5 Ways to Optimize Your Code for Android 5.0 Lollipop

Introduction
With the release of Android 5.0 Lollipop*, an innovative default runtime environment was introduced, called ART* (short for Android RunTime). It includes a number of enhancements that improve performance. In this paper, we introduce some of the new features in ART, benchmark it against the previous Android Dalvik* runtime, and share five tips for developers that can further improve application performance.

What’s new in ART?
Profiling many Android applications on the Dalvik runtime identified two key pain points for end users: the time it takes to launch an app, and the amount of jank. Jank occurs when an application is stuttering, juddering or simply halting because an app isn’t keeping up with the screen refresh rate, and is the result of the frame setup taking too long. A frame is defined as janky when it is much faster or slower than the previous frame. Users see jank as jerky motion which makes the user experience less fluent than users and developers would wish for. To address these issues, there are several new features in ART:

- **Ahead of time compilation**: At install time, ART compiles apps using the on-device dex2oat tool and generates a compiled app executable for the target device. By comparison, Dalvik used an interpreter and just-in-time compilation, converting an APK into optimized dex bytecode at installation time, and further compiling the optimized dex bytecode into native machine code for hot paths when the application is run. The result is that applications launch faster under ART, although the price is that they take longer to install. Applications also use more flash memory space on the device under ART because the code compiled at install time takes up extra space.

- **Improved memory allocation**: Applications that need to allocate memory intensively might have experienced sluggish performance on Dalvik. A separate large object space and improvements in the memory allocator help to alleviate this.

- **Improved garbage collection**: ART has faster and more parallel garbage collection, resulting in less fragmentation and better use of memory.

- **Improved JNI performance**: Optimized JNI invoke and return code sequences reduce the number of instructions used to make JNI calls.

- **64-bit support**: ART makes good use of 64 bit architectures, improving the performance of many applications when run on 64-bit hardware.

Together, these features improve the user experience of applications written using the Android SDK alone, as well as applications that make lots of JNI calls. Users may also benefit from longer battery life because applications compile only once and execute faster, and consume less power during routine use as a result.
Comparing performance in ART and Dalvik

When ART was first released as a preview on Android KitKat 4.4, there was some criticism of its performance. That wasn’t a fair comparison because an early preview version of ART was being compared to the fully matured and optimized Dalvik, with the result that some applications ran slower under ART than under Dalvik.

We now have an opportunity to compare the consumer-ready version of ART against Dalvik. Because ART is the only runtime in Android 5.0, a side by side comparison of Dalvik and ART is only possible if you compare devices that have recently been updated from Android KitKat 4.4 to Android Lollipop 5.0. For this paper, we have conducted tests using the TrekStor SurfTab xintron i7.0* tablet with an Intel® Atom™ processor, initially with Android 4.4.4 running Dalvik, and then updated to Android 5.0 running ART.

Since we are comparing different versions of Android it is possible that some of the improvements that we see come from Android 5.0 Improvements other than ART, but based on our internal performance analysis we found that ART is the cause of most of the improvements.

We ran benchmarks where Dalvik’s ability to aggressively optimize code that is repeatedly executed might be expected to give it an advantage, as well as Intel’s own gaming simulation.

Our data shows that ART outperforms Dalvik on the five benchmarks we tested, in some cases significantly.

For more information on these benchmarks, see these links:

- Quadrant 2.1.1*
- CaffeineMark 3.0*
- Smartbench 2012 productivity*
- Antutu 4.4 Overall* (note that the version we tested is no longer available for download)
IcyRocks version 1.0 is a workload developed by Intel to mimic real-world gaming applications. It uses the open source Cocos2d* library along with JBox2D* (A Java Physics Engine) for most of its computations. It measures the average number of animations (frames) it can render per second (FPS) at various load levels and then computes the final metric by taking a geometric mean of the FPS at these various load levels. It also measures the degree of jank (jank per second), which is the mean of janky frames per second at various load levels. It shows improved performance in ART compared to Dalvik:

Based on this performance evaluation, it is clear that ART is already delivering a better user experience and better performance than Dalvik.
Moving code from Dalvik to ART
The transition from Dalvik to ART is transparent and most applications that run on Dalvik should run on ART without requiring modification. As a result, many applications will see a performance improvement when users upgrade to the new runtime. It’s still a good idea to test your application with ART, especially if it uses the Java Native Interface, as ART’s JNI error handling is stricter than Dalvik’s, as explained in this article.

Five tips for optimizing your code
Most applications will experience a performance increase as a result of the improvements in ART detailed above. Additionally, there are several practices you can adopt that may help to optimize your application further for ART. For each technique below, we’ve provided some simplified code to illustrate how it works.

Because all applications differ and the resulting performance depends so much on the surrounding code and context, it’s not possible to provide an indication of the performance increase you can expect. However, we will explain why these techniques increase performance, and we recommend that you test them in the context of your own code to see how they affect your performance.

The tips we provide here are broadly applicable, but in the case of ART, the dex2oat compiler that generates binary executable code from a dex file will implement these optimizations.

Tip #1 – Use local variables instead of public class fields when possible.
By limiting the scope of variables, you can not only make your code more readable and less error-prone, but also more optimization-friendly.

In the unoptimized code below, the value of $v$ is calculated when the application runs. That’s because $v$ is accessible from outside the method and can be changed by any code, so its value is not known at compilation time. The compiler doesn’t know whether the `some_global_call()` operation changes $v$ or not, because $v$ can be changed from outside the method by any code.

