How to Optimize Java* Code in Android* Marshmallow

Introduction

As the Android* ecosystem evolves, so do the components that provide performance and a good user experience to the end user. The Android Marshmallow release incorporates new features and capabilities, along with a number of enhancements to the Android Runtime* (ART) that result in improved application performance, lower memory overhead, and faster multitasking. This article is about the Marshmallow release; there is a similar article for Lollipop.

With each release, application developers must understand what's changed in the Android Virtual Machine. In other words, what used to work to get the best performance might no longer be as efficient, and new techniques may produce better results. There is generally little published information about the Android release changes, so developers must figure out these differences through trial and error.

This article describes the Android Runtime for Marshmallow's code generation from a programmer's perspective and gives techniques that developers can use to provide the best user experience to the end user. The tips and tricks provided will help you achieve better code generation and performance. The article also describes why certain optimizations are enabled or independent of the developer's Java* code.

Java Programming Tips for Better ART Performance

The Android ecosystem is complex. One element of that complexity is the compiler that transforms Java into Intel® x86 code for Intel-based devices. Android Marshmallow includes a new compiler, the Optimizing compiler, which optimizes Java programs with better code performance than Lollipop's legacy Quick compiler. Currently in Marshmallow, almost all of the program is compiled using the Optimizing compiler. The Android System Framework methods are compiled using the Quick compiler in order to provide better way for Android developers to debug.
Library Selection and Precision Loss Considerations

Floating-point calculations offer multiple variations of similar operations. In Java, Math and StrictMath offers different precision levels for floating-point operations. Though StrictMath provides a more repeatable calculation, in most cases the Math library will suffice. The following method calculates the cosine:

```java
public float myFunc (float x) {
    float a = (float) StrictMath.cos(x);
    return a;
}
```

However, a developer can use Math.cos(x) instead of StrictMath.cos(x) if precision loss is acceptable. The Math class has been optimized to use the Android Bionic library for Intel® architecture. Intel’s implementation of ART calls math library functions in the Bionic library, which are 3.5 times faster than the equivalent StrictMath methods.

There are cases where StrictMath class is required and should not be replaced by the Math class. However, Math is acceptable for most cases. It is really a question of precision loss, and also depends on the algorithm and implementation.

Support for Recursive Algorithms

Recursive calls are more efficient in Marshmallow compared to Lollipop. When code is written in a recursive way for Lollipop, the self method argument is always loaded from the Dalvik® Executable Format (dex) cache. In Marshmallow a recursive function uses the same self method argument from the initial argument list rather than reloading it from the dex cache. Of course, the greater the recursion depth, the more the performance difference between Lollipop and Marshmallow. However, when an iterative version of an algorithm is available, it still performs better for the Marshmallow release.

Using `array.length` to Eliminate a Bound Check

The Optimizing compiler in Marshmallow is able to eliminate certain array bound checks. See this article for a discussion of bound checking elimination.

An empty loop is:
for (int i = 0; i < max; i++) { }

The variable \( i \) is called the Induction Variable (IV). If the IV is used to access an array, and the loop is iterating over each element, then the array bound check can be removed if \( max \) is explicitly defined as being the array’s length. See this article for more information about induction variables.

**Example**

Consider the following example where the code uses a variable \( size \) as the maximum value for the IV:

```java
int sum = 0;
for (int i = 0; i < size; i++) {
    sum += array[i];
}
```

In this program, the array index \( i \) is compared to \( size \). Let us assume \( size \) is either defined outside the method, passed as an argument, or defined elsewhere in the method. In any of these cases, the compiler may not be able to infer that \( size \) is the length of the array. Due to that uncertainty, the compiler must generate a runtime bound check on \( i \) as part of each array access.

Rewriting the code as below, the runtime bound checks would be eliminated by the compiler.

```java
int sum = 0;
for (int i = 0; i < array.length; i++) {
    sum += array[i];
}
```

**Loops with Two Arrays: Advanced Bound Check Elimination (BCE)**

The previous section showed a simple case of the BCE optimization and what to do to activate it in the compiler. However, there are algorithms where a single loop is handling multiple separate arrays that all have the same length. In this case the compiler must do null and bound checks on both accesses.
The following section takes a closer look at BCE and how to enable it when using multiple arrays. A rewrite of the code is often required to enable the compiler to optimize the loop.

This section’s example considers multiple array accesses in the same loop:

```java
for (int i = 0; i < age.length; i++) {
    totalAge += age[i];
    totalSalary += salary[i];
}
```

There is a problem in this code. The program is not checking the length of `salary` and risks an array index out of bounds exception. The program should check that the lengths are the same before entering the loop, for instance:

```java
for (int i = 0; i < age.length && i < salary.length; i++) {
    totalAge += age[i];
    totalSalary += salary[i];
}
```

Now that the code is correct, there still is a problem because BCE does not work in this case.

