Enabling Automated, Rich, and Versatile Data Management for Android Apps with BlueMountain

Sharath Chandrashekhara, Kyle Marcus, Rakesh G. M. Subramanya, Hrishikesh S. Karve,
Karthik Dantu, Steven Y. Ko
Department of Computer Science and Engineering
University at Buffalo, The State University of New York
Email: {sc296, kmarcus2, rakeshgu, hkarve, kdantu, stevko}@buffalo.edu

Abstract

Today's mobile apps often leverage cloud services to manage their own data as well as user data, enabling many desired features such as backup and sharing. However, this comes at a cost; developers have to manually craft their logic and potentially repeat a similar process for different cloud providers. In addition, users are restricted to the design choices made by developers; for example, once a developer releases an app that uses a particular cloud service, it is impossible for a user to later customize the app and choose a different service.

In this paper, we explore the design space of an app instrumentation tool that *automatically* integrates cloud storage services for Android apps. Our goal is to allow developers to treat all storage operations as local operations, and automatically enable cloud features customized for individual needs of users and developers. We discuss various scenarios that can benefit from such an automated tool, challenges associated with the development of it, and our ideas to address these challenges.

1 Introduction

A unique aspect of current mobile app development is integrated data management which leverages both local and cloud storage. This integration enables rich forms of interaction such as backup and sharing across various devices belonging to one or more users. However, this richness comes at a cost; developers are presented with several design choices such as which cloud providers to use and what kinds of consistency to guarantee. Developers also need to decide which of these decisions should be configurable by their users, and the users in turn are constrained by the decisions of the developers. These challenges are mostly tangential to app logic, and developers need to repeat a similar process for different apps, taking precious developer resources away from core app development.

Further, cloud storage services provide their own APIs that are not compatible with each other in their interfaces and semantics. As a result, choosing a particular cloud service binds an app and its data interaction to the semantics of the API provided by the cloud service. In addition, if a developer wants to leverage multiple cloud services or migrate from one to another, she needs to learn and understand the semantics for each cloud service provider that she wants to leverage. This might involve reasoning about different consistency models offered by various providers and manually tailoring to them, which is a challenging process.

In order to address these problems, we propose to take the burden off app developers; we aim to reduce the development effort involved in learning, customizing, and adapting to different semantics and interfaces of various cloud providers, and allow developers to focus instead on their app logic. Our key idea is *automatically virtualizing storage operations* in a mobile app. We perform this by statically analyzing an app binary, and augmenting the storage calls with richer forms of data interaction. Ultimately, we aim to allow developers to treat all storage operations as local storage operations, and not concern themselves with any particular cloud service, including APIs and semantics.

To achieve our goal, we are developing *BlueMountain*, a framework that automatically integrates cloud storage services with Android apps. BlueMountain decompiles an Android app, correctly identifies all storage-related code points to instrument, and enables cloud service integration automatically. This instrumentation has several interesting challenges to address, which we discuss in this paper.

In the remainder of this paper, we first discuss a few scenarios that can benefit from our approach (Section 2). We then discuss the challenges and ideas for BlueMountain (Section 3) as well as our prototype (Section 4). We describe the related work in Section 5. Lastly, we discuss our future work and conclusions in Section 6.

2 Motivation

As mentioned in Section 1, our key insight into enabling richer data management for mobile apps is *storage virtualization*. In this section, we will motivate its benefits through specific use cases where such virtualization would greatly simplify mobile app development.

2.1 Use Case 1: Private Corporate Cloud

Bob, an Android app developer, has built AwesomeNotes, a corporate note taking app. Bob understands the need of reliability for this app and spends much development time to make sure AwesomeNotes backs up all user data on a server. Paranoid Inc. and Stealthy Inc. are two big corporate houses who are interested in using AwesomeNotes but are concerned about their proprietary notes being transferred out to the public Internet and stored on a third-party server. Paranoid Inc. and Stealthy Inc. have their own cloud where they prefer to securely store their data. So they ask Bob for a custom version of AwesomeNotes that backs up data to their private cloud. This requires Bob to create and maintain different apps for each corporate house, and spend time to understand custom settings for each potential client. Similarly, Paranoid Inc. and Stealthy Inc. have to do this for each and every app they are using. They have to share their cloud backup APIs and ask the developers to design customized apps for their settings.

