Cuckoo: A Computation Offloading Framework for Smartphones

Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal

Vrije Universiteit, De Boelelaan 1081A, Amsterdam, The Netherlands
{rkemp,palmer,kielmann,bal}@cs.vu.nl

Abstract. Offloading computation from smartphones to remote cloud resources has recently been rediscovered as a technique to enhance the performance of smartphone applications, while reducing the energy usage.

In this paper we present the first practical implementation of this idea for Android: the Cuckoo framework, which simplifies the development of smartphone applications that benefit from computation offloading and provides a dynamic runtime system, that can, at runtime, decide whether a part of an application will be executed locally or remotely. We evaluate the framework using two real life applications.

Keywords: Mobile Computing, Computation Offloading.

1 Introduction

In the last decade we have seen, and continue to see, a wide adoption of advanced mobile phones, called *smartphones*. These smartphones typically have a rich set of sensors and radios, a relatively powerful mobile processor as well as a substantial amount of internal and external memory. A wide variety of operating systems [1,2,3,4] have been developed to manage these resources, allowing programmers to build custom applications.

Centralized market places, like the Apple App Store [5] and the Android Market [6], have eased the publishing of applications. Hence, the number of applications has exploded over the last several years – much like the number of webpages did during the early days of the World Wide Web – and has resulted in a wide variety of applications, ranging from advanced 3D games [7], to social networking integration applications [8], navigation applications [9], health applications [10] and many more.

Not only has the number of third-party applications available for these mobile platforms grown rapidly – from 500 [12] to 200,000+ [13] applications within two years for the Apple App Store –, but also the smartphones' processor speed increased along with its memory size (see Figure 1), the screen resolution and the quality of the available sensors. Furthermore, the cell networking technology grew from GSM networks allowing for 14.4 kbit/s to the current 4G networks that will provide around 100 Mbit/s, while simultaneously the local wireless networks increased in bandwidth [14,15].

M. Griss and G. Yang (Eds.): MOBICASE 2010, LNICST 76, pp. 59–79, 2012.

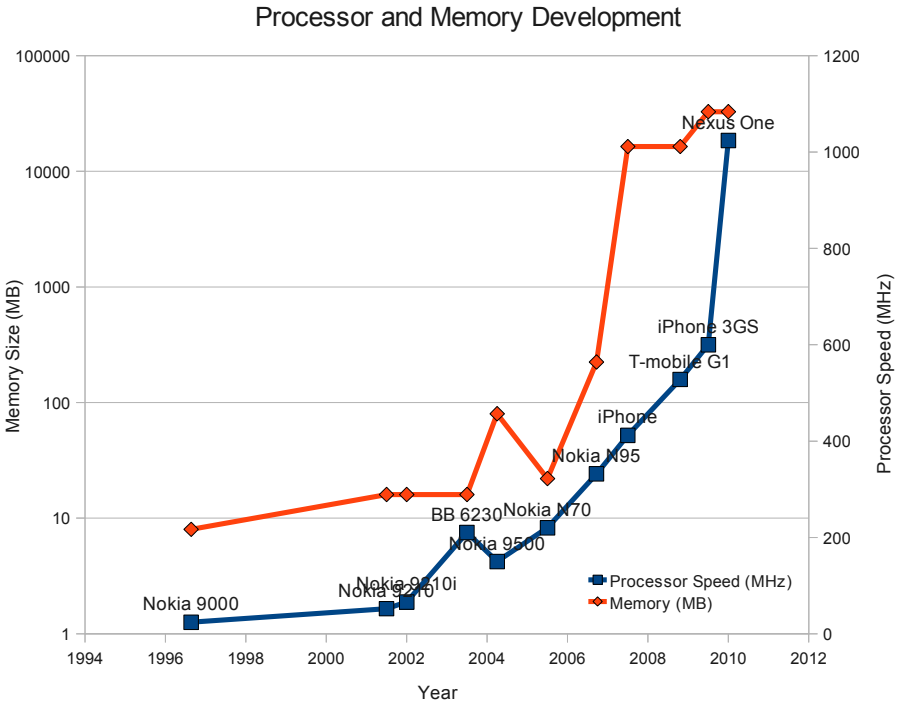

Fig. 1. Starting from the earliest Nokia 9000 Communicator, the processor speed as well as the memory size have grown enormously. In this graph, we show the increase in processor speed (linear scale) and the increase of the memory size (logarithmic scale) of some of the top model smartphones over the last 15 years. Data gathered from `http://www.gsm-arena.com`.

Todays' smartphones offer users more applications, more communication bandwidth and more processing, which together put an increasingly heavier burden on its energy usage, while advances in battery capacity do not keep up with the requirements of the modern user.

The original idea of offloading computation from a thin client to a rich server is well known from a speed performance point of view. Only recently, it has been recognized that offloading computation using the available communication channels to remote cloud resources can also help to reduce the pressure on the energy usage [16,17,18].

In this paper we eleborate on the idea of computation offloading and present a practical system, called *Cuckoo*, that can be used to easily write and efficiently run applications that can offload computation. Cuckoo is targeted at the Android platform, since Android provides an application model that fits well for computation offloading, in contrast with other popular platforms, such as the iPhone.

Cuckoo offers a very simple programming model that is prepared for mobile environments, such as those where connectivity with remote resources suddenly

disappears. It supports local and remote execution and it bundles both local and remote code in a single package. Cuckoo integrates with existing development tools that are familiar to developers and automates large parts of the development process. Furthermore, it offers a simple way for the application user to collect remote resources, including laptops, home servers and cloud resources.

