2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering

Uniport: A Uniform Programming Support Framework for Mobile Cloud Computing

Pengfei Yuan, Yao Guo, Xiangqun Chen
Key Laboratory of High-Confidence Software Technologies (Ministry of Education)
School of Electronics Engineering and Computer Science, Peking University
{yuanpf12, yaoguo, cherry}@sei.pku.edu.cn

***Abstract*—Personal mobile devices (PMDs) have become the most used computing devices for many people. With the introduction of mobile cloud computing, we can augment the storage and computing capabilities of PMDs via cloud support. However, there are many challenges in developing mobile cloud applications (MCAs) that incorporate cloud computing efficiently, especially for developers targeting multiple mobile platforms.**

This paper presents Uniport, a uniform framework for developing MCAs. We introduce a uniform architecture for MCAs based on the Model-View-Controller (MVC) pattern and a set of programming primitives and runtime libraries. Not only can Uniport support the creation of new MCAs, it can also help transform existing mobile applications to MCAs efficiently. We demonstrate the applicability and flexibility of Uniport in a case study to transform three existing mobile applications on iOS, Android and Windows Phone, to their mobile cloud versions respectively. Evaluation results show that, with very few modifications, we can easily transform mobile applications to MCAs that can exploit the cloud support to improve performance by 3–7x and save more than half of their energy consumption.

I. Introduction

Personal mobile devices (PMDs) including smartphones and tablet computers are continuously gaining popularity worldwide. Thanks to the sophisticated operating systems, middleware and programming languages equipped in mobile platforms, we have abundant mobile applications to choose from. Although we are getting richer and richer experience from mobile applications on high-end PMDs, their storage and computing abilities are still one of the most important limiting factors, especially when energy consumption and limited battery capacity are taken into consideration.

Recent years have seen the trend of incorporating cloud computing support to mobile applications. Siri, Instagram, Snapchat and Google Now are just a subset of products on mobile platforms that benefit from cloud computing and enrich user experience of PMDs.

Compared with the cloud, both computing capability and storage capacity of PMDs are extremely limited because their major design goal is energy efficiency. PMDs will be greatly enhanced if we equip them with cloud computing support. Expanded storage capacity and augmented computing capability are only part of the advantages we gain; PMDs may also save energy via offloading computation to the cloud [1].

Many solutions have been proposed to explore the possibilities of uniting mobile platforms and cloud computing services, such as COMET [2], CloneCloud [3], Cuckoo [4] and Paranoid Android [5] for Android, and MAUI [6] for Windows Mobile. Most of these solutions are attempting to offload or migrate some computation-intensive tasks to the cloud, in order to improve performance and reduce energy consumption on PMDs. To the best of our knowledge, all the existing work on mobile cloud computing are platform-dependent, providing mostly ad-hoc solutions at the system level.

With the rapid evolution of both PMDs and cloud computing technologies [7, 8], we can see a trend of mobile cloud computing in the near future. So it is very important to provide solutions to the development of mobile cloud applications (MCAs), which can incorporate cloud computing and storage services efficiently into normal mobile applications, especially for developers targeting multiple mobile platforms, such as the most popular iOS, Android and Windows Phone platforms.

Although development support for mobile applications are well-studied and many commercial and open-source tools are available, they can not be directly applied to MCAs. Most popular mobile applications support multiple mobile platforms, and a few cross-platform solutions to mobile application development have been proposed, such as Mobl [9] and PhoneGap [10]. However, these solutions do not support mobile cloud computing.

Developing MCAs is hard because it is difficult to identify the computation-intensive tasks that can be offloaded to execute in the cloud. For example, most device-dependent tasks related to user interactions and display can only be executed on the devices. Moreover, developing MCAs requires the knowledge of many low-level details such as data serialization, network communication and cloud computing service invocation, which becomes even harder when providing cross platform support for MCAs.

In order to overcome the above challenges, we present Uniport, a uniform programming support framework for mobile cloud computing, which includes the following key components:

- *A uniform architecture for cross-platform MCAs*, which is derived from the *Model-View-Controller* (MVC) pattern that can be applied to a wide range of applications on multiple mobile platforms.
- *A set of mobile cloud computing primitives*, which provide programming interfaces for mobile cloud computing. These primitives are platform and programming

978-1-4799-8977-5/15 $31.00 © 2015 IEEE
DOI 10.1109/MobileCloud.2015.10

language-independent, such that developers are able to design and implement MCAs for multiple platforms in a uniform way.
- *A set of runtime libraries* implementing the mobile cloud computing primitives, which provide the runtime supporting framework for MCA execution. We have implemented the runtime libraries on three popular mobile platforms including iOS, Android and Windows Phone.
- *A set of development tools* including a code generator, which can be used to generate code skeletons for different mobile platforms, and a static analyzer that analyzes existing mobile applications and checks them against the constraints introduced in the Uniport architecture. The purpose of these supporting tools is to help improve productivity.

In order to demonstrate the applicability and flexibility of Uniport, we perform a case study to transform three existing mobile applications on iOS, Android and Windows Phone, to their mobile cloud versions respectively. Only a very small fraction of lines of code (ranging from 3–9%) are modified (added/removed) to make them take advantage of cloud computing. Evaluation results show that the transformed MCAs reduce both execution time and energy consumption significantly.

The rest of this paper is organized as follows: Section II describes the basic principles of our framework. Section III discusses how our framework provides developers with programming support and Section IV describes details about implementation. Section V presents how we apply the Uniport framework to three existing mobile applications and shows the evaluation results. Section VI discusses the applicability of our framework and the evaluation results. Section VII compares the Uniport framework with related work and Section VIII concludes this paper and its future work.

