Static Dalvik Bytecode Optimization for Android Applications

Jeehong Kim, Inhyeok Kim, Changwoo Min, Hyung Kook Jun, Soo Hyung Lee, Won-Tae Kim, and Young Ik Eom

Since just-in-time (JIT) has considerable overhead to detect hot spots and compile them at runtime, using sophisticated optimization techniques for embedded devices means that any resulting performance improvements will be limited. In this paper, we introduce a novel static Dalvik bytecode optimization framework, as a complementary compilation of the Dalvik virtual machine, to improve the performance of Android applications. Our system generates optimized Dalvik bytecodes by using Low Level Virtual Machine (LLVM). A major obstacle in using LLVM for optimizing Dalvik bytecodes is determining how to handle the high-level language features of the Dalvik bytecode in LLVM IR and how to optimize LLVM IR conforming to the language information of the Dalvik bytecode. To this end, we annotate the high-level language features of Dalvik bytecode to LLVM IR and successfully optimize Dalvik bytecodes through instruction selection processes. Our experimental results show that our system with JIT improves the performance of Android applications by up to 6.08 times, and surpasses JIT by up to 4.34 times.

Keywords: Dalvik bytecode, static optimization, LLVM, Android.

Manuscript received Jan. 29, 2014; revised June 2, 2015; accepted July 30, 2015.

This work was supported by Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (10041244, SmartTV 2.0 Software Platform).

Jeehong Kim (jjilong@skku.edu), Inhyeok Kim (kkojiband@skku.edu), Changwoo Min (changwoo@gatech.edu), and Young Ik Eom (corresponding author, yieom@skku.edu) are with the College of Information and Communication Engineering, Sungkyunkwan University, Suwon, Rep. of Korea.

Hyung Kook Jun (hkjun@etri.re.kr), Soo Hyung Lee (soohyung@etri.re.kr), and Won-Tae Kim (wtkim@koreatech.ac.kr) are with the SW & Contents Research Laboratory, ETRI, Daejeon, Rep. of Korea.

I. Introduction

Mobile devices have limited processing power, memory, and battery life; thus, optimizing mobile applications for better performance is critical for their successful deployment [1]. However, since optimization requires high-level technical expertise, it is a daunting task for an application developer. Therefore, automatic code optimization techniques are widely used to achieve high performance in applications for mobile devices.

An Android application is written in Java language for developer productivity and code mobility. On desktop and server environments, a Java program is compiled to Java bytecodes, which is an intermediate representation (IR) for Java Virtual Machine (JVM); JVM runs the Java bytecodes. A Java bytecode is based on a stack-based instruction set (hence the term "stack-based Java bytecode") and has object-oriented features in Java language. On the other hand, Android applications written in Java language are run on Dalvik Virtual Machine (DVM) [2], which is an optimized virtual machine for the Android platform. However, since the performance of stack-based Java bytecode on resource-limited mobile devices is poor due to slow interpretation, Android researchers have created a new bytecode set for DVM — Dalvik bytecode — to improve the performance of Android applications. In contrast to the Java bytecode, the Dalvik bytecode is based on a register-based instruction set; therefore, it can reduce the code size and running time [3]–[4].

In terms of automatic code optimization, JVM researchers have devoted most of their efforts in developing just-in-time (JIT) compilers rather than in developing static Java compilers [5]–[6]. However, when applications run, considerable runtime

overhead, a byproduct of JIT compilation, is incurred in the detection of hot spots and their subsequent transformation into machine codes [7]. Moreover, since JIT compilation has additional processing and memory space overhead, DVM JIT [8] adopts only simple optimization techniques that have low overhead. Optimization techniques implemented in Dalvik JIT can be classified into the following three classes: local optimization, global optimization, and simple peephole optimization. The local optimization techniques in Dalvik JIT are constant propagation, register allocation, redundant load/store elimination, redundant null-check elimination, and heuristic scheduling such as load hoisting and store sinking. The global optimization techniques include redundant branch elimination and induction variable optimization.