The contributions of this paper are:

- We show that computation offloading can be implemented elegantly, if the underlying operating system architecture differentiates between interactive user-oriented activities and computational background services.
- We present Cuckoo[1]: a complete framework for computation offloading for Android, including a runtime system, a resource manager application for smartphone users and a programming model for developers, which integrates into the Eclipse build system. The runtime system supports custom remote implementations optimized for remote resources and can be configured to maximize computation speed or minimize energy usage.
- We evaluate the framework with two real life example applications.

The outline of this paper is as follows. In Section 2 we will discuss the Android operating system, which is the platform that we have selected for our research. Section 3 then discusses the design of the Cuckoo framework and Section 4 presents important details about the implementation. In Section 5 we evaluate our system with two example applications. The related work is covered in Section 6 and we conclude with presenting our future work and conclusions in Sections 7 and 8.

2 Background: Android

First, we will turn our attention to the Android platform, as we need to understand how mobile applications on Android are composed internally, before detailing the design and implementation of Cuckoo.

Android is an open source platform including an operating system, middleware and key applications and is targeted at smartphones and other devices with limited resources. Android has been developed by the Open Handset Alliance, in which Google is one of the key participants. Android applications are written in Java and then compiled to Dalvik bytecode and run on the Dalvik Virtual Machine.

2.1 Android Application Components

The main components of Android applications can be categorized into *Activities*, *Services*, *Content Providers*, and *Broadcast Receivers*, which all have their own specific lifecycle within the system.

[1] The framework is named after the Cuckoo bird which offloads egg brooding to other birds.

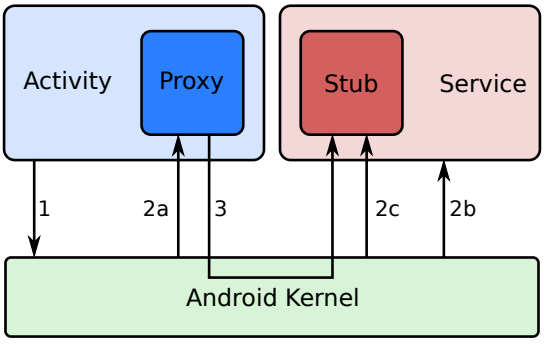

Fig. 2. A schematic overview of the Android IPC mechanism. An activity binds to a service (1), then it gets a proxy object back from the kernel (2a), while the kernel sets up the service (2b) containing the stub of the service (2c). Subsequent method invocations by the activity on the proxy (3) will be routed by the kernel to the stub, which contains the actual implementation of the methods.

Activities are components that interact with the user, they contain the user interface and do basic computing. Services should be used for CPU or network intensive operations and will run in the background, they do not have graphical user interfaces. Content Providers are used for data access and data sharing among applications. Finally, Broadcast Receivers are small applications that are triggered by events which are broadcasted by the other components in the system.

For computation offloading, we focus on activities and services, because the separation between the large computational tasks in the services and the user interface tasks in the activities form a natural basis for the Cuckoo framework. We will now have a closer look at how activities and services communicate in Android.

2.2 Android IPC

When a user launches an application on a device running the Android operating system, it starts an activity. This activity presents a graphical user interface to the user, and is able to bind to services. It can bind to running services or start a new service. Services can be shared between multiple activities. Once the activity is bound to the running service, it will communicate with the service through inter process communication, using a predefined interface by the programmer and a stub/proxy pair generated by the Android pre compiler (see Figure 2). Service interfaces are defined in an interface definition language called AIDL [19]. Service methods are invoked by calling the proxy methods. These proxy methods can take primitive type arguments as well as *Parcelable* arguments. Parcelable arguments can be serialized to Parcels and created from Parcels, much like Java Serialization serializes and deserializes Objects from byte arrays. The Android

IPC also supports callbacks, so that the service can invoke a method on the activity, allowing for asynchronous interfaces between activities and services.

2.3 Android Application Development

Android applications have to be written in the Java language and can be written in any editor. However, the recommended and most used development environment for Android applications is Eclipse [20], for which an Android specific plugin is available [21].

Eclipse provides a rich development environment, which includes syntax highlighting, code completion, a graphical user interface, a debugging environment and much more convenient functionality for application developers.

The build process of an Android application will be automatically triggered after each change in the code, or explicitly by the developer. The build process will invoke the following builders in order:

- **Android Resource Manager**: which generates a Java file to ease the access of resources, such as images, sounds and layout definitions in code.
- **Android Pre Compiler**: which generates Java files from AIDL files
- **Java Builder**: which compiles the Java source code and the generated Java code
- **Package Builder**: which bundles the resources, the compiled code and the application manifest into a single file

After a successful build, an Android package file (.apk) is created, which can be installed on a device running the Android operating system.

3 Cuckoo Design

After initial research on the topic of computation offloading in [18], we found that computation offloading as a technique is very useful to enhance the performance and reduce the battery usage of applications with heavy weight computation. However, adding offloading to such applications requires additional effort and skills of application developers. We focus on minimizing this effort by:

- offering a very simple programming model that is prepared for connectivity drops, supports local and remote execution and bundles all code in a single package.
- integrating with existing development tools that are familiar to developers.
- automating large parts of the development process.
- offering a simple way for the application user to collect remote resources, including laptops, home servers and other cloud resources.

In this section we show which parts of the development process we automated, how we integrated our system into the existing development tools and what the programming model looks like. We will also discuss how the user can collect remote cloud resources. In Section 4, we will present the details of the implementation.

3.1 Programming Model

One of the key contributions of Cuckoo is the programming model that is offered to application developers. This programming model acts as the interface of the system to the developers and therefore should be easy to use and to understand.

