COFA : Automatic and Dynamic Code Offload for Android

Deepak Shivarudrappa
Department of ECEE
University of Colorado, Boulder
deepak.shivarudrappa@colorado.edu

MingLung Chen
Department of CS
University of Colorado, Boulder
tyge318@gmail.com

Shashank Bharadwaj
Department of ECEE
University of Colorado, Boulder
shashank.bharadwaj@colorado.edu

ABSTRACT

Personal computing devices like smartphones and tablets are taking over the market; with such devices becoming more ubiquitous, users expect more from their devices. Since the mobile device does not have enough processing power, one prevalent programming model among application developers is to adopt a client-server model such that the computationally heavy work is transferred to a server while the mobile device is configured as a client. We believe that such a programming model would hinder growth in the long run, to that end we propose the COFA system to solve this problem. COFA is a system that can automatically, based on network conditions, offload computation to a server. With COFA system in place, the application developer would have to target only the Android phones. Also users of the applications would instantly gain the advantages of the system with faster responses and better overall experience. In our paper we present the complete design of the COFA system.

Categories and Subject Descriptors

D.4 [**Software**]: Operating Systems; D.4.1 [**Operating Systems**]: Process Management—*threads*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Mobile cloud computing, code offload, energy management, partitioning, migration, smartphones

1. INTRODUCTION

Writing software for mobile phones is getting popular by the year, as is evident from the huge growth of applications written for them[20]. Applications are getting more complex as application developers try to leverage higher processing power of the newer devices. Manufacturers are fighting a constant battle of trying to provide more computation power while still maintaining acceptable battery life.

One popular way to write complex applications (such as face recognition or to process speech input) is to execute the computationally intensive parts on a server[11]. Although this makes the application perform faster and exerts lesser stress on the battery, the downside is that the application ceases to function if a data connection is unavailable. In this paper we propose a solution to this problem by having the application developers write the whole code so as to perform on the application. We modify the application framework such that computationally-intensive parts are automatically offloaded to servers if a data connection is present.

With the COFA system in place, applications would perform the complex operations on the device, although this might take a while, the application does not fail to perform the user requested action. If the user has a good data connection, the computation is automatically exported to a server which will perform the computation there. All of this will be invisible to the application user. The application developer too has little role to play here, other than write the application. The system we envision analyses the application code and performs a static code analysis to mark different parts of the program which can executed outside the device. When a network connection is present, such methods are executed on the server.

In this paper we present the complete design of the COFA system. We detail the design decision we made. Our only design goal has been to keep it highly modular such that this system would work on a variety of servers and configurations. We have used standard techiniques almost everywhere.

The paper is organized as follows: we present the current state-of-art and other related works in Section 2, Section 3 presents the system architecture; Section 4 we provide the details of our implementation; in Section 5 we explain the current status of our project and present what we would like to implement in the future;

we provide some concluding remarks in Section 6.

2. RELATED WORK

There has been recent research focused on code offload especially for mobile phones. Cuervo et. al. demonstrated in their work on MAUI[8] the feasibility of performing code-offload on mobile phones. The authors of MAUI design a code offload system for Microsoft's Common Language Runtime (CLR). The authors propose a pre-compiler which will use `@remotable` annotations provided by the application developer to inject code that handles the offload of code. Our work here is heavily based on the MAUI system with one key difference. Our work does not require any additional work from the application developer. Also, since we operate on application binaries, existing binaries in Android Market or Amazon's App Store could be repackaged and could directly become *offloadable*. Pathak et. al. in 2011 develop a system called XRay [15], which is an event-tracing-based profiling tool that identifies methods in a smart-phone application that can be offloaded to a remote server, and determines whether and when offloading the methods will benefit the application. Both MAUI and XRay partition the applications at the method-level granularity.

Mobile code-offloading is not a new problem, in essence it is the decade-old problem of application partitioning, and running parts on servers, applied to the context of mobile phones. More recently this problem has been studied by Donaldson et. al. in their Offload C++ application where the authors develop a compiler and runtime system for automatically offloading general-purpose C++ code to run on the Synergistic Processor Elements (SPEs) of the Cell Broadband Engine (BE) processor[12]. Georay et. al. showed that a Java Virtual Machine (JVM) could be modified to support code-offloading [13]. The authors develop a Micro Virtual Machine (MVM) as the core of the system and JnJVM as the extensible VM including a JIT. They use this system to develop code-offloading mechanism for Java applications. The authors in this paper are mainly concerned about large web applications. Since the different VMs were interconnected via a network, the authors also solve a lot of networking aspects along the way.

