Intel Optimizations in the Android* Compiler

By Jean Christophe Beyler

Acknowledging (alphabetical):
Johnnie L Birch Jr, Dong-Yuan Chen, Olivier Come, Chao-Ying Fu, Jean-Philippe Halimi, Aleksey V Ignatenko, Rahul Kandu, Serguei I Katkov, Maxim Kazantsev, Razvan A Lupusoru, Yixin Shou, Alexei Zavjalov

Introduction

As the Android* ecosystem continues to evolve, Intel is working with OEMs to provide an optimized version of the Android runtime, thus providing better performance on Intel® processors. One of the ecosystem components is the compiler, which has been available for a few years but has recently undergone massive changes.

In the Android ecosystem, the application developer generally writes a section of the application in the Java* language. This Java code is then transformed into an Android-specific bytecode, called the Dex bytecode. In the various flavors of Android's major releases, there have been several ways to go from the bytecode format to the actual binary format that is run on the processor. For example, in the Jelly Bean* version, there was a just-in-time (JIT) compiler in the runtime, called Dalvik* VM. From KitKat* on, the appearance of a new Android runtime (ART), added the existence of an ahead-of-time (AOT) compiler in ART.

When the compiler is young, there are often stepping-stones to achieving the state-of-the-art transformations one can expect from a more mature compiler. Since the ART compiler is young, the way a programmer writes the code can impact the ability of the compiler to generate optimal code.(For tips on how you can optimize your code for this compiler, see 5 Ways to Optimize Your Code for Android 5.0 Lillipop.) On the other hand, developing a more generic compiler is a huge effort, and it takes time to implement the initial infrastructure and optimizations.

This article describes the basic infrastructure and two optimizations that Intel implemented in the ART compiler. Both optimizations provide a means to simplify compile-time evaluated loops. These elements are only part of Intel's commitment—amid much more infrastructure and optimizations—to provide the best user experience for the Android OS and are presented here to show how synthetic benchmarks are greatly optimized (improved??) by these optimizations.

Optimizations

Since 2008, there have been three variations of Google's Android compiler. The Dalvik Jit compiler, available since Froyo, was replaced by the Quick AOT compiler in the Lollipop version of Android. This succession represented a major change in the paradigm for transforming the Dex bytecode, which is Android's version of the Java bytecode, into binary code. As a result of this change, most of the middle-end optimization logic from the Dalvik VM was copied over to Quick, and, though AOT compilation might provide the compiler more time for complex optimizations, Quick provided infrastructure and optimizations similar to that of the Dalvik VM. In late 2014, the ASOP added a new compiler named Optimizing. This latest version of the compiler is a full rewrite of the Quick compiler and, over time, seems to be adding more and

more infrastructure to enable more advanced optimizations, which were not possible in previous compiler versions.

At the same time, Intel has been working on its own infrastructure and optimizations into the compiler to provide the best user experience for Intel processor-based devices. Before presenting the optimizations that were implemented by Intel in the Android compiler, the next section shows two small optimizations that are classic in production compilers and help provide a better understanding of the two Intel optimizations.

Constant Folding and Store Sinking

Constant Folding and Store Sinking are very known optimizations in compiler literature. This section presents them and shows how the Android* compiler builds upon them to deliver more complex optimizations. Constant Folding used in the Android* compiler was developed by Google, while the Store Sinking algorithm is implemented by Intel.

Consider the code:

```
a = 5 + i + j;
```

If the compiler is able to determine that *j* is actually equal to 4, the compiler simplifies the code into:

```
a = 5 + i + 4;
```

Then, with a bit of rewriting, it obtains:

```
a = 5 + 4 + i;
```

At this point, the compiler is generally able to simplify this into:

```
a = 9 + i;
```

The transformation *(5 + 4)* to *9* is called constant folding. In a way, the compiler has deemed that the calculation of the addition can be done during compilation and doesn't not have to wait for the generated code to figure out the result is 9. This is a key concept used during the Trivial Loop Evaluator optimization presented next.

Now what happens when we have a loop and the constant is added:

```
for (i = 0; i < 42; i++) {
  a += 4;
}
```

At this point, it is not clear that Constant Folding can simplify the code. However, as the article will show, both the Constant Calculation Sinking and the Trivial Loop Evaluator optimizations can.