Secondly, since smartphones are not always connected to networks, making cloud resources sometimes unreachable, the programming model must support both local and remote method implementations.

Furthermore, the programming model must specifically support remote implementations to be different from the local implementation, so that, for instance, parallelization can be used at the remote implementation to get the full performance out of a remote multiprocessor resource.

As a last requirement, the programming model must bundle all local and remote code to be together, so that the user will always have a compatible remote implementation.

To make the programming model easy to use and to understand, we decided to use the existing *activity/service model* [22] in Android that makes a separation between compute intensive parts (services) and interactive parts of the application (activities), through an interface defined by the developer in an interface definition language (AIDL). For Android applications that contains compute intensive operations, there will be already such an interface available if it is well programmed. Otherwise an interface can easily be extracted from the code. This interface will be implemented as a local service that has, when used, a proxy object at the activity.

Next to this local service implemented by the programmer, the Cuckoo framework generates an implementation of the same interface for a remote service. Initially this implementation will contain dummy method implementations, which the programmer has to replace with real method implementations that can be executed at the remote location. The real methods can be identical to the local service implementation, but may also be completely different, since the remote implementation can run a different algorithm, use a different library, run in parallel, etc.

3.2 Integration into the Build Process

The Cuckoo framework provides two Eclipse builders and an Ant [23] build file that can be inserted into an Android project's build configuration in Eclipse.

The first Cuckoo builder is called the *Cuckoo Service Rewriter* and has to be invoked after the Android Pre Compiler, but before the Java Builder. The Cuckoo Service Rewriter will rewrite the generated Stub for each AIDL interface, so that at runtime Cuckoo can decide whether a method will be invoked locally or remote.

The second Cuckoo builder is called the *Cuckoo Remote Service Deriver* and derives a dummy remote implementation from the available AIDL interface. This remote interface has to be implemented by the programmer. Next to generating the dummy remote implementation, the Cuckoo Remote Service Deriver also

generates an Ant build file, which will be used to build a Java Archive File (jar) that contains the remote implementation, which is installable on cloud resources. The Cuckoo Remote Service Deriver and the resulting Ant file have to be invoked after the Java Builder, but before the Package Builder, so that the jar will be part of the Android Package file that results from the build process. A schematic overview of how the Cuckoo components integrate into the default Android build process is shown in Figure 3, whereas Table 1 shows the order of the development process.

3.3 Integration into the Runtime

At runtime method invocations to services are intercepted at the service side of the Stub/Proxy pair, which is at the Stub. The Cuckoo framework will then decide whether or not this method should be invoked locally or remotely. The Cuckoo framework will query the Cuckoo Resource Manager for any available resources to see whether these resources are reachable and, if so, use them.

3.4 Cloud Resources

An application which uses Cuckoo for computation offloading, can offload its computation to any resource running a Java Virtual machine, either being machines in a commercial cloud such as Amazon EC2 [24] or private mini clouds such as laptops, desktops, home servers or local clusters. The user runs a simple Java application, the *server*, on such a resource to enable it to be used for computation offloading. The server that runs on such a resource does nothing by itself, however, services available on a phone can be installed onto such a server.

A part of the Cuckoo framework is a Resource Manager application that runs on the smartphone. In order to make a remote resource known to a phone, the remote resource has to register its address to this Resource Manager using a side channel. If a resource has a display, starting a server will result in showing a two dimensional barcode – a QR code [25] – on the resources' screen. This QR code contains the address of the server. Smartphones are typically equiped

Table 1. Application Development Process. An overview of what steps need to be performed during the development process of an Android application that supports computation offloading.

step	actor	action
1	developer	creates project, writes source code
2	developer	defines interface for compute intensive service
3	build system	generate a stub/proxy pair for the interface and a remote service with dummy implementation
4	developer	writes local service implementation, overwrites remote service dummy implementation
5	build system	compiles the code and generates an apk file
6	user	installs the apk file on its smartphone

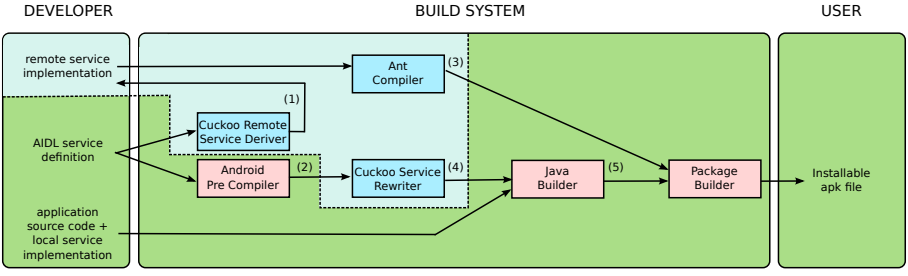

Fig. 3. Schematic overview of the build process of an Android application, the area within the dashed line shows the extensions by Cuckoo for computation offloading. The figure shows the process of building an Android application from source code to a user installable apk file. The developer has to write application source code and if applicable AIDL service definitions and implementations for these services. Then the build system will use these to generate Java Source files from the AIDL files (2), compile the source files into Dalvik bytecode (5) and bundle them into an installable apk file. To enable an application for computation offloading, the only thing the developer has to do is to overwrite remote service dummy implementations (1) generated by the Cuckoo Remote Service Deriver. The Cuckoo Service Rewriter will insert additional code into the generated Java Source Files and these rewritten Source Files (4) will be compiled and subsequently bundled with the compiled remote implementation (3) to again an installable apk file.

with a camera and can scan this QR code using a special resource manager application (see Figure 4). If the resource does not have a visual output, a resource description file can be copied from the resource to the phone to register the resource. When the resource is known to the Resource Manager application, it can be used repeatedly for any application that uses the Cuckoo computation offloading framework.

