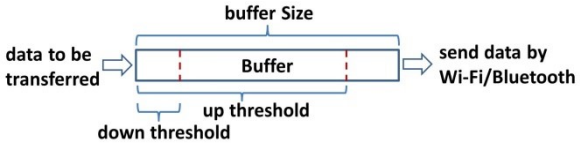

Figure 4: WBB monitors the buffer and choose suitable network interface for data transfer based on the size of data in the buffer.

This design guarantees that by properly configuring *S*, *D* and *U*, WBB can always make the desired choice regardless of the data production rate and the size of data. For example, if a high quality image needs to be transferred, WBB will choose Wi-Fi, because the size of the image can easily exceed the up threshold. Another example is an application which sends small data at high frequency. Due to the high data production rate, the size of data in the buffer will soon exceeds the up threshold, so Wi-Fi will be used to transfer data. In contrast, if small data is generated at low frequency, the size of data in the buffer could remain smaller than down threshold, so Bluetooth will be used for data transfer and Wi-Fi is kept in PSM.

E. Jade Programming Model

When designing applications which support code offloading, the application needs to be partitioned into sub-parts which can be offloaded to the server. There are different levels at which to partition an application (e.g., class, method, process, thread). In Jade, an application is partitioned at the class level. Developers can produce an initial partition of their applications with minimal effort by using the Jade programming model. A class simply needs to implement the *RemotableTask* interface if it should be considered for offloading by the Jade runtime engine.

In Jade, a class which implements the RemotableTask interface is called *remotable class*. An instance (object) of remotable class is called *remotable object* (i.e., offloadable code mentioned in previous sections). A remotable object can be executed on the client (locally) or on the server (remotely). An application developed using the Jade programming model is called Jade compatible application. At runtime, a Jade compatible application contains remotable objects which can be executed locally or remotely.

Some types of code must be executed locally, if a class contains the following code, it should not be considered for offloading:

- code that creates the user interface of the application.
- code that handles user interaction (e.g., callback method for clicking a button).
- code that access special hardware of the client which could be unavailable on the server (e.g., some smart phones are equipped with temperature sensor but some tablets are not).
- code that is not suitable for re-execution (e.g., code which perform online transaction) [1].

Our goal of implementing the RemotableTask interface is that developers don't need to guess if a class is suitable for offloading (in terms of energy consumption and performance). Once a class doesn't contain the above code, it can implement the RemotableTask interface.

The RemotableTask interface is the key construct in the Jade programming model, it defines the following methods which will be called by the Jade runtime engine in sequence:

- *preExecution*, which is used to do some preparations before the remotable object performs the main task, for instance, initializing the data.
- *loadData*, which is used to load data from the file system of the client. If the remotable object is running on the server, invoking this method will cause the client to read data from the file system and send it to the server.
- *execution*, which is invoked to perform the main task.
- *updateData*, which is the counterpart of loadData. After the task finishes, update data in the file system. If the remotable object is running on the server, the data will be sent back to the client to update its file system.
- *postExecution*, which is the counterpart of preExecution. If there are any things need to be done after the task completes (e.g., disconnect with database, update log and send notification to user), we do it here.

By implementing the RemotableTask interface, the life of a remotable object can be divided into three stages: 1) before execution; 2) execution; and 3) after execution. This division

is natural because it matches the steps of a general computation: 1) preparation (e.g., load data, connect to network); 2) execution, which performs computation on the data; and 3) update, which wraps up the execution (e.g., update database, notify user).

All remotable objects of an application will be considered for offloading by the Jade optimizer, if a remotable object is chosen for offloading, the Jade runtime engine will handle its remote execution following the steps shown in Figure 5.

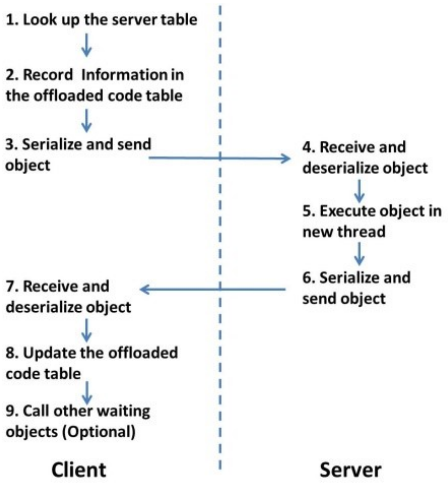

Figure 5: Life-cycle of remotable object which is offloaded to the server.

