Amniote: A User Space Interface to the Android Runtime

Zachary Yannes[a] and Gary Tyson
Department of Computer Science, Florida State University, Tallahassee, U.S.A.

Keywords: Android, Zygote, Runtime, Dalvik, Virtual Machine, Preloading, ClassLoader, Library.

Abstract: The Android Runtime (ART) executes apps in a dedicated virtual machine called the Dalvik VM. The Dalvik VM creates a Zygote instance when the device first boots which is responsible for sharing Android runtime libraries to new applications.
New apps rely heavily on external libraries in addition to the runtime libraries for everything from graphical user interfaces to remote databases. We propose an extension to the Zygote, aptly named Amniote, which exposes the Zygote to the user space. Amniote allows developers to sideload common third-party libraries to reduce application boot time and memory. Just like the Android runtime libraries, apps would share the address to the library and generate a local copy only when one app writes to a page.
In this paper, we will address three points. First, we will demonstrate that many third-party libraries are used across the majority of Android applications. Second, execution of benchmark apps show that most page accesses are before copy-on-write operations, which indicates that pages from preloaded classes will infrequently be duplicated. Third, we will provide a solution, the Amniote framework, and detail the benefits over the traditional Zygote framework.

1 INTRODUCTION

Recent studies have shown that Android devices have maintained substantially higher market shares compared to iOS, Windows, Blackberry, and other operating systems (Katariya, 2017). With the increase of Android devices, application development has also increased. A recent report by AppBrain shows that as of January 1, 2018, there are roughly 3.5 million applications in the Google Play Store. The growing trend of Android development indicates that it is an ideal candidate for research.

Android applications contain a combination of Java bytecode, native code, and resource files. Unlike traditional Java applications, Android apps execute in a dedicated virtual machine called DalvikVM. DalvikVM has several unique features tailored towards mobile devices which have less resources than those of a laptop or desktop computer. For example, Dalvik VMs utilize a register-based construct rather the stack-based execution of traditional Java VMs. This improves execution speed and reduces the complexity of switching between applications.

The primary difference between DalvikVM and traditional VMs is the *Zygote*. The zygote process, which is started after the kernel is loaded, preloads common runtime classes and resource files from the Android and javax packages into a read-only boot image. When a new application is ready to launch, zygote forks a new VM and starts the process. The new app contains a reference to the boot image rather than a full copy. This allows multiple applications to share common classes and resource files, reducing memory utilization.

The Android kernel employs a copy-on-write mechanism for the boot image. Whenever an application writes to an address mapped to the shared boot image, the kernel copies that page to that application's address space. This allows each application to maintain a local copy of written pages.

Since the initial Android version release in 2009, Google has released 14 versions, with the newest, Pie, currently in beta (Wikipedia, 2018). This is a rapid development cycle for a relatively new operating system. To keep up with the rapid changes to the Android API, Google released a dedicated integrated development environment (IDE), Android Studio. Android Studio provides support for Maven repositories, which simplifies the process of linking to external libraries. This caused a paradigm shift of relying on third-party libraries for common Android util-

[a] https://orcid.org/0000-0003-3098-0807

Yannes, Z. and Tyson, G.
Amniote: A User Space Interface to the Android Runtime.
DOI: 10.5220/0007715400590067
In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2019)*, pages 59-67
ISBN: 978-989-758-375-9

ities such as graphic user interfaces (GUI) (Android, 2018a; Android, 2018e; Android, 2018f), data storage mechanisms (Google, 2018b; Google, 2018c; Android, 2018c), and code annotations (Android, 2018d; Google, 2018a; Android, 2018b).

There has been widespread adoption of many third-party libraries which are utilized by a large number of Android applications. Not only does this reduce coding time but it also provides an opportunity optimization. Just as the Android/javax runtime classes and resource files are referenced by many applications, so are the classes and resource files in a number of third-party libraries.

The other option is to extend the functionality of the zygote process to the user space. The ability to append classes and resource files to the zygote process's memory space from user space would open up many avenues for optimization. Therefore, we introduce a framework called *Amniote* which provides methods of managing the zygote's boot image from the user space. The Amniote framework also contains utilities for calculating common classes between a large set of applications.

The Amniote framework could also benefit mobile device vendors and Android Things developers. Mobile vendors often fork Android with customized launchers, stock applications, and power monitoring (Peters, 2017). The applications in these forked Android versions likely share proprietary code which would benefit from adding to the boot image. An additional benefit would be quicker deployment of new Android version. Instead of vendors implementing optimizations on the operating system, "profile" applications could be written which optimize the zygote boot image depending on the type of applications the user will be running. Similarly, Android Things developers deploy applications with a specific suite of required libraries.