Peephole optimization performs power-reduction in multiplication and division operations. Due to the high runtime overhead, more sophisticated optimization techniques such as aggressive loop optimization and peephole optimization are not implemented in DVM JIT. In addition, the scope of *method inlining* is limited only to string, math, and atomic operations.

In Android 4.4, ART runtime [9] was introduced, which is a new Android runtime to boost the performance of Android devices based on ahead-of-time compilation. ART runtime precompiles Dalvik bytecode to binary during installation of an application, and the resulting binary code is then executed directly at runtime of an application. However, this scheme requires considerable install time of an application and more space and memory footprint than existing Dalvik bytecode.

Previous studies to improve the performance of Android applications can be largely classified into two areas — those that adopt complementary compilations of DVM [10]–[11] and those that adopt modifications of DVM [4], [11]–[13]. However, previous studies have yet to adequately apply their optimization techniques to new versions of DVM because their techniques adopt the unique compilation on a particular version of DVM and a special hardware support. With regard to the first of these issues, since the Android platform is under active development, it requires a lot of effort to maintain the various modifications of DVM when a new version becomes available. Moreover, because Android applications are composed of Dalvik bytecodes, previous transformation techniques for Java codes are not helpful in optimizing Android applications [14]–[16]. With regard to the second of these issues, development of a new optimization system [17]–[18] requires considerable efforts on the part of developers to develop and verify any such new systems; thus, it is costly in terms of time and effort. From this perspective, using a mature compiler infrastructure, such as Low Level Virtual Machine (LLVM) [19]–[20] and GCC [21], has clear advantages in terms of development efforts and maturity. While previous studies [14]–[15] also used mature compiler infrastructures to transform Java programs to machine codes, they are not considered for execution on mobile devices due to their high resource requirements.

In this paper, we introduce a novel static Dalvik bytecode optimization system to complement the DVM. We elaborate on optimizations to transform the Dalvik bytecodes into optimized Dalvik bytecodes. Since the final output of our system is in that of a Dalvik bytecode format, we can improve the performance of Android applications without any runtime overhead, while maintaining code mobility. We make the following contributions in this paper:

- We find that there are further opportunities to optimize Dalvik bytecodes in Android applications.
- We design and implement a static Dalvik bytecode optimization framework by exploiting optimization passes in LLVM. We discuss the challenges involved in handling high-level language features in LLVM and determine how to optimize the Dalvik bytecode.
- We evaluate our Dalvik bytecode optimization framework by using several benchmarks and show its effectiveness.

The remainder of this paper is organized as follows. In Section II, we describe the design and implementation of our static Dalvik bytecode optimization framework. Experimental results are shown in Section III. In Section IV, related work is described. Finally, in Section V, we conclude the paper.

II. Static Dalvik Bytecode Optimization

In this section, we discuss how our framework optimizes the Dalvik bytecode while preserving the high-level language features of the bytecode itself. Our system at first transforms Dalvik bytecode to LLVM IR and statically optimizes the transformed LLVM IR. It then finally generates the optimized Dalvik bytecode from the optimized LLVM IR. For an explanation, we use the sample code in Fig. 1 and show the optimized code in Figs. 2, 5, and 6.

1. Motivating Example

Figure 1 illustrates the motivating example of our research. The Java code shown in Fig. 1(a) is transformed into the Java bytecode shown in Fig. 1(b) and again into the Dalvik bytecode shown in Fig. 1(c). Since the summation statement, $sum = a + b$, shown in dark gray is invariant regardless of the loop iterations, it can be removed from the loop represented by the dotted box to avoid unnecessary computation. This optimization technique is called loop-invariant code motion — a widely used compiler optimization technique. Lines 4 to 7 in Fig. 1(b) and Line 9 in Fig. 1(c) are Java bytecode and Dalvik bytecode corresponding to Line 6 in Fig. 1(a), respectively. As

ETRI Journal, Volume 37, Number 5, October 2015
http://dx.doi.org/10.4218/etrij.15.0114.0036

