BlueMountain: An Architecture for Customized Data Management on Mobile Systems

Sharath Chandrashekhara, Taeyeon Ki, Kyungho Jeon, Karthik Dantu, Steven Y. Ko
Department of Computer Science and Engineering
University at Buffalo, The State University of New York
{sc296,tki,kyunghoj,kdantu,stevko}@buffalo.edu

ABSTRACT

In this paper, we design a pluggable data management solution for modern mobile platforms (e.g., Android). Our goal is to allow data management mechanisms and policies to be implemented independently of core app logic. Our design allows a user to install data management solutions as apps, install multiple such solutions on a single device, and choose a suitable solution each for one or more apps. It allows app developers to focus their effort on app logic and helps the developers of data management solutions to achieve wider deployability. It also gives increased control of data management to end users and allows them to use different solutions for different apps.

We present a prototype implementation of our design called BlueMountain, and implement several data management solutions for file and database management to demonstrate the utility and ease of using our design. We perform detailed microbenchmarks as well as end-to-end measurements for files and databases to demonstrate the performance overhead incurred by our implementation.

1 INTRODUCTION

Mobile systems have gradually become the predominant platform for everyday computing, and their computational, memory, network, and storage capabilities are getting powerful by the day. Apps on mobile systems span diverse categories such as games, personal activity, banking, productivity, and business applications. Consequently, mobile apps employ sophisticated data management solutions, including local file and database storage as well as cloud services for user/app data. These solutions enable desirable features such as backup, sync, and sharing.

However, data management in mobile apps is currently rigid, and does not provide enough flexibility for their developers and users. Specifically, individual app developers need to make policy and mechanism decisions on how user and app data should be managed. Examples include, but not limited to, which cloud provider to use for data backup (e.g., Dropbox vs. Google Drive), what policies to use when transferring data over the Internet (e.g., only when re-charging, only over Wi-Fi, etc.), and what mechanisms to use for synchronization (e.g., delta update, de-duplication, etc.). As a result, developers have to spend additional effort in putting their data management logic into each app, and users do not have any control over how their data is managed. Some app developers expose these decisions to users as configuration options, but those options are typically limited. In short, current mobile apps have a tight coupling between data management and app logic, which leads to added development time for developers and inflexibility for users.

In order to overcome this problem, we envision a new ecosystem where there are two kinds of apps. The first kind of apps are regular apps that one can find currently from online app stores such as Google Play. The second kind is what we call *data management apps* that are also downloaded from online app stores. However, these data management apps are meant to be *pluggable* and always used in conjunction with regular apps for the purpose of managing their data. A regular app delegates all its data management decisions to a data management app, and users choose a data management app to use for each regular app depending on their needs. For example, a user in this ecosystem could install a regular note-taking app with a data management app that backs up all notes to a public cloud. At the same time, another user could install the same note-taking app, but with a different data management app that uses a private cloud instead of a public cloud. This new ecosystem achieves decoupling of data management and app logic, and overcomes the current limitation of rigid data management in mobile apps.

In order to make this vision a reality, we have built BlueMountain, a framework for pluggable data management apps on Android that has three distinctive features. First, BlueMountain defines a clean interface for data management apps, which mimics the interface for Android's filesystem, database, and key-value storage (`SharedPreferences`). Using this interface, separate data management apps can be written solely for the purpose of data management such as backup, sync, and sharing. Second, BlueMountain enables dynamic binding between a data management app and a regular app, so that a user can choose which data management app to use for each regular app the user has. BlueMountain achieves this by dynamically loading a data management app's code into a regular app, and redirecting the regular app's storage calls to the data management app's code. Third, BlueMountain automatically transforms existing apps to leverage BlueMountain's functionality via Java bytecode instrumentation. BlueMountain's instrumentation finds all uses of local storage APIs in an Android app (filesystem APIs, database APIs, and `SharedPreferences` APIs) and translates them, so that it automatically uses BlueMountain-injected code. This means that regular app developers do not need to bother integrating BlueMountain for their local app data. Users can also benefit from BlueMountain immediately as legacy apps can be

MobiCom '17, October 16–20, 2017, Snowbird, UT, USA

ACM ISBN 978-1-4503-4916-1/17/10…$15.00
https://doi.org/10.1145/3117811.3117822

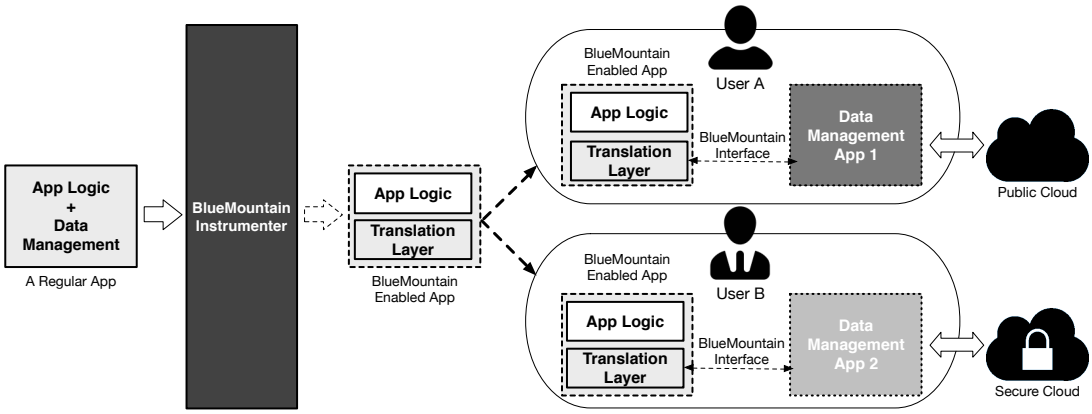

Figure 1: BlueMountain Workflow Example

BlueMountain-enabled automatically. Figure 1 shows a visualization of the envisioned workflow with BlueMountain.

To demonstrate the usefulness of our system, we have built five different data management apps, each of which implements one strategy for managing regular app data. The first data management app implements a personal data storage system, inspired by PSCloud [2], which uses all of the user's devices to store data. We use WiFi Direct [40], local network, and the Internet to communicate with a user's mobile devices, desktop, and cloud, respectively. The second data management app implements an efficient data synchronization mechanism, inspired by QuickSync [8] and implements file chunking to improve network utilization and reduce storage footprint. Further, we implement three mechanisms to provide database backup in the cloud. The first strategy locally logs all database operations and remotely replays the log for data backup. The second strategy divides a database file into multiple chunks, keeps track of modifications made to each chunk, and backs up individual chunks with new changes. This is inspired by the rsync library [31]. The last strategy automatically performs database sharding on Android, i.e., storing and managing each database table as a separate file. This strategy can be used in conjunction with other strategies for more efficient database backup.

In order to evaluate BlueMountain, we measure the overhead using microbenchmarks as well as the five data management apps described above. Our results show that the performance degradation is minimal (<10%) in most of the cases and stays less than 30% even in the worst case. This can be further reduced by using a more efficient memory management in our system.

We also have downloaded 400 apps from Google Play and automatically transformed them to leverage BlueMountain. Our instrumentation results show that BlueMountain's instrumentation does not incur much overhead in terms of code size increase (<12%), heap usage (<16%), and energy usage (<3.5%). We have also tested the correctness of our instrumentation via an automated UI testing system and verified that BlueMountain's instrumentation neither crashes apps nor throws new exceptions.

In summary, we make the following contributions:

- We show that it is *feasible* to decouple data management and app logic for mobile apps. We do this by proposing a new framework that enables the decoupling, as well as showing that the framework incurs minimal overhead.
- We demonstrate that the decoupling is *useful* by creating five data management apps and showing their practical performance. This includes re-implementing two file-based solutions previously proposed for mobile systems.
- We show that our approach *supports existing apps* by automatically transforming 400 apps downloaded from Google Play to leverage our framework and verifying the correctness of their functionality.

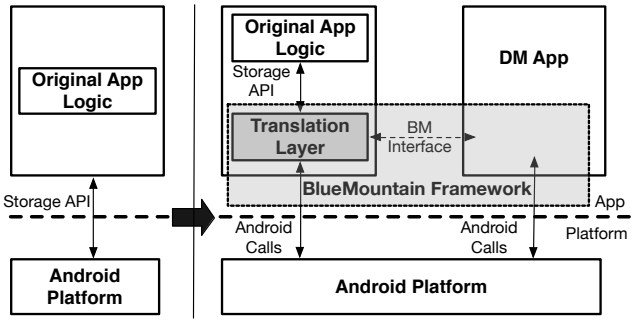

Figure 2: BlueMountain System Overview

2 BLUEMOUNTAIN DESIGN OVERVIEW

BlueMountain is designed with three goals in mind—*decoupling of data management and app logic*, *supporting widespread deployment*, and *supporting legacy apps*. We overview these goals and our design in this section.