3.5 Limitations

Cuckoo does not yet support callbacks, although they are supported by the Android interface definition language. Implementing asynchronous callback communication for remote services is challenging and is part of our future work.

Method arguments can only be used as input parameters and cannot be used in a C-style way as output parameter. With Cuckoo, only the return object of a method will be available to the activity. Currently, Cuckoo does not support any form of security, which means that the remote resources can be accessed by untrusted phones, which in turn can install any code onto the system. However, setting up a security infrastructure can be realized and will be part of our future work.

Cuckoo supports only stateless services. Although the programming model does not forbid a service to maintain internal state, Cuckoo can arbitrarily change from local to remote execution or from one remote resource to another without transfering state. We do not support such state transfers, because at the

Fig. 4. A screenshot of the resouce manager collecting the address of a remote resource. From now on this resource is known to the smartphone and will be used by any application that uses Cuckoo for computation offloading.

moment the state needs to be transfered, it is generally too late to access the state, because the connection to the running service has been lost. Nonetheless, this limitation is acceptable, since application developers understand well how to transfer state between stateless services by using the Representational State Transfer (REST) architectural style [26].

4 Implementation

In this section we will highlight the important implementation details of the Cuckoo framework. We will describe the Ibis communication library that we use to communicate with the remote resources and the protocol between the phone and the remote resources. Finally we show how Cuckoo can be used in different configurations.

4.1 Offloading Decision

When an activity invokes a method of a service, the Android IPC mechanism will direct this call through the proxy and the kernel to the stub. Normally, the stub will invoke the local implementation of the method and then return the result to the proxy. The Cuckoo system intercepts all method calls and then decides whether it is benificial to offload the method invocations or not. We plan to use a combination of heuristics, context information and history to evaluate an offloading decision. For now, we use the very simple heuristic to always prefer remote execution. The only context information we use is to check whether the remote resource is reachable. We will incorporate more advanced intelligence in Cuckoo in our future work (see Section 7).

4.2 Communication: Ibis

In order to execute methods on a remote resource, the phone has to communicate with the remote resource. We use the Ibis communication middleware [27] for this purpose, because it offers a simple and effective interface that abstracts from the actual used network, being either WiFi, Cellular or Bluetooth.

The Ibis communication middleware is an open source scientific software package developed for high performance distributed computing in Java. Since it is written in Java, it can also run on Android devices. While targeted at high performance distributed applications that typically run on large supercomputer clusters, the Ibis middleware has proven to be useful on mobile devices too [18,28].

The Ibis middleware consists of two orthogonal subsystems, the Ibis Distributed Deployment System, which deploys applications onto remote resources and the Ibis High-Performance Programming System, which handles the communication between the individual parts of a distributed application.

Cuckoo has been implemented on top of the Ibis High-Performance Programming System, which offers an interface for distributed applications in the form of several high-level programming models and a low-level communication library. The low-level communication library in turn abstracts from the actual implementations of this library. Currently the library has implementations for TCP, UDP, SmartSockets, Bluetooth and Myrinet. The Cuckoo framework is built directly on top of the communication library and offers a programming model that is at the same abstraction level as other existing programming models, such as Remote Method Invocation (RMI).

The communication library offers unidirectional communication channels between so called *ports* and allows for messages to be sent between multiple ports. Send Ports can be connected to Receive Ports in one-to-one, many-to-one, one-to-many and many-to-many mode. Receive Ports receive messages either explicitly or implicit with upcalls. Every port has a unique identifier, which include a program-wide Ibis identifier and the name of the port.

4.3 Client / Server Protocol

In the Cuckoo system we set up Receive Ports at the server to handle requests from clients. The clients can find a server using its Ibis identifier and bind a Send Port to the matching Receive Port at the server to exchange messages with the server.

Normally, the client will request the server to execute a particular method with particular arguments and return the result of the method execution to a Receive Port that the client has set up especially for this method invocation. However, if the service is not available on the server, the client has to install the service onto the server. It does so, by sending a message to the install Receive Port. This message contains the jar file created in the build process (see Section 3.2), all the required external jars, and the package name of the service.

After a successful installation, the client can invoke methods on the remote service. Although the installation of a service introduces the overhead of sending

the jars to the remote server, this occurs only once per service, while many method invocations can be done.

Since the client runs on a mobile phone in a dynamic networking environment, it will sooner or later experience disconnections with resources. These disconnection can be due to network switching or limited network coverage. Cuckoo will react on disconnection by switching to another remote resource. As a final fallback, it can use the always available, but less preferred, local implementation.

A Cuckoo server can be shared with multiple clients. For instance a family home server can be used to enhance the computation of each of the family's mobile device. Since the supported services are stateless, multiple invocations cannot interfer with each other. Users can exchange remote resources by simply presenting the QR-code containing the Ibis identifier on their phone, while another person uses the camera of the phone to scan the QR-code.

4.4 Configuration of Cuckoo: Trade Offs

Now that we have explained how programmers can write remote implementations and how these implementations eventually will be executed on the remote server, we will turn our attention to the different configuration options of Cuckoo. We implemented several configuration points in the run time system of Cuckoo to make it flexible and therefore valuable across a large number of applications with different requirements.