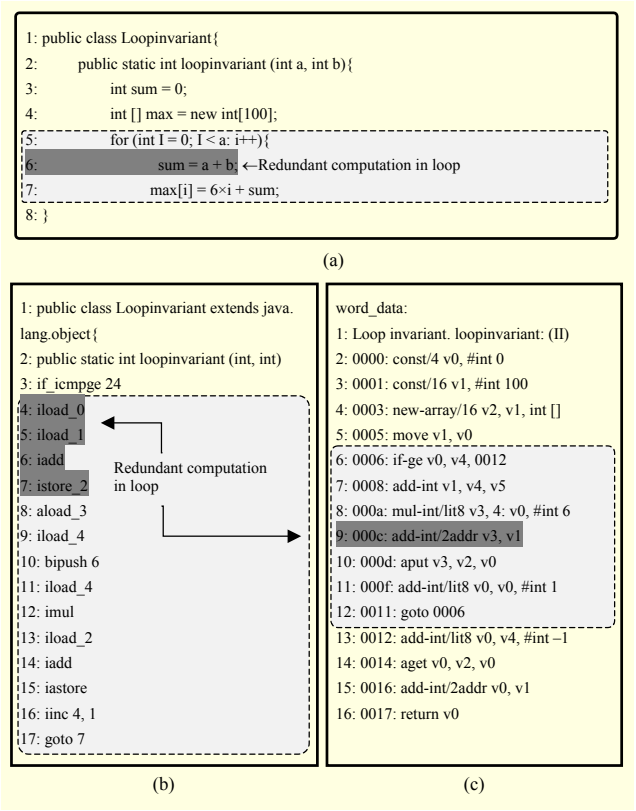

Fig. 1. Motivating example: (a) Java code, (b) Java bytecode, and (c) Dalvik bytecode.

shown in Figs. 1(b) and 1(c), although the Java code in Fig. 1(a) is compiled by the Java compiler and transformed by the Dx tool [22], we find that the loop-invariant statement, *add-int/2addr v3*, *v1*, in the dark gray box remains inside the loop; consequently, this statement becomes a redundant operation, directly decreasing program performance.

To measure how redundant statements decrease the performance, we run the motivating example in Fig. 1(a) on an Android device. When we manually optimized the code of Fig. 1(a) by removing the loop-invariant code, its performance improvement is 21% in only-DVM interpretation and 90% in JIT-enabled DVM interpretation (even though the resulting sample code is relatively simple). This demonstrates that performance can significantly be improved by performing static aggressive optimization before JIT compilation. Since static optimization can perform aggressive optimization techniques without any runtime overhead, it is beneficial in particular to Android mobile devices with limited processing power, memory, and battery.

2. Comparison between Dalvik Bytecode and LLVM IR

Our Dalvik bytecode optimization system exploits LLVM for extensive static optimization of the Dalvik bytecode as a complementary compilation for DVM. LLVM [19]–[20] is a mature compiler infrastructure that uses a language-independent low-level instruction set, called LLVM IR. LLVM front-end generates the LLVM IR code from application code. After the LLVM optimizer optimizes the LLVM IR code in a language-independent way using aggressive optimization passes, the LLVM back-end optimizes the LLVM IR code in a target architecture–dependent way; finally, the machine codes of the target architecture are generated. Therefore, resolving the impedance mismatches between the high-level Dalvik bytecode and the low-level LLVM IR code is challenging when attempting to create a static optimization system for Dalvik bytecode using LLVM.

Table 1. Comparison between Dalvik bytecode, LLVM IR, and machine code.