Another common optimization is to delay the store of a variable until after the loop is done. For example:

```
for (i=0; i < 42; i++) {
   a += 4;
   final_value = a;
}
```

Since the loop is not using the value stored in *final_value*, the compiler can sink the assignment after the loop, transforming the code into:

```
for (i=0; i < 42; i++) {
   a += 4;
}
final_value = a;
```

Note that the current implementation only does this for class members and not local variables. Both these simple optimizations showcase simple transformations that the compiler can do. The following two optimizations are more complex but build on these two concepts: if you know that you can delay the calculation, then you can put the code *after* the loop.

The next sections present two optimizations implemented in the Android compiler and build on Google's implementation of Constant Folding and Intel's implementation of Store Sinking.

Constant Calculation Sinking

The Constant Calculation Sinking optimization sinks any calculation based on constants that can be evaluated at compile time.

Consider the following loop:

```
public void AddEasyBonusPoints(int basePoints) {
  for (int i = 0; i < GetBricks(); i++) {
     basePoints += GetEasyPointsPerBrick(i);
  }
}
```

Assume that the function is from a Breakout-style game, where the player tries to destroy all the bricks using a bounding ball. In the game, the user can trigger a bonus score. The bonus score is the sum of the brick points of the level.

GetBricks simply returns the integer 42 because that version of the level has 42 bricks and GetEasyPointsPerBrick simply returns 10 as the value because, for example, at the Easy level each brick has the same cost. This is a plausible, albeit simple, piece of code and if the compiler has a good inliner, the loop is transformed into the version shown below.

Inlining

Before performing this optimization, the first step is to inline the two methods as stated previously. Once inlined, the code looks like this:

```
public void AddEasyBonusPoints(int basePoints) {
  for (int i = 0; i < 42; i++) {
     basePoints += 10;
  }
}
```

At this point, it is clear that the calculation does not require a loop. Intuitively, the calculation can be simplified to:

```
public int AddEasyBonusPoints(int basePoints) {
 return basePoints + 420;
}
```

The rest of this section explains how the compiler supports and handles this and how the optimization handles other operations such as subtraction or multiplication.

Inner Workings: Integer Cases

The optimization rewrites statements of the form:

```
var += cst;
```

by moving the calculation after the loop (an operation known as *sinking*). The calculation can also be a subtraction, remainder, division, or multiplication. The following table shows the possible transformations.

Operation	Transformed Into
var += cst	var += cst * N, where N is the number of iterations
var -= cst	var -= cst * N, where N is the number of iterations
var %= cst	var %= cst
var *= cst var /= cst	var = result, if result does not overflow and operation can be safely sunk var = 0, if 0 is obtained during evaluation of the calculation

Of course, for the optimization to be safe, a few checks must take place. First, the variable must not be used in the loop. If the accumulation is being used, sinking the end result creates a problem because the intermediate value is required *during* the loop's execution. A second requirement is that the bounds of the loop must be known at compile time.

For multiplication and division, as the algorithm calculates the value's progression, the calculation hits a fixed point if the value becomes zero. Zero times or divided by anything is zero; therefore the calculation can be simplified to being 0.

Supporting Floating-Point Cases

While working on the integer version, the multiplication and division case spurred the idea that a similar technique could be used for the floating-point cases. The optimization can do two things: first, it can sink the result if fully calculated, meaning the compiler was able to evaluate the full calculation. Second, if the iteration count is high enough, the compiler sinks zero. That mechanism brought the idea that multiplying by a given value in floating-point might lead to infinity and dividing by a value might lead to 0.0.

However, compared to the integer case where the compiler can determine that regardless of the initial value the result would lead to 0 in certain cases; in the floating-point case the initial value is required to ensure calculation convergence.

Impact on CF-Bench*

The Constant Calculation Sinking optimization is performed on small loops where the accumulation by a constant can be sunk, resulting in better performance. When considering the various existing Android benchmarks, the CF-Bench benchmark suite contained a few of those loop types.

Before showing the results of applying the Constant Calculation Sinking optimization, Intel's ART team added a second small optimization, which removed unused loops; that is, loops that have no side effects, invokes, possible exceptions, and no calculations used by code after the loop. This is a common optimization that allows the compiler to essentially remove dead code. This process allows the loop to be completely removed after all the constant calculations have been sunk, which would otherwise leave only the skeleton of the *for* loop.