The remainder of this paper is organized as follows. In sections 2 and 3 we describe Java's ClassLoader and how it can benefit from prefetching in both traditional Java VMs and in Android's Dalvik VMs. In section 4, we further explain the zygote process. In section 5, we analyze the commonly used third-party classes across a large sample of open-source applications. In Section 6, we evaluate the executions of applications in the *Agave* benchmark suite and stock Android applications. In Section 7, we describe the Amniote framework. In Section 10, we conclude our findings and discuss future work on this topic.

2 THE JAVA ClassLoader

Java VMs rely on a boot classpath which contains paths to all Java class files. This allows Java to support *lazy loading*, where classes are loaded on-demand via a *ClassLoader* (Liang and Bracha, 1998). *ClassLoader*s also provide support for multiple namespaces, which allows a VM to have non-unique class names which are resolved by unique package namespaces. For example, the Android SDK provides the class *android.app.Application* containing generic app data that is frequently extended to contain app-specific data. In this case, the system *ClassLoader* will resolve the Android SDK's *Application* while a second application *ClassLoader* will resolve the

As shown in (Liang and Bracha, 1998), *ClassLoader*s are typically implemented with a cache of loaded class data referenced by the class name. For example, when the application requires a specific class for the first time, a *ClassLoader* searches the classpath for the class file, loads the class file into memory, and adds an entry into the *ClassLoader*'s cache. Each subsequent use of that class then utilizes the class data found in the *ClassLoader*'s cache.

3 PREFETCHING

ClassLoaders demonstrate a common cache problem where compulsory (or cold) misses accumulate since the cache is initially empty and the first reference of a class results in a cache miss. To circumvent this problem, *prefetching* fills the cache with classes it will likely access to avoid costly searches and disk I/O operations. Cache prefetching is not a new solution. In fact, preloading process works for both hardware and software caches. Yet we limit the scope of this paper to software caches.

3.1 Java VM Prefetching

An early approach, greedy prefetching, was used in (Luk and Mowry, 1996) for recursive C program data structures. This was later extended in (Cahoon and McKinley, 1999) to Java programs, using intra-procedural data flow analysis to detect objects to prefetch. Their approach demonstrates the effectiveness of greedy prefetching which relies upon inlining to remove unnecessary method calls. We will use a similar approach in Amniote to prefetch third-party library classes.

Zaparanuks et. al present a Markov predictor for speculative class-preloading (Zaparanuks et al.,

2007). Using a Markov predictor allows accurate prediction of which classes should be preloaded and when is the best time to do so. A similar approach is taken by (Fulton, 2016). While these are novel and robust methods, they only preload classes available in the application. Android apps have the capability of linking to third-party libraries in external *jar* files which would have to be located and extracted first.

3.2 Dalvik VM Prefetching

In Android's Dalvik VM, the class files are precompiled into bytecode or native code and packaged into *DEX*, *OAT*, *APK*, or *JAR* files. This provides the executable resources in a compressed format that saves space on resource-limited mobile devices.

The Android Runtime (ART) executes Android apps quite differently from traditional Java applications. Primarily, ART is typically executing many apps, via Dalvik VMs, concurrently. Each time an app references a new class it results in a compulsory miss, requiring all needed class data to be copied into the process's virtual memory.

Unlike the previous implementations the Dalvik VM contains a naive prefetching mechanism, initiated by the zygote process, to mitigate these compulsory misses. A file containing common Android and javax runtime classes, *preloaded-classes* is read by the zygote when preloading. The *preload-classes* text file is created by running class dependency analysis on a stock set of applications. These preloadable classes are contained in a precompiled system boot image, *boot.art*. While this suffices for executing stock apps, it does not fully exploit the frequency of third-party library classes.

*ClassLoader*s provide the opportunity to load from third-party libraries dynamically. Maly et. al utilized *ClassLoader*s to dynamically load modules in Android apps (Maly and Kriz, 2015). They posit that Android apps can contain the main application logic while business logic, which is often changing, can be dynamically loaded in a context-aware adaptable environment. This allows changes to be continuously deployed on the device without any user interaction, such as through manually upgrading the application through the Play Store.