II. Overview of the Uniport Framework

The main objective of the Uniport framework is to provide uniform programming support for MCAs on different mobile platforms. In order to achieve this goal, the framework introduces a platform-independent MCA architecture derived from the MVC pattern and a few constraints to the architecture to make it suitable for simplified development of MCAs on different platforms and in different programming languages.

A. Mobile Cloud Application

An MCA is a mobile application which incorporates cloud computing and cloud storage to augment the capabilities of PMDs. Generally, an MCA can be divided into two major parts: one running on PMDs; the other running in the cloud. The users of an MCA are not aware of the separation of the two parts because they collaborate together to execute as a single mobile application, with the capabilities of PMDs enhanced and usage scenarios expanded transparently.

In reality, many mobile applications adopt the MVC pattern or its derivatives, such as Model-View-Adapter, Model-View-Presenter, Model-View-ViewModel. The major goal of the MVC pattern is to simplify the application architecture by decoupling models and views, and to make source code more flexible and maintainable.

MVC splits the interactions between the user and the application into three roles: the model, the view, and the controller. In MVC, the view represents the user interface which consists of contents and styles applied to the contents. The model describes the core of an application that uses business logic to manipulate data. The controller is an event handler and a bridge which connects the view and the model, passes input from the view to the model and updates the view from the execution results of the model.

Following the MVC pattern can greatly benefit the development of MCAs, as the model part is a natural fit for cloud execution, while the view and the controller should run on PMDs.

To simplify MCA development, we propose an architecture in which the model running on the device and the model running in the cloud are the same. In practice, this can be achieved via sharing the same code base for the model part in the cloud and on the device, thus simplifying the development procedure of MCAs.

To achieve the goal of simplicity, we introduce a couple of constraints to MCAs. Based on our observation, the middleware on mobile platforms and their counterparts on corresponding desktop platforms share a lot of constants, data structures and APIs in common and they use the same programming languages. Table I shows the comparison of middleware on different mobile platforms and their corresponding desktop platforms. This characteristic enables us to achieve the simplicity of the MCA architecture by enforcing the following constraints:

- The model part of an MCA should not use the noncommon features in the middleware, either directly or indirectly. This constraint is reasonable because the majority of those noncommon features are either device-dependent or used for user interfaces and interactions.
- The model part of an MCA should not use static or global variables directly because they are considered as part of the local storage data source and need to be synchronized via serialization mechanisms. To use static or global variables indirectly, we can copy them to non-static member variables before entering the model part.

B. The Uniport Architecture

Figure 1 presents the general architecture of MCAs based on the Uniport framework, which is divided into two parts: the device (client) and the cloud (server).

The MCAs based on the Uniport framework adopt an architecture similar to the original MVC. They still include views (that consist of contents and styles) and controllers (as event handlers), which are exactly what most applications on different mobile platforms have. What is different from the original MVC pattern is the model part. In the Uniport architecture, the model part of an MCA is now encapsulated into the Uniport client to leverage cloud computing services.

TABLE I: Comparison of three mobile platforms

Mobile Platform	*Middleware*	*Desktop Counterpart*	*Language Used*
Android	Java-compatible	Java Runtime	Java
iOS	Cocoa Touch	Cocoa	Objective-C
Windows Phone	Silverlight	.NET Framework	C#/VB

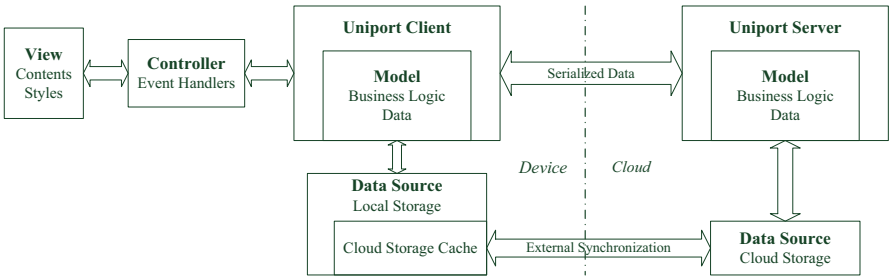


Fig. 1: Overview of the Uniport architecture

In the Uniport architecture, the controller passes user input from the view to the Uniport client and the client handles computation offloading of the actual MCA model. The controller receives execution results from the Uniport client, the process of which is event-driven, and updates the view asynchronously.

In the cloud, the server application does not contain any views because it does not need any user interfaces. The role of the controller is partly acted by the Uniport server which encapsulates the actual MCA model. The Uniport client passes the user input it receives from the controller to the Uniport server via network communication. The Uniport server executes the actual MCA model in the cloud and sends execution results back to the Uniport client in return, which will then be passed to the controller for updating the view. This is how cloud computing services are leveraged in MCAs based on the Uniport framework.

III. Programming Support in Uniport

In order to simplify the development procedure of MCAs, the Uniport framework provides programming support including programming primitives, data source and runtime policies. We discuss the details of these features in this section.

A. Programming Primitives

In the Uniport framework, we provide programming primitives to implement the communications between the different parts in the Uniport architecture (Figure 1). These primitives also provide cloud computing service invocation support for MCAs.

When developing an MCA, developers need to consider three kinds of communications: the communication between the controller and the Uniport client, the communication between the Uniport client and the actual MCA model, and the communication between the Uniport server and the actual MCA model.

To handle these communications and support cloud computing service invocations, we provide a small set of programming primitives. Table II shows the primitives we adopt and their semantics. These primitives are programming language and platform-independent, suitable for cross-platform MCA development.