2.1 Data Management and App Decoupling

Figure 2 shows how BlueMountain decouples data management and app logic. Shown on the left is a regular Android app that uses the Android storage APIs to store data locally. Shown on the right is the architecture enabled by BlueMountain that has a separate data management app. BlueMountain defines a clean interface (which we call the BlueMountain interface) that data management app developers use to provide their data management solutions. BlueMountain also has the translation layer, which translates all

Android storage APIs into the BlueMountain interface and redirects all storage API calls to a data management app. We note that for each regular app, we dynamically load a data management app's code into the same address space of the regular app. Figure 2 shows a regular app and a data management app side-by-side only to highlight our logical decoupling. It is not meant to show that they run as separate processes. Section 3 details the BlueMountain interface as well as the translation layer.

This design provides two benefits that overcome the limitation of the rigid data management that current mobile apps experience. First, it *simplifies development* for regular app developers since they do not need to implement various policies and mechanisms for data management. They can focus on their app logic and delegate the complexity of data management to separate data management apps. Second, it provides *flexibility* to users since users can choose different data management apps depending on their needs. This is possible since BlueMountain allows the binding between a regular app and a data management app to occur at *use time* instead of development time.

2.2 Supporting Widespread Deployment

BlueMountain supports widespread deployment by restricting all its implementation to take place within individual apps. This means that we do not modify anything in the Android platform—this is because modifications to the Android platform are tightly regulated by Google as well as other mobile platform vendors such as Samsung and Amazon. If any platform modification were to be required for BlueMountain, we would need to rely on Google or other vendors to approve the use of our system and properly distribute it, which is impractical. In contrast, if all modifications were restricted to individual apps, we could widely distribute our system by using an online app store such as Google Play. Therefore, BlueMountain restricts *all modifications to be self-contained in individual apps* and makes no changes to the Android platform.

2.3 Supporting Legacy Apps

Android is a popular platform for app development, and many app developers are now used to its programming model and interfaces for app design. If we were to require app developers to use a new storage interface for BlueMountain, it would involve learning and hinder the pace of development. Further, previously-developed apps would need to be re-written using the new interface, which puts an excess burden on the developers. To mitigate both of these challenges, we use Java bytecode instrumentation to automatically transform an app to leverage BlueMountain. For this purpose, we use Reptor [23], a tool that enables automated API class replacement for Android apps. Reptor takes an app binary, decompiles it, performs program analysis, correctly replaces Android API classes with custom classes, and recompiles it into a new app. This makes the process of replacing Android storage APIs with BlueMountain APIs seamless and transparent to regular app developers.

We now describe the BlueMountain system implementation.

3 SYSTEM IMPLEMENTATION

The BlueMountain architecture is shown in Figure 3. Any app that wishes to use our system has to include our framework within the app and use our interface for all data storage functionality. For this, we provide a modular implementation of our framework that can be used as a library. Given such a library, we use an instrumentation tool (Reptor [23]) to automatically inject the whole BlueMountain framework into existing apps without developer's intervention. This allows the existing apps to easily benefit from BlueMountain.

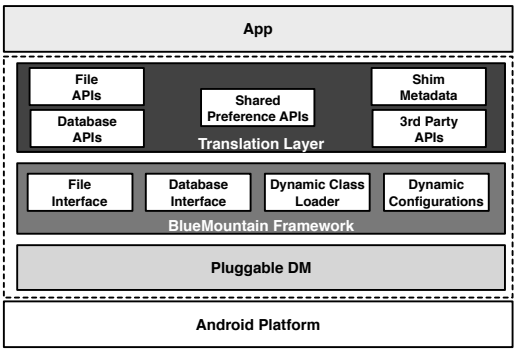

Figure 3: BlueMountain System Architecture

As described earlier, the BlueMountain framework dynamically loads a data management app's code. We define a clean interface for a data management app to implement, and our translation layer translates all Android storage API calls to the calls to our own interface. Below, we first describe our BlueMountain interface. We then describe our translation layer. Lastly, we describe how we inject the BlueMountain framework and dynamically load a data management app's code.

3.1 The BlueMountain Interface

We define a clean interface for data management apps which mimics the three storage options that Android provides. They are (a) files, (b) databases, and (c) key-value stores called `SharedPreferences`. The reason why we mimic Android's storage options for our interface design is to allow data management app developers to provide customized solutions for different storage options available on Android. BlueMountain intercepts all calls to these storage options and translates them into our own interface calls, so that the data management apps can handle them.

Table 1 describes our interface for data management apps. We design our interface using *objects* as our storage abstraction instead of files, which helps us keep our interface design simple. Our data objects are of a fixed size, but it can be configured by the individual data management apps. By default, we use 4MB, which is a typical choice for cloud vendors like Dropbox [11]. This means that if an app stores a single file, BlueMountain divides into multiple 4MB chunks. Below, we discuss how we use the object abstraction in our interface that mimics the Android storage APIs.

Files: We provide a simple CRUD interface for the data management apps that wish to manage files. Our read and update interfaces include two optional parameters to indicate the offset within the object and the requested length. This allows our interface to mimic Android APIs when apps want to optimize operations on files. Generally, the flush operation on a `FileIOStream` is a NOP. However, apps can force a `fsync` using low level Java constructs like `FileChannel` or `FileDescriptor`. To support this behavior, we add

Type	Interface	In	Out	Description
File: CRUD	fcreate	id, buffer[], length	none	Creates an object id of size length from buffer bytes
	fread	id, buffer[], [offset, length]	buffer[]	Reads data from object id into buffer. offset and length are optional parameters indicating the offset within the object and the length to read
	fupdate	id, buffer[], [offset, length]	none	Updates the object id with buffer bytes. offset and length are optional parameters indicating the offset within the object and the update length
	delete	list: id	none	Deletes all objects in the list
File: State	close	list: id	none	Indicates operations on all objects in the list is completed
	flush	list: id	none	Flushes all objects in the list to a persistent storage
	initFS	none	none	Initializes the system
	clearFS	none	none	Deletes all the data and formats the system
DB: State	open	db-path, flags	cnx-id	Opens the database in db-path with the provided access flags in flag and returns the connection id cnx-id
	close	cnx-id	none	Closes the previously opened connection cnx-id
	cancel	cnx-id	none	Cancels the current operation on the connection cnx-id
DB: Execute	executeWrite	cnx-id, SQL-String, args[]	rows	Executes statement SQL-String which modifies database, with args an Object array of bind arguments, on connection cnx-id. Returns rows, a row id or number of rows modified
	executeQuery	cnx-id, SQL-String, args[]	Cursor	Executes SQL-String query with args as an Object array of bind arguments, on the connection cnx-id. Returns Cursor for the result.
DB: Info	statementInfo	cnx-id, SQL-string	Info	Returns information about statement SQL-String on connection cnx-id in an Info Object

Table 1: BlueMountain Interface for Data Management Apps

flush to our interface. Since the app calls flush at the file level, this may affect all the objects belonging to a file, including the metadata file. Therefore, our flush operation takes a list of object ids which have to be flushed. Similarly, we include close as part of our interface. This provides a mechanism to indicate to the data management app that the app is done operating on the object, it can release the resources and flush it to a persistent storage.

Databases: To support databases, we provide open, close and cancel methods in our interface. Open returns a connection ID which all the subsequent calls should use to perform operations on the same database. For the execution of a database operation (e.g., insert, update, delete, query, etc.), we provide two methods—`executeWrite` and `executeQuery`. `executeQuery` is used in queries, `executeWrite` is used for insert, delete, and update operations and all other SQL operations. These two methods accept a raw SQL statement string as a parameter, which provides the most flexibility for a data management app in handling SQL operations. The `statementInfo` method allows us to get useful information about a SQL string from the database such the number of columns, the column name, or if a particular query is a read-only query.

There are certain types of database APIs that our current interface does not support—(1) APIs for controlling database queries issued from multiple threads, (2) APIs for compiled SQLite statements, and (3) APIs that use an app-provided `CursorFactory`, which handles query results in a customized fashion. These features are *not essential* for using a database but rather provide more support for app programmers. For example, multi-threading support provides an easier way to control queries that are executed in parallel from different threads; compiled SQLite statements make it easier to reuse frequently-issued queries; and `CursorFactory` provides support for customized output access. For each of these features, there are alternative ways for app programmers to accomplish the same tasks.