These previous works demonstrate preloading and dynamic loading of Java libraries via *ClassLoader*s. Yet there is an untapped opportunity for optimization with third-party libraries. This paper is an attempt to address this issue by employing a novel framework, Amniote, which decreases overall cache compulsory misses by preloading commonly used library classes. But since the number of overlapping libraries is constantly changing based on the user's application preferences, it is difficult to determine which library classes should be preloaded. Therefore, Amniote provides a dynamic preloading mechanism to user space so developers can exploit app category *profiles*. Our profiles are constructed based on app categories such as games, internet, and navigation, but could be based on any suite of apps.

4 ANDROID ZYGOTE PROCESS

The Android Runtime (ART) uses Dalvik VMs to execute application code. When the device finishes booting, the first process to start is the Zygote process. The zygote process then starts the preloading mechanism discussed in section 3.2. The zygote process then acts as a server, responding to incoming requests to launch applications, and forking virtual machines to fulfill the requests.

When a new process is forked, the boot classpath contains the app code along with the zygote's boot.art. Since applications are stripped of the common Android runtime libraries, they are instead found in boot.art when an Android runtime class is referenced.

Android's Linux kernel is responsible for performing the copy-on-write when an application tries to write data to the boot.art space. Since the virtual memory address (VMA) of the boot.art data is marked as read-only, a write is redirected to an anonymous VMA owned by the application. Any consecutive reads would then be performed with the anonymous VMA instead of the zygote's boot.art VMA.

Using a copy-on-write mechanism allows the common classes to be initially shared by any applications, at the expense of copying a page when a write occurs. Initially, this requires a much lower amount of memory for all the applications to access these classes. However, as the number of unique page writes increases, the memory required will increase since each application maintains its own copies. The worst case scenario is every application performs a complete copy of the entire class. Then the read-only copy managed by the zygote is eventually useless and still held in memory. Therefore, copy-on-write must be used carefully to avoid such a scenario.

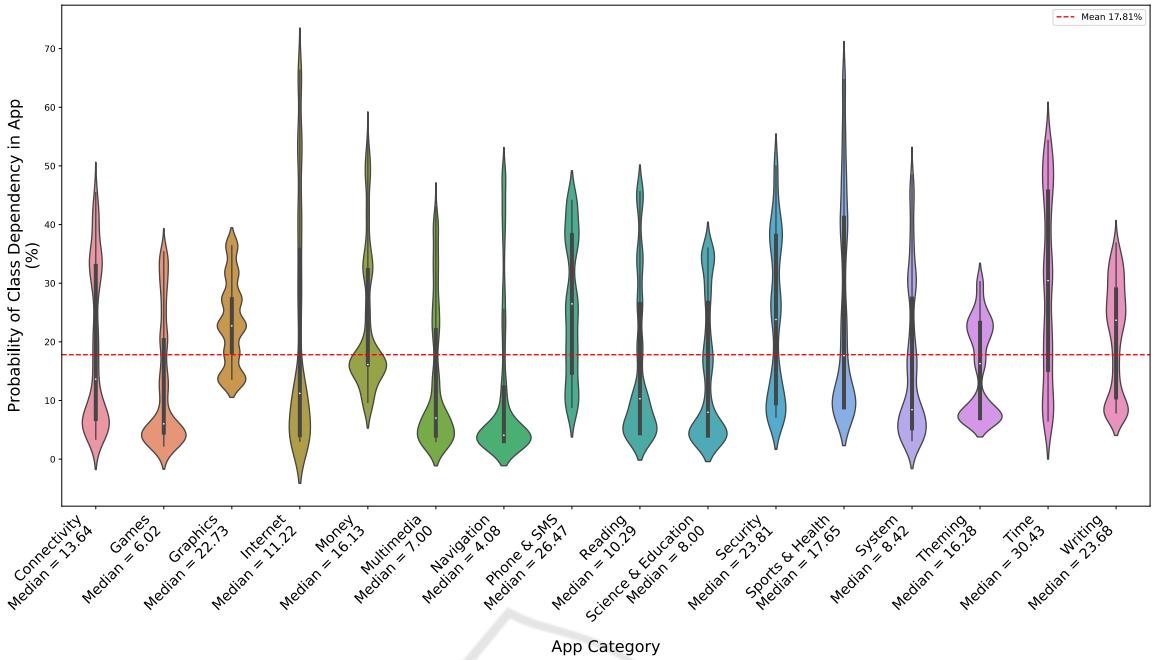

Figure 1: Distributions of Third-Party Classes across App Categories.