TABLE II: Primitives in the Uniport framework

`client execute`	The controller invokes the client to run the model
`execution run`	The model applies its business logic to the data
`execution will begin`	The client notifies the controller before the model starts running
`execution did end`	The client notifies the controller after the model finishes running
`restore session`	The model restores session data
`backup session`	The model backs up session data
`prepare sync`	The model prepares for a full synchronization
`finish sync`	The model finishes synchronization

To demonstrate how the primitives `client execute`, `execution run`, `execution will begin` and `execution did end` are used to support mobile cloud computing, let's take image processing as an example. The user browses photos stored in Google Drive, iCloud or OneDrive and chooses one for face recognition. The controller receives user input from the view, analyzes it and invokes `client execute` primitive with appropriate parameters. Since data is provided by the cloud storage data source which will be explained later, the Uniport client will only serialize necessary parameters for the actual MCA model. Before serialization and data transmission, the Uniport client invokes `execution will begin` primitive asynchronously. The Uniport server receives the serialized data, deserializes it, invokes `execution run` primitive for executing the actual MCA model and returns execution results. When the actual MCA model runs in the cloud, it fetches the chosen photo from Google Drive, iCloud or OneDrive. After the Uniport client receives the results, it invokes `execution did end` primitive asynchronously, notifying the controller to update the view to show face recognition results.

The usage of primitives `restore session`, `backup session`, `prepare sync` and `finish sync` will be discussed later.

B. Data Source

At the bottom of the Uniport architecture shown in Figure 1, *data source* provides the actual MCA model with data to process. In the Uniport architecture, we also divide the data source into two parts: one is the local storage data source, the other is the cloud storage data source. The two kinds of data sources operate together for full leverage of cloud computing and storage services without too much alteration to the existing design.

The *local storage data source* consists of data stored in files, databases that only exist in the local storage of the device, as well as static or global variables mentioned in the previous section. The *cloud storage data source* consists of data stored in the cloud storage services and cached locally. The actual MCA model may use both data provided by the local storage data source and data provided by the cloud storage data source for processing.

To leverage cloud computing and storage services in the Uniport architecture, we need to synchronize the data sources used by the actual MCA model on the device and the data sources used in the cloud. Data from the local storage data source is synchronized to the cloud via serialization, a mechanism provided by C#, Java, Objective-C and many other programming languages. The serialization and synchronization procedure is implemented in the Uniport client and transparent to MCAs based on the framework.

The cloud storage data source is synchronized via external mechanisms such as network file system, Google Drive, iCloud, OneDrive.

C. Runtime Policy

In addition to the communications and data sources discussed previously, we design some runtime policies for better support of mobile cloud computing. The key concerns of the policies are availability, security, performance and energy consumption of the MCAs based on our framework.

1) Availability: Many adversities may impact the smooth execution of an MCA. For example, there are circumstances where network is unavailable, network connection is slow and unstable, or even the cloud service is unavailable. These are availability issues.

To deal with the unavailabilities and guarantee the availability of the MCAs based on the Uniport framework, we design some runtime policies for the Uniport runtime libraries on different mobile platforms.

The network availability is ensured by socket creation and connection. If network is unavailable for MCAs, socket creation and connection will fail. The speed and stability of network connection is ensured by socket communication timeout and the serialization mechanisms. The availability of the cloud service is ensured by socket connection, communication timeout and the serialization mechanisms.

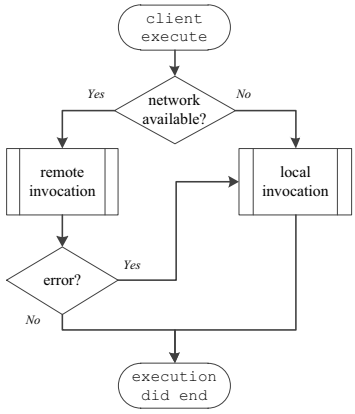


Fig. 2: Execution flow

When any part is unavailable, the Uniport client will invoke `execution run` primitive for executing the actual MCA model locally on the device to guarantee the availability of the MCA. This policy is under the hood and its detailed mechanisms are transparent to the controller and the actual MCA model.

With availability taken into consideration, the execution flow of MCAs based on the Uniport framework is shown in Figure 2.

2) Security: Security is another key concern, including the security of the device, the security of the cloud and the security and privacy of the data exchanged between them.

To protect the security of the device, the Uniport client catches exceptions when receiving the results. To protect the security of the cloud, the Uniport server only accepts invocations of known MCA models from authorized clients and catches exceptions during the execution of the MCA model.

To protect the security and privacy of the data exchanged between the device and the cloud, Secure Socket Layer/Transport Layer Security (SSL/TLS) is used for encryption. SSL/TLS also provides mechanisms for authentication and authorization [11]. Authentication and authorization can also be combined with access control mechanisms.

3) Performance and Energy Consumption: The default runtime policy of the Uniport framework shown in Figure 2 does not guarantee the shortest response time, but is rather a trade-off between performance and energy consumption.

The Uniport client first tries invoking the `execution run` primitive remotely in the cloud. It will only invoke `execution run` locally if remote invocation fails. The response time is $T_r + T_d$ if remote invocation succeeds, where T_r is the time of remote execution and T_d is the time of data transmission. The response time is $T_t + T_l$ if remote invocation fails, where T_t is the default timeout and T_l is the time of local execution.

To achieve the shortest response time, remote invocation and local invocation can be performed simultaneously, which

will of course result in more energy consumption. With this policy, the response time is $min(T_l, min(T_r + T_d, T_t))$. It is no more than the response time with the default runtime policy and will be shorter if any forms of unavailabilities occur.

The choice between the two runtime policies can be configured statically by the MCA developer. It can also be determined by the Uniport runtime library dynamically, according to the real-time CPU utilization, battery level and network connection quality.

