Android: The Road to JIT/AOT Hybrid Compilation-Based Application User Experience

Introduction

As the Android Open Source Project (AOSP) evolves, we often want to fully understand how design choices will affect User Experience (UX). This article looks at compilation paradigm shifts: how an Android application gets transformed into binary executable code in the next Android release. The primary audience is any Android developer who wants to understand how the evolution of AOSP will impact UX. Product manufacturers also must understand how the Android ecosystem is evolving to ensure that their users get the most performance at first boot, first launch for often-used applications and during system updates.

Lollipop (5.0) Android release introduced a new Virtual Machine (VM) called the Android Run Time (ART). ART replaced Kitkat’s (4.4) Dalvik VM and was deemed to provide a faster, smoother, and more powerful computing experience. With the Marshmallow (6.0) Android release, ART followed up with specific VM tuning to improve UX. These consecutive transformations in the way applications are executed on Android were primarily driven by higher performance expectations.

Lollipop included a new method-based compilation technology at application install time. Next, Marshmallow introduced memory management and Garbage Collection (GC) improvements with the intent of enhancing performance and battery life. In both Lollipop and Marshmallow, ahead-of-time (AOT) compilation forces all methods in an Android application package file (.apk) to be compiled during application installation.

There are several shortcomings to this compilation approach, including noticeably longer application install and compile times. These are critical metrics because end users notice the impact on every application install or over-the-air (OTA) software update, and OEMs rely on fast first boot time during product validation.
In both Lollipop and Marshmallow, AOT installation forces all methods inside an application to be compiled with the same optimization level, with an exception of large methods which might be interpreted, depending on device storage limitations. AOT compiled application binaries also consume significant storage space; as a result, low storage devices sometimes leave applications interpreted. In reality, each user is different and his or her interaction with an Android application is potentially unique. Typical users interact with some applications more than others. Furthermore, some features of a particular application are more commonly used than others.

The rest of this article shows how the AOSP master branch fixes major Marshmallow shortcomings and takes a first step towards obtaining the best performance possible without the shortcomings of long installation, update and boot times and with the benefits of reduced storage space and memory footprint. The next sections will explain how the AOSP master also generates better native code for user applications, and its impact on user perception of application install and launch times, RAM usage, binary size and overall Java performance.

**Application UX: Dynamic compilation, background compilation using profiles**

This section presents the new application compilation workflow. Images are often worth a thousand words, so this section is built around a discussion of the following figure:

![Workflow Diagram](image)

Based on recent development in AOSP master branch (as shown in the N developer preview), the upcoming Android version is expected to no longer force applications to be compiled at install time. The default is to not do any compilation, so that applications install much faster. In
order to maintain the same level of performance, application compilation becomes a hybrid mechanism involving Just-In-Time (JIT) compilation while the application is running, and background compilation occurring when the device is plugged in and stays idle in screen-off mode for long time. These are currently not the default settings in the AOSP master, however JIT enabled by default as part of N developer preview. In AOSP master, JIT and background settings should be enabled explicitly via Android system settings.

At first launch of an application, none of its methods have been compiled. Without AOT compiled code, an interpreter initially runs all the bytecode. This approach improves application installation time, but has negative impact on application startup and runtime performance. For each method, the interpreter starts out counting the number of times a particular method is being entered, the number of loop iterations executed within it and the number of virtual calls it makes. A particular method is considered often-executed, also called warm, when the total of these counts exceeds a threshold. Once warm, more information is gathered about control flow and the actual targets of virtual calls. Once the total of the counts exceeds yet another higher threshold, the warm method becomes hot and is compiled by the JIT compiler into native executable code, which is stored in the JIT code cache along with the collected profile information. The next time the hot method is invoked, native code is executed instead of the interpreter.

One major development seen in the AOSP master based JIT compiler is that ART will record which methods are hot and save their names for later AOT recompilation. When the device is unused (with screen-off) for over long duration, a service will compile the hot methods and save the generated code, expected to happen once every day. When the user launches the application again, the compiled code will be loaded directly into memory and executed. There will be no need to interpret or JIT-compile these hot methods. For this reason, the compiled code can be seen as divided into two parts: AOT loaded methods and JIT’ed methods. After a few days, if a user plays with all the major features of a given application, the often-used code corresponding to those features will all be compiled and the performance should be optimal.

In order to remember which methods are hot, Android now contains a mechanism known as the profile saver. It polls the code cache periodically to obtain the list of hot methods and writes them into a file for later use by the background compiler (Based on N developer preview).