Our storage virtualization approach could greatly simplify this customization process. For example, it could allow Bob to write his app as if it only uses local storage; it could then take the app binary and automatically instrument the app so that it could use a cloud service. This automation could adapt to different (private) cloud services without requiring Bob to go through the same process of manual customization repeatedly.

This automation could also allow Bob to easily add new cloud storage features. For example, suppose Bob wants to add a new feature in AwesomeNotes that would allow people to collaborate and edit notes simultaneously. Our automated storage virtualization could make this process as easy as running the app through our tool once again with a different set of configuration parameters, specifying various requirements for the new feature such as cloud providers to use and consistency guarantees.

2.2 Use Case 2: Alternative to Cloud APIs

EasyBox is a new cloud storage company. Like all major cloud storage companies, EasyBox is working on releasing its API libraries for different platforms so that app developers can use them as a backup option in mobile apps. These API libraries provide basic support for uploading and downloading files, leaving all the logic of dealing with backup logic (e.g., diffs, consistency guarantees) to app developers.

EasyBox could benefit from our automated storage virtualization approach; they could release an EasyBox compiler instead of API libraries, i.e., instead of asking developers to download and use their API libraries, EasyBox could simply provide a Web-based tool that automatically instruments each app to use their cloud service. Bob the Android developer now could develop an app using just local storage and upload the binary to EasyBox, which would return a new version of the app capable of using EasyBox as a backup service. This would increase his productivity significantly.

2.3 Use Case 3: Closed Group Sharing

Paranoid Inc., the large company mentioned earlier, is using an internal app throughout the company called MeetingScheduler. Marvin, a neurotic manager, has certain preferences for the app, such as the set of resources to use, the maximum time a meeting should be held, specific dietary restrictions for food orders, and certain rooms in which he would like his team meetings to be held. MeetingScheduler allows its users to set these preferences inside the app and Marvin wants everyone in his group to use the same settings that he uses. Being fickle minded, Marvin changes these settings often, causing everyone in his group to also change their settings frequently. To simplify this process, he asks the developers of MeetingScheduler to design a customized version so that all the settings can be shared among a closed group of users. Bob the application developer spends much time developing customized code for the requirements of Marvin.

This scenario could benefit from our automated storage virtualization. Instead of asking Bob to redesign the app, Marvin could use our tool to generate a customized version of MeetingScheduler with a cloud sharing feature. With this group sharing feature of the app, Marvin could easily have all his group members use his settings. Everyone would install the instrumented app and sync from Marvin's settings. Whenever Marvin changes the settings, the group members would automatically get their settings updated. Sharing and security would be completely handled by the automated instrumentation.

3 BlueMountain Design

The uses cases described above show the utility of automated storage virtualization. Unlike libraries that developers need to use manually, automated instrumentation can potentially reduce the burden of development and even allow end users to modify and enhance downloaded apps.

We are currently exploring this automated approach with a framework called BlueMountain. Below, we present the overall architecture and the research challenges for BlueMountain. Although our discussion in this section is mostly centered around cloud storage services, we envision that we can apply similar concepts in a hybrid cloud-peer setting.

3.1 BlueMountain Overview

Figure 1 shows the overall architecture of BlueMountain with two main components—*BlueMountain com-*

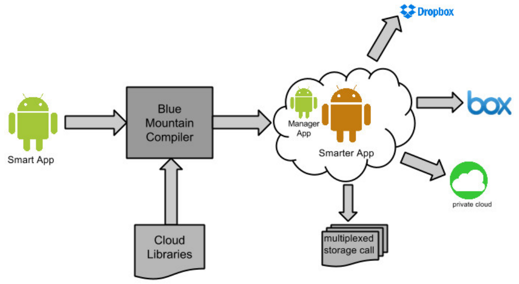

Figure 1: BlueMountain System Overview

piler and *BlueMountain manager*. The BlueMountain compiler takes an Android app that wants storage virtualization and a configuration file that describes the cloud service API of the provider of choice, e.g., DropBox and Google Drive. It then uses static analysis to intercept storage calls at the bytecode-level and customizes their functionality. The BlueMountain manager is an app installed on a user's mobile device [1] that manages user preferences, settings, and runtime configurations.