On a somewhat related note, Cliff Click demonstrated that a live Java VM could support global code motion, where parts of the computation in loops can be hoisted out of loops and executed else-where[7]. In recent past, Azul Systems has designed a segmented VM [18] which divides the VM into two parts, a shell VM and a core VM. The shell VM is assigned to always work with the UI and perform all interactions while the core VM is assigned to primarily computation and data management.

In 2009, Chun et. al. presented a similar idea, *cloud execution*[6] where they offloaded some tasks to a nearby computer, so that it could be executed in a cloned whole-system image of the device and after completion which the results were reintegrated on the smartphone. In 2011 the authors revisited the same topic with *CloneCloud* [5] in which they provide more motivations to the same execution model. CloneCloud performs both static and dynamic profiling to determine how to partition applications after which the application is automatically optimized for execution time and energy.

Quite recently, Chonglei Mei et. al. in their work [3] [4] show that offloading code would be highly benificial in improving the user experience. It would enable more complex applications to run on a resource-limited hardware. They provide a complete analysis on how to distribute binaries and present a multi-server-multi-user design. The work that we are doing in the COFA system can be applied directly to the work being done by these authors to build a completely dynamic system.

Satyanarayanan et. al talk about VM-based cloudlets [16], where a cloudlet is a nearby resource-rich computer. The authors suggest that a coffee shop could deploy such a cloudlet, and also provides low latency, one-hop, high-bandwidth wireless access to the cloudlet. The authors configure the mobile device as a thin client, which exports all significant computation to a nearby cloudlet. This work performs the computation on the cloudlet in a way very similar to that of VM-migration; the authors call it *dynamic VM synthesis* where only a smaller VM overlay is migrated between the client and server. The VM overlay is derived from the base VM, but the state contained in the VM overlay is significantly smaller. In our COFA system, we take advantage of the application binary so only remotely execute those threads which are marked offloadable. Due to this, we think that in our system the state transfer would be even smaller than the VM overlay.

3. ARCHITECTURE

In this section we present the system architecture for our project. This section will first give a short introduction to Android's architecture and then describe our system architecture.

3.1 How to partition the program?

In our project, our goal is to export as much as computation as possible to the server, when it is more energy efficient. Towards this end, we have to identify what parts of the given application which can be safely executed in the server without changing the program semantics. There are four major ways that a program can be partitioned such that parts of it can be executed remotely:

- **Pre-defined Interface:** Our system would specify a predefined interface and any class that implements our interface would be marked as being offloadble. This method inherently requires input from the application developer, which does not match our goals, so we quickly discarded this method.

- **VM-mirgration:** The complete state of the VM would be migrated into the server. Recent work by Satyanarayanan et. al. [16] uses a variant of this method called *dynamic VM synthesis*, where a smaller state (VM overlay)[1] is transferred instead, but a complete VM migration requires a lot of state transfer which could cause more computation to be performed locally.

- **Method Level:** Prior work from Cuervo et. al. [8] uses this approach. Although this approach provides fine grained granularity in terms of control on what methods to execute remotely, it would require significant state transfer as each method gets called. Also the static analysis phase (which would mark parts of the code as being offloadable), would be quite hard to get it right when looking at methods. Since the static analysis should not affect program semantics, it would have to be very conservative which would imply that most methods would not get marked as offloadable. Due to these reasons, we avoid this approach.

- **Thread Level:** We advocate this approach in this paper. In what follows, we explain the reasons why we choose this approach over the other we just discussed, and also present an overview of the static analysis pass that would mark the threads as offloadable.

3.1.1 Why Thread-level partitioning?

In this section we describe in a little more detail as to why we choose thread-level partitioning and then move on to describe the current state-of-art in thread migration and discuss how we propose to do it in the COFA system.