We can still support these features by adding them to our interface. However, our current prototype does not support them in the interest of keeping our interface clean and simple. Adding these features to our interface means that each data management app will need to support the features. For example, in order to support multiple threads, a data management app should be able to handle queries properly even if they are issued by multiple threads. Similarly, in order to support compiled SQLite statements, a data management app should be able to generate compiled SQLite statements even when it does not use a SQLite database underneath.

In Section 5.4, we present our detailed analysis on how many apps use these features.

SharedPreferences: `SharedPreferences` on Android is implemented as an XML file and is typically a small file where apps store key-value pairs like user settings. We can treat the XML file as any regular file and use our file interface to read and write the `SharedPreferences` file through the data management app. Thus, we do not provide a new interface to deal with `SharedPreferences`.

3.2 BlueMountain Translation Layer

The translation layer translates the Android Storage API calls into the BlueMountain interface calls. It is either directly built into an app (for newly-developed, BlueMountain-aware apps) or injected into it at a later stage (for legacy apps). Architecturally, we separate the translation layer from the rest of the framework. This is because this design enables third party data management libraries to be plugged into our system if they can provide a translation from their APIs into BlueMountain APIs. Below, we discuss the strategies for translation we have used for Android storage APIs.

Files: The translation layer needs to translate the Android file operations into the CRUD interface shown in Table 1. The files on Android are accessed through the following classes—`FileInputStream`, `FileOutputStream`, `FileReader` and `FileReader` for sequential access; `RandomAccessFile` for random file access; `File` for metadata, `FileDescriptor`, `FileChannel` as low level representations of a file and `FileLock` for locking. This requires us to translate the streaming I/O model into an object-based model. We have developed wrappers around the `FileInputStream` and `FileOutputStream`, which translates the calls to an object based I/O. As mentioned earlier, we have chosen the default object size to be *4 MB*, but this is a configuration parameter. Therefore, the translation layer also needs to break down files larger than 4MB into objects of 4MB chunks.

This also requires us to maintain mappings from files to objects as metadata. The metadata itself can be treated as an object and stored persistently using the same data management app that is plugged into the app. Since object size could have a significant impact on performance, we allow the data management app to change the default object size according to its design.

Our current interface translation supports most file operations on Android. However, there are some obscure lower-level representations of files through the `FileChannel` and `FileDescriptor` classes that we currently do not handle. Our preliminary examination of real apps suggests that most apps do not use these interfaces for data management. Therefore, we leave the handling of these interfaces as part of our future work.

File Metadata: Apart from metadata file which maintains the mapping from files to objects, we need a mechanism to deal with the system metadata as well. This includes all the operations from the Java `File` interface. Moving the management of this metadata to the data management app makes the design more complex, and is not particularly advantageous. Therefore, in our design, we let the Android framework deal with the filesystem metadata. For example, operations such as creating directories, empty files, maintaining access times of files, directory structures, permissions, etc., are left to the Android system. We do this by creating the entire directory structure with empty files using the Android platform interface whenever a file is accessed through the BlueMountain translation layer. Thus, while the translation layer through the BlueMountain framework delegates the responsibility of handling the file data to the data management app, the associated metadata is maintained locally.

Databases: In order to use a SQLite database, apps use the `SQLiteOpenHelper` and the `SQLiteDatabase` classes to open and issue requests to the database. These app-facing classes provide a variety of functionality not just to read and write from the database but also to perform database management. In our design, we intercept calls to these classes and translate them into our own.

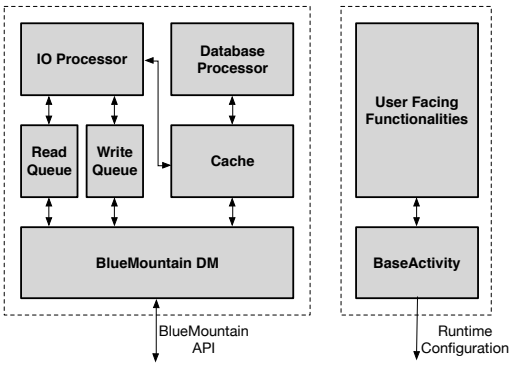

Figure 4: BlueMountain Data Management App SDK

For the Android database APIs that ultimately issue a database operation (e.g., insert, update, delete, query, etc.), we interpret and translate them into raw SQL strings and invoke either `executeWrite` or `executeQuery` (which we discuss in Section 3.1).

SharedPreferences: To handle SharedPreferences, we provide wrappers to access and edit the SharedPreferences and notify the data management app of any changes made to the corresponding XML file. While serving the reads, we check if the SharedPreferences file has been updated outside of the app, and if it is, we simply reload the SharedPreferences file.

3.3 BlueMountain Data Management App SDK

In order to support the development of a data management app, we have developed an SDK which provides common functionality and a base implementation for all APIs. Any data management solution that wishes to override the default behavior can override one or all the methods of the SDK or can simply extend the functionality that is provided. Figure 4 shows an overview of the SDK and the modules. For files, the base implementation writes all operations to the local storage cache with a FIFO policy, and adds them to the write queue. It also serves the reads from the cache, raising an exception when data is not found. The requested object is then added to the read queue. A background thread dispatches these requests to the I/O processing queue. The data management apps need to extend the I/O processor to handle these requests. For databases, the base implementation uses a local database as a cache and the data management apps have to override this behavior to provide additional functionality.

The SDK also provides a `BaseActivity`—a UI which identifies a set of BlueMountain enabled apps installed in the system and lets the user choose a regular app to which the data management app has to be plugged. A data management app can extend this functionality to enable runtime configurations. The data management apps can pass on these runtime configurations to the BlueMountain framework (running within the regular apps) using Androids `Intents`. The data management app might also implement other user functionality that allows users to modify some of these configurations—for instance, an interface that allows users to

provide access to a particular cloud account or an interface to set the network data limit.

Our current version of SDK provides an implementation to handle object chunking, data caching, and a client to access cloud servers over a REST-based interface. In the future releases, we intend to add more functionality to the SDK to aid the development of data management apps.

3.4 Dynamic Loading and Instrumentation

At run time, we enable the data management apps to be plugged into any regular app having the BlueMountain framework. We achieve this by dynamically loading the data management app's code into the regular app. We use `DexClassLoader`, a Java class that Android provides, which allows an app to dynamically load another app's code. Once loaded, we use Java reflection to invoke data management app code. Such a mechanism binds the data management app with the regular app at runtime allowing the user to configure any regular app with a data management app of her choice. The overhead due to reflection can be mitigated by using a pre-defined interface and invoking the data management app methods through this interface. As we show in Section 5, the overhead due to reflection did this way is fairly minimal.

If a regular app is not compiled with the BlueMountain framework, we need to inject the BlueMountain framework into the app before a data management app can be plugged into it. We use Reptor [23] for this purpose. Reptor enables *API class replacement* for an Android app—a developer can replace an API class (such as `FileInputStream`) with a custom class that the developer writes. Thus, we use Reptor to replace all storage API classes with our own classes, which redirect all storage API calls to our translation layer.

4 PLUGGABLE DATA MANAGEMENT APPS

In this section, we discuss the design and implementation of five pluggable data management solutions built using our SDK and BlueMountain framework. We discuss two solutions that handle files and three solutions that handle databases. Our intent is to demonstrate the ease of implementing data management solutions using BlueMountain. Our first two data management apps are based on two previously published systems, PSCloud [2] and QuickSync [8]. For databases, we implement three strategies for efficiently synchronizing an Android app database with the cloud as separate data management solutions for databases. The key-value store is stored as a file. One of the two data management solutions for files can be used for the key-value store as well.

4.1 Personal-Device Cloud

Our first use case system is a data management solution for personal storage cloud inspired by PSCloud [2]. Briefly, PSCloud envisions the use of storage on all mobile devices of a single user, along with home servers and cloud storage services. It creates a single unified personal storage system where data is automatically cached, replicated, and placed to enable reliable access across all devices. It also minimizes network access and storage costs using a per-device network context graph that tracks connectivity relationships between a user's devices.