5 THIRD-PARTY LIBRARY ANALYSIS

The Amniote framework operates on the basis that many third-party libraries are depended on by many different applications. In order to determine this, we analyzed a large dataset of open-source apps pulled from fdroid.org (fdr, 2018). Each app was sorting into one of the following 16 categories: Connectivity, Games, Graphics, Internet, Money, Multimedia, Navigation, Phone & SMS, Reading, Science & Education, Security, Sports & Health, System, Theming, Time, and Writing. Using the AOSP tool, *dexdump*, we determined the third-party library classes required by each app. Each library class was counted over all apps in each category.

Figure 1 shows the resulting probability distributions of third-party library classes being referenced in apps across 16 app categories. While a third-party class is only used by 17.8% of apps in each category on average, several classes are used by the majority of apps in a category. For example, the *android.support.v4.app.Fragment* class is reference by 66% of the apps in the Internet category. This class provides functionality for displaying a portion of an Activity's user interface (Android, 2018g). Additionally, the *android.support.v4.view.ViewPager$SavedState* class has a probability of 64% being referenced in Sports & Health apps.

By splitting the apps into categories, we can create app category "profiles". These profiles will contain a precompiled list of classes with a high-probability of reference for a given category. Category profiles could be used by vendors or developers to create app launchers or lock screens that preload common classes. For example, a user frequently uses Internet-based applications such as Google Chrome and Gmail. The user could then switch to an Internet app launcher that preloads the Internet category profile. The categories do not necessarily have to be exclusive either. A category could pertain to a specific user's morning routine. For example, checking a messaging app, followed by an email app, followed by a news feed app. Since the app classes are available before runtime, the preload classes candidates can be selected ahead-of-time either offline or on the device.

6 APPLICATION EXECUTION ANALYSIS

The current Dalvik VM utilizes copy-on-writes to duplicate modified pages to the app's virtual address space before applying changes. While preloading classes reduces compulsory cache misses, there is a tradeoff with memory consumption as the ratio of CoWs to page accesses increases. The greater the ra-

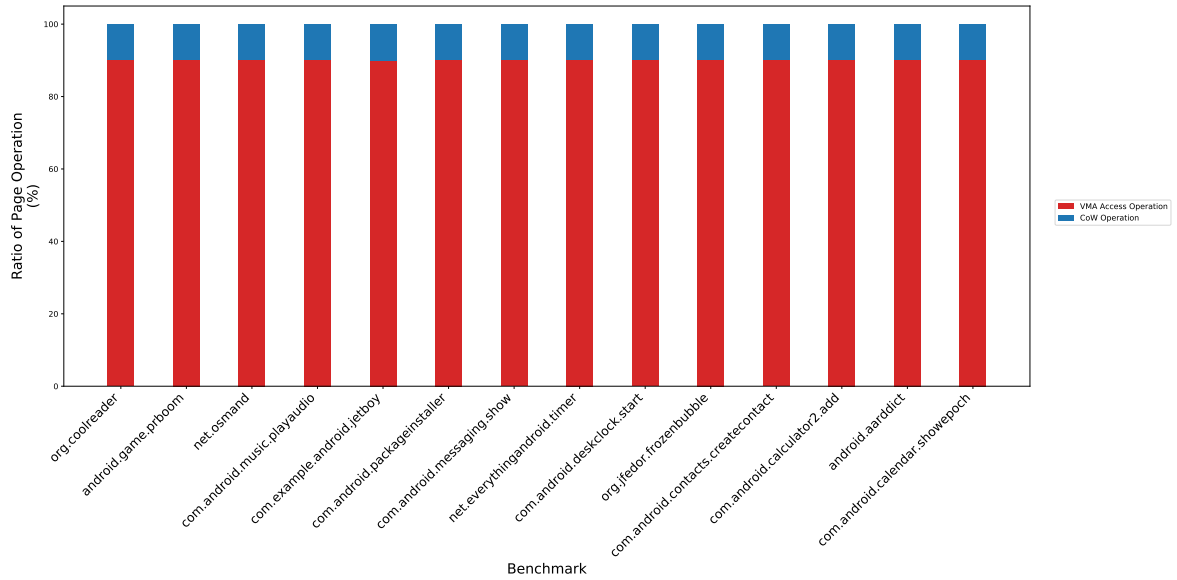

Figure 2: Ratio of Page Access and CoW Operations Per Benchmark.

tio, the more apps will produce local page copies and less references will be made to the shared page. In order to evaluate the CoW-to-page access ratio, we analyze the execution of apps in the *Agave* Android benchmark suite (Brown et al., 2016). The Agave benchmark suite provides 14 apps across multiple app categories.