In order to concretely realize this architecture, there are a number of questions we need to answer. First, we need to identify the set of storage operations available on Android, so we can enumerate all possible ways that an app can store data. Second, given all the storage operations, we need a mechanism to analyze and instrument an app binary for storage call interception and virtualization. Third, we need to match the storage semantics of an app to the storage semantics of the cloud services the app wants to use. This requires us to bridge the gap (if any) between (i) the local Android storage interface and cloud providers' interfaces, and (ii) the storage semantics of an app and the semantics provided by cloud services. In the rest of this section, we detail each question and report our findings so far.

3.2 Android Storage Options and APIs

We first need to identify the set of storage operations in Android, which will allow us to correctly identify which operations we need to intercept with app instrumentation. We answer this question by examining Android APIs.

Storage Access on Android: Unlike desktop apps, Android apps use storage in a more managed fashion. Every app has its own storage sandbox, i.e., specific locations for read/write operations controlled by Linux file permissions. In addition, Android provides APIs such as `Activity.getDir()` to access these locations. These APIs abstract away the detail of where files are stored in the filesystem, which ensures portability across devices and OS versions. The use of path strings to access storage is likely to only work on a specific platform, hence discouraged.

Storage on Android: Android provide four storage options for an app: (i) files, (ii) SQLite database, (iii) `SharedPreferences`, a key-value store commonly used for storing user settings, and (iv) `ContentProvider`, a storage interface that an app can implement which is commonly used to provide structured data to other apps, e.g., contact information. All of these options are well-documented and we can easily enumerate the list of API calls available for these options using the Android documentation [1].

These two characteristics (managed storage access and well-documented storage options) give us the opportunity to tap into the storage operations and virtualize them. The next challenge is to instrument an app to virtualize storage calls, and this is more challenging than simply finding and replacing storage method calls as we discuss below.

3.3 App Instrumentation

Identifying all possible storage options and their APIs gives us a good starting point for storage call virtualization. However, there are additional questions we need to answer. First, we need to be able to instrument an Android app binary without requiring its source code. Second, we need to correctly intercept storage calls when an app makes the calls. For the first question, there are various off-the-shelf Java/Android instrumentation tools available that we can leverage, among which we have chosen to use Soot [8]. Soot allows us to decompile an Android app, and statically analyze the decompiled source with its intermediate representation called Jimple [9].

At first glance, it may seem that all that we need to do is a text search to identify all call sites where storage APIs are used, and rewrite those call sites to virtualize the storage operations. However, it turns out that it is much more involved than that. For example, consider the code snippet shown in Figure 2. In lines 1 to 6, an app class `MyFileOutputStream` extends a Java library class `FileOutputStream` and overrides `write()`. In lines 10 to 12, `main()` creates a `MyFileOutputStream` object and passes it to `myWrite()`, at which point type casting happens to `FileOutputStream`. This means that when `write()` is called in line 17, it is called with the type `FileOutputStream`, but the actual object is of type `MyFileOutputStream`.

If we want to intercept `FileOutputStream.write()`, a simple search for the type `FileOutputStream` and the method `write()` will determine that we need to intercept the call site at line 17. How-

[1] BlueMountain manager can also be injected into the app itself without requiring a user to install an app—an alternative architecture we are exploring.

```
public class MyFileOutputStream
        extends FileOutputStream {
                public int write(Bytes b) {
                // Overriding
                }
}
public class Main {
        public static void main(String args[]) {
                Bytes b = 10;
                MyFileOutputStream obj1 =
                    new MyFileOutputStream();
                myWrite(obj, b);
        }

        public static void myWrite
            (FileOutputStream obj, Bytes b) {
                        obj.write(b);
        }
}
```

Figure 2: Call Interception Example

ever, as the example demonstrates, the actual type can be different, e.g., `MyFileOutputStream`, which is not what we want to intercept. Another simple approach that one can easily perceive is to rewrite `FileOutputStream` itself and put interception code in its `write()`. However, modifying `FileOutputStream` would require us to modify system libraries that are part of the Android platform. Unless we could redistribute the Android platform, it would be impossible for us to deploy our interception code.