Inspired by the design of PSCloud, we have implemented a data management app that is capable of performing file transfers using three mechanisms—WiFi Direct (i.e., peer-peer wireless link between devices), desktop sync (i.e., local network access to desktop when possible), and Dropbox sync (i.e., cloud access when network connectivity is available). Our current implementation employs a simple file placement algorithm—a file is copied to other devices based on a device priority list. Currently, we give other mobile devices the highest priority. If no device is found, the system attempts to copy the file to a desktop. If that is also not available, the user's Dropbox account is used for backup (when network is available). Although we have implemented this policy based on the brief description from PSCloud [2], the file placement algorithm itself is orthogonal to our objectives and more sophisticated techniques can be implemented using our framework for improved efficiency. Once again, our intent is to demonstrate the feasibility of implementing previously proposed mobile data management systems using the BlueMountain framework.

4.2 Efficient Data Synchronization

Our second data management app is an efficient cloud data syncing system inspired by QuickSync [8]. QuickSync uses strategies such as chunking, bundling, delta encoding, de-duplication, and data compression to create an adaptive chunking and de-duplication strategy based on network conditions. Its goal is to reduce network traffic and have uninterrupted data syncing. It uses separate data and control servers. Inspired by QuickSync, we have implemented a data management app that efficiently syncs files by breaking them into finer chunks, and use a local Desktop as a control server to store metadata and Dropbox as a public cloud data server. Though chunking is just one of the optimizations provided in QuickSync, our simplified implementation demonstrates the ability to develop a system like QuickSync as a pluggable data management app for BlueMountain framework. Again, we do not attempt to replicate all of the advanced algorithms described in the original work as they are orthogonal to our design objectives.

When QuickSync is implemented as a data management app, it receives explicit notifications from the BlueMountain framework whenever an object is modified or read. In our implementation, when we receive a create notifications from the BlueMountain framework, we split the object into multiple chunks, update the metadata server with this information, and upload the chunks to the cloud. On read notifications, we first fetch the metadata and then fetch the required chunks if it is not present in the local cache. This differs from the library implementation where the reads will be directly handled by the Android framework. Similarly, on update notifications, only the affected chunk is uploaded and the metadata is updated.

4.3 Database Management

Our next set of data management apps are implementations of efficient backup mechanisms for databases. There are various libraries which assist in backing up databases ranging from transferring the database as a JSON string to using a rsync-like library to continuously back it up in the cloud. A sync library should be fast and efficient in using network bandwidth and energy. We implement

three approaches for backing up the database with the cloud, each one designed as a data management app.

Log and Replay: Our first app for backing up a database to a server is to log all SQL statements on the mobile device and reconstruct the database on a server by replaying the logs. On the execution of every SQL statement, we synchronously log the SQL statement into a journal file. A background thread which monitors this file uploads file updates to a server. The server, after reading the statements, replays them on its SQLite server to recreate the database. Whenever there is data loss or when the user wants to restore to a previous version of the database, we can simply download it from the server and use it. Our current implementation of this works as a sync-only system, where we still maintain a local copy of the database and serve all the queries through it.

Backing Up The Database File: In our second database management app, we implement an approach similar to librsync, where we break up the database file into multiple virtual chunks. We monitor the database for every change, identify the chunk that was modified using a hashing function and upload that chunk. On the server side, we reconstruct the original database by merging the chunks. While we have used this approach as an example for database backup, this can be a light-weight solution to backing up any file between the device and the cloud.

Multiple Databases: In our third database management app, we split the database into multiple databases, each containing just one table. This is commonly referred to as database sharding in web services domain. Separate database files are created whenever a new table is created with the corresponding table in it. SQLite supports the SQL ATTACH which lets multiple databases to be accessed as one virtual database through a single connection. Using this, we attach all the databases, each containing one table and access it through a single virtual connection. SQLite can support up to 10 attached databases and can transparently handle modifications to the table.

The biggest advantage of this approach is that an update on a table will modify only the corresponding database file, and other database files will be untouched. This would enable us to use other back up mechanisms more efficiently as the database files are now smaller and updated independently. To enable this, our implementation starts with one database for the first table created and examine the SQL operations. On every create table operation, we create a new database and attach it to the existing database. One complication in our implementation is that this approach cannot handle foreign keys from different tables due to limitations of SQLite; tables with a foreign key relationship cannot be placed in two different database files. Thus, our implementation detects foreign key relationships between any two tables, and whenever a new table is created, we insert the table to the same database file that has the corresponding foreign key table.

5 EVALUATION

We evaluate BlueMountain in four aspects. The first is performance—we use microbenchmarks to show that BlueMountain incurs very little overhead (Section 5.1). The second is usefulness—we measure the performance of our prototype data management solutions for both files and databases, and show that they provide comparable performance to current Android storage APIs (Section 5.2). The third is support for automation—we inject BlueMountain into 400 existing apps, show that they work correctly under our testing scenarios, and report instrumentation statistics as well as energy and heap consumption for some of the well-known apps (Section 5.3). Lastly, we analyze 2000 apps and provide an insight into improvising the support for databases (Section 5.4). All our experiments, except when noted, were conducted on Google Nexus 5 phone, running Slim Rom [30] version of Android 6.0.1 (API 23).

5.1 Microbenchmarks

We first measure the performance of BlueMountain using microbenchmarks. As BlueMountain framework introduces multiple layers, we expect some performance degradation compared to directly using the Android APIs. The overhead can occur at two layers. First, due to our API translation layer, every storage API gets translated to our APIs. We call this overhead the *framework* overhead ("B framework" in the plots). In order to quantify this overhead, we intercept all storage APIs and translate it into our APIs, but instead of redirecting the calls to a pluggable data management app, we directly perform local I/O within our system. This also serves as a default data management policy when a user does not install any data management apps.

The second overhead is due to invoking a pluggable data management app. As we use Java reflection in this layer, we expect to have some overhead. We call this the *pluggable system* overhead. In order to quantify this overhead, we create a data management app which just uses the local storage on the phone through the Android Storage APIs and perform our measurements. This overhead indicates the base overhead incurred by any pluggable data management app ("DM App" in the plots).

Sequential I/O: In this section, we evaluate the overhead of BlueMountain for sequential file I/O. Figures 5 and 6 shows the total time taken for writing and reading files of 5 different sizes plotted as a CDF. [1] "Native" is the I/O performance through Android storage API classes, `FileInputStream` and `FileOutputStream`. Once again, "B Framework" indicates the performance of the BlueMountain framework and the translation layer, and "DM App" indicates the performance over a plugged in data management app. We conducted the experiments by measuring the average time taken for write/read for files of different sizes over 1000 iterations. We have used an object size of 4MB for our translation layer. This parameter can be configured by the pluggable data management app and our system used 4MB as the default settings. We study the impact of this object size in Section 5.1. Our write tests include flushing the data to persistent storage, while we use warm cache for our read performances. Photos and videos which are the most common type of data on mobile apps mainly use sequential I/O. Thus our tests are reflective of the performance we can expect on real apps. The performance numbers show that BlueMountain incurs minimal overhead over the native storage APIs. Further, we can see that the pluggable layer barely has any overhead above the framework overhead and has an almost-identical performance. This is because the two experiments are functionally equivalent as the

[1] We have also conducted the same experiments for file sizes up to 100MB, and the performance had the same pattern; thus, we do not include the results here.