	Dalvik bytecode	LLVM IR	Machine code
Class information	Yes	No	No
SSA form	Yes (conditional)	Yes	No
Register type	Dynamic	Static	Static

To understand the characteristics of the Dalvik bytecode and LLVM IR, we compare the Dalvik bytecode, LLVM IR, and the machine code in Table 1. An Android application written in Java language is compiled to *.class* file (Java bytecode) by the Java compiler. The Dx tool transforms the *.class* file to a Dalvik executable file (*.dex*) that is composed of Dalvik bytecodes. During transformation, the Dx tool performs simple optimizations including register allocation, dead code elimination, constant propagation, and constant folding [23]. In contrast to the LLVM IR, the Dalvik bytecode uses an infinite number of virtual registers. A virtual register's type is determined by the mnemonic code of the Dalvik bytecode that uses it. Moreover, the Dalvik bytecode has high-level language features that are derived from the Java program to describe class information and class inheritance. However, similar to the IR of most compilers, LLVM IR does not have such high-level language features. LLVM includes only four derived types — pointer, array, structure, and function [19]. High-level data types are represented as a combination of these four types. These four derived types are used in complicated language-independent analyses and optimizations. The register of LLVM IRs is already set according to an instruction and is in static single assignment (SSA) form to facilitate analyses and optimization passes.

3. Translation to LLVM IR

When we regenerate the Dalvik bytecode from LLVM IR

```
1: define i32 @loopinvariant (i32 %arg0, i32 %arg1) {
2: bb:          %0 = call i32 @llvm.dalvik.newArray (i32 100, i32 4)
3: br label %bb15
4:
5: bb15:
6: %11.0 = phi i32 [0, %bb], [%2, %bb4]
7: %10.0 = phi i32 [0, %bb], [%5, %bb4]
8: %1 = icmp uge i32 %10.0, %arg0
9: br i1 %1, label %bb5, label %bb4
10:
11: bb4:
12: %2 = add i32 %arg0, %arg1← Redundant computation in loop
13: %3 = mul i32 %l0.0, 6
14: %4 = add i32 %3, %2
15: call void @llvm.dalvik.aput (i32 %4, i32 %0, i32 %10.0)
16: %5 = add i32 %10.0, 1
17: br label %bb15
18:
19: bb5:
20: %6 = add i32 %arg0, –1
21: %7 = call i32 @llvm.dalvik.aget (i32 %0, i32 %6)
22: %8 = add i32 %7, %11.0
23: ret i32 %8
```

Fig. 2. LLVM IR code of motivating example.

```
1: llvm.dvk.strings = !{!0, !1, !2, !3, !4, !5, !6, !7, !8, !9, !10, !11}
2: llvm.dvk.types = !{!1, !3, !4, !6, !8, !9}
3: llvm.dvk.class = !{!12}
4: init = !{!13}
5: loopinvariant = !{!14}
                                        ⋮
13: !0 = metadata !{metadata !"<init>"}
14: !1 = metadata !{metadata !"I"}
15: !2 = metadata !{metadata !"III"}
16: !3 = metadata !{metadata !"LLoopinvariant;"}
17: !4 = metadata !{metadata !"Ljava/lang/Object;"}
18: !5 = metadata !{metadata !"Loopinvariant.java"}
19: !6 = metadata !{metadata !"V"}
20: !7 = metadata !{metadata !"VL"}
21: !8 = metadata !{metadata !"[I"]
22: !9 = metadata !{metadata !"[Ljava/lang/String;"]
23: !10 = metadata !{metadata !"loopinvariant"}
24: !11 = metadata !{metadata !"main"}
25: !12 = metadata !{metadata !"LLoopinvariant;", metadata !"Ljava/lang/Object;", i32
1}
26: !13 = metadata !{metadata !"<init>," i32 65537, i32 3, i32 0}
27: !14 = metadata !
{metadata !"loopinvariant," i32 9, i32 0, i32 2, i32 0, i32 0}
```

Fig. 3. Metadata in LLVM IR code.

code, we should preserve the high-level language features of the original Dalvik bytecode. The beginning portion of Dalvik bytecode is composed of several constant pool sections, including string ids, type ids, method ids, class definitions, and word data. These sections represent strings; types of variables and methods; methods of classes; class information; and data in classes. To handle the impedance mismatch between Dalvik bytecode and LLVM IR code described in Section II-2, we annotate these high-level language features of the Dalvik bytecode into LLVM IR code using metadata, and create the intrinsic functions for the Dalvik instructions, which do not have any direct correspondence in LLVM IR code [24]–[26]. In the LLVM, intrinsic functions are used to add new fundamental types and new instructions to extend LLVM IR [27]. In our system, since an LLVM transformation pass does not change the metadata, an intrinsic function maintains the metadata for high-level language features, such as class information and inheritance, during optimization and code regeneration.