Android programming guidelines behooves the application developer to execute performance intensive tasks in a worker thread in Android[10]. Almost all worker threads are created in a way similar to the example presented in Listing 1. The UI or the main thread of the program spawns a new thread where the work is performed. Line 5 in the example shows the work that must be done by the worker thread (in this case it calculates the 28^th fibonacci number). The call-back mechanism is widely used to communicate results back to the UI or main thread. In our example the `resultCallback` is

[1]see Section 2 for a more detailed discussion

```
1  public void doInBackground() {
     Thread fibTh = new Thread(new Runnable(){
3
       @Override
5      public void run() {
         int fib = fibonacci(28);
7        resultCallback(fib);
       }
9    });
     fibTh.start();
11 }

13 public void resultCallback(int fib) {
     System.out.println("Fib: " + fib);
15   displayOnUI(fib);
   }
```

Listing 1: Example Android program doing computation in a thread

the call-back method which then displays the result on the application UI.

Since most application programmers already write code this way, we thought it would be most appropriate to support this programming model. Also, partitioning the user program at the thread-level has the advantage that it'd require less state transfer between the client and server VMs. This is essentially because the thread is created anew at the server and hence most of the state for the new thread is created with it, which would mean we would only need little state transfer. In our project, we support offloading at the thread-level.

3.1.2 Thread-level migration

Thread-migration as a mechanism has been studied quite well in the literature. In 2000 Truyen et. al. discussed how to add support for transparent thread migration[19], a few years later Zhu et.al presented the Jessica2 system[21] which is a distributed JVM with transparent thread migration support. In the what follows, we present a short description of this technique.

Thread migration is a mechanism to continue the current execution of a VM at another location. In order to migrate, the current VM saves the current state and transfers it to the other location. The target VM would re-establish the state and continue execution. In this context, by state we mean the state of an object (or a cluster of them).

In Java, each object consists of the following states:

- **Program state:** This is essentially the byte code of the class.

- **Data state:** This is the state of the instance variables of the object.

- **Execution state:** This is essentially the current state of execution of the thread. It consists of the

program counter and a private Java stack. This gets created when the thread is initialized [14].

Each JVM thread has its own program counter and has a private Java stack. The Java stack is equivalent to the stack of conventional languages. A Java stack stores frames. A new frame is created each time a Java method is invoked. A frame is destroyed when its method completes. Each frame holds local variables and an operand stack for storing partial results and passing arguments to methods and receiving return values. Conventional thread-migration techniques use the following approach to migrate a thread[19]:

1. execution of the current thread is suspended
2. the Java stack, program counter (PC) (execution state) and the data state are captured (in serializable format) and then this state is sent to the target.
3. stack is re-established at the target VM and PC set to the old code position.
4. finally, the thread is rescheduled for execution.

For resource-constrained mobile devices, we think it is important to minimize point 2 above, so that we keep state-transfer to a minimum. Towards this end, we simplify this process even further by adding a requirement that the complete life-cycle of a thread is migrated to the target device. This has the advantage that all the state gets recreated on the target device removing the need to transfer it from the client. As we saw in subsubsection 3.1.1 this model fits well with the existing Android programing paradigm.

3.2 Client-server architecture

In the COFA system, we propose to use a simple client-server architecture as shown in Figure 1. The server is a high-performance machine hosted over a high-bandwidth line. Our approach to this project has been to keep the code highly modular, we envision that the user would be able to run the server on their personal laptop or desktop too. This could enable a wide array of potentially one-hop server which would improve the performance of our system.

After having presented an overview of the architecture in this section, we now proceeed to describing the implementation in detail.

4. IMPLEMENTATION

In this section we will discuss the detail of our implementation of the COFA system. The section begins with a short introduction about Android and the Dalvik virtual machine and then continues onto presenting on how we modified the Dalvik machine to support automatic code offload.