Energy consumption is another important concern for MCAs. To calculate the amount of energy saved via offloading, we use the following formula proposed by Kumar and Lu [1].

$$\frac{C}{M} \times \left(P_c - \frac{P_i}{F} \right) - P_{tr} \times \frac{D}{B}$$

In the formula, C is the amount of computation. M is the CPU speed of the device and the cloud is F times faster than the device. P_i, P_c and P_{tr} are power consumed by the device in idle state, for computing and for network communication. D is the bytes of data exchanged between the device and the cloud. B is the network bandwidth. In order to reduce energy consumption as much as possible, we need to reduce D as much as possible. There are two kinds of cases to consider for the reduction of D.

In one kind of cases, like the image processing example discussed previously, different invocations to the actual MCA model are independent of each other, which implies that the model does its work independently among different invocations. In this kind of cases, we can take advantage of the cloud storage data source to reduce the amount of exchanged data. Redundant data can also be removed by the Uniport server while the cloud is returning results to the device.

In another kind of cases, a sequence of invocations to the actual MCA model is doing one job. A simple practice is to exchange the data representing the state every time. But it is not energy efficient. To reduce the amount of exchanged data, we can use a strategy in which only the data representing the differences between states are exchanged.

In the optimized strategy, the device and the cloud need to be synchronized in the state of MCA models. This is why we introduce primitives `restore session`, `backup session`, `prepare sync` and `finish sync` in the Uniport framework.

Before the Uniport client tries invoking the `execution run` primitive remotely in the cloud, it checks whether the device and the cloud are synchronized in state. If not, it invokes `prepare sync` to notify the MCA model for preparing synchronization. After receiving results from the cloud, it invokes `finish sync` to notify the MCA model for cleaning up synchronization data. In the cloud, each client has a corresponding session storing the current state of MCA models. Before invoking `execution run`, the Uniport server invokes `restore session` to notify the MCA model for recovering the state from the session. After invoking `execution run`, it invokes `backup session` to notify the MCA model for backing up its state to the session.

IV. Implementation

In this section, we present the implementation details of the Uniport framework on different mobile platforms, how different programming languages are supported, and how the design goals of the Uniport framework can be achieved.

A. Platform and Language Characteristics

We implement the Uniport framework on three most popular mobile platforms: iOS, Android and Windows Phone, with Objective-C, Java and C# respectively. Fortunatedly, the three mobile platforms and the three programming languages share the following common characteristics, which enable us to achieve the goal of simplicity and uniformity:

- The iOS platform runs upon the XNU kernel, which is the same as the kernel of Mac OS X. Applications in iOS are built upon a middleware layer named Cocoa Touch, which corresponds to Cocoa in Mac OS X. Cocoa Touch includes Objective-C frameworks such as `Foundation` and `Core Foundation` that are platform independent and subsets of their counterparts in Cocoa.
- Android runs a Linux kernel slightly different from the stock Linux kernel [12]. Applications in Android are built upon a Java-compatible middleware layer, which is based on Apache Harmony [13] and provides many libraries in common with the standard Java runtime.
- Windows Phone runs the Windows CE/NT kernel. Applications in Windows Phone are built upon either Silverlight or the XNA framework with C# or Visual Basic. The latter is mainly used in video games while the former is used in normal applications and is a subset of the .NET framework.

The similarities between these mobile platforms and their corresponding desktop platforms are the key to our effort to achieve simplification in the Uniport framework. Because of the common features, most of the model parts in MCAs can run both on the device and in the cloud without any modification, which simplifies the development process of MCAs.

B. Data Source and Serialization

As discussed in the previous section, the cloud storage data source relies on external implementations. Popular mobile platforms such as iOS, Android and Windows Phone have built-in cloud storage support like iCloud, Google Drive and OneDrive respectively. For Android, there are also network file system implementations [14, 15].

For user input and data from the local storage data source, the Uniport client serializes them before sending them to the cloud side. The Uniport server also serializes the results before sending them back to the device. The serialization mechanism is supported in Objective-C, Java and C# via their corresponding middlewares.

On iOS, `NSKeyedArchiver` and `NSKeyedUnarchiver` from the `Foundation` framework are used for serialization and deserialization. To make the serialization mechanism on iOS work, the MCA model must implement `NSCoding` protocol.

On Android, `ObjectOutputStream` and `ObjectInputStream` from Java language infrastructure are used for serialization and deserialization. To make the serialization mechanism on Android work, the MCA model must implement either `Serializable` or `Externalizable` interface. In the Uniport framework, we choose the latter because it allows serializing data members selectively.

On Windows Phone, we use `DataContractJsonSerializer` for serialization and deserialization because `BinaryFormatter` from the .NET framework is not available in Silverlight. To make the serialization mechanism on Windows Phone work, the MCA model must mark itself with `DataContract` attribute and mark its fields that need to be serialized with `DataMember` attribute.

If anything fails during serialization, deserialization or invocation to the `execution run` primitive, an exception will be caught by the Uniport framework. The runtime libraries of the framework will then use the policies described in the previous section to guarantee the availability of the MCA.

C. Execution and Error Handling

On mobile platforms, it is typically unacceptable to block the main/UI thread [16], which will make the whole application unresponsive, resulting in bad user experience. The Uniport client uses background threads to avoid blocking the main/UI thread. After the controller invokes the `client execute` primitive, the Uniport client starts running in a background thread. However, the Uniport client should invoke primitives `execution will begin` and `execution did end` in the main/UI thread to avoid concurrency faults. The cross thread invocations are implemented with Grand Central Dispatch on iOS, `android.os.Handler` on Android and `System.Windows.Threading.Dispatcher` on Windows Phone.