Cuckoo can do both early and late binding to remote resources. When speed is of key importance to an application, early binding will give the best results: Cuckoo will try to find and, if needed, install a remote resource at the moment the activity binds to the service. Then, when service methods are invoked, the resource discovery process has already been done and the remote methods can be invoked right away. In contrast with early binding, using late binding the resource discovery process will be delayed until a method will be invoked. This will avoid energy overhead in the form of unnecessary reconnections, which can occur during the time between the binding of the activity to the service and the moment a method of the service is invoked. During this time the established connection may already be lost, making a reconnection to a new resource necessary. Although early binding optimizes for speed, late binding optimizes for energy usage. A developer can specify per service, whether the service is optimized for speed or for energy usage.

Another property that the developer can configure is the maximum number of remote resources that Cuckoo will try before giving up and use the local implementation. A higher number of resources will give a bigger chance to find one that can be used, but can also introduce more overhead in terms of time and energy if Cuckoo cannot connect to any of the resources. Determining whether a resource can be used involves a simple ping-pong message exchange between the client and the resource. If the cost of multiple ping-pong messages is neglectable compared to the benefit of a potential remote execution, the developer should specify a high maximum number of remote resources that will be tried, to increase the chance of remote execution. Ideally, the developer should be able to

Fig. 5. This figure shows a screenshot of the object recognition application *eyeDentify*. The application can recognize objects in the center of the eye, which shows the camera preview. Users can also teach the name of new objects to the application.

specify a budget in time and/or energy that Cuckoo might spend in searching for a remote resource. We plan to include this in our future work.

To improve responsiveness of an application – that is execution speed – the developer can also specify a method of the service to be executed in parallel on multiple resources, including the local implementation. The result of the first implementation that returns, will then be forwarded to the proxy in the activity. Note that this method is energy hungry, because there always will be some execution of code that will be discarded.

Finally, the developer can configure individual methods of a service to be not offloadable at all, which is useful for instance if methods interact with sensors on the smartphone.

5 Example Applications

In this section we will evaluate the Cuckoo computation offloading framework with two smartphone applications that contain heavy weight computation. We will show which parts of the applications are offloaded and how the computation offloading impacts the performance of the applications in both speed and functionality.

5.1 eyeDentify

Our first example application is *eyeDentify* [18], a multimedia content analysis application that performs object recognition of images captured by the camera of a smartphone (see Figure 5). The idea of the application is similar to the Google Goggles application which can recognize contact info, places, logos, landmarks,

Table 2. eyeDentify parameters. This table shows the parameters used for the local and the remote implementation of the service that converts an image into a feature vector. The quality of the feature vector is higher if the image size is larger, the number of receptive fields in the algorithm is higher, if there are more color models and if there are more bins. See [18] for an explanation of the *-marked terms.

	image size	receptive fields*	color models*	bins*
local	64 x 48	19	1	500
remote	512 x 384	37	3	1000

artworks, and books [29]. In addition to the recognition of objects, eyeDentify allows the user to teach new objects to the application.

The key algorithm in eyeDentify is an algorithm that converts an image into a *feature vector*, a mathematical representation of the image that can be compared to other feature vectors. This algorithm is both compute and memory intensive and therefore suitable for computation offloading. There are several parameters of the algorithm that influence the computation time and the memory footprint. For instance, a larger input image will result in a bigger memory footprint and a larger computation time. Other parameters can be tuned to reduce the memory and computation requirements, while also reducing the quality of the feature vector and thereby the quality of the object recognition.

In [18] we show that it is possible to perform object recognition on a T-Mobile G-1 smartphone, with a 528 MHz processor and 16 MB available application memory within a reasonable time, however, the algorithm used can take at most 128 x 96 pixel images as input, with the algorithm parameters set to poor quality object recognition. By offloading the computation needed for this algorithm, we have shown that we can speed up the computation with a factor of 60, reduce the battery consumption with a factor of 40 and increase the quality of the recognition [18].

We have rewritten eyeDentify to use the Cuckoo computation offloading framework. First, we specified an interface in AIDL for a service that hosts the algorithm that converts an image into a feature vector (see Figure 6). Then we implemented the local service with parameters that are suitable for local computation (see Table 2, local). Cuckoo generated a dummy remote implementation

```
package interdroid.eyedentify;

import ibis.dog.shared.FeatureVector;

interface FeatureVectorServiceInterface {
  FeatureVector getFeatureVector(in byte[] jpegData);
}
```

Fig. 6. The AIDL interface definition of the service that converts an image into a feature vector. The eyeDentify application has a local and a remote implementation of this interface, where the only differences between the implementation are the value of accuracy parameters of the algorithm.

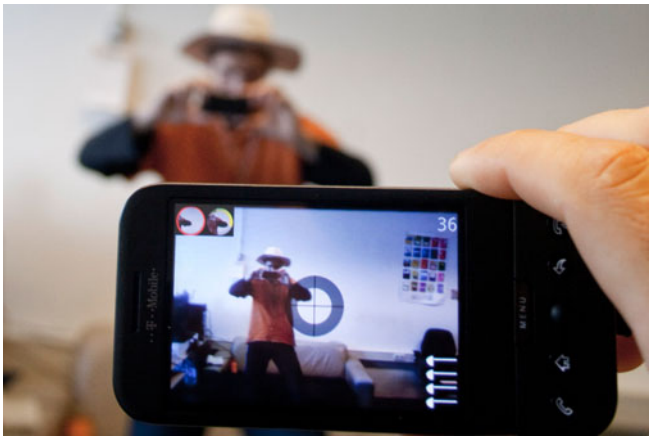

Fig. 7. A screenshot of PhotoShoot - The Duel, a distributed augmented reality smartphone game in which two players fight a virtual duel in the real world. Both players have 60 seconds and six bullets to shoot at each other. They shoot with virtual bullets, photographs of the smartphones' camera, and use face detection to evaluate whether shots are hit. The area covered in the crosshairs will be used for face detection.

and we replaced the dummy implementation simply with the same implementation as for the local one and then changed the algorithm parameters (see Table 2, remote), so that the remote implementation would perform high quality object recognition on large images. Cuckoo then generated the jar file for the remote service and the application was ready to be deployed on a smartphone.