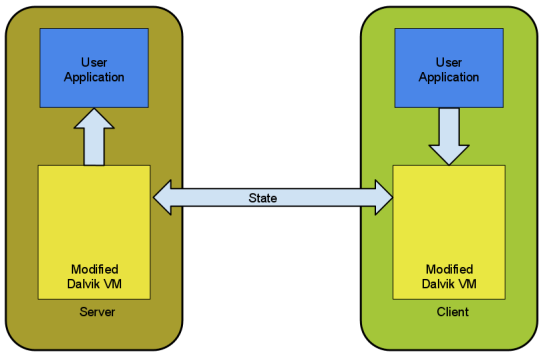

Figure 1: Basic architecture of the COFA system

Language	Files	Lines of Code
C	455	95,117
Assembly	991	46,344
Java	697	53,357

Figure 2: Dalvik Code Statistics (from [17])

Android is an open-source mobile operating system. Figure 3 shows the architecture of the android operating system[9]. The core of Android is the Dalvik Virtual Machine. Apart from the Dalvik, Android OS also consists of linux kernel and libraries which usually run on the core directly. The higher layers i.e. application framework and application layers are java source files translated to dex executables by dx tool. These dex executable are then interpreted by the Dalvik virtual machine.

Dalvik is a open source project in itself. It is implemented mainly in C, Assembly and Java. Dalvik VM[1] is a register-based virtual machine which operates on `dex` (Dalvik Executables) files. It has been optimized for low memory requirements and has been designed to allow multiple VM instances to run simultaneously.

Dalvik presently does not have a port for x86 architecture. Hence for our implementation we have an emulator running the modified Dalvik VM. Since all our modifications are restricted to the Dalvik VM this is sufficient. With x86 ports of the Dalvik in future, we could possibly extract much more performance since VM will be directly running on server hardware rather than an emulator.

Dalvik Virtual Machine will now have three modes of operations namely normal mode, server mode and client mode. In the normal mode of operation, there will be no code offload and operation will be similar to a unmodi-

Figure 3: Android system architecture

fied VM. In server mode the VM will host a server and wait for client to contact with requests. In the client mode, the VM has the capability to contact the server and transfer execution in case of offloadable methods present in the source. Different modes of operation are forced by arguments to the VM during initialization. Just in Time compilation (JIT) is presently disabled to reduce complexity.

- `dalvikvm -cp test.jar Main`
- `dalvikvm -cp test.jar -s Main`
- `dalvikvm -cp test.jar -c Main`

The first of the commands instructs the emulator to start up a Dalvik VM and run `test.jar` in the present directory. The function `Main` is interpreted by the Virtual machine. The arguments -s and -c force the Dalvik virtual machine into server mode and client mode respectively.

4.1 Communication

Our implementation uses Transmission Control Protocol (TCP) to communicate between the server and client. Using TCP makes sense because TCP provides more reliability than UDP. The present implementation supports single server, single client model only. Going forward this could extended to accommodate multiple servers and clients.

The server after initializing the VM, will execute the code to host server rather than interpret function `main` in `test.jar`. It initializes a socket and binds it to a predefined port number. Parameters `AF_INET` and `SOCK_STREAM` are used which result in TCP being the default protocol to be used. `SOCK_STREAM` makes the link connection oriented meaning an end to end connection is first established before any data is sent. Initialization of the socket is followed by a blocking call to `listen(...)` and waits for a request from the client. When a connection request is received server-client perform a three-way handshake to establish a connection. At this time a new descriptor is created and used in all further communications. The server then invokes a blocking call to `recv(...)` to receive state data from the client. If no errors occur on the transmission, the server unpacks the received state data and DVM interpreter resumes execution on the server. After successful execution on server, it sends back sate information to the client. Server then loops back, makes a call to `recv(...)` where the server waits for more requests from the client.

On the client side, Dalvik requests for a connection to the server. A socket descriptor is created and used for all further communications. When a offloadable thread is trapped, it sends a request to the server with the state information for execution on the server. It then makes a blocking call to `recv(...)`. Once it receives request for callbacks or the state information from the server, it takes the appropriate actions. After completion it applies the state information from server and resumes execution.

4.2 Offload.xml

In this part of the implementation we will describe the structure of the file `offload.xml` that will be created by the static analysis. This file describes all the threads that could potentially be executed on the server without changing the semantics of the program. We plan to provide a static analysis tool which will study the application byte code and generate such an xml file. For the purposes of this project, the offload.xml file was hand-written.

Before we understand the structure of the xml file itself, it is important to understand the structure of the android binary (APK file) and where we place the xml file.