In order to address this problem, we are exploring *class wrapping* as a solution. In our approach, we generate a wrapper class for each storage class we want to intercept and modify the original type to its corresponding wrapper type. While this gives us an opportunity to correctly intercept storage API calls, there are further challenges to overcome due to intricate interplays between generated wrappers and the rest of the code. This is mainly caused by many language features offered by Java such as synchronization, class loading, and native code integration (JNI). We are currently identifying and addressing these challenges in our full implementation.

3.4 Bridging the Gap: Interfaces and Semantics

The last challenge we need to address is bridging the gap between local and remote storage operations. This comes in two flavors: (i) interface mismatch— translating local storage operations into operations that perform the same way on a hybrid local/cloud storage, and (ii) semantics mismatch— providing the same semantics of local storage (e.g., consistency), while using cloud services. This, in particular the second mismatch, is quite challenging to deal with and in some cases, may not even be possible to address. As a first step, we discuss the interfaces provided by current cloud services and how we are addressing specific mismatches.

Cloud Storage APIs: Generally, all the major cloud storage providers (e.g., Google Drive and DropBox) use an object store model, i.e., they provide the ability to upload and download files through a Web-based CRUD interface. In addition, each cloud service provides alternative interfaces for development convenience. These alternative interfaces internally leverage Web-based CRUD APIs and provide advanced features. For example, DropBox provides Sync API that resembles a local filesystem interface, which internally syncs data to the DropBox cloud. DropBox also has DataStore API that provides a convenient way to store structured data (e.g., a table). This DataStore API automatically performs conflict resolution as well. Similarly, Google's RealTime API (though only available as a JavaScript library) offers a way to use objects of various granularities (e.g., strings and lists) to sync across multiple clients.

Interface Mismatch: The above discussion about cloud provider interfaces reveals three facts: (i) the common denominator for all cloud providers is Web-based basic CRUD APIs, (ii) there is a diverse set of interfaces that they provide, and (iii) CRUD APIs only assume an object-based access model. Due to first two facts, we believe that using the Web-based basic CRUD APIs is a better choice for an automated tool like BlueMountain. It will make the job of customizing cloud storage access easier, since it will mostly be mechanical. On the other hand, if we were to use the alternative APIs such as DropBox DataStore API, it would be difficult for us to make it adaptable across different providers.

The last fact poses a challenge for us since we need to deal with various granularities of objects. As discussed earlier, Android provides four storage options, and these options operate at different object granularities. For example, Android stores an entire SQLite database instance as a single file, but apps access it at a much finer granularity, e.g., individual rows and columns. This means that a naive way of using a CRUD API, i.e., updating an entire file whenever there is any change, might not be the best way to deal with Android storage options.

Semantics Mismatch: In addition to the interfaces, there is a potential mismatch between app consistency requirements and cloud providers' guarantees. This stems from the fact that all cloud services only provide eventual consistency through their CRUD interfaces. This is true even with private cloud offerings such as OpenStack [4]. While this model works well in most of the cases, it is not suitable for all apps. For instance, creating a collaborative mobile app requires strong consistency guarantees with automatic conflict resolution. Using alternative APIs might help; however, cloud service providers have completely different semantics with their alternative APIs, making it difficult for us to leverage. In order to address this, we are exploring a hybrid cloud-peer approach, where we leverage a cloud service both as a stable storage and a message rendezvous point [2], allowing clients to talk with each other to provide strong consistency guarantees. Additionally, we plan to use an instrumentation-time configuration file that specifies various parameters. This file would contain parameters such as the cloud service to use (e.g., DropBox), the desired consistency model (e.g., near-real-time), etc.