While the invocation to `execution run` is in progress, the main/UI thread can still access the local storage data source. Therefore developers should take care and avoid concurrency bugs.

In the Uniport framework, we use the middleware and programming language-supported exception handling facilities to deal with errors. We place code that might generate errors in `try`-blocks and handle errors with runtime policies in the corresponding `catch`-blocks. In some cases, such as the invocation to `finish sync`, we use extra `finally`-blocks.

To handle network connection issues, we set up `SO_RCVTIMEO` and `SO_SNDTIMEO` options for the BSD compatible sockets that are used for communications between the device and the cloud.

If any error occurs during the remote invocations to `execution run` in the cloud, the synchronization between the device and the cloud discussed in the previous section is broken. The Uniport client will invoke the `prepare sync` primitive before the next remote invocation to the `execution run` primitive.

D. MCA Development Support

In order to improve productivity for MCA developers using the Uniport framework, we provide a set of developing tools for MCA development. Currently the toolset consists of a code generator and a static code analyzer.

The code generator takes advantage of the simplicity and uniformity of the Uniport architecture and can generate code skeletons for iOS, Android and Windows Phone from XML-based MCA configurations. With the configuration and the target mobile platform specified, the code generator produces two projects, one for the device, the other for the cloud. The former project contains the configured Uniport client and the defined MCA models. The latter project contains the configured Uniport server and the defined MCA models.

The configuration of an MCA is an XML file that defines modules and specifies various attributes of the MCA, such as cloud server address and port number, runtime policies, timeout values.

Using the code generator, new MCAs for different mobile platforms can be easily created. First, the developers design the application and write an MCA configuration for it. Then the developers choose from the supported mobile platforms and generate necessary projects and code skeletons with the code generator. Afterwards the developers can start implementing the MCA. During implementation, the developers can reuse code forming the model parts of the MCA in both the client project and the server project. After these steps are completed, the developers can build the projects, and deploy the client application and the server application to the device and the cloud respectively.

We also notice that there is demand for refactoring existing code to use cloud computing services as well as demand for constructing an MCA from scratch. While the code generator is mainly used for developing new MCAs based on the Uniport framework, the static code analyzer helps developers apply the Uniport framework to existing code.

The analyzer is based on the ANTLR parser generator [17]. It parses Objective-C, Java and C# source code and checks whether it conforms with the Uniport architecture and its constraints for MCAs. The analyzer checks against direct constraint violations while parsing and maintains a reference graph for checking against indirect constraint violations.

We build the analyzer upon a parser instead of a lexer because there are cases that a lexer can not detect accurately, such as qualified names in Java and C#.

Another important job in MCA development is to identify computationally-intensive code that is *suitable* for cloud execution. Modern mobile platforms include profiling tools that can help developers with the identification job, such as Instru-

ments [1], Traceview [2], Windows Phone Application Analysis [3]. With these profiling tools, developers can pick out candidates that are suitable for cloud execution, according to the formula mentioned previously with respect to energy efficiency. After dynamic profiling, the analyzer helps developers refactor the existing code and apply the Uniport framework.

As mentioned in Section II, the project for the device and the project for the cloud share the same code for the model parts of the MCA without any modification. This goal of simplicity is achieved via the Uniport architecture and with the help of compilers. If code for the model parts of an MCA fails to compile, either in the device project or in the cloud project, one possible reason is that the code violates Uniport's constraint. MCA developers should check the implementation of the model and modify it to the way that conforms with the Uniport framework.

E. Cloud Side Support

In our implementation, the Uniport server supports cloud computing services provided for iOS, Android and Windows Phone devices. It uses threads to serve multiple clients concurrently. For one client, it deserializes the received data, invokes primitives `restore session`, `execution run` and `backup session` successively, serializes the execution results and sends them back to the client.

Only data that represents the model part of an MCA known to the Uniport server will be deserialized. We embed the three primitive invocations in a `try`-block. If any exception is caught, service to the client will stop.

For iOS clients, the server application can run on Mac OS X or with GNUstep, which is an open source implementation of the Cocoa middleware. For Android clients, the server application can run on platforms that support the Java virtual machine. For Windows Phone clients, the server application can run on the .NET framework or Mono, which is an open source .NET implementation. GNUstep and Mono are cross-platform replacements for Cocoa and the .NET framework. So iOS and Windows Phone clients do not actually require Mac OS X and Windows environments in the cloud.

V. Case Study and Evaluation

In this section, we study how the Uniport framework can be applied to existing mobile applications to transform them into MCAs. We also evaluate the Uniport framework with the three applications to demonstrate how the framework and its runtime policies perform on three different mobile platforms.

A. Case Study

We choose three open source mobile applications, Kigomoku [18], Fivestones [19] and GomokuPro [20] for iOS, Android and Windows Phone respectively because it is not feasible to do binary code refactoring on all mobile platforms. The three applications implement Gomoku, an abstract strategy board game, in which users play with the AI.

All three chosen applications adopt the MVC pattern. The model part is the Gomoku AI that thinks and plays with the user. The view part is the Gomoku board where the game is displayed and the user responds. The controller part handles user inputs and updates the game board from user inputs and the AI's responses.

The Gomoku model is a model where a sequence of invocations is doing the same job, i.e., playing a round of Gomoku game with the user. To be energy efficient, we only serialize and transmit data that represents the differences between game states.

We transform the three applications to MCAs respectively based on the Uniport framework. Table III shows the modifications applied to the three applications, in lines of source code added, removed and modified in total. We can see that with the Uniport framework, transforming an existing mobile application to an MCA needs relatively small amount of modifications, ranging from 3–9% of the total lines of code.