4.2.1 APK File Format

The general structure of the APK file[2] is shown in Figure 4.

The APK files consist of the following:

- A “META-INF Folder” which contains meta-data and signature for contents of the APK.
- A “res Folder” which contains theme information of this application.
- Both the “layout folder” and the “xml folder” contain the XML files for theme. The layout folder stores the general layout structure of the applications graphic user interface; XML files for other purposes are stored in the “xml folder”.
- A “resources.arsc” file which contains the compressed source files.

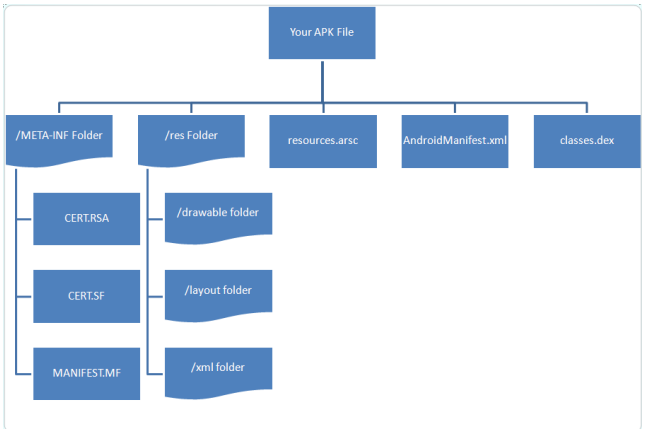

Figure 4: Structure of the APK file

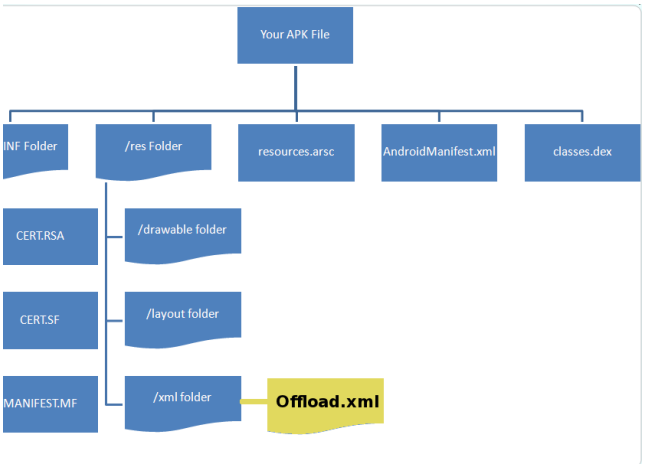

Figure 5: Structure of the APK file after including offload.xml

- A "AndroidMenifest.xml" file which stores the information required for the android to execute this APK file.
- A "classes.dex" which is the compiled java application.

We place our `offload.xml` file under `/res/xml` folder as highlighted in Figure 5.

A file named as `offload.xml` and placed under the `/res/xml` folder is identified by the Dalvik VM as the offloadable-xml. On a user's device, the presence of this xml file triggers the Dalvik to start-up in the client mode such that if network conditions are good, code gets offloaded to the server.

The structure of the `offload.xml` file is shown in Listing 2. Everything defined in the offload xml file should be enclosed by tags `<offloadable>` and `</offloadable>`. Since multiple offloadable threads are possible, using `<thread> </thread>` tags with different values of "id" attribute to specify them. The

```
1 <?xml version = "1.0"?>
  <offloadable>
3   <thread id="1">
      <name>Foo</name>
5     <class>java.foo</class>
    </thread>
7   <thread id="2">
      <name>comp</name>
9     <class>java.comp</class>
    </thread>
11 </offloadable>
```

Listing 2: Sample structure of the offload.xml file

`<thread>` and `</thread>` enclose the required information when offloading it: the thread name and the java class involved. The offload xml file is created either by static analysis or by hand-written. We manually wrote this xml file in our implementation.