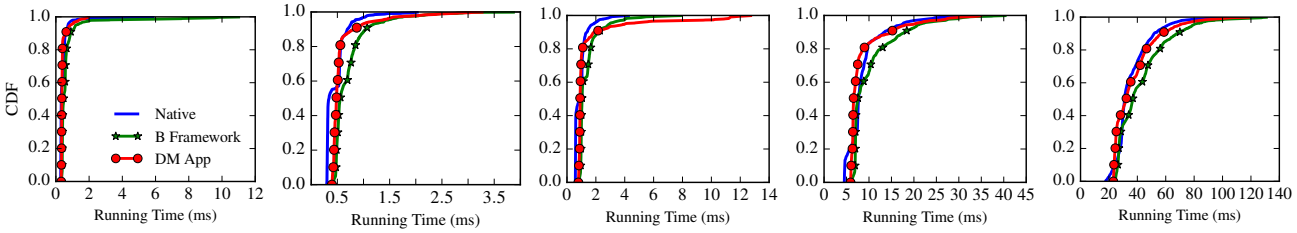

Figure 5: CDF: Sequential Writes with Various File Sizes, 4KB, 20KB, 100KB, 1MB, and 4MB (left to right)

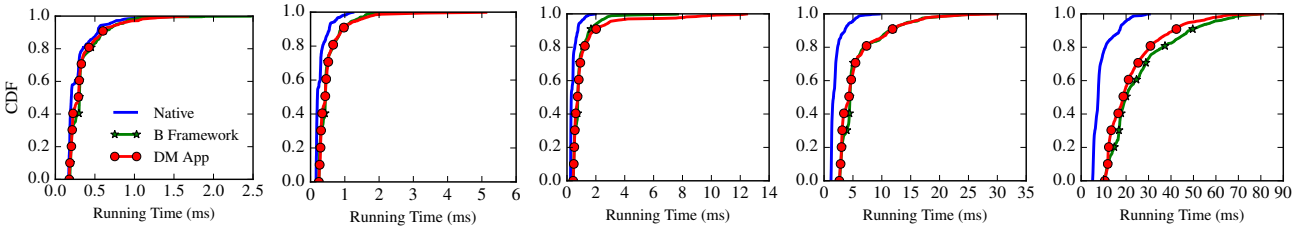

Figure 6: CDF: Sequential Reads with Various File Sizes, 4KB, 20KB, 100KB, 1MB, and 4MB (left to right)

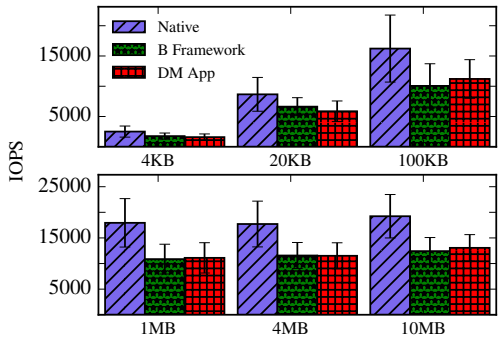

Figure 7: 4K IOPS for Random Writes

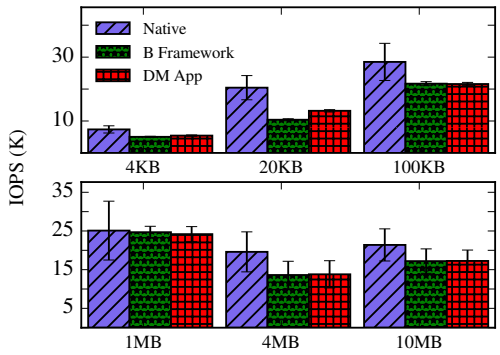

Figure 8: 4K IOPS for Random Reads

overhead of reflection is mitigated due to the use of our pre-defined interface as described in Section 3. The higher overhead for larger files is likely due to a higher memory pressure on the system; our current implementation allocates one large memory chunk for all data and does not perform any optimization. We expect that our future implementation can optimize this further.

Random I/O: In our next measurement, we evaluate our system against random I/O operations. We measure the number of IOPS by issuing I/O operations at a size of 4KB, which is typically the smallest size an app would read/write from/to a filesystem (4KB is the default block size of the ext4 filesystem—Android's default filesystem). Like the previous experiment, we measure the IOPS for files of different sizes over 1000 iterations and then calculate the average and standard deviation. Although random file access is not as common as streaming I/O, they are still used by plenty of apps, typically for accessing a documents or data files. BlueMountain's random access interface is based on Android's `RandomAccessFile` API. We measure the overhead of our system and compare it with the performance of when `RandomAccessFile` is directly used. Figure 7 and Figure 8 shows write and read IOPS respectively and compare Android's native performance, BlueMountain framework performance and the pluggable data management's performance. We incur some overhead on random access compared to the native performance. This is because Android's `RandomAccessFile` directly uses the low level file primitives while we build the random access support on top of a object based model. This means random writes on BlueMountain will results in updates to the object, which are typically slower. This is an optimization problem rather than an architectural limitation. For instance, in a app which primarily uses random I/O a much more optimized `update` which can handle small random I/O can be implemented by using techniques like chunking. The performance can also be improved by tuning the object size of the BlueMountain layer. As this parameter configurable, it enables the data management app to choose the best size depending on the targeted workload.

Effect on Object Size: As discussed in our system architecture, we divide large files into smaller objects of a particular block size. The choice of this block size will have an impact on the I/O performance. To measure the impact of the block size on the system performance, we vary the block size between 256KB and 8MB and

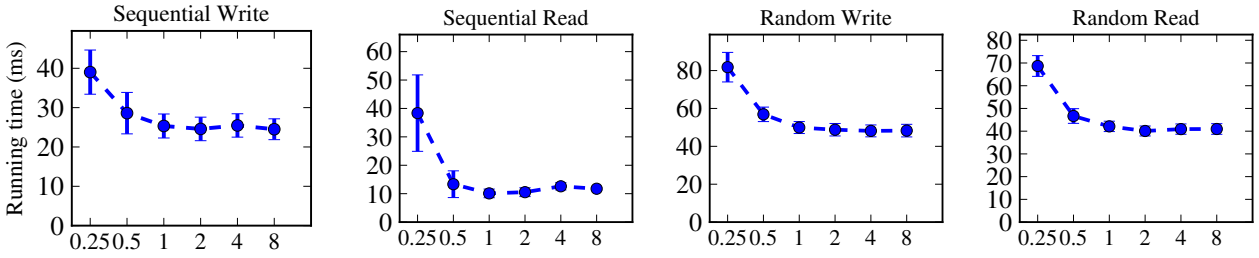

Figure 9: I/O performance of 4MB file for various block sizes (The x-axis shows block size in MB)

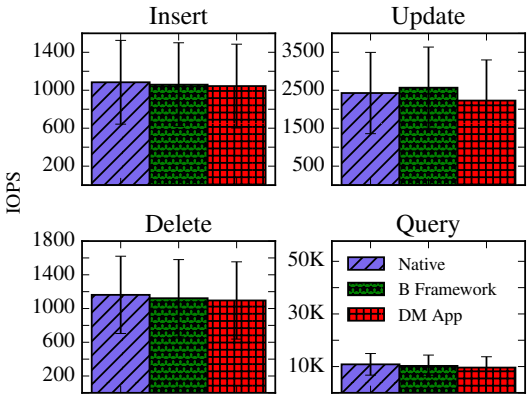

Figure 10: Database Microbenchmark

measure the I/O latency for a file of 4MB, for both sequential and random I/O. For random I/O, we issue I/O at 4KB chunks.

Figure 9 shows the performance for sequential and random I/O. With smaller block sizes, BlueMountain has to create multiple objects for the same file and I/O involves reading and writing to multiple objects. This adds additional overhead in our translation layer and degrades the performance. For instance, when the object size is 512KB, our system creates eight objects for a 4MB file. Hence, the overhead is higher compared to the case where we use the object size of 4MB—this is due to the difference in the number of objects that need to be created. With the object size of 8MB, there is no improvement as we still end up creating only one object. However, smaller objects might be beneficial with an app that issues many random I/O operations with frequent updates, since each update operation only needs to affect an object of a small size. Smaller object sizes can also reduce the network sync traffic if the app frequently updates the files after they are created.

Database operations: We have measured the overhead for database operations on Android's native database interface, with BlueMountain framework, and with a pluggable data management app. Just as in files, for framework experiments, the database operations go through our translation layer, and instead of redirecting the database operations to the data management app, we execute these operations on a local database. Similarly, to measure the pluggable overhead, we have developed a data management app for database which just uses a local database and Android's `SQLiteDatabase` interface to access it. Figure 10 shows the measurements in operations per second and for the four basic database operations—insert, update, delete and query. We have performed these operations on a table with 5 columns and having 10 rows to start with. We run 1000 iterations for each operation and measure the average and standard deviation. As we can see from the measurements, the overhead due to our framework and pluggable system is fairly minimal.

System	Throughput (write)
Native Android	68
Personal Device Syncing app (4.1)	52
Efficient Cloud Syncing app (4.2)	60

Table 2: Full Stack File Throughput (MB/s)

5.2 End-to-End BlueMountain Evaluation

We have measured the end-to-end throughput by plugging in the different data management solutions we have created as explained in Section 4. When the data management systems are plugged in, the app might experience slowdown due to two reasons. First is a combination of the "framework" and "pluggable" overhead as evaluated in our microbenchmarks. Second is due to the slowdown of the system when the data management apps are working in background. To quantify these, we have measured the throughput of our system, as MB/s writes for files and as the number of database operations per second for databases. We compare both these measurements with the corresponding measurements obtained from directly using the Android file/database APIs for local I/O—the goal here is to measure the maximum performance degradation that an app can experience when a data management app with cloud syncing abilities is plugged in.