Our system first parses the Dalvik bytecode and then generates metadata from the constant pool sections. The metadata of a method is described as a sequence of the method name, access flag, return type, parameter count, and parameter type. If a Dalvik bytecode instruction has a corresponding LLVM IR instruction, then our system generates the LLVM IR instruction as shown in Fig. 2 for the Dalvik bytecode instruction shown in Fig. 1(c). Otherwise, it generates an intrinsic function [27] — the name of which is the same as the Dalvik bytecode instruction. For example, since "aput" instruction at Line 10 in Fig. 1(c), which stores a register value to a given array element with given array index, has no corresponding LLVM IR instruction, an intrinsic function "void @llvm.Dalvik.aput" underlined at Line 15 in Fig. 2 is generated. In LLVM optimization passes, an intrinsic function is treated as an unanalyzable function. In this way, 43 Dalvik bytecode instructions among 256 bytecode instructions are translated to intrinsic functions: for example, "aput, aget, sput, sget," and so on.

Figure 3 shows the metadata translated from the Dalvik bytecode shown in Fig. 1(c) through the front-end compiler of our system. Each metadata in a dotted box includes variables and methods in a class as follows:

- Types of string, variables, and methods in the Dalvik bytecode (from "metadata !0" at Line 13 to "metadata !11" at Line 24).
- Class information ("metadata !12" at Line 25).
- Methods information (from "metadata !13" at Line 26 and "metadata !14" at Line 27).

For example, in "metadata !14" at Line 27 of Fig. 3, the first argument, "loopinvariant," describes the method name. The second argument is an access flag of the "loopinvariant" method, where "i32 9" represents public static method whose type is determined in the DVM library. The third argument is the return type of the method, where "i32 0" indicates the first argument (!1) of "llvm.dvk.types" at Line 2, and the metadata "I" at Line 14 (!1) refers to an integer type. The fourth argument is the number of parameters, and the fifth and sixth arguments are the presented parameter types. Since the fourth, fifth, and sixth arguments are "i32 2," "i32 0," and "i32 0,"

respectively, the number of parameters is two, and the type of the two parameters is integer, as described above.

4. Optimization and Generation of Dalvik Bytecode

An LLVM static optimizer [28] optimizes the LLVM IR code generated by our front-end compiler, in both a target-dependent way and a target-independent way.

Figure 4 illustrates how our static Dalvik bytecode optimization improves the code quality of Android applications, where the gray shaded boxes indicate our optimization components. The "Dalvik bytecode instruction selection" phase optimizes LLVM IR code using Dalvik bytecode descriptions to select instructions and operands target-dependently [29]–[30]. Dalvik bytecode descriptions consist of language characteristics for Dalvik bytecodes, including instruction and register information. In the "Dalvik bytecode instruction selection" phase, LLVM IR code is translated into an initial directed acyclic graph (DAG), in which the nodes specify the operations and operands of each instruction. Using an initial DAG helps LLVM to optimize the Dalvik bytecodes target-independently on a very low level. The "Optimize DAG" phase simplifies the initial DAG by eliminating redundancies exposed by the previous step, before and after the "Legalize DAG" phase. The "Legalize DAG" phase transforms the DAG to eliminate the types and operations that are not supported by the Dalvik VM. The "Instruction selection" phase for the DAG linearizes DAG into Dalvik bytecode instructions by using pattern matching. Then, the Dalvik bytecode is optimized using SSA-based optimization, including loop-invariant code motion, dead code elimination, common subexpression elimination, and peephole optimization. In the "Register allocation" stage, the number of