At runtime this application can always perform object recognition, even when no network connection is available, in contrast to, for instance, Google Goggles that only works when connected to the cloud. However, if a network connection is available, then eyeDentify can use remote resources to speed up the recognition and reduce the energy consumption, while providing higher quality object recognition. Another advantage is that the client and server code of eyeDentify are bundled and will remain compatible with each other.

Compared to our earlier work on computation offloading, using Cuckoo results in similar performance improvements.

5.2 PhotoShoot

The second example that we will consider is a distributed augmented reality game, called *PhotoShoot* [28], with which we participated in the second worldwide Android Developers Challenge [30] and finished at the 6^{th} place in the category 'Games: Arcade & Action'. This innovative game is a two-player duel that takes place in the real world. Players have 60 seconds and 6 virtual bullets to shoot at each other with the camera on their smartphone (see Figure 7). Face detection will determine whether a shot is a hit or not. The first player that hits the other player will win the duel.

The major compute intensive operation in this game is the face detection. The Android framework comes with a face detection algorithm, so it is possible to create a local implementation to detect faces in an image. However, due to memory limitations this algorithm works only for images up to 3.2 MegaPixels – which is rather low for a modern smartphone – and consumes about 9 seconds of compute time on the Google Nexus One with a 1.0 GHz processor or 44 seconds on the older T-Mobile G-1 with a 528 MHz processor. Thus, without offloading, the slower the processor of the smartphone, the longer it takes for the shot to be analyzed, which gives the user of a slow smartphone a significant disadvantage. Offloading can, however, be used to make the game fair again.

We have modified PhotoShoot, by refactoring the face detection into a service (see Figure 8), so that it can use computation offloading using the Cuckoo framework. Using offloading, slower phones can get fast face detection and phones with very high resolution cameras will not experience memory shortage.

We used a different face detection algorithm for the remote implementation to demonstrate that offloading does not only result in faster execution, but gives also more precise results. The remote implementation is based on the Open Source Computer Vision library (OpenCV [32]) and can, next to frontal faces, also detect profile faces, and will therefore give more accurate results (see Figure 9).

A face *recognition* algorithm would further improve the application so that only pictures containing the face of the opponent would count as valid hits. This would open up possibilities for team-based augmented reality multiplayer games. But to our knowledge, existing face recognition algorithm libraries for Java are not available in the public domain.

6 Related Work

Computation offloading is a technique that dates back to the era of dumb clients that used mainframes for computation. With the introduction of *personal computers*, the need for computation offloading decreased, but with the introduction of todays' portable devices, a new need for remote compute power emerged.

```
package interdroid.photoshoot.market;

import interdroid.photoshoot.market.Face;

interface FaceDetectorInterface {
  List<Face> findFaces(in byte[] jpegData, int maxFaces);
}
```

Fig. 8. The interface of the face detection service in AIDL. The services will search for a maximum number of faces in the provided image and returns a list of Face objects that it has found. The local implementation will use the face detection library available on Android, which can only detect frontal faces, while the remote implementation uses the OpenCV library, which can also detect profile faces.

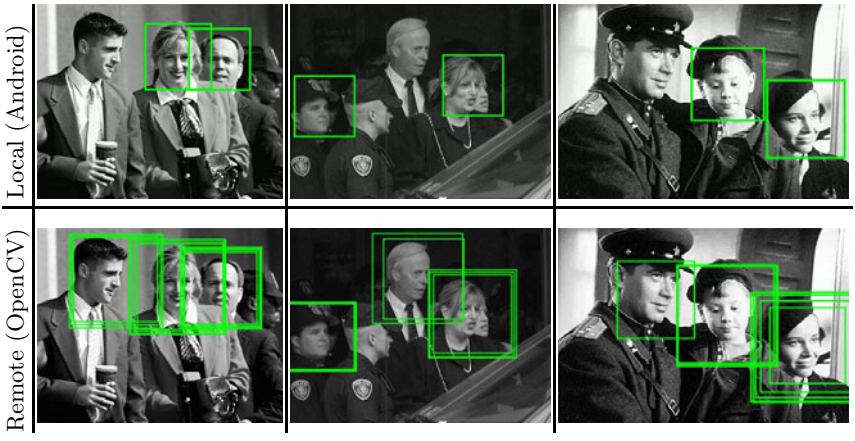

Fig. 9. Comparison of the local and remote implementation of the face detection service on several test images from [31]. The local version (top row) uses the face detection algorithm provided by the Android framework, while the remote version (bottom row) is able to use a much more powerful algorithm, which for instance can also detect profile images. Since the remote implementation runs multiple detectors over the image, some faces are detected several times.

In this section we give an overview of what has been proposed by others with respect to computation offloading for smartphones and how that relates to the Cuckoo framework.

In the early days of the portable handheld devices and before the popularity of the commercial cloud, Satyanarayanan [33] proposed a computation offloading model called *cyber foraging*, in which portable resource constrained devices can offload computation to more powerful remote resources called *surrogates*. Important in cyber foraging are the discovery of surrogates, the trust relation between the client and the surrogates, load balancing on the surrogates, scalability and seamlessness.