The Agave benchmarks were executed on Android with a modified Linux kernel 3.14 using a modified Gem5 simulator (Binkert et al., 2011). The Linux kernel was modified to support gem5 operations and provide size information for different kernel structures. The gem5 simulator was modified to monitor for page access and CoW operations.

The total number of page access and copy-on-write operations were calculated for each benchmark. Figure 2 shows the ratio of the page access operations to CoW operations per Agave benchmark app. The percentage of CoW operations is consistent in each benchmark at roughly 10%. This indicates that on average an app only copies 10% of the memory from the shared boot classpath. Clearly the remaining 90% of shared memory accesses can be exploited by preloading the shared memory. The app category profiles discussed in Section 5 can be used to determine which classes to preload into shared memory.

7 THE AMNIOTE FRAMEWORK

In this section, we describe the implementation and capabilities of the Amniote framework.

Amniote is implemented as a userspace layer over the Zygote[1]. The Amniote framework is implemented over the AOSP Android 6.0.1 written primarily in C++, with the userspace library implemented in Java.

The purpose of the Amniote framework is to manage a secondary shared address space that can be manipulated from userspace, unlike the Zygote shared address space which is read-only after the Zygote process is initialized. From the perspective of the Dalvik VMs, there will still be a single boot classpath to find system libraries. Amniote modifies the secondary address space by communicating with the Zygote's socket. When Amniote sends an add image or remove image command to the Zygote socket, it also sends the path to the image file. The Zygote process has been modified to append this image to the secondary boot classpath. Similarly, when a preload or unload command is sent, the classSignature is also sent, and then either preloaded or unloaded, respectively, using the method currently available for the first boot classpath.

Amniote is initialized as the system server is started by the Zygote. Upon creation, a secondary shared address space is created and added to the default bootpath. Figure 3 shows the basic Amniote framework fields and methods. From user space, apps can get an instance of Amniote by executing *Amniote.getInstance()*. Then an app can add or remove a DEX, JAR, APK, or OAT image to Amniote's classpath with *addImage(..)* or *removeImage(...)*, respectively. Adding an image checks that the image

[1]The Amniote framework is named after the biological term *amniotic sac* which contains the embryo after the zygote implants in the uterus.

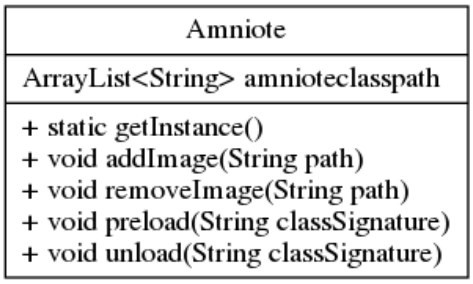

Figure 3: Amniote Framework.

file exists before appending to the classpath. Next, *preload(...)* or *unload(...)* can be called to preload or unload a specific class, respectively. Once a class is preloaded, a reference from any app will find the class in the shared address space as if it were preloaded by the zygote process. If an application makes a write to a page in the shared address space, it will perform a copy into the apps's address space through the Linux copy-on-write mechanism.

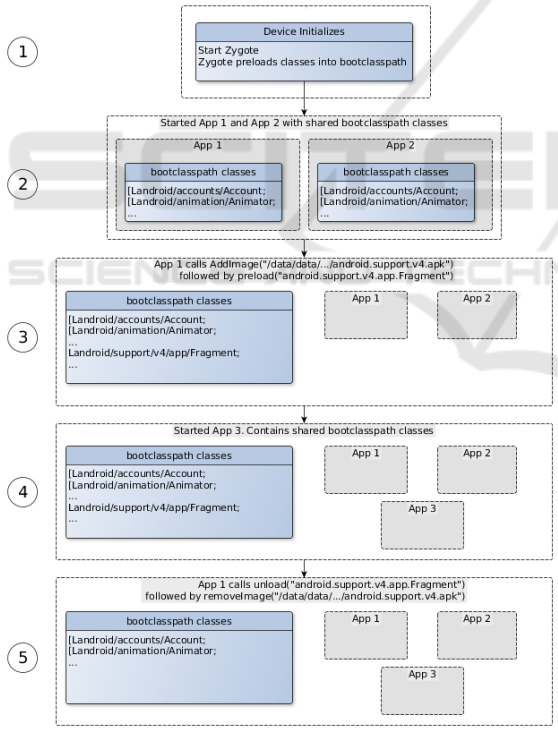

Figure 4: Example of the Amniote Framework.