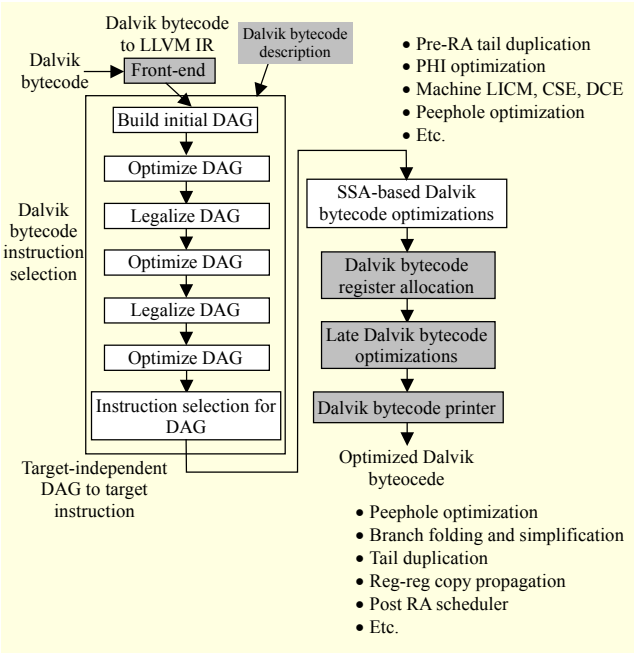

Fig. 4. Static Dalvik bytecode optimization.

```
1: define i32 @loopinvariant (i32 %arg0, i32 %arg1) {
2: bb:
3: %0 = call i32 @llvm.dalvik.newArray (i32 100, i32 4)
4: %1 = add i32 %arg0, %arg1 ← Hoisting redundant computation out of the loop
5: br label %bb15
6:
7: bb15:
8: %10.0 = phi i32 [0, %bb], [%5, %bb4]
9: %2 = icmp uge i32 %10.0, %arg0
10: br i1 %2, label %bb5, label %bb4
11:
12: bb4:
13: %3 = mul i32 %10.0, 6
14: %4 = add i32 %3, %1
15: call void @llvm.dalvik.aput (i32 %4, i32 %0, i32 %10.0)
16: %5 = add i32 %10.0, 1
17: br label %bb15
18:
19: bb5:
20: %6 = add i32 %arg0, –1
21: %7 = call i32 @llvm.dalvik.aget (i32 %0, i32 %6)
22: %8 = add i32 %7, %1
23: ret i32 %8
24: }
```

Fig. 5. Optimized LLVM IR code of motivating example.

```
1: .method public static loopinvariant (II)I
2:         .registers 6
3:         .parameter
4:         .parameter
5:
6:         .prologue
7: const/4 v0, 0x0
8: const/16 v1, 0x64
9: new-array v2, v1, [I
10: move v1, v0
11: add-int v1, p0, p1 ← Hoisting redundant computation out of the loop
12:
13: :goto_6
14: if-ge v0, p0, :cond_12
15: mul-int/lit8 v3, v0, 0x6
16: add-int/2addr v3, v1
17: aput v3, v2, v0
18: add-int/lit8 v0, v0, 0x1
19: goto :goto_6
20:
21: :cond_12
22: add-int/lit8 v0, p0, –0x1
23: aget v0, v2, v0
24: add-int/2addr v0, v1
25: return v0
26: .end method
27: .method public static
```

Fig. 6. Optimized Dalvik bytecode of motivating example.

virtual registers in SSA form is decreased to 15, which is the number of general-purpose registers in our target architecture, ARMv7. Subsequently, Dalvik bytecodes are optimized by final Dalvik bytecode-dependent optimization including peephole optimization. Finally, the optimized Dalvik bytecodes are printed out using the LLVM MC infrastructure.

Figure 5 shows the optimized LLVM IR code generated by our static optimization. In Fig. 5, the redundant computation at Line 4, "%1 = add i32 %arg0, %arg1," shown in the dark gray box was removed from the loop in the dotted box. Finally, the back-end of our system generates the optimized Dalvik bytecode, where the summation instruction is removed from the loop (as shown in Fig. 6).