Cuckoo provides resource discovery through QR-codes as explained in Section 3.4 and addresses scalability by allowing for both commercial and private cloud resources. In our future work we will investigate the security aspects of computation offloading and load balancing on the surrogates.

Yang et al [34] describe an offloading system based on cyber foraging that, like Cuckoo, uses a Java stub/proxy model. Using profiling, surrogate matching and graph partitioning algorithms their system decides what part of the computation can be offloaded. They provide an evaluation of the system using an automatic translation application that uses OCR and Machine Translation. The system runs on the HP iPaq platform.

From the field of security Chun et al [17] propose an architecture, called *CloneCloud*, that can offload computation, such as virus scans and taint checking to a clone of the smartphone in the cloud. An implementation of their proposed architecture should support the following types of augmented execution:

- primary functionality outsourcing: offloading of expensive computation
- background augmentation: offloading of background processes
- mainline augmentation: offloading light weight computation, for heavy weight analysis (taint checking, debugging, profiling).
- hardware augmentation: offloading of computation because of hardware limitations of the smartphone
- augmentation through multiplicity: parallel execution of offloaded computation, for computation speedup or speculative execution.

The Cuckoo framework implements all the proposed execution types in its single programming model. It does not run a complete clone of the smartphone at the remote cloud resource, as proposed by Chun et al, but rather runs a temporary clone limited to only the service that the application is using, thereby avoiding the costly process of keeping the smartphone synchronized with an application clone in the cloud.

Another offloading model, called AlfredO, which is based on the modular software framework OSGi, is proposed in [35]. They contribute partitioning algorithms to optimize the distribution of OSGi software modules between the phone and remote resources and evaluate their system with a home interior design application. Since AlfredO will use the same modules on the phone as on the remote resource, it is not possible to differentiate between local and remote implementations, as is possible with Cuckoo.

In [16], Kumar et al describe a cost/benefit model for offloading with respect to energy usage. This model takes into account the speed of both the smartphone and the remote cloud resource, the number of bytes that need to be transfered, the network bandwidth, the energy consumption of the smartphone in idle, computing and communicating state. They assume that the number of local instructions is identical to the number of remote instructions and the number of bytes that need to be communicated are known beforehand. The assumptions do not completely hold for Cuckoo, since the remote implementation might be different from the local implementation and since the number of bytes that will be received by the smartphone after a remote method execution can be unknown beforehand. However, we plan to extend the model and incorporate it into Cuckoo.

Next to computation offloading systems, where computation components are transfered from a smartphone to the remote resource, there exist many applications that are separated into a light weight client and a heavy weight server hosted in the cloud. Examples of such applications are the music search service Shazam [36] and image search service Goggles [29]. The remote parts of these services typically have to interact with very large databases and therefore are not suitable to be bundled with the client application. The drawback of unbundled services is that the application provider has to provide the remote service, which will include hosting and maintanance costs. Futhermore, since these services are typically commercial, users might not want to hand over input data, because of privacy concerns. With an offloading system, like Cuckoo, users will run their

heavy weight computations on their private machines, or machines that they trust.

In case studies about computation offloading we found that the applications that benefit from computation offloading exist in the following domains:

- **image processing**: object recognition [18], OCR [34], face detection [28], barcode analysis [37]
- **audio processing**: speech recognition [38]
- **text processing**: machine translation [34]
- **artificial intelligence for games**: chess [16]
- **3D rendering**: 3D home interior design [35]
- **security**: taint analysis and virus scans [17,39]

The Cuckoo framework supports the development of applications from these domains in a simple and developer friendly environment.

Several other solutions, complementery to computation offloading, have been proposed at different abstraction levels to reduce the pressure on the energy usage. At the hardware level, manufacturers build processors that can switch to lower frequencies to save energy. Techniques to harvest energy from surrounding sources, such as movement, light and wireless signals, have been proposed to charge the battery during operation [40]. Furthermore, some modern smartphone operating systems have been tuned to be more energy efficient. Finally, the users of smartphones have developed habits to turn off sensors or radios when they are not needed.

7 Future Work

In this section we present our research agenda, we describe what parts of Cuckoo will be enhanced and which parts we feel that are missing in Cuckoo.

In our future work we will improve the intelligence of the offloading by improving the heuristics and the addition of more context information, such that the estimation whether or not offloading is going to save energy or increase the computation speed will be more accurate. Although computation offloading can speed up computation and save energy, it is not guaranteed that it does. Offloading introduces additional communication, which consumes energy and takes time. To take this overhead into account we plan to incorporate the energy analysis model for computation offloading described by [16].

We also want to add context information of the remote resources to the system, such as processor speed, number of processors, available memory, etc., which could be added to the address encoded in the QR-code. In addition we want to use context information available on the phone, such as location, network status, etc.

Another direction of our future work is to investigate which security measures need to be taken to secure the communication between the smartphone and the remote cloud resources. We also have to pay attention to the security implications of multiple users using a single remote resource, running foreign code on

the remote resources, and making sure that remote services cannot disturb the working of other remote services.

Furthermore, we will extend the programming model to support callbacks from the remote resource to the smartphone and investigate whether we can support method parameters to be used as return values, like the AIDL specification supports.

We will improve the Eclipse integration, by integrating the builders with the graphical user interface of Eclipse.

8 Conclusions

In this paper we have presented Cuckoo, a framework for computation offloading for smartphones, a recently rediscovered technique, which can be used to reduce the energy consumption on smartphones and increase the speed of compute intensive operations.