Files: Table 2 shows the throughput for the two file data management systems we discussed in Section 4.1 and 4.2 and compare them with the native file throughput. We plugged in the two data management systems to our test suite app, one at a time, and enabled syncing to Dropbox and a private server which was hosted on our local network. We issued a large write from our test app, so the data management apps were continuously syncing the data to the cloud in the background. When the app is under full load, we measure the time taken to write a 1GB file. As we can see in the table, there is a slight degradation in the throughput compared to the native on both our file data management systems. We attribute this to the overhead caused by a background task which was actively syncing data to Dropbox and local server as described in Section 4.

Database Syncing Systems: Next, we show the performance of the three database syncing systems we have developed for BlueMountain, as previously described in Section 4.3. The goal of these systems is to synchronize the database with a remote server in the

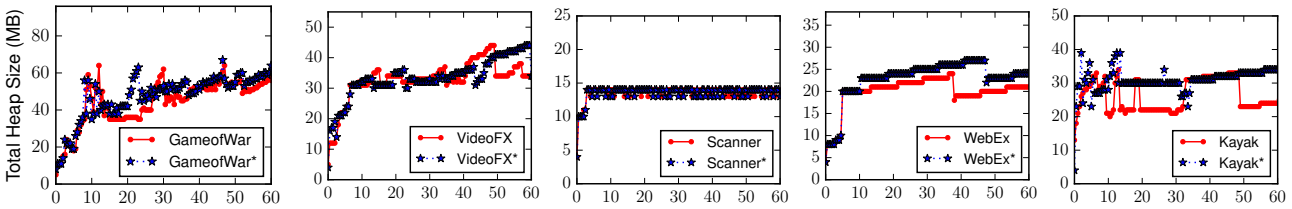

Figure 11: Heap Usage (*Instrumented apps): The x-axis shows elapsed time in seconds.

System	Insert	Query	Update	Delete
Native Android	1782	6300	1865	1835
File-Sync	764	6245	803	486
Log and replay	1367	6106	1485	1374
Multiple Databases	444	5188	417	407

Table 3: Full Stack Database Throughput (IOPS)

background and we capture the performance overhead of these systems when they are under full load and syncing data in the background.

For these measurements, we have a similar experimental setup for all three systems. We create eight tables, each having five columns and fifty rows. Our testing app, for which we plug in one database management app at a time, will run the four database operations—insert, update, delete and query, for 1,000 iterations. We then repeat the same experiment by directly using the Android's database APIs. All our database management systems synchronize data with a server. For our measurements, we used a local server hosted on our local network. Table 3 shows the throughput of four database operations using the three data management systems discussed in Section 4.3 along with the Android's native database APIs. We can see the throughput for write operations with these data management apps are considerably lower than that of the native database. In the next three paragraphs, we explain the possible reasons for this.

Log and replay: The log and replay system (described in Section 4.3) will synchronously log SQL operations to a persistent log-file on every database write operation. Since the log file is synchronously updated, we can observe a noticeable overhead for operations that modify the database—insert, update and delete. As queries are not logged, there is no additional overhead and the performance is comparable to the native performance. We also have a background task that continuously syncs the log-file to our server. When the system is under full load, this would affect the performance as well.

File sync: The file sync system (described in Section 4.3) has to notify a background task whenever a database write operation is issued. This, along with the computationally intensive background task of identifying the file chunks that are updated on each database operation, adds to the overhead. Like the log and replay, the queries are relatively unaffected by this and is comparable to the native system.

Multiple databases: In the multiple database system (described in Section 4.3), the overhead mainly comes from using multiple databases through a single connection. Due to this reason, the way database is accessed is changed and we can observe a higher penalty on queries as well. Like the previous two systems, this system is also continuously syncing the database files, but in this case, only the file which was affected by the operation. Thus, like the previous two database systems, we observe that the background sync slows down the system in this case well. The trade-off in this system is that we can achieve lower network traffic while the operations are slower.

5.3 Large Scale Instrumentation

In order to evaluate our support for automation, we inject BlueMountain interface into 400 existing apps by bytecode instrumentation. This means that these 400 apps, after our instrumentation, can leverage BlueMountain automatically without any involvement from their original developers. By default, BlueMountain uses the local filesystem for all storage calls if there is no pluggable data management app. We use this default setting to evaluate our instrumentation.

Instrumentation Correctness: To verify the correctness of our instrumentation, we have used the following three methods:

(1) **Robustness Testing:** In our first experiment, we have run all 400 apps using our automated UI testing system. This system uses Android UI Automator [15] to generate random UI events such as button clicks and text input. We have run each app (unmodified version and BlueMountain instrumented version) for 10 times and 30 seconds each time. With every run, we have recorded all the exceptions an app throws. Then, we have compared the exceptions thrown by the unmodified app with the exceptions thrown by the same app *with* BlueMountain instrumentation. If there were any exception with BlueMountain that does not appear in the unmodified app, we consider it a result of incorrect instrumentation. For all 400 apps, we have verified that there is no app that throws new exceptions with BlueMountain.

(2) **Log Analysis:** For our second experiment, we have modified our framework to track the number of I/O related calls that are invoked and successfully handled by our framework. This was done to make sure that our automated tests were indeed resulting in I/O calls. With the logs enabled, we have re-run the automated tests previously described and observed that every method in the BlueMountain interface was triggered during our runs. Table 6 summarizes the number of calls our framework handled in the course of the testing 400 apps. These results show that our tests successfully triggered many I/O calls which were successfully handled by our system.

(3) **Manual Testing:** Lastly, we selected five apps (a camera app, a video editor, a text editor, a shopping list manager, and a game) and manually tested each one of them. Through

Category	Examples	Inst. Time Avg. (Min./Max.)	DEX Size Avg. (Min./Max.)	DEX Size Increase Avg. (Min./Max.)	Lines of Code Avg. (Min./Max.)	Lines of Code Increase Avg. (Min./Max.)
Game	Game of War - Fire Age, Fruit Slice	27.7s (5.5s/53.4s)	2.6M (169.7K/4.4M)	330.1K (16.3K/818.9K)	231.34K (9.44K/379.40K)	3.65K (3.65K/3.70K)
Entertainment	VideoFX Music Video Maker, ESPN Fantasy Sports	31.7s (2.8s/74.9s)	3.6M (293.5K/9.8M)	404.3K (37.1K/1.3M)	294.75K (7.51K/748.96K)	3.67K (3.65K/3.73K)
Media	Radi Music, Google News & Weather	27.4s (1.8s/63.2s)	3.2M (17.9K/8.5M)	348.0K (31.5K/1.5M)	259.40K (0.97K/676.75K)	3.66K (3.65K/3.75K)
Education	NASA, ABC Preschool Free	19.3s (2.3s/52.3s)	2.0M (159.1K/6.5M)	289.0K (14.1K/980.1K)	160.69K (5.06K/537.00K)	3.65K (3.65K/3.69K)
Personalization	Asteroids 3D live wallpaper, Twemoji	18.0s (1.6s/57.9s)	1.9M (18.6K/7.1M)	246.5K (5.5K/553.9K)	164.80K (0.23K/559.70K)	3.65K (3.65K/3.68K)
Productivity	Google Now Launcher, Barcode Scanner	16.1s (1.5s/71.8s)	1.7M (4.9K/8.8M)	272.8K (30.1K/1.3M)	145.22K (0.16K/803.07K)	3.65K (3.65K/3.71K)
Business	Cisco WebEx Meetings, Indeed Job Search	29.4s (1.9s/56.7s)	3.5M (38.2K/7.7M)	362.1K (21.5K/1.2M)	282.34K (1.72K/607.75K)	3.67K (3.65K/3.87K)
Social	KAYAK Flights, Hotels & Cars, Airbnb	32.9s (8.5s/64.7s)	3.8M (1.2M/8.0M)	439.4K (59.6K/952.1K)	315.21K (66.94K/650.81K)	3.66K (3.64K/3.72K)
Total	N/A	25.3s (1.5s/74.9s)	2.8M (4.9K/9.8M)	336.5K (5.5K/1.5M)	231.72K (0.16K/803.07K)	3.66K (3.64K/3.87K)

Table 4: Instrumentation Results for 400 Popular Apps (50 apps in each category)

App Name	Average (J)	Std Dev (J)
Game of War	1340.7J	26.3J
Game of War*	1381.2J	30.5J
VideoFX	949.3J	15.6J
VideoFx*	939.0J	22.9J
Barcode Scanner	666.7J	8.1J
Barcode Scanner*	678.4J	8.6J
Cisco WebEx	767.1J	21.5J
Cisco WebEx*	787.0J	8.5J
Kayak	494.9J	10.0J
Kayak*	496.0J	10.1J

Table 5: Energy Consumption (*Instrumented apps)

API	Number of Calls
Files	331,759
Databases	28,660
Shared Preferences	100,052

Table 6: API Invocation Count

this experiment, we have verified that (1) the experience of using these apps was the same as using the unmodified app, (2) there was no noticeable lag when using these apps, and (3) all the I/O was successfully redirected to a preconfigured location.