In the loop above, the programmer is accessing two single-dimensional arrays: `age` and `salary`. Even though the induction variable, the variable `i`, is set to check for the length of both arrays, the compiler cannot eliminate the bound check for the multiple condition case.

In the loop shown, both arrays are not using the same memory. Therefore, logical operations done on fields of both arrays are independent of each other. Instead, split the operations into two separate loops, like below:

```java
for (int i = 0; i < age.length; i++) {
    totalAge += age[i];
}
for (int i = 0; i < salary.length; i++) {
    totalSalary += salary[i];
}
```

After separating the loops, the Optimizing compiler will eliminate array bound checks from both loops. Java codes can be sped up three to four times for similar simple loops.
BCE works here, but now the function contains two loops resulting in potential code bloat. Depending on the target architecture and the sizes of the loops, or number of times done, this could have an effect on the final generated code size.

Multithreaded Programming Techniques

In a multithreaded program, developer must be careful when accessing data structures. Assume a program spawns four identical threads before the loop shown below. Each thread then accesses an array of integers named thread_array_sum, one cell per thread accessed through a variable myThreadIdx, a unique integer identifying each thread.

```c
for (int i = 0; i < number_tasks; i++) {
    thread_array_sum[myThreadIdx] += doWork(i);
}
```

Some device architectures, such as the Intel® Atom™ x5-Z8000 processor series, do not have a LastLevelCache (LLC) shared by all processor cores. While response time can be better with separate LLCs (since the caches are “reserved” to one or two processor cores), maintaining coherence between them can result in the line “bouncing” between LLCs. Such bouncing can cause performance degradation and processor core scaling issues. See this article for more information.

Because of the cache layout, multiple threads writing to the same array risk performance degradation due to a potential high level of cache thrashing. The programmer should use a local variable to store the intermediate result and then only later update the array. The loop would then become:

```c
int tmp = 0;
for (int i = 0; i < number_tasks; i++) {
    tmp += doWork(i);
}
thread_array_sum[myThreadIdx] += tmp;
```

In this case, the array element thread_array_sum[myThreadIdx] is independent of the inner loop, and the accumulated value from doWork() can be stored in the array element outside the loop. This reduces potential cache thrashing significantly. Thrashing can still happen during the instruction thread_array_sum[myThreadIdx] += tmp but is far less likely.
It is not a good practice to store shared data structures in loops unless the stored values must be visible to other threads at the end of each loop iteration. Such cases will generally require at least the use of volatile fields and/or variables, but this discussion is beyond the scope of this article.

Optimal Code Performance Tips for Low Storage Devices

Android devices are available in a broad range of memory and storage configurations. Java programs should be written to be easily optimized on devices regardless of memory size. Low-storage devices are likely to optimize for space, which is an ART compilation option. In Marshmallow, methods larger than 256 bytes in size are not compiled in order to save storage space on the device, hence Java programs containing large hot methods will execute in the interpreter and perform poorly. For best performance in Marshmallow, implement often-used code into small methods to fully enable the compiler optimizations.

Java programs written as small methods are more likely to be compiled by ART regardless of device storage limitations, resulting in up to three times better performance on large Android applications.

Summary

Every Android release comes with new elements and different technologies. As was the case for KitKat and Lollipop, Marshmallow comes with big changes to the compiler technology in the Android ecosystem.

As with Lollipop, the ART uses an Ahead-of-Time compiler, which generally transforms user applications into native code at install time. However, instead of using the Lollipop Quick compiler, Marshmallow uses a new compiler called the Optimizing compiler. Though there are cases where the Optimizing compiler defers to the Quick compiler, the Optimizing compiler is the new centerpiece of Android Java binary code generation.

Each compiler has its own quirks and optimizations, so each compiler may generate different binary code depending on how the Java program is written. This article presented a few of the major differences one can see with the Marshmallow release and what a developer should be aware of when using it.
From math library usage to bound check elimination, there are many new features in the Optimizing compiler. Some are difficult to show by example since much optimization is done “under the hood.” What we do know is that the maturity of the compiler in Android is starting to show as its technology is becoming more advanced and slowly catching up with other optimizing compilers. As the compiler evolves, developers can be confident that the code they write will be well optimized and provide a better end-user experience, something everyone can appreciate.

Acknowledgements (alphabetical)

Johnnie L Birch, Jr., Dong Yuan Chen, Chris Elford, Haitao Feng, Paul Hohensee, Aleksey Ignatenko, Serguei Katkov, Razvan Lupusoru, Mark Mendell, Desikan Saravanan, and Kumar Shiv

About the Authors

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.

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.

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.