Cuckoo integrates with the popular open source Android framework and the Eclipse development tool. It provides a simple programming model, familiar to developers, that allows for a single interface with a local and a remote implementation. Cuckoo will decide at runtime where the computation will take place. Furthermore, the Cuckoo framework comes with a generic remote server, which can host the remote implementations of compute intensive services. A smartphone application to collect the addresses of the remote servers is also included.

We have evaluated the Cuckoo framework with two real world smartphone applications, an object recognition application and a distributed augmented reality smartphone game and showed that little work was required to enable computation offloading for these applications using the Cuckoo framework.

References

1. Android, http://developer.android.com/
2. iPhone OS, http://developer.apple.com/iphone/
3. Symbian, http://developer.symbian.org/
4. Windows Phone 7 Series, http://www.windowsphone7.com/
5. iPhone App Store, http://www.apple.com/iphone/appstore
6. Android Market, http://www.android.com/market/
7. Raging Thunder, http://www.polarbit.com/our-games/raging-thunder-2/
8. Motorola MOTOBLUR, http://www.motorola.com/Consumers/US-EN/Consumer-Product-and-Services/MOTOBLUR/Meet-MOTOBLUR
9. Google Maps Navigation, http://www.google.com/mobile/navigation/
10. Health to Go, http://www.healthymagination.com/
11. Data gathered from: http://www.gsmarena.com
12. iPhone 3G on Sale Tomorrow, http://www.apple.com/pr/library/2008/07/10iphone.html
13. Steve Jobs. Thoughts on Flash, http://www.apple.com/hotnews/thoughts-on-flash/

14. IEEE Standard for Information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 5: Enhancements for Higher Throughput. IEEE Std 802.11n-2009 (Amendment to IEEE Std 802.11-2007 as amended by IEEE Std 802.11k-2008, IEEE Std 802.11r-2008, IEEE Std 802.11y-2008, and IEEE Std 802.11w-2009), 29 (2009)
15. Bluetooth High Speed. Product Zone, Bluetooth High Speed Technology, http://bluetooth.com/
16. Kumar, K., Lu, Y.-H.: Cloud computing for mobile users. Computer 99 (2010)
17. Chun, B.-G., Maniatis, P.: Augmented smart phone applications through clone cloud execution. In: Proceedings of the 12th Workshop on Hot Topics in Operating Systems, HotOS XII (2009)
18. Kemp, R., Palmer, N., Kielmann, T., Seinstra, F., Drost, N., Maassen, J., Bal, H.E.: eyeDentify: Multimedia Cyber Foraging from a Smartphone. In: IEEE International Symposium on Multimedia (2009)
19. AIDL, http://developer.android.com/guide/developing/tools/aidl.html
20. Eclipse, http://www.eclipse.org/
21. ADT Eclipse plugin, http://developer.android.com/sdk/eclipse-adt.html
22. AIDL, http://developer.android.com/guide/topics/fundamentals.html
23. Apach Ant, http://ant.apache.org/
24. Amazon Elastic Computing, http://aws.amazon.com/ec2/
25. Denso Wave's QR website, http://www.denso-wave.com/qrcode/index-e.html
26. Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. ACM Transactions on Internet Technology 2(2), 115–150 (2002)
27. van Nieuwpoort, R.V., Maassen, J., Wrzesińska, G., Hofman, R.F.H., Jacobs, C.J.H., Kielmann, T., Bal, H.E.: Ibis: a flexible and efficient Java-based Grid programming environment. Concurrency and Computation: Practice and Experience 17(7-8), 1079–1107 (2005)
28. Kemp, R., Palmer, N., Kielmann, T., Bal, H.: Opportunistic Communication for Multiplayer Mobile Gaming: Lessons Learned from PhotoShoot. In: MobiOpp 2010: Proceedings of the Second International Workshop on Mobile Opportunistic Networking, pp. 182–184. ACM (2010)
29. Google Goggles, http://www.google.com/mobile/goggles/
30. Android Developer Challenge 2, http://code.google.com/android/adc
31. Schneiderman, H., Kanade, T.: A Statistical Model for 3D Object Detection Applied to Faces and Cars. In: IEEE Conference on Computer Vision and Pattern Recognition. IEEE (June 2000)
32. Open Source Computer Vision library, http://opencv.willowgarage.com/wiki/
33. Satyanarayanan, M.: Pervasive computing: Vision and challenges. IEEE Personal Communications 8, 10–17 (2001)
34. Yang, K., Ou, S., Chen, H.H.: On Effective Offloading Services for Resource-Constrained Mobile Devices Running Heavier Mobile Internet Applications. IEEE Communications 46, 56–63 (2008)
35. Giurgiu, I., Riva, O., Juric, D., Krivulev, I., Alonso, G.: Calling the Cloud: Enabling Mobile Phones as Interfaces to Cloud Applications. In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 83–102. Springer, Heidelberg (2009)
36. Shazam website, http://www.shazam.com
37. Kallonen, T., Porras, J.: Use of distributed resources in mobile environment. In: International Conference on Software in Telecommunications and Computer Networks, pp. 281–285 (2006)

38. Goyal, S., Carter, J.: A lightweight secure cyber foraging infrastructure for resource-constrained devices. In: WMCSA 2004: Proceedings of the Sixth IEEE Workshop on Mobile Computing Systems and Applications, pp. 186–195. IEEE Computer Society (2004)
39. Portokalidis, G.: Using Virtualisation to Protect Against Zero-Day Attacks. PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands (February 2010)
40. Paradiso, J.A., Starner, T.: Energy scavenging for mobile and wireless electronics. IEEE Pervasive Computing 4, 18–27 (2005)