4.3 Profiler

Our system follows other existing systems and includes a profiler [8], [5]. The need for the profiler is to determines if a thread should execute on the server or locally. The profiler consists of three parts:

1. **Device Profiler:** Evaluates the device characteristics and monitors the energy consumption. A thread execution that consumes too much power is suggested to be offloaded to a server.
2. **Program Profiler:** Records the time/instructions of a threads previous execution to predict the cost of its next execution. A thread execution that takes too much time is suggested to be offloaded to a more powerful server (if there exits one).
3. **Network Profiler:** Examines current wireless network quality. This is the most fundamental part of the profiler. Network quality has the major impact in decision to offload a thread. Offload is impossible when no wireless network, and poor network quality results in expensive cost of state transmission.

4.4 Walk through

Consider an example program as shown in Listing 3. As discussed earlier this is prevalent structure followed in Android application development. The computationally intensive task is relegated to a newly created thread. In the example computation of n^th number in a fibonacci sequence is relegated to thread `fibTh`. The thread is defined in Listing 4. `fibTh.start` shifts control to the `run()` in thread object. The thread object then completes the computation and makes a call to the callback function. If observed, this function belongs to

```
1  public class Main {
     public Main(){
3          Thread fibTh = FibTh(this);
           fibTh.start();
5    }

7    public void resultCallback(int fib){
       System.out.println( Fib:     + fib);
9      displayOnUI(fib);
     }
11 }
```

Listing 3: Example Android program doing computation in a thread

```
   public class FibTh extends Thread {
2      private Object callbackObj;
       public FibTh(Object caller) {
           callbackObj = caller;}
4

6      @Override
       public void run() {
8        int fib = fibonacci(28);
         callbackObj.resultCallback(fib);
10     }

12     private int fibonacci(int f){
         if(f==0 || f==1)
14         return f;
         else
16         return fibonacci(f-1)
               + fibonacci(f-2);
18     }
   }
```

Listing 4: Definition of thread class used by Listing 3

the caller who created the thread. Callback is the mechanism used where caller passes a reference to the callee thread to contact once thread is done. This does not require the caller to block till the thread returns.

The client begins by executing the main in Listing 3. The client would create an instance of class `FibTh` i.e. would spawn a thread. The `fibTh.Start` would start the execution. At this point we trap the thread within the Dalvik VM. The thread name is compared with the list of offloadable threads specified in the offloadable XML file described earlier. In case of a match with an element in the list, the thread is offloaded to the server. In the above example the following information is passed to the server

- **Thread Index** in the dex source files. This will help the server to locate the source to be executed in the apk file located on the server
- **Class Descriptor** which is the fully qualified class name

The server on receiving the above information, creates an instance of the thread by calling the constructor with custom arguments. The control is shifted to the function `run()` in the thread object. All the subsequent instructions are interpreted on server. In case of callbacks the thread refers to objects/ references passed by the caller to the constructor of the thread. Using the custom arguments passed to the thread constructor in the server we trap any callbacks. The information is sent to the client to execute the callbacks. This is required to maintain the program's semantics and sane behaviour. Once the execution terminates the thread is killed in the server and the client intimated of the current state of the Dalvik virtual machine. In this particular case since there is no actual state that needs to be transferred the server after intimating the client to execute the callback, will kill the thread. The client on receiving the callback will execute the call back function. Once it learns of completion of the thread, it resumes the execution of the code. Above process is repeated for any other offloadable thread.

5. STATUS AND FUTURE WORK

Currently we have implemented the TCP communication modules. Dalvik VM is able to execute in server mode with the `-s` argument specified, in client mode with the `-c` argument specified, or in normal mode with neither of these arguments are specified. Dalvik VM in server mode is able to allocate a socket for client to contact and serve the next request from the client when it completes a task. Dalvik VM in client mode is also able to allocate a socket to contact the server and offload its state to server through the TCP connection. Unpacking the state received from client in server is resolved.

However, there are still issues we need to further work on. Handling callback functions and side effects has not been implemented yet. The current profiler only checks for network connection and does not profile the device or program characteristics.

5.1 Future Work

We define our future work as follows:

- **Removing the offload.xml:** One limiting factor of our design is that we need a static analysis tool to generate an offload.xml. We would like to move to a design where the xml file is generated dynamically too.
- **Design a More Advanced Profiler:** We expect a more advanced profiler that is able to measure the network quality, instead of just testing if theres a connection, and choose the best server to offload its task when multiple servers exists. It should also be capable of adjusting its offload policy according to the overall evaluation on the profiler (i.e. the device, program and network characteristics). And, for the long term goal of a profiler, it should be intelligent enough to examine the APK file and tell which thread is offloadable, so that theres no need to explicitly run a static analysis to the APK files and create the offloadable xml file.