All the three applications have "undo" buttons, the action of which actually breaks the synchronization. So we need to notify the Uniport client about that. Figure 3a demonstrates part of our modifications to the controllers of Kigomoku to support the "undo" action. Similar modifications to Fivestones and GomokuPro are demonstrated in Figures 3b and 3c.

B. Evaluation

In order to evaluate the three applications, we do not use the models contained in the three applications. Instead, we implement a Gomoku model that uses depth-first search and α–β pruning [21]. Using the same Gomoku model among the three applications helps us analyze platform and language characteristics. Unlike the original models used in the three applications, we do not bring any randomness into our AI algorithm. With the same user input, the response of the model is always the same. So we can reproduce the evaluation results.

The machine we use to simulate the cloud is a laptop with 2.66GHz CPU and 4GB RAM. The devices we use for evaluation are the second generation iPod Touch, Google Nexus S and Nokia Lumia 800.

The response time is calculated as the time between two adjacent invocations to primitives `execution will begin` and `execution did end`. On iOS and Android, energy consumption is measured via battery level APIs. On Windows Phone, it is measured in the Diagnostics application. To improve accuracy, we repeat the same round of Gomoku game for ten times and calculate an average energy consumption for one round.

For the first part of our evaluation, we run the three applications in emulation environments: iOS simulator, Android-x86 [22] virtual machine and Windows Phone emulator. The iOS simulator executes x86 native code and the CLR of Windows Phone emulator is based on x86. The emulator from the Android SDK is too slow because it interprets ARM instructions. So we choose Android-x86 instead.

[1] https://developer.apple.com/technologies/tools/

[2] http://developer.android.com/tools/help/traceview.html

[3] http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj215908\%28v=vs.105\%29.aspx

TABLE III: Amount of modifications applied to the three applications

Application	Total Lines	Part	Lines Added	Lines Removed	Total Modifications
Kigomoku	1805	Model	119	0	161 lines, 8.9%
		Controller	37	5	
Fivestones	4424	Model	90	0	133 lines, 3.0%
		Controller	33	10	
GomokuPro	3428	Model	86	2	120 lines, 3.5%
		Controller	31	1	

```
- (void)undoRedoMove:(id)sender {
  if (sender == undoButton) {
    int moves_to_undo = 2;
    if ([self game].gameStarted == NO
        && [self game].currentPlayerIndex == 1)
      moves_to_undo = 1;
    NSLog(@"undo UNDO pressed, undoing %d moves",
        moves_to_undo);
    [((GomokuViewController *)self.mainController).model
        setSyncFlag:NO];
    for (int i = 0; i < moves_to_undo; i++)
      [[self game] undoLastMove];
  } else if (sender == redoButton) {
    NSLog(@"undo redo pressed");
  }
}
```

(a) Kigomoku

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
  if (keyCode == KeyEvent.KEYCODE_BACK) {
    handler.makeStepBack();
    ((AndroidEnemy)enemy).model.setSyncFlag(false);
    return true;
  }
  return super.onKeyDown(keyCode, event);
}
```

(b) Fivestones