Figure 4 shows an example usage of the Amniote framework. In step 1, the device boots up, starting the zygote process. The zygote process initializes the *bootclasspath* and preloads the default classes. In step 2, two different apps are launched, each containing their own classpaths, but also sharing the *bootclasspath*. In step 3, App 1 adds a local apk to the shared bootclasspath. Next, the *Fragment* class is preloaded. In step 4, a third app is launched. Three running apps can reference any of the preloaded classes in the shared *bootclasspath*, including the *Fragment* class preloaded by App 1. Additionally, each app can still access their private classpaths. Finally, in step 5, App 1 removes the *Fragment* class the local apk. Note that App 1 adds and removes the local image file because the image is in a private directory. However, if the image is in a directory readable by any process, such as on the sdcard, then any app could add or remove the image to the *bootclasspath*.

One side effect of using Amniote is that APKs do not need to include external libraries. Common libraries such as *com.android.support.** and *com.google.android.material.** which are frequently utilized in applications need only be stored on the device once. This greatly reduces APK code size and disk utilization on the device. Furthermore, it solves the common method count limit when targeting older APIs. In order to support this, libraries are stripped out from the APK via Proguard during packaging and deployed through the mechanisms discussed in Section 9.

8 AMNIOTE ANALYSIS

The benefits of the Amniote Framework are most prevalent when multiple apps are running that utilize the same classes. Amniote can preload the overlapping classes to reduce memory usage.

In order to test the effectiveness of the Amniote Framework, we evaluated the third-party library usage across 9 apps which were hand-classified into three different categories (TeamAmaze, 2019; federicoiosue, 2019; k9mail, 2019; byoutline, 2018; TeamNewPipe, 2019; nickbutcher, 2019; AntennaPod, 2019; esoxjem, 2018; videolan, 2019). Ta-

Table 1: A suite of benchmarks classified into categories.

Category	Benchmark	Description
Phone & SMS	AmazeFileManager	Show files and directories in root directory of sdcard.
	Omni Notes	Create and save new checklist with a single item.
	K9 Mail	Attempt to login with sample username and password.
Internet	Kickstarter	Display list of sample kickstarter campaigns.
	NewPipe	Fetch and display list of recent Youtube videos.
	Plaid	Fetch and display recent blog posts about material design.
Multimedia	AntennaPod	Fetch and display list of available podcasts and select an item.
	MovieGuide	Fetch and display list of recent movies and select an item.
	VLC	Play sample 5 second video.

Table 2: The total and overlapping library class count for each category and the total size of the duplicated classes in bytes.

Category	Total Library Class Count	Overlapping Library Class Count	Size of Duplicated Classes (in B)
Productivity	2318	27	126054
Social	5014	97	3388
Media	2451	205	6986

ble 1 shows the benchmark apps in each category and provides a description of each benchmark. The benchmarks were selected based on releasing versions within the last year and utilizing third-party libraries. Additionally, we only chose apps for a category that provide unique functionality and would likely be run sequentially by the user. For example, consider the *Productivity* category consisting of *AmazeFileManager*, *Omni Notes*, and *K9 Mail*. A user might create a todo list with Omni Notes, and then open a file in AmazeFileManager, and finally share that file over email via K9 Mail. When a user sequentially runs apps, the Android Runtime maintains the virtual machine running each app unless the user explicitly exits the app or the device runs out of resources required to run the app. Therefore, it is probable to have multiple running apps in the same category.

Since the effectiveness of Amniote is determined by the number of accesses to overlapping third-party libraries, we modified the Android OS to log every access to a library class.

Each benchmark app was executed on the modified Android OS using the Android Emulator. Then, for each category, the total set of third-party library classes and the union set of third-party library classes were determined. Table 2 shows the total third-party library class count and the overlapping third-party library class count for each category. The table shows between 1.9% and 8.3% of third-party classes are duplicated across each category. The more interesting result is the amount of memory consumed by duplicate classes. For example, in the Productivity category, the 27 classes that overlap result in 123 KB of unnecessary memory consumption that would be prevented by Amniote.

While the results of our experiment demonstrate a benefit for employing the Amniote Framework, our benchmarks were designed to perform a single task. Real-world users are likely to use an app to do more than one task. In fact, as a user runs more concurrent apps, the Amniote Framework could potentially reduce total memory usage even further.

9 DISCUSSION

In section 2, we discussed how Java *ClassLoader*s can load non-unique classes as long as they are contained in different package namespaces. This produces a unique ClassLoader entry for each class in each package namespace. However, since third-party libraries are compiled directly into Android apps, the apps on a device may depend on conflicting library versions. Since each Dalvik VM uses an local app *ClassLoader* to resolve these library classes, there are currently no issues. But when the library classes are linked via the system *ClassLoader* instead of the app *ClassLoader*, non-unique classes will produce a conflict.