As mentioned above, our approach finally generates optimized Dalvik bytecode before it is executed in DVM. After starting the execution of the Dalvik bytecode, a typical DVM JIT compiler handles the reference types and garbage collection. Because of our aggressive optimization, our approach can give positive effects on dealing with these operations (reference types and garbage collection) during executing the Dalvik bytecode.

III. Evaluation

To evaluate our static Dalvik bytecode optimization system, we first ran four micro-benchmarks and two real benchmarks, which were downloaded from the Google application market (Benchmark Pi [31], EmbeddedCaffeineMark 3.0 [32]). We compared the scores of the benchmarks in four configurations — baseline, JIT, baseline with static optimization, and JIT with static optimization. In the baseline configuration, DVM performs only interpretation without either JIT or our static optimization. We use the baseline configuration as the baseline for measuring normalized performance improvement of other schemes. In the JIT configuration, DVM performs both the interpretation and JIT compilation. In the configurations with static optimization, the Dalvik bytecodes are statically optimized before running on DVM. In addition, we compare the performance of our static optimization scheme with other studies [10], [15]. All experiments are performed on a reference phone, Galaxy Nexus [33], running Android 4.0.1, with 1.2 GHz TI OMAP 4660 ARM Cortex-A9 dual core and 1 GB memory. In addition, to evaluate our system on another Android device, we performed EmbeddedCaffeineMark 3.0 on Galaxy S4, running Android 4.4.2, with 2.2 GHz Qualcomm Snapdragon 800 Krait 400 quad core and 2GB memory.

1. Performance of Static Dalvik Bytecode Optimization

We developed four micro benchmarks — loop-invariant code motion (Fig. 1(a)), loop splitting, loop unswitching, and partial redundancy elimination. Each micro benchmark is a program that can be optimized by the above-mentioned optimization techniques. Figure 7 shows that our static optimization scheme significantly improves performance, where the performance measure is execution speed for each micro benchmark. As we expected, the best performance is achieved by using both the JIT compilation and static optimization. The average performance improvement in the micro benchmarks is 64%.

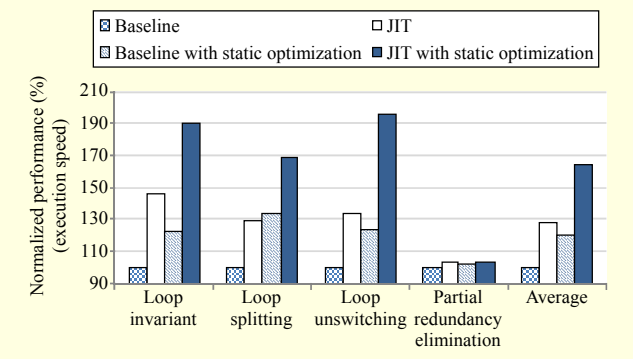

Fig. 7. Normalized performance improvement of micro benchmarks.

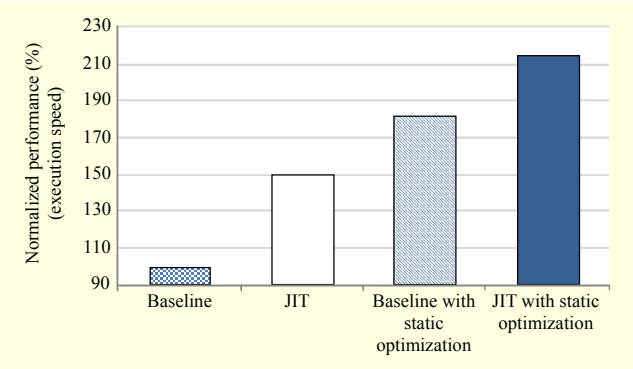

Fig. 8. Normalized performance improvement of Benchmark Pi.