From these experiments, we conclude with reasonable confidence that there was no error in the functionality of the apps we tested after instrumenting them with BlueMountain.

Instrumentation Overhead: Table 4 shows the apps and their code metrics of our instrumentation. As shown, injecting BlueMountain's framework code (which uses the local filesystem by default) does not have significant overhead in terms of instrumentation time and code size increase.

Heap and Energy Usage: Figure 11 shows the heap usage of five well-known apps—Game of War [44], VideoFX [12], Barcode Scanner [37], Cisco WebEx [6], and Kayak [21]. Even in terms of heap usage, BlueMountain does not incur much overhead compared to the original apps. Table 5 shows energy consumption of the five apps with and without BlueMountain. We have used a Nexus S phone, connected with a Monsoon Power Monitor for measurement, and run the five apps for ten minutes five times. The averages are all within the standard deviation, which indicates that there is little statistical difference in energy consumption.

Database Feature	Percentage of Apps Using
Thread control calls	1.6%
Compiled statements	19.2%
Custom `CursorFactory`	7.8%

Table 7: Unsupported Database Features

5.4 Unsupported Database APIs

As we discuss in Section 3, our current prototype does not support non-essential database APIs—APIs for control over parallel queries, APIs for compiled SQLite statements, and APIs for customized output access—in the interest of keeping our interface clean and simple. In order to understand how popular these unsupported APIs are, we have downloaded 2000 popular Android apps from Google Play, statically analyzed them, and gathered statistics on how many apps use the APIs. Through this experiment, we have found out that 511 out of 2000 apps use at least one of the APIs. Further, Table 7 shows the percentage of apps that use the features. As shown, compiled SQLite statements are the most popular feature. As mentioned earlier, there are alternative ways for app programmers to accomplish the same tasks for these features and we can still support them by including them in our interface.

6 DISCUSSION

Security: While our system enables flexibility of data management for end users, one important challenge is security, where a malicious app pretends to be a data management app and potentially compromises user data. Although our current work does not directly address security questions, mishandling of user data has been one of the biggest and known concerns for malicious apps on Android [17]. Moreover, techniques for mitigating such a concern are already in place [3] and will be applicable for malicious data management apps as well.

Switching Data Management Apps: A potential feature of interest is the ability to hot-swap different data management apps at run time. Since each data management app would implement their own way of managing data, enabling hot-swap between data management apps would require an ability to export from one data management app and importing to another. We are currently investigating various ways to provide this feature.

Extending Coverage: In our current implementation, we still depend on Android to manage the filesystem metadata and structure for a data management app. This can be a limitation when a user

wants to use a highly secure data management system without leaking any information to the platform. We plan to address this issue in our future work by providing wrappers around the Android metadata management APIs.

7 RELATED WORK

Integrating Cloud Services: Previous work has tackled various challenges with mobile devices accessing cloud storage services. This ranges from optimizing data sync mechanism and providing better consistency models to providing more user functionality. In our own previous work, which laid the foundation for this work, we have explored the possibility of automatically integrating various cloud services into existing apps [5]. Many other systems provide libraries to be integrated with the apps or suggest modifications to underlying operating system/framework or to the cloud servers; our focus is complementary to these approaches, as we provide a mechanism through which some of these solutions can be easily deployed and used by the end user.

Call Interception: Our mechanism of intercepting Android storage calls and redirecting them is similar to FUSE [22]. GlusterFS [28] also uses `LD_PRELOAD` to intercept the POSIX filesystem calls and replace them with a custom implementation of GlusterFS core libraries. Recently, other techniques, such as Java bytecode rewriting and instrumentation, are utilized to change the behavior of applications without modifying their source code [4, 9, 10, 20, 25, 29, 32, 33, 38]. In the following, we discuss lines of research in data management issues arising from mobile apps and cloud-based storage services.

Cloud-Based File Synchronization Services: As cloud-based file synchronization services become popular, their shortcomings, challenges, and solutions have been discussed in the literature. Viewbox [43] proposes a design that integrates local file systems with cloud-based storage services to detect and recover from both inconsistency and data corruption that may happen in loosely coupled file synchronization services. QuickSync [8] and UDS [24] improve the efficiency of data synchronization in mobile devices using a variety of techniques. Metasync [18] implements a secure and reliable file synchronization mechanism by utilizing multiple cloud storage services.

Novel Data Management in Mobile Apps: A line of work discusses issues arising from how modern mobile apps utilize both filesystems and databases at the same time. Pebbles [35] argues that mobile apps do not correctly delete related data stored in filesystems and database and designs a system that can find the implicit relationships in an app's data stored in mobile devices. Simba [13] is a data-sync service for mobile apps and provides stronger consistency guarantees than file-based synchronization service. Diamond [42] designs a data management platform for reactive applications. Additionally, many commercial platforms such as Firebase [16] and Kinvey [34], enable app developers to link their apps to cloud backend which provides functionality like user management, push notifications, data management etc., through an emerging trend known as *Mobile Backend as a Service (MBasS)*. However, to take advantage of these services, the client components have to be integrated into the app at development time.

Context and Constraint-Based Data Management: Procrastinator [27], analyses and rewrites app binaries to do context based pre-fetching of the data. PSCloud [2] and Wherestore [36] create a location based data storage model which lets the users store their data on multiple devices. Cimbiosys [26] proposes partial replication of content shared among mobile devices and cloud services. WearDrive [19] is a storage system for wearable devices that offloads energy intensive tasks on the phone to improve the performance of apps and battery life of wearable devices.

Providing Flexibility and Cloud Adaptability for Mobile Data Management: Modern web/mobile applications rely on cloud-based storage services and there are various such services with different APIs. SPANStore [41] and Cloudlib [1] unify multiple cloud storage services with a single, key-value store interface. BlueMountain can easily transform an existing mobile application from using a single cloud storage provider to utilizing a layer like them. Zumero [45] provides a extension to SQLite database which enables database files to be synced across many devices. Lastly, to support the trend of employees using their personal mobile devices at workplace (BYOD), various Enterprise Mobility Management (EMM) systems, dedicated to manage user-owned devices, have come into existence. Systems like Airwatch [39], XenMobile [7], Android for Work [14], etc., provide a suite which includes device management, access control, data management (e.g. Airwatch Content Locker), etc., enabling personal devices to securely access corporate data. These platforms typically require a custom enterprise app to take advantage of their solutions while BlueMountain targets general apps which can be used by any user.

8 CONCLUSION

In this paper, we have presented a novel approach for customized data management on mobile devices. We have made a case for a new kind of app ecosystem where data management apps and regular apps exist. Our approach lets app developers focus on app logic while providing a way for data management app developers to provide innovative data management mechanisms. Our approach also gives end users better control over their data.

We demonstrate the usefulness of our approach by creating five data management apps. Each of the apps implements a unique mechanism for mobile data management, showcasing the flexibility of our approach. Our results show that our prototype system, BlueMountain, incurs only modest overhead in terms of read/write latency and throughput.

9 ACKNOWLEDGMENTS

We would like to acknowledge the works of Ajay Pratap Singh Chhokar, Ramanpreet Singh Khinda, and Aniruddh Ramesh Adkar who helped us immensely in developing the initial prototype of our data management apps. We would also like to thank the anonymous reviewers and our shepherd, Aruna Balasubramanian, for their valuable feedback. Lastly, we want to thank everyone working in our lab (Reliable Mobile Systems), for providing timely feedback. This work was supported in part by the generous funding from the National Science Foundation, CNS-1350883 (CAREER) and CNS-1618531.

REFERENCES

[1] Apache. 2017. Apache Libcloud. (Jan 2017). Retrieved July 10, 2017 from http://libcloud.apache.org

[2] Sobir Bazarbayev, Matti Hiltunen, Kaustubh Joshi, William H Sanders, and Richard Schlichting. 2013. Pscloud: a durable context-aware personal storage cloud. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems*. ACM, 9.