Using Jade programming model, the application development is intuitive. Figure 6 shows an example. Imagine we have an application which collects information entered by the user (e.g., name, phone number, address and company), verifies the information, and finally saves the information to the database on a remote server. The application could contain three steps. Step three is a good candidate for computation offloading, especially when the database operation is heavy. In the Jade programming model, we create a remotable class (UpdateInfo) for step three. The pseudocode looks like:

```
class UpdateInfo implements RemotableTask{
    preExecution(){
        open database connection;
    }
    loadData(){
        do nothing;
    }
    execution(){
        if new user
            insert user info into database;
        else
            update user info;
    }
    updateData(){
        do nothing;
    }
    postExecution(){
        close database connection;
    }
}
```

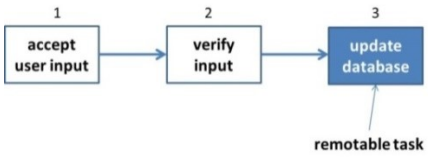

Figure 6: An application which contains task that could be executed remotely.

In Jade, a remotable class should perform task independently. The advantage of this design includes: 1) avoid the complexity introduced by dependencies when the Jade optimizer makes the offloading decision. In fact, the algorithm of the Jade optimizer is light weight, so it consumes less energy; and 2) the application benefits from parallel execution. Since each remotable object performs independent task, they can be executed simultaneously locally or remotely, which greatly enhances the performance of the application.

IV. Evaluation

In this section we evaluate Jade's ability to improve energy consumption and performance of mobile applications. We implemented two applications that contain heavy computation for the evaluation.

Our first application is FaceDetection. It asks user to choose some pictures from the photo gallery, then detects faces appearing in these pictures, and finally highlights them by putting a rectangle around each face. We build FaceDetection using the Jade programming model and the face detection library in Android. The code that detects faces is implemented as a remotable class, so it can be offloaded.

The second application is TextSearch, which searches text files against a library of 1000 key words. It counts how many times each key word appears in the text files. The search algorithm is implemented in a remotable class. At runtime, the text files can be searched locally or remotely.

We used an HTC One smart phone as the client. HTC one is a high end smart phone equipped with Qualcomm Snapdragon 600 quad-core 1.7GHz CPU and 2GB RAM. For the server, we use a Samsung Galaxy Tab 3 tablet and a Samsung Galaxy S4 smart phone. Galaxy Tab 3 is equipped with Intel Atom Z2560 dual-core 1.6GHz processor and 1GB RAM. Galaxy S4 has 1.9GHz quad core processor and 2GB RAM. All devices run Android 4.2.2. The client and the servers are connected by Wi-Fi and Bluetooth. We measure the energy consumption of the two applications on the client by the Trepn Plug-in for Eclipse and the PowerTutor.

At runtime, Jade can dynamically change its offloading strategy according to the battery level of devices. For example, if the battery level of the client is low and the server is charging, Jade will offload as much computation as possible to the server (aggressive mode). If the battery level of the server is low, Jade will offload less computation to the server to avoid draining its battery (moderate mode). In the following tests, the battery level of the client is kept below 20%, and the server is charged, so Jade works in aggressive mode.

We vary the number of servers to see if it has impact on the performance of applications. For tests with one server, we use the Galaxy Tab 3 as single server. For tests with two servers,

we add the Galaxy S4 as the second server. To see the impact of WBB on the energy consumption, we do each test with WBB enabled and disabled. When WBB is disabled, devices are connected with both Wi-Fi and Bluetooth, but only Wi-Fi is used for data transfer.