4 Preliminary Prototype and Evaluation

We have a preliminary prototype of BlueMountain with an ability to automatically create a backup-aware version of an Android app. Our prototype implements search-and-replace instrumentation, hence limited in terms of instrumentation correctness.

To test our implementation, we created a sample Android app similar to the MeetingScheduler app mentioned in Section 2.3. Our prototype instruments the app to back up all the app data including the settings of the app to DropBox. It also enables the app to restore everything from DropBox at installation time either on the same device (a re-installation) or a different device (a fresh installation and re-installations afterwards).

With this instrumentation, we have conducted some preliminary experiments. From a fresh installation, starting the app takes 0.17s on average *without* instrumentation, and 0.67s on average *with* instrumentation. This extra latency comes mostly from our injected code that checks if there is any data to restore from DropBox.

After a re-installation, data gets restored at startup, and it takes 1.65s with a few KBs of data to restore. The extra latency comes from data restoration, and it will depend on the size of the data to restore. These results show that the instrumentation itself will not add much overhead. Rather, most of the overhead will stem from the way we interact with cloud services, particularly when stronger synchronization is required.

5 Related work

There is no directly related work on automated tools for customization and adaptation. However, previous work has tackled various challenges with mobile devices accessing cloud services. Procrastinator [7], developed for Windows phones, analyzes and rewrites app binaries to optimize caching and reduce network traffic. Viewbox [10] and Simba [5] both deal with providing fault tolerance and consistency guarantees for data communication between mobile devices and cloud services. Viewbox uses checksums to provide fault tolerance and syncs only consistent views of local data. Simba provides consistency guarantees by providing programmers with a high-level local programming abstraction unifying tabular and object data. This provides strong consistency guarantees under all failure conditions. Cimbiosys [6] deals with content sharing between mobile devices and cloud services. It has an eventual consistency model where data is updated when a connection exists between a device and a cloud service. Apache Cloudlib [2] is a Python library that provides a unified API for interacting with many of the popular cloud service providers. CloudRail [3] is a startup which is creating a universal cloud API. With this API, programmers have access to all popular cloud service providers and they extend this advantage to the users.

6 Conclusion and Discussions

In this paper, we have argued that an instrumentation tool may enable mobile apps to access cloud storage services automatically. By separating data management from app logic, we believe the process of app development will be easier and faster. We are currently developing BlueMountain, a prototype tool that aims to demonstrate the power of automated storage virtualization.

Our future work includes, (i) concretely exploring consistency and interface mismatch problems, (ii) analyzing apps in the Google Play Store to identify a subset of apps that are most likely to benefit from BlueMountain, and (iii) exploring the possibility of automatically adapting an app developed for one cloud service to use a different cloud service.

Acknowledgments

This work has been supported in part by an NSF CAREER award, CNS-1350883.

References

[1] Android Documentation. `http://developer.android.com/`.

[2] Apache Libcloud. `http://libcloud.apache.org`.

[3] Cloud Rail. `http://cloudrail.com/`.

[4] OpenStack Swift. `https://swiftstack.com/openstack-swift/`.

[5] Y. Go, N. Agrawal, A. Aranya, and C. Ungureanu. Reliable, consistent, and efficient data sync for mobile apps. In *FAST'15*. USENIX Association, Feb. 2015.

[6] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. In *NSDI'09*. USENIX Association, 2009.

[7] L. Ravindranath, S. Agarwal, J. Padhye, and C. Riederer. Procrastinator: Pacing mobile apps usage of the network. In *Proc. ACM MobiSys*, 2014.

[8] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *CASCON'99*. IBM Press, 1999.

[9] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. 1998.

[10] Y. Zhang, C. Dragga, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Viewbox: Integrating local file systems with cloud storage services. In *FAST'14*. USENIX, 2014.

[2] In a mobile environment, all devices are behind firewalls, which makes it difficult to directly communicate with each other.