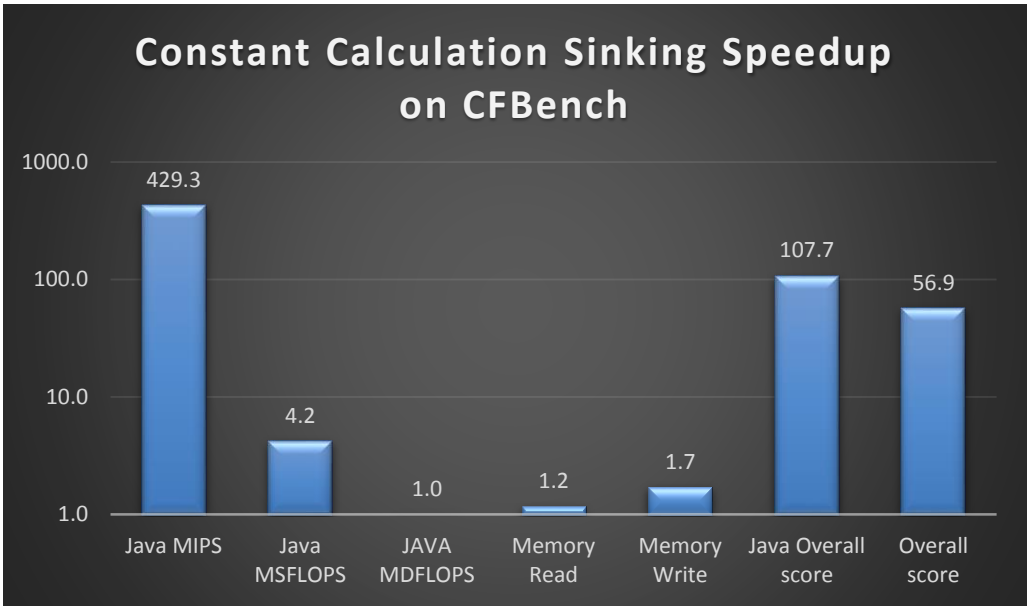

Figure 1: Constant Calculation Sinking speedup on CF-Bench* on the TrekStor SurfTab xintron i 7.0

Figure 1 shows the speedup achieved on various subtests of the CF-Bench benchmark suite. There is a major speedup on the integer versions due to the compiler removing the whole loop

of each case after sinking the various calculations. For the single-precision floating-point version, named Java MSFLOPs, the optimization does provide better performance, though it is only a 4x improvement.

Summary

Constant Calculation Sinking is an important optimization. When considered individually, it is not immediately clear whether it can be applied to real world cases, but the game example shows how candidate loops are recognized. Integrating the optimization provides a means to optimize loops that get transformed by the inliner method, for example.

For the Constant Calculation Sinking optimization, the synthetic benchmark CF-Bench benefited in terms of score for a few of its subtests. In the compiler domain, it is rare to have speedups on the order of magnitude such as the one obtained for the benchmark. It shows that the transformations to the MIPS subtest score shadow any future optimization applicable to other subtests.

Trivial Loop Evaluator

The second optimization is a generalization of Intel's Constant Calculation Sinking optimization. Let's look at another example from the Breakout method and assume the developer wants to provide the maximum number of points for the level:

```
public int CalculateMaximumPoints() {
    int sum = 0;
    // Starting points per brick is 10.
    int points_per_brick = 10;
    int num_colors = GetBrickColors();
    // Go through the number of colors.
    for (int i = 0; i < num_colors; i++) {
      int num_bricks = GetNumberOfBricks ( i );
      int points_in_level = num_bricks * points_per_brick;
      points_per_brick += GetPointIncrementPerLevel ( i );
      sum += points_in_level;
    }
    return sum;
  }
Now the three helper functions are imagined to be:
  public final int GetBrickColors() {
    // Six colors: red, blue, green, black, white, yellow.
    return 6;
  }
  public final int GetNumberOfBricks(int i) {
    // We want less and less bricks as the brick level goes up.
    //   This provides us with: 84, 75, 64, 51, 36, 19
    //  We assume i is < 7, if it is not, we are negative :), call it a feature for this example.
    return 100 - (i + 4) * (i + 4);
  }
  public final int GetPointIncrementPerLevel(int i) {
    // 10 points per level, makes the progression a bit less linear.
    return 10 * i;

  }
```

These are simple methods that perform a calculation, but it is not trivial to pre-calculate this loop by hand, and, in any case, as the developer tweaks the code of the helper functions, updating the maximum number of points is not manageable. That being said, let's focus on what the compiler does with this.