We strongly believe that with this additional work, the COFA system has a bright future. If network providers like ATT or T-mobile or even companies like Google (with their Android Market) or Amazon (with their App Store) could offer this service. They could repackage the apk files from the developers to include the offload.xml (or skip this step completely if we get to a point where we can dynamically calulate the offloadable threads) and then it would be immediately be recognized by the Dalvik VM such that a client mode is triggred. This would enable a seamless experience for both the end user and the developer, who gain automatic code offloading.

6. CONCLUSIONS

We have designed the COFA system, with automatic and dynamic code offload capabilities for the Android system. Our solution is to add support directly into the Dalvik virtual machine in Android. We have studied the Android programming model and the Java VM and decided that thread-level partitioning would be most appropriate for Android since it would be the best tradeoff of granularity versus amount of state to be transferred. We have implemented the system on the Davik project under the open source Android project for gingerbread. Although we have not been able to completely implement and test the whole system, we have transferred state between the server and client and have good confidence that the system would work.

7. ACKNOWLEDGMENTS

We would like to thank Prof. Richard Han for his constant support and encouragement without which this project would not have started. We would also like to thank all the Android developers (both in Google and from the open source community) for creating and releasing the source code for the Android operating system without which this project would have stalled.

8. REFERENCES

[1] D. Bornstein. Dalvik vm internals. In *Google I/O Developer Conference*, volume 23, pages 17–30, 2008.

[2] Chibucks. General structure of an apk. *http://forum.sdx-developers.com/index.php?topic=3472.0*.

[3] C. W. A. C. Chonglei Mei, Daniel Taylor and J. Weissman. Mobilizing the cloud: Enabling multi-user mobile outsourcing in the cloud. 2011.

[4] C. W. A. C. Chonglei Mei, James Shimek, J. Weissman, and C. Mei. Dynamic outsourcing mobile computation to the cloud. 2011.

[5] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[6] B. Chun and P. Maniatis. Augmented smartphone applications through clone cloud execution. In *Proceedings of the 12th conference on Hot topics in operating systems*, pages 8–8. USENIX Association, 2009.

[7] C. Click. Global code motion/global value numbering. *SIGPLAN Not.*, 30:246–257, June 1995.

[8] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[9] A. Developers. What is android? *Internet: http://developer.android.com/guide/basics/what-is-android.html [Sept, 5 2011]*.

[10] A. Developers. Painless threading. *http://developer.android.com/resources/articles/painless-threading.html*, 2011.

[11] A. Developers. Speech input API in Android. *http://developer.android.com/resources/articles/speech-*

input.html,
2011.

[12] A. Donaldson, U. Dolinsky, A. Richards, and G. Russell. Automatic offloading of c++ for the cell be processor: a case study using offload. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*, pages 901–906. IEEE, 2010.

[13] N. Geoffray, G. Thomas, and B. Folliot. Transparent and dynamic code offloading for java applications. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4276 of *Lecture Notes in Computer Science*, pages 1790–1806. Springer Berlin / Heidelberg, 2006.

[14] B. Joy, G. Steele, G. Bracha, and A. Buckley. The java language specification. 2011.

[15] A. Pathak, Y. Hu, M. Zhang, P. Bahl, and Y. Wang. Enabling automatic offloading of resource-intensive smartphone applications. Technical report, Tech. rep, 2011.

[16] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, 8(4):14 –23, oct.-dec. 2009.

[17] SourceForge. Count lines of code. *http://cloc.sourceforge.net/*.

[18] G. Tene and S. Pillalamarri. Segmented virtual machine, May 19 2009. US Patent 7,536,688.

[19] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable support for transparent thread migration in java. *Agent Systems, Mobile Agents, and Applications*, pages 377–426, 2000.

[20] T. van Agten. Windows phone 7 marketplace: One year later. *http://www.distimo.com/publications*, November 2011.

[21] W. Zhu, C. Wang, and F. Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 381–388. IEEE, 2002.