A major advantage to the hybrid approach is that the dynamic (JIT) compiler and the background (AOT) compiler need not be identical or perform the same set of optimizations. That means that the JIT compiler can be a quicker compiler in terms of compile time whereas the background compiler can have a more extensive optimizer. By differentiating the two compilers, the AOSP master branch provides a door to classic big optimizations that have
existed in mainstream compilers for decades. On the flip side, if the device does not have any down-time, then the background compiler cannot do its job. In modern day common phone and tablet use, this is an unlikely event.

First Boot, App Install and Launch Time, RAM and Storage, App Performance

Minimizing the time between when a user performs an action to when the system responds is critical to the best user experience. First boot of the device, waiting for an application to install or launch, and application runtime performance are some of the most critical user perception metrics. Typical users are also concerned with system update time, application memory usage and storage space limitations.

Due to AOT optimizing compilation, device first boot in both Android Lollipop and Marshmallow versions takes significantly longer compared to previous versions of Android. In the upcoming version of Android, first boot should be faster since the system relies on JIT compilation to provide good performance. Application and system update times should also improve. As a result of compiling only methods associated with often-used application features, users can expect greatly reduced application binary size, which saves storage space.

In Lollipop and Marshmallow, application installation takes a noticeably long time due to AOT compilation of the entire application. The larger the application, the worse the problem. In AOSP master with the JIT enabled, the system relies on compiling methods at runtime. This significantly reduces compilation time and RAM footprint, which is important for low memory devices. Application startup, however, is a bit slower, but the AOSP master branch contains a fast interpreter, which helps alleviate the problem.

There is a downside to the shift to the hybrid JIT/AOT compilation model (shown in N preview). As a result of first using an interpreter and having to wait for JIT compilation to finish, some applications may experience sluggishness compared to Marshmallow until the code is compiled. However, the application is expected to recover performance as the interpreter calls the JIT optimizing compiler on commonly used methods. Finally, as the previous section stated, hot code will be interpreted only until it is JIT or AOT compiled. After background compilation, the interpreter is no longer used at all since the previously hot code will have be compiled for the next launch.

Conclusion

AOSP master brings dynamic compilation back to the next generation of Android by reintroducing a Just-In-Time (JIT) compiler. This is a necessary evolution compared to
Marshmallow in order to address excessive application install and first boot time, and memory and disk space consumption. The AOSP master based JIT is not the same as the one used in Android previously, which was phased out in the Lollipop release. It has a larger optimization scope (method based JIT vs. trace based JIT) and has a much more complex infrastructure. Current the AOSP master with JIT enabled compilation infrastructure is able to retain hot method profiles and use them to background recompile hot methods when the phone is idle (screen-off) for a long time and is charging (should be pretty close to behavior in N preview). Generated code performance is thus improved based on a particular user’s application use the next time the application is launched.

This switch to an interpreter plus hybrid JIT/AOT compilation system in AOSP master should lead to a much better user experience with far shorter first boot, install and over-the-air update times with the additional benefits of reduced RAM and storage usage. The interpreter and JIT compiler combination provides a good application launch experience while background compilation should deliver excellent performance after a few days of use. The two elements together should bring the performance of the AOSP master branch to the same or better level as Marshmallow.

Acknowledgements (alphabetical)

Dong-Yuan Chen, Chris Elford, Aleksey Ignatenko, Serguei Katkov, Razvan Lupusoru, Mark Mendell, Dmitry Petrochenko, Desikan Saravanan, Nikolay Serdjuk

About the Authors

Rahul Kandu is a software engineer in the Intel Software and Solutions Group (SSG), Systems Technologies & Optimizations (STO), Client Software Optimization (CSO). He focuses on Android performance and finds optimization opportunities to help Intel's performance in the Android ecosystem.

Jean Christophe Beyler is a software engineer in the Intel Software and Solutions Group (SSG), Systems Technologies & Optimizations (STO), Client Software Optimization (CSO). He focuses on the Android compiler and ecosystem but also delves into other performance-related and compiler technologies.

Paul Hohensee is a principal engineer in the Intel Software and Solutions Group (SSG), Systems Technologies & Optimizations (STO), Client Software Optimization (CSO). He focuses on the runtime and library aspects for Java virtual machines, and is helping change the way we measure Java performance to be application and UX oriented.
Notice

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm
Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.

Intel, the Intel logo, and Intel Atom are trademarks of Intel Corporation in the U.S. and/or other countries.

Copyright © 2016 Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.