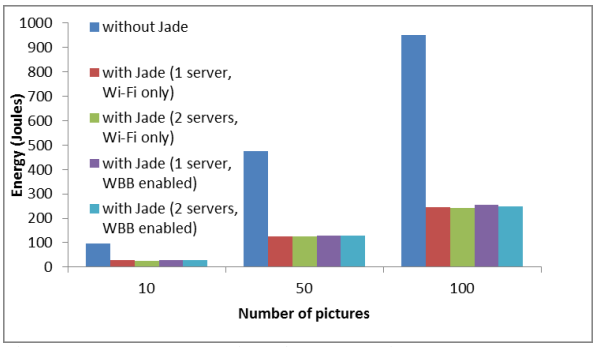

Figure 7: Energy consumption of FaceDetection.

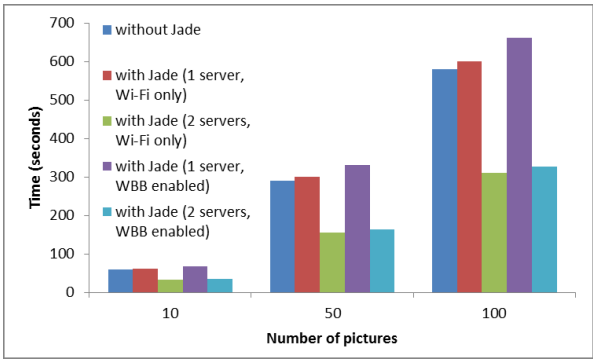

Figure 8: Execution time of FaceDetection.

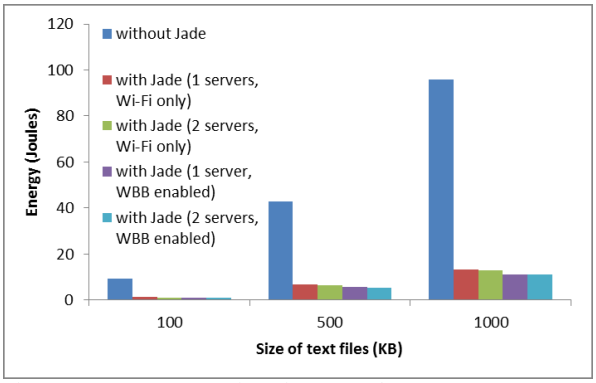

Figure 9: Energy consumption of TextSearch.

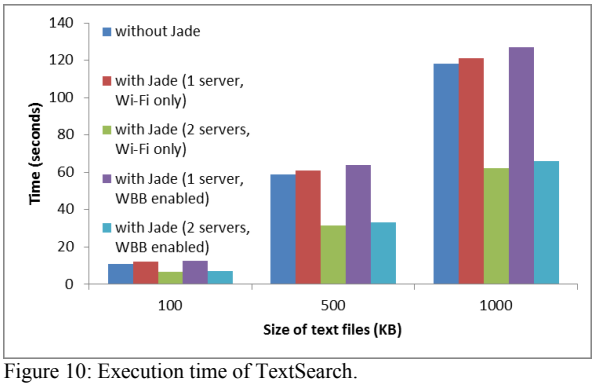

Figure 10: Execution time of TextSearch.

To evaluate how much energy Jade saves for FaceDetection, we execute it on the client with Jade enabled and disabled, and varying the number of pictures to detect. We only use pictures smaller than 200KB. The results are shown in Figure 7 and Figure 8.

We use similar method to evaluate TextSearch: execute it with Jade enabled and disabled, and varying the total size of text files that are searched. Each text file is 50KB. The energy consumption and execution time are shown in Figure 9 and 10.

From the tests, we can see that Jade saves 74% of energy for FaceDetection and 86% of energy for TextSearch. For execution time, when there is only one server, execution without Jade is a little faster than execution with Jade: there is 3.7% and 3% slowdown for FaceDetection and TextSearch. This is due to two reasons: 1) the server is less powerful than the client (CPU and memory); and 2) transfer overhead. When there are two servers, the performance improves: Jade reduces 43% and 48% of execution time for FaceDetection and TextSearch.

For FaceDetection, WBB does not reduce energy consumption compared with Wi-Fi only mode. This is because each remotable object contains an image, its size exceeds the up threshold of the buffer, so Bluetooth is never used for data transfer. For TextSearch, the application send small data (50KB per file) at long interval, so WBB effectively reduced 10% more energy compared with Wi-Fi only mode.