The current Amniote implementation has no version control for libraries. In other words, it is assumed that the libraries shared amongst applications are all using the exact same version. Yet this is rarely the case. We present three possible solutions to the versioning problem, each putting the responsibility on a different party with different levels of practicality.

The first possibility is for the Play Store to perform library dependency analysis and to hold or lock new app versions until all the apps using a specific library use the same version. This puts the onus of version control on the Google Play Store. For example, App A and App B rely on a library. App A releases a new version that depends on library version 2.0, while App B depends on library version 1.0. The Play Store would hold App A until App B releases a new version depending on library version 2.0. Clearly this would cause many problems. If the developers for App B never released a new version, App A would never be released. It is simply too impractical to rely on the continuous deployment cycles of apps.

The second possibility is for Amniote to perform the library dependency analysis to find the greatest overlapping version numbers of each library and only include those library versions when prefetching. While this solves the problem of library compatibility and eliminates any responsibility from app and library developers, it significantly decreases the effectiveness and greatly increases the complexity of Amniote. Every image that is added to Amniote requires rerunning dependency analysis, adding additional runtime and memory requirements. Furthermore, many potential libraries could be excluded due to a version mismatch. In extreme cases, an image may contain all version mismatches and be completely incompatible.

The final and most practical solution is to provide library support in the Google Play Store. Having the Play Store support libraries would share the responsibility between the library and Google Play Store developers, instead of the app developer. As long as the

library has backwards compatibility, new versions can be deployed on the Play Store and automatically updated via the Play Store update protocol. The Google Play Store could also provide support for live updates by simply having Amniote reload any classes implemented by the new library, which would be immediately reflected in individual apps. Linking libraries through the Google Play Store has the added benefit of security. Since all applications now utilize the same library version, it is easier to ensure the library is directly from the vendor and has not been tampered.

10 CONCLUSION AND FUTURE WORK

In this paper we discussed the growing dependency of third-party libraries in Android apps. We demonstrated that many third-party library classes are referenced by the majority of apps in their specific app category. We then showed that the probability distribution of third-party classes can be used to construct an app category profile which statistically informs our decision to preload a given class.

Next we evaluated the execution of apps in the Agave benchmark suite on a modified gem5 simulator. The simulations demonstrated that only a relatively small proportion of operations are copy-on-writes compared to page accesses.

Finally we introduced the Amniote framework, a user-space interface to the Dalvik VM. Amniote provides a simple interface for dynamically adding and removing a variety of image files to the shared boot classpath, as well as preloading or unloading specific classes. This framework provides developers and vendors greater opportunity for fine-tuning a device based on user preferences.

In the future we plan to extend the Amniote framework with several additional features.

First, we will define a file format for the app category profiles and provide direct support for these files. Then the preloading would be performed automatically by providing the image file and the profile file.

Second, we plan to provide support for resource files in addition to classes. The naive preloader in the Dalvik zygote process preloads classes and resource files, which Amniote can easily extend.

Third, our evaluations used a large, but static dataset of benchmark apps for selecting preloading class candidates. In reality, a user may have more variability in the apps executed such that the user may run apps across multiple categories in a single session. In order to target this use case, machine learning could be utilized to dynamically build the category profiles.

Finally, we plan to develop the Library Play Store discussed in Section 9. One benefit of Amniote is that app developers no longer need to focus on third-party libraries. Library version updates and library version interdependencies are no longer a concern since the libraries are stripped from the APK and installed independently.

The Library Play Store would provide a single repository for libraries which would additionally add security and a possibility for live updates. This would move Android development one step closer to liquid software deployment, a conceptual goal of continuous deployment proposed by Gallidabino et. al (Gallidabino et al., 2017). Each entity involved in the final product, the app developers and the library developers, could independently deploy updates without concerning themselves with the updates of dependencies.

REFERENCES

(2018). F-droid. https://f-droid.org/en/. Online; accessed 30-April-2018.

Android (2018a). Android AppCompat Library V7. https://mvnrepository.com/artifact/com.android.support/appcompat-v7. Accessed: 2018-11-26.

Android (2018b). Android Room Compiler. https://mvnrepository.com/artifact/android.arch.persistence.room/compiler. Accessed: 2018-11-26.

Android (2018c). Android Room Runtime. https://mvnrepository.com/artifact/androidx.room/room-runtime. Accessed: 2018-11-26.