Inliner

The first step lies once again in inlining the methods. Since the three methods are simple enough, they are inlined directly in the method, which means that the compiler then considers the method as if it were actually written as:

```
public int CalculateMaximumPoints() {
    int sum = 0;
    // Starting points per brick is 10.
    int points_per_brick = 10;

    int num_colors = 6;
    // Go through the number of colors.
    for (int i = 0; i < num_colors; i++) {
      int num_bricks = 100 - (i + 4) * (i + 4);
      int points_in_level = num_bricks * points_per_brick;
      points_per_brick += 10 * i;
      sum += points_in_level;
    }
    return sum;

  }
```

Once this is done, the loop can actually be entirely computed at compile time, since it is using only local variables and does not rely on data from memory. Therefore, it can be evaluated and simplified at compile time into:

```
public int CalculateMaximumPoints() {
    sum = 9520;
    return sum;
 }
```

This is much simpler and much more efficient! The implementation of the Trivial Loop Evaluator is implemented in a simple manner: First calculate all the input values and their initial constant values. Then during the loop evaluation, remember the state of all the registers and emulate instruction-by-instruction and update the register state. Finally, once the loop completes, calculate the outputs that require updating and emit the instructions.

Impact on Quadrant*

Figure 2 shows the benefit of the Trivial Loop Evaluator on the Quadrant benchmark suite. The suite contains a few subtests that benefit from it. Note that the scores are improved by a factor of 6 for the Double subtest and to 60x for the Long and Short subtests. Finally, the overall score improved by 26x.

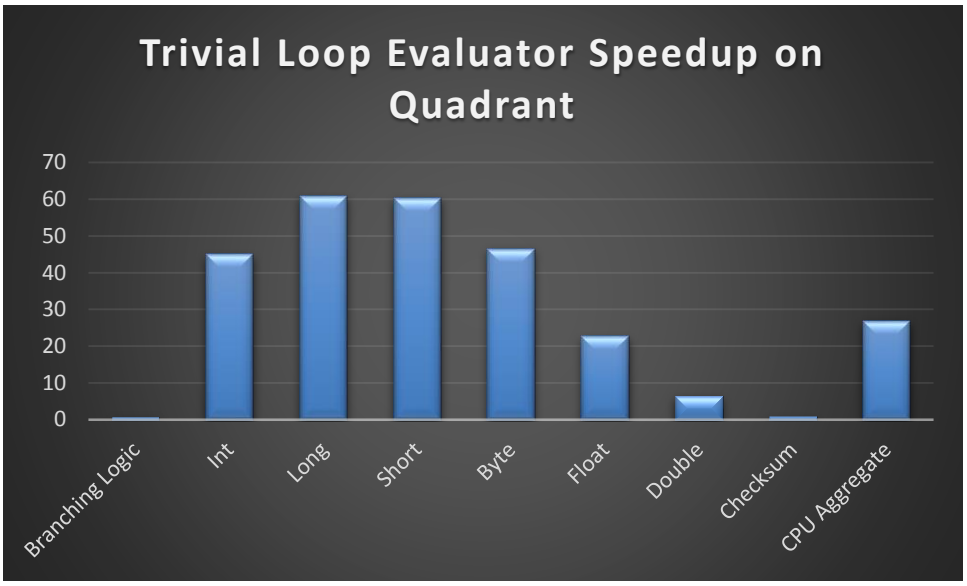

Figure 2. Trivial Loop Evaluator's impact on the Quadrant* benchmark on the TrekStor SurfTab xintron i 7.0

Inherently, when the optimization is used, the loop is removed. Therefore, when it is applied, the score of the benchmark jumps if that is the loop being timed.

Summary

The Trivial Loop Evaluator is a powerful optimization when every loop input is available to the compiler. For this optimization to apply realistically, other optimizations such as inlining must occur as well. This is paramount to the two optimizations discussed in this article: developers generally don't write loops that are applicable per se.

Conclusion

This article described two optimizations—Constant Calculation Sinking and Trivial Loop Evaluation—that the Intel team worked on. These optimizations are a stepping-stone for more advanced optimizations for two reasons: they are used at the end of the code simplification process and they also simplify loops, which helps other optimizations later further the process.

The team's future work lies in enabling more calculations and loops to be simplified and removed. More aggressive inlining is the first task that comes to mind. As the examples showed, the first step in simplifying the code is often inlining code and then optimizing the new version.

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.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

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. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

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.