WBB keeps monitoring the buffer, and data needs to be stored in the buffer before transfer. This introduces overhead, so for both applications, the execution time with WBB is longer than Wi-Fi only mode, the execution time increased at most 10%.

The results demonstrate that Jade can effectively reduce battery consumption of applications while improving the performance. It also shows that WBB can dynamically choose suitable network interfaces. For applications which send small piece of data at low frequency, WBB can further reduce the energy consumption for applications.

V. Related Work

Mobile devices have limited resources such as battery capacity, storage and processor performance. Computation offloading is an effective method to alleviate these restrictions by sending heavy computations to resourceful servers and receiving the results from these servers. Many issues related to computation offloading have been investigated in the past decade: making offloading feasible, making offloading decisions, and developing offloading infrastructures.

Jade is built upon previous research done in program partitioning, code offloading, and remote execution. In this section, we give an overview of what has been proposed by these researches and how they relate to Jade.

Cuervo et al. proposed MAUI [1], a system that enables energy-aware offloading of mobile code to the infrastructure. MAUI enables developers to produce an initial partitioning of their applications by annotating methods and/or classes as remotable. At runtime, MAUI solver decides which remotable methods should execute locally and which should execute re-

motely. Unlike MAUI, Jade provides the programming model which enables developers to create remotable object. This has one significant benefit: the profiling and optimization overhead is low. Because In Jade, each remotable object is an independent unit performing some tasks, for the optimizer, it only needs to decide if a remotable object should be offloaded regardless of the other code of the application.

Chun et al. proposed CloneCloud [2], an application partitioner and execution runtime that enables unmodified mobile applications running in an application-level virtual machine to seamlessly offload part of their execution from mobile devices onto device clones operating in a computational cloud. In CloneCloud, to offload computation, threads need to be paused, all states of the threads need to be transferred to the server, and finally threads resume on the server. The offloading is expensive, especially when the client and the server are both resource constraint mobile devices. In contrast, code offloading in Jade is lightweight. Remotable objects are serialized, transferred and deserialized, the overhead is much lower than thread migration.

VI. Future Work

In our future work we will improve the Jade profiler by providing automatic application profiling. This will further reduce the burden on application developers. We will also study the impact of automatic profiling on the effectiveness of Jade, since it introduces more overhead.

VII. Conclusion

In this paper, we have presented Jade, a system which enables computation offloading for wireless ad-hoc networked mobile devices. Jade can effectively reduce the energy consumption of mobile devices (over 75% in our examples), and dynamically change its offloading strategy according to the status of devices.

We have evaluated Jade with two applications, a face detection application and a text search application. The result shows that Jade can reduce the energy consumption effectively for both applications while maintaining/improving the performance.

References

[1] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making smartphones last longer with code offload. In MobiSys, 2010.

[2] B. Chun, S. Ihm, P. Maniatis, M. Naik, A. Patti. CloneCloud: Elastic Execution between Mobile Device and Cloud. In ACM EuroSys, 2011.

[3] T. Pering, Y. Agarwal, R. Gupta, R. Want. CoolSpots: Reducing the Power Consumption of Wireless Mobile Devices with Multiple Radio Interfaces. In Proceedings of the 4th international conference on Mobile systems, applications and services, 2006.

[4] P. Rong and M. Pedram. Extending the Lifetime of a Network of Battery-Powered Mobile Devices by Remote Processing: A Markovian Decision-based Approach. In the 40th annual Design Automation Conference, 2003.

[5] S. Gurun C. Krintz and R. Wolski. NWSLite: A Light-Weight Prediction Utility for Mobile Devices. In 2nd international conference on Mobile systems, applications, and services, 2004.

[6] C. Wang and Z. Li. Parametric Analysis for Adaptive Computation Offloading. In ACM SIGPLAN 2004 conference on Programming language design and implementation, 2004.

[7] Dalvik Debug Monitor Server (DDMS), http://developer.android.com/tools/debugging/ddms.html, 2014.

[8] Android Developer Tools. http://developer.android.com/tools/index.html, 2014.

[9] PowerTutor. http://powertutor.org/, 2014.
Offloading Decisions. In IEEE International Symposium on Parallel and Distributed Processing, 2008.

[10] Trepn Plug-in for Eclipse. https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-plug-eclipse, 2014.