Android (2018d). Android Support Library Annotations. https://mvnrepository.com/artifact/com.android.support/support-annotations. Accessed: 2018-11-26.

Android (2018e). Android Support Library V4. https://mvnrepository.com/artifact/com.android.support/support-v4. Accessed: 2018-11-26.

Android (2018f). Android Support RecyclerView V7. https://mvnrepository.com/artifact/com.android.support/recyclerview-v7. Accessed: 2018-11-26.

Android (2018g). Fragments — Android Developers. https://developer.android.com/guide/components/fragments. [Online; accessed 2018-11-26].

AntennaPod (2019). AntennaPod. https://github.com/AntennaPod/AntennaPod. [Online; accessed 20-Feb-2019].

Binkert, N., Beckmann, B., Black, G., Reinhardt, S., Saidi, A., Basu, A., Hestness, J., Hower, D., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M., and Wood, D. (2011). The gem5 simulator. ACM SIGARCH Computer Architecture News, 39(2):1–7.

Brown, M., Yannes, Z., Lustig, M., Sanati, M., McKee, S., Tyson, G., and Reinhardt, S. (2016). Agave: A benchmark suite for exploring the complexities of the android software stack. In Performance Analysis of

Systems and Software (ISPASS), 2016 IEEE International Symposium on, pages 157–158. IEEE.

byoutline (2018). Kickmaterial. https://github.com/byoutline/kickmaterial. [Online; accessed 20-Feb-2019].

Cahoon, B. and McKinley, K. S. (1999). Tolerating latency by prefetching java objects. In Workshop on Hardware Support for Objects and Microarchitectures for Java. Citeseer.

esoxjem (2018). MovieGuide. https://github.com/esoxjem/MovieGuide. [Online; accessed 20-Feb-2019].

federicoiosue (2019). Omni-Notes. https://github.com/federicoiosue/Omni-Notes. [Online; accessed 20-Feb-2019].

Fulton, M. S. (2016). Conservative class preloading for real time java execution. US Patent 9,477,495.

Gallidabino, A., Pautasso, C., Mikkonen, T., Systä, K., Voutilainen, J.-P., and Taivalsaari, A. (2017). Architecting liquid software. J. Web Eng., 16(5&6):433–470.

Google (2018a). Dagger. https://mvnrepository.com/artifact/com.google.dagger/dagger. Accessed: 2018-11-26.

Google (2018b). Firebase Database. https://mvnrepository.com/artifact/com.google.firebase/firebase-database. Accessed: 2018-11-26.

Google (2018c). Firebase Firestore. https://mvnrepository.com/artifact/com.google.firebase/firebase-firestore. Accessed: 2018-11-26.

k9mail (2019). K-9. https://github.com/k9mail/k-9. [Online; accessed 20-Feb-2019].

Katariya, J. (2017). Apple Vs. Android. https://android.jlelse.eu/apple-vs-android-a-comparative-study-2017-c5799a0a1683. [Online; accessed 1-August-2018].

Liang, S. and Bracha, G. (1998). Dynamic class loading in the java virtual machine. Acm sigplan notices, 33(10):36–44.

Luk, C.-K. and Mowry, T. C. (1996). Compiler-based prefetching for recursive data structures. In ACM SIGOPS Operating Systems Review, volume 30, pages 222–233. ACM.

Maly, F. and Kriz, P. (2015). Techniques for dynamic deployment of modules in context-aware android applications. In Computational Intelligence and Informatics (CINTI), 2015 16th IEEE International Symposium on, pages 107–111. IEEE.

nickbutcher (2019). Plaid. https://github.com/nickbutcher/plaid. [Online; accessed 20-Feb-2019].

Peters, A. (2017). How Android Differs Depending on the Hardware Manufacturer. https://www.makeuseof.com/tag/android-differs-hardware-manufacturer/. Accessed: 2018-11-26.

TeamAmaze (2019). AmazeFileManager. https://github.com/TeamAmaze/AmazeFileManager. [Online; accessed 20-Feb-2019].

TeamNewPipe (2019). NewPipe. https://github.com/TeamNewPipe/NewPipe. [Online; accessed 20-Feb-2019].

videolan (2019). VLC. https://github.com/videolan/vlc. [Online; accessed 20-Feb-2019].

Wikipedia (2018). Android version history. https://en.wikipedia.org/wiki/Android_version_history. [Online; accessed 2-April-2018].

Zaparanuks, D., Jovic, M., and Hauswirth, M. (2007). The potential of speculative class-loading. In Proceedings of the 5th international symposium on Principles and practice of programming in Java, pages 209–214. ACM.