In the optimized code, $v$ is a local variable and its value can be calculated at compilation time. As a result, the compiler can put the result directly into the code and avoid the calculation at runtime.

Unoptimized code
```java
class A {
    public int v = 0;

    public int m(){
        v = 42;
        some_global_call();
        return v*3;
    }
}
```

Optimized code
```java
class A {
```
public int m(){
    int v = 42;
    some_global_call();
    return v*3;
}

Tip #2 – Use the final keyword to hint that a value is constant

The final keyword can be used to protect your code from accidentally modifying variables that should be constant, but can also improve performance by giving the compiler a hint that a value is constant.

In the unoptimized code below, the value of \( v^2 \) must be calculated at runtime, because the value of \( v \) could change. In the optimized code, using the keyword final when assigning a value to \( v \) tells the compiler that this value won’t change, so the calculation can be performed during compilation and the result can be added into the code, removing the need to calculate it at runtime.

Unoptimized code

```java
class A {
    int v = 42;

    public int m(){
        return v*v*v;
    }
}
```

Optimized code

```java
class A {
    final int v = 42;

    public int m(){
        return v*v*v;
    }
}
```

Tip #3 – Use the final keyword for class and method definitions

Because all methods in Java are potentially polymorphic, declaring a method or class as final tells the compiler that the method is not redefined in any subclass.

In the unoptimized code below, \( m() \) must be resolved before making the call.

In the optimized code, because the method \( m() \) was declared as final, the compiler knows which version of \( m() \) will be called. As a result, it can avoid method look-up and inline the call, replacing the call to \( m() \) with the contents of its method. This results in a performance increase.

Unoptimized code

```java
class A {
    public int m(){
        return 42;
    }
}
```
public int f(){
    int sum = 0;
    for (int i = 0; i < 1000; i++)
        sum += m(); // m must be resolved before making a call
    return sum;
}

Optimized code
class A {
    public final int m(){
        return 42;
    }

    public int f(){
        int sum = 0;
        for (int i = 0; i < 1000; i++)
            sum += m();
        return sum;
    }
}

Tip #4 – Avoid JNI calls for small methods.
There are good reasons to use JNI calls, such as when you have a C/C++ codebase or library to reuse, you need a cross-platform implementation, or you need increased performance. But it’s important to minimize the number of JNI calls, because each one carries a significant overhead. When JNI calls are used to optimize performance, this overhead can result in not realizing the expected benefits. In particular, frequently calling short JNI methods can be counter-productive, and putting JNI calls in a loop can amplify the overhead.

Code example
class A {
    public final int factorial(int x){
        int f = 1;
        for (int i = 2; i <= x; i++)
            f *= i;
        return f;
    }

    public int compute (){)
        int sum = 0;
        for (int i = 0; i < 1000; i++)
            sum += factorial(i % 5);
        // if we used the JNI version of factorial() here
        // it would be noticeably slower, because it is in a loop
        // and the loop amplifies the overhead of the JNI call
        return sum;
    }
Tip #5 – Use standard libraries instead of implementing the same functionality in your own code

Standard Java libraries are highly optimized and often use internal Java mechanisms to get the best possible performance. They might work significantly faster than when the same functionality is implemented in your own application code. Attempts to avoid the overhead of calling a standard library might actually result in lower performance. In the unoptimized code below, there is custom code to avoid calling Math.abs(). However, the code that uses Math.abs() works faster because Math.abs() is replaced by an optimized internal implementation in ART at compile time.

**Unoptimized code**
```java
class A {
    public static final int abs(int a){
        int b;
        if (a < 0)
            b = a;
        else
            b = -a;
        return b;
    }
}
```

**Optimized code**
```java
class A {
    public static final int abs (int a){
        return Math.abs(a);
    }
}
```

**Intel optimizations in ART**

Intel worked with OEMs to provide an optimized version of Dalvik that provides better performance on Intel processors. Intel is making the same investment in ART, so performance will further increase on the new runtime. Optimizations will be made available through the Android Open Source Project (AOSP), and/or directly through device manufacturers. The optimizations will, as before, be transparent to developers and users so there will be no need to update applications to benefit.

**Find out more**

To find out more about optimizing your Android applications for Intel processors, and to discover Intel® compilers, visit the Intel Developer Zone at [https://software.intel.com](https://software.intel.com).

**Disclaimers**

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY
WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENCE IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

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.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

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

- Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.
- Intel does not control or audit the design or implementation of third party benchmarks or Web sites referenced in this document. Intel encourages all of its customers to visit the referenced Web sites or others where similar performance benchmarks are reported and confirm whether the referenced benchmarks are accurate and reflect performance of systems available for purchase.
- Relative performance is calculated by assigning a baseline value of 1.0 to one benchmark result, and then dividing the actual benchmark result for the baseline platform into each of the specific benchmark results of each of the other platforms, and assigning them a relative performance number that correlates with the performance improvements reported. No computer system can provide absolute security. Requires an enabled Intel® processor and software optimized for use of the technology. Consult your system manufacturer and/or software vendor for more information.
- Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families: Go to: Learn About Intel® Processor Numbers

Copyright © 2015, Intel Corporation. All rights reserved. Intel, the Intel logo, Intel Core, and Intel Atom are trademarks of Intel Corporation in the U.S. and other countries.

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

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel.

Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804