[3] Google blog. 2017. Shielding you from Potentially Harmful Applications. (Feb 2017). Retrieved July 10, 2017 from https://goo.gl/KosA6d

[4] A. Chander, J.C. Mitchell, and Insik Shin. 2001. Mobile Code Security by Java Bytecode Instrumentation. In *DARPA Information Survivability Conference and Exposition II, 2001. DISCEX '01. Proceedings*, Vol. 2. 27–40.

[5] Sharath Chandrashekhara, Kyle Marcus, Rakesh GM Subramanya, Hrishikesh S Karve, Karthik Dantu, and Steven Y Ko. 2015. Enabling Automated, Rich, and Versatile Data Management for Android Apps with BlueMountain.. In *HotStorage*.

[6] Cisco. 2017. Cisco WebEx Meetings. (Jan 2017). Retrieved July 10, 2017 from https://play.google.com/store/apps/details?id=com.cisco.webex.meetings&hl=en

[7] Citrix. 2017. XenMobile. (Jan 2017). Retrieved July 10, 2017 from https://www.citrix.com/products/xenmobile/

[8] Yong Cui, Zeqi Lai, Xin Wang, Ningwei Dai, and Congcong Miao. 2015. QuickSync: Improving Synchronization Efficiency for Mobile Cloud Storage Services. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*. ACM, 592–603.

[9] Benjamin Davis and Hao Chen. 2013. RetroSkeleton: Retrofitting Android Apps. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '13)*.

[10] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. 2012. I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. In *Proceedings of the IEEE Mobile Security Technologies (MoST '12)*.

[11] Dropbox. 2014. Streaming File Synchronization. (Jul 2014). Retrieved July 10, 2017 from https://blogs.dropbox.com/tech/2014/07/streaming-file-synchronization/d

[12] FuzeBits. 2017. VideoFX Music Video Maker. (Jan 2017). Retrieved July 10, 2017 from https://play.google.com/store/apps/details?id=com.videofx&hl=en

[13] Younghwan Go, Nitin Agrawal, Akshat Aranya, and Cristian Ungureanu. 2015. Reliable, Consistent, and Efficient Data Sync for Mobile Apps. In *FAST'15*. USENIX Association. https://www.usenix.org/conference/fast15/technical-sessions/presentation/go

[14] Google. 2017. Android for Work. (Jan 2017). Retrieved July 10, 2017 from https://www.android.com/work

[15] Google. 2017. Android UI Automator. (Jan 2017). Retrieved July 10, 2017 from https://developer.android.com/topic/libraries/testing-support-library/index.html

[16] Google. 2017. Firebase. (Jan 2017). Retrieved July 10, 2017 from https://firebase.google.com/

[17] Google. 2017. The Google Android Security Team's Classifications for Potentially Harmful Applications. (Feb 2017). Retrieved July 10, 2017 from https://source.android.com/security/reports/Google_Android_Security_PHA_classifications.pdf

[18] Seungyeop Han, Haichen Shen, Taesoo Kim, Arvind Krishnamurthy, Thomas Anderson, and David Wetherall. 2015. MetaSync: File synchronization across multiple untrusted storage services. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 83–95.

[19] Jian Huang, Anirudh Badam, Ranveer Chandra, and Edmund B Nightingale. 2015. WearDrive: fast and energy-efficient storage for wearables. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 613–625.

[20] Galen Hunt and Doug Brubacher. 1999. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd Conference on USENIX Windows NT Symposium - Volume 3 (WINSYM '99)*.

[21] Kayak. 2017. KAYAK Flights, Hotels & Cars. (Jan 2017). Retrieved July 10, 2017 from https://play.google.com/store/apps/details?id=com.kayak.android&hl=en

[22] Linux Kernel. 2017. FUSE. (Jan 2017). Retrieved July 10, 2017 from http://fuse.sourceforge.net/

[23] Taeyeon Ki, Alex Simeonov, Bhavika Jain, Chang Min Park, Keshav Sharma, Karthik Dantu, Steven Y. Ko, and Lukasz Ziarek. 2017. Reptor: Enabling API Virtualization on Android for Platform Openness. In *Proceedings of the 15th annual International Conference on Mobile Systems, Applications, and Services (MobiSys '17)*. ACM.

[24] Zhenhua Li, Christo Wilson, Zhefu Jiang, Yao Liu, Ben Y Zhao, Cheng Jin, Zhi-Li Zhang, and Yafei Dai. 2013. Efficient batched synchronization in dropbox-like cloud storage services. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 307–327.

[25] Microsoft. 2017. Windows Hooks. (Jan 2017). Retrieved July 10, 2017 from http://goo.gl/r32d7B

[26] Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C. Marshall, and Amin Vahdat. 2009. Cimbiosys: A Platform for Content-based Partial Replication. In *NSDI'09*. USENIX Association. http://dl.acm.org/citation.cfm?id=1558977.1558995

[27] Lenin Ravindranath, Sharad Agarwal, Jitendra Padhye, and Chris Riederer. 2014. Procrastinator: Pacing mobile apps' usage of the network. In *Proc. ACM MobiSys*.

[28] RedHat. 2017. GlusterFS. (Jan 2017). Retrieved July 10, 2017 from https://www.gluster.org/

[29] Jonathan Rentzsch. 2016. Mach Inject. (Nov 2016). Retrieved July 10, 2017 from https://github.com/rentzsch/mach_inject

[30] Slim Roms. 2017. Slim Roms. (Jan 2017). Retrieved July 10, 2017 from https://slimroms.org/

[31] Rsync. 2017. Lib Rsync. (Jan 2017). Retrieved July 10, 2017 from http://librsync.sourceforge.net/

[32] Algis Rudys and Dan S. Wallach. 2003. Enforcing Java Run-time Properties Using Bytecode Rewriting. In *Proceedings of the 2002 Mext-NSF-JSPS International Conference on Software Security: Theories and Systems (ISSS '02)*.

[33] Haichen Shen, Aruna Balasubramanian, Anthony LaMarca, and David Wetherall. 2015. Enhancing Mobile Apps to Use Sensor Hubs Without Programmer Effort. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '15)*. ACM, New York, NY, USA, 227–238. https://doi.org/10.1145/2750858.2804260

[34] Progress Software. 2017. Kinvey BaaS. (Jan 2017). Retrieved July 10, 2017 from https://www.kinvey.com/

[35] Riley Spahn, Jonathan Bell, Michael Lee, Sravan Bhamidipati, Roxana Geambasu, and Gail Kaiser. 2014. Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems. In *OSDI'14*. USENIX Association. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/spahn

[36] Patrick Stuedi, Iqbal Mohomed, and Doug Terry. 2010. Wherestore: Location-based data storage for mobile devices interacting with the cloud. In *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*. ACM, 1.

[37] ZXing Team. 2017. Barcode Scanner. (Jan 2017). Retrieved July 10, 2017 from https://play.google.com/store/apps/details?id=com.google.zxing.client.android&hl=en

[38] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*.

[39] VmWare. 2017. Air-Watch Enterprise mobility platform. (Jan 2017). Retrieved July 10, 2017 from https://www.air-watch.com/

[40] Wi-Fi.org. 2017. Wi-Fi Direct. (Jan 2017). Retrieved July 10, 2017 from http://www.wi-fi.org/discover-wi-fi/wi-fi-direct

[41] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. 2013. SPANStore: Cost-effective Geo-replicated Storage Spanning Multiple Cloud Services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 292–308.

[42] Irene Zhang, Niel Lebeck, Pedro Fonseca, Brandon Holt, Raymond Cheng, Ariadna Norberg, Arvind Krishnamurthy, and Henry M. Levy. 2016. Diamond: Automating Data Management and Storage for Wide-Area, Reactive Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 723–738. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhang-irene

[43] Yupu Zhang, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. ViewBox: Integrating Local File Systems with Cloud Storage Services. In *FAST'14*. USENIX. https://www.usenix.org/conference/fast14/technical-sessions/presentation/zhang

[44] Machine Zone. 2017. Game of War - Fire Age. (Jan 2017). Retrieved July 10, 2017 from https://play.google.com/store/apps/details?id=com.machinezone.gow&hl=en

[45] Zumero. 2017. SQL Datasync for Mobile Apps. (Jan 2017). Retrieved July 10, 2017 from http://zumero.com