```
private void Withdraw_Click(object sender,
                            RoutedEventArgs e)
{
  if (ChessBoard.Mode == GameMode.PlayMode)
  {
    ChessBoard.Withdraw();
    ChessBoard.Withdraw();
    if (engine.model != null)
      engine.model.syncFlag = false;
  }
}
```

(c) GomokuPro

Fig. 3: MCA modifications to support "undo" (*code added shown in italic*)

In each set of the results, we run the mobile versions of the applications (without cloud support) and the mobile cloud versions (with cloud support) separately, and compare their performances.

Figure 4a shows the evaluation results. We can see that the CLR in Windows Phone emulator is not as fast as that of the .NET framework and that the Dalvik virtual machine performs worse than the Java virtual machine. From the result of Kigomoku, we can see that the Uniport framework does not introduce performance overhead.

Then we run the three applications on real devices. Figure 4b shows the comparisons of response time and Figure 4c shows the comparisons of energy consumption. With cloud support, the Gomoku game achieves 3–7x speedup and 48–66% decrease in energy consumption on different mobile platforms.

On real devices, GomokuPro is the fastest among the three applications because the CPU of Lumia 800 is faster. Although Nexus S has a better CPU than the second generation iPod Touch, Fivestones is still slower than Kigomoku because the runtime efficiency of the Dalvik virtual machine is inferior. Kigomoku consumes the least amount of energy among the three applications because iOS applications execute native code directly without the overhead of virtual machines.

The case studies and evaluation results show that, with the application of the Uniport framework, existing mobile applications can be converted to MCAs easily to improve performance and reduce energy consumption.

VI. Discussion

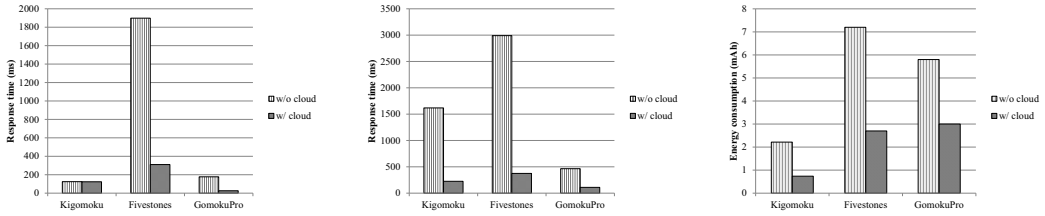
A. Applicability of Uniport

The Uniport framework support the transformation of mobile applications based on the MVC pattern or its derivative patterns into MCAs. This does not restrict the applicability of the Uniport framework because the MVC and its derivative patterns are widely adopted in mobile applications on different platforms. The iOS Developer Library officially states that "MVC is central to a good design for a Cocoa application" [23]. Questions and answers about the MVC pattern in Android applications on Stack Overflow are rated as very useful [24, 25].

Although the Uniport framework is designed specifically with the MVC pattern in mind, it can also be applied to non-MVC applications with a little extra effort. As the programming primitives and the runtime libraries can still be used in any MCAs, the only extra effort we need for a non-MVC application is to identify the modules that *can* be offloaded to the cloud, which actually have similar roles to the models in the MVC pattern. The identification task can be accomplished either (semi-)automatically or manually, which have been demonstrated in previous work [6]. The techniques can be incorporated to the Uniport framework to support MCAs in more general forms.

B. Performance and Energy Results

One of the major benefits of mobile cloud computing is that it enables offloading computation from the mobile device to the cloud in order to improve performance and save energy. The performance of computation offloading highly depends on the speed of the cloud and the quality of the network connection between the mobile device and the cloud. In our evaluation, we use a laptop as the cloud and wireless local area network as the communication channel between the devices and the cloud. If we use faster cloud servers and higher-speed network connection, MCAs will have higher performance and lower energy consumption. However, if the network speed is slower when using the remote cloud, it might increase the offloading time due to the transmission latency. Many



(a) Response time on emulators (b) Response time on devices (c) Energy consumption on devices

Fig. 4: Evaluation results

tradeoffs need to be made in reality for MCAs, which have been studied extensively in the mobile computing and systems community [2, 3, 6, 26].

Because the focus of this paper is developing a programming framework to support MCA development, many runtime decisions are beyond the scope of this paper. So the performance and energy results shown in this paper are only a proof of concept, and are not meant to reflect the complicated reality of mobile cloud computing.

VII. Related Work

A. Mobile Cloud Computing

Carzaniga, Picco and Vigna propose mobile code paradigms for distributed applications [27]. Weinsberg and Ben-Shaul propose a model for disconnected-aware applications on resource-constrained devices [28].

In the Uniport framework, we adopt a new paradigm where both the device and the cloud know how and own resources. So MCAs based on the framework can run locally when disconnected from the network, which guarantees availability.

CloneCloud [3] discusses the possibility of cloning mobile device to a virtual machine running in the cloud and dynamically transferring the execution of mobile applications. MAUI [6] discusses the possibility of using managed code environment to provide a fine-grained energy-aware offloading solution. Virtualized screen [29] presents a way for cloud-mobile convergence by rendering screen of mobile device in the cloud. Cuckoo [4] presents a framework which simplifies the development of mobile applications that benefit from computation offloading. Paranoid Android [5] focuses on protecting mobile devices with replay in the cloud. COMET [2] leverages distributed shared memory for migrating execution transparently on Android. Sapphire [30] provides a general-purpose distributed programming platform for simplifying the design and implementation of mobile/cloud applications via the separation of application logic from deployment logic.

Sapphire, COMET, CloneCloud, Cuckoo and Paranoid Android are specifically designed for Android. The implementation of MAUI relies on the reflection mechanism supported by the CLR of Windows Mobile. Our Uniport framework, however, does not rely on platform specific features. It uses features and characteristics that are common among different platforms and programming languages and makes low level details transparent to MCAs based on the framework.

In the Uniport framework, we use the formula proposed by Kumar and Lu [1] to calculate the reduction in energy consumption. Recently, more fine-grained energy models [31] and scheduling policies [26] have been proposed, which will help MCAs make better decisions about code offloading.

B. Models for Mobile Applications

The MVC pattern was formulated in the 1970s by Trygve Reenskaug at Xerox PARC [32]. During the past 40 years, its application scenarios have been studied in-depth. Many other patterns derive from MVC, such as Model-View-ViewModel, Model-View-Presenter and Model-View-Adapter. But to the best of our knowledge, no previous work has explored the possibility of combining MVC and mobile cloud computing.

Christensen examines architectural considerations for mobile application created with RESTful web services [33], with data transfer size optimization and offloading intensive calculations taken into consideration. Medvidovic and Edwards conclude the state-of-the-art and research challenges of mobile software and systems [34]. Bronsard discusses framework constraints [35], which inspire us in designing the Uniport framework and the constraints to the Uniport architecture.

C. Cross-Platform Solutions

There are a few cross-platform solutions to mobile application development. Mobl [9] is a language designed to declaratively construct web applications for different mobile platforms. PhoneGap [10] is a framework that enables development of native application for different mobile platforms with web technologies.

Both Mobl and PhoneGap use web technologies including HTML5, CSS and JavaScript, which are mainly concerned with interactive user experience. Mobile applications developed with them actually run on a web view control provided by the mobile platform, instead of the mobile platform itself, limiting their runtime performance and energy efficiency, making them suboptimal solutions to mobile cloud computing.

In the Uniport framework, we provide a conceptually cross-platform solution. The primitives are platform and programming language-independent and the runtime libraries support multiple mobile platforms. The principles of the Uniport architecture are simplicity and uniformity.

VIII. Conclusion and Future Work

In this paper, we present Uniport, a uniform framework for developing mobile cloud applications (MCAs). The Uniport framework introduces a new architecture for MCAs, a set of programming primitives and runtime libraries for MCA development and runtime support, and runtime policies to efficiently leverage cloud computing and storage services and improve energy efficiency. We have implemented the Uniport framework on three popular mobile platforms. Case studies on different platforms show that, with relatively small amount of modifications to existing mobile applications, developers can create MCAs that reduce execution time and energy consumption significantly.

Security features such as SSL/TLS have not been used in the Uniport framework. In the future, we will consider adding SSL/TLS to our framework for better protection of security and privacy.

Other automatic tools would also be helpful to MCA developers in addition to the code generator and analyzer, such as a profiler that helps developers determine the partition of modules for computation offloading, and IDE plugins for Xcode, Eclipse and Visual Studio that integrate the code generator, analyzer and profiler together to further improve productivity.

ACKNOWLEDGMENT

This work is supported by the High-Tech Research and Development Program of China under Grant No.2013AA01A605, and the National Natural Science Foundation of China under Grant No.61421091, 61103026.

References

[1] K. Kumar and Y.-H. Lu, "Cloud computing for mobile users: Can offloading computation save energy?" *Computer*, vol. 43, no. 4, pp. 51–56, Apr. 2010.

[2] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "Comet: Code offload by migrating execution transparently," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 93–106.

[3] B.-G. 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*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 301–314.

[4] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, "Cuckoo: a computation offloading framework for smartphones," in *Mobile Computing, Applications, and Services*. Springer, 2012, pp. 59–79.

[5] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid Android: Versatile protection for smartphones," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. New York, NY, USA: ACM, 2010, pp. 347–356.

[6] 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*, ser. MobiSys '10. New York, NY, USA: ACM, 2010, pp. 49–62.

[7] H. Mei and Y. Guo, "Network-oriented operating systems: status and challenges," *SCIENCE CHINA Information Sciences*, vol. 43, no. 3, pp. 303–321, 2013, (in Chinese).

[8] W. Zheng, "An introduction to Tsinghua Cloud," *SCIENCE CHINA Information Sciences*, vol. 53, no. 7, pp. 1481–1486, 2010.

[9] Z. Hemel and E. Visser, "Declaratively programming the mobile web with Mobl," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 695–712.

[10] Adobe Systems Inc., "PhoneGap," http://phonegap.com.

[11] M. Brown and R. Housley, "Transport Layer Security (TLS) Authorization Extensions," http://tools.ietf.org/html/rfc5878.

[12] Embedded Linux Wiki, "Android Kernel Features," http://elinux.org/Android_Kernel_Features.

[13] Apache Software Foundation, "Apache Harmony," http://harmony.apache.org.

[14] Y. Guo, L. Zhang, J. Kong, J. Sun, T. Feng, and X. Chen, "Jupiter: Transparent augmentation of smartphone capabilities through cloud computing," in *Proceedings of the 3rd ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds*, ser. MobiHeld '11. New York, NY, USA: ACM, 2011, pp. 2:1–2:6.

[15] Y. Dong, H. Zhu, J. Peng, F. Wang, M. P. Mesnier, D. Wang, and S. C. Chan, "RFS: A network file system for mobile devices and the cloud," *SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, pp. 101–111, Feb. 2011.

[16] Android Open Source Project, "Keeping Your App Responsive," https://developer.android.com/training/articles/perf-anr.html.

[17] T. J. Parr and R. W. Quong, "ANTLR: A predicated-LL (k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[18] K. Gredeskoul, "Kigomoku," https://github.com/kigster/kigomoku.

[19] tungi52, "Fivestones project," http://code.google.com/p/fivestones-project.

[20] DragonGame, "GomokuPro," http://windowsphone.com/s?appid=bf68c082-1ff4-4a15-ba31-02ed83643453.

[21] G. Heineman, G. Pollice, and S. Selkow, *Algorithms in a Nutshell*. Oreilly Media, 2008, ch. Path Finding in AI, pp. 217–223.

[22] C.-W. Huang and Y. Sun, "Android-x86 Project," http://www.android-x86.org.

[23] Apple Inc., "Cocoa Core Competencies: Model-View-Controller," https://developer.apple.com/library/ios/documentation/general/conceptual/devpedia-cocoacore/MVC.html.

[24] Burjua, "MVC pattern in Android?" http://stackoverflow.com/q/2925054/2748784.

[25] JustDanyul, "Which design patterns are used on Android?" http://stackoverflow.com/a/6770903/2748784.

[26] W. Zhang, Y. Wen, and D. O. Wu, "Energy-efficient scheduling policy for collaborative execution in mobile cloud computing," in *INFOCOM, 2013 Proceedings IEEE*, ser. INFOCOM '13. IEEE, 2013, pp. 190–194.

[27] A. Carzaniga, G. P. Picco, and G. Vigna, "Designing distributed applications with mobile code paradigms," in *Proceedings of the 19th International Conference on Software Engineering*, ser. ICSE '97. New York, NY, USA: ACM, 1997, pp. 22–32.

[28] Y. Weinsberg and I. Ben-Shaul, "A programming model and system support for disconnected-aware applications on resource-constrained devices," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 374–384.

[29] Y. Lu, S. Li, and H. Shen, "Virtualized screen: A third element for cloud-mobile convergence," *IEEE MultiMedia*, vol. 18, no. 2, pp. 4–11, Apr. 2011.

[30] I. Zhang, A. Szekeres, D. V. Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy, "Customizable and extensible deployment for mobile/cloud applications," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 97–112.

[31] C. Wang, F. Yan, Y. Guo, and X. Chen, "Power estimation for mobile applications with profile-driven battery traces," in *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 120–125.

[32] T. Reenskaug, "MVC," http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html.

[33] J. H. Christensen, "Using RESTful web-services and cloud computing to create next generation mobile applications," in *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. New York, NY, USA: ACM, 2009, pp. 627–634.

[34] N. Medvidovic and G. Edwards, "Software architecture and mobility: A roadmap," *J. Syst. Softw.*, vol. 83, no. 6, pp. 885–898, Jun. 2010.

[35] F. Bronsard, "Practical framework constraints," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 273–276.