For real benchmarks, we evaluated EmbeddedCaffeinemark 3.0 [32] and Benchmark Pi [31]. These are popularly used benchmarks for testing the performance of DVM JIT [8], [10]. First, Benchmark Pi calculates the ratio of the circumference of a circle to its diameter. Figure 8 shows the normalized performance improvement of Benchmark Pi under the four configurations, where the performance measure is execution speed. The baseline with static optimization outperforms the baseline by 1.81 times and the JIT with static optimization outperforms the baseline by 2.13 times. The experimental results clearly show that our system significantly improves the performance of Android applications by statically optimizing the Dalvik bytecodes.

Second, EmbeddedCaffeineMark 3.0 is used for our evaluation.

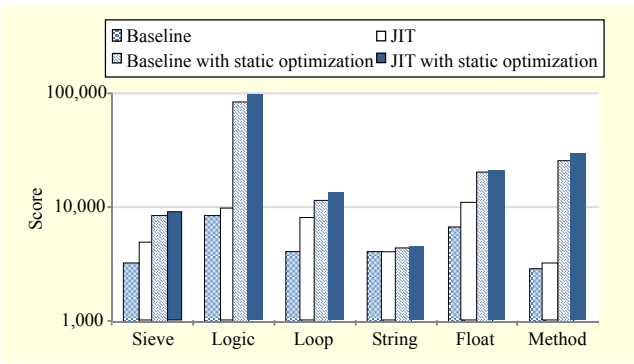

Fig. 9. EmbeddedCaffeineMark 3.0 scores on Galaxy Nexus.

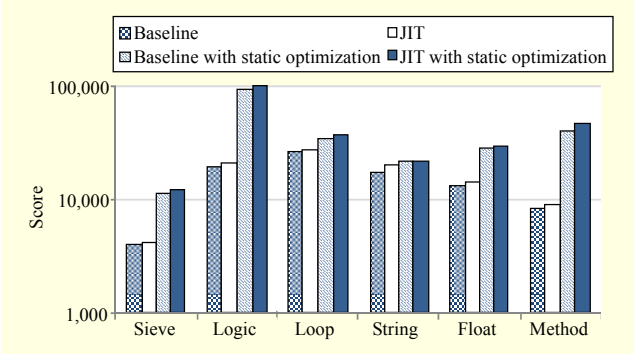

Fig. 10. EmbeddedCaffeineMark 3.0 scores on Galaxy S4.

Table 2. Average execution time of each benchmark.

	Baseline (s)	JIT (s)	Baseline with static optimization (s)	JIT with static optimization (s)
Loop invariant	7.65	5.24	6.28	4.02
Loop splitting	10.98	8.54	8.27	6.53
Loop unswitching	9.87	7.45	8.04	5.06
Partial redundancy elimination	6.16	6.01	6.07	6.01
Benchmark Pi	0.66	0.41	0.34	0.29
EmbeddedCaffeineMark 3.0	23.80			

It is a subset of the complete CaffeineMark suite, including Sieve, Loop, Logic, Method, Float, and String [32]. First, Sieve is the classic Sieve of Eratosthenes (for prime numbers). Loop performs sorting and sequence generation for measuring the compiler optimization of loops. Logic measures the execution speed of decision-making instructions in a virtual machine. Method performs recursive function calls on a virtual machine. Float simulates a 3D rotation of objects around a point. Finally, String performs string concatenation, which is the operation of joining character strings end-to-end. The score is the number of executed instructions per second. Figure 9 shows the scores of EmbeddedCaffeineMark 3.0 under the four configurations in a logarithmic scale. As expected, the benchmark scores are the best when we use both the JIT compilation and static optimization; normalized performance acceleration relative to the baseline is increased by up to 6.08 times. Also, interpreting with static optimization achieves performance improvement by up to 5.22 times on average. Therefore, when an application is statically optimized, the JIT-compiled performance is 4.34 times faster on average than that of no static optimization. Especially, in the case of the Logic benchmark in Fig. 9, we can observe that our static optimization system, simultaneously with DVM JIT, improved the performance by about 10 times because of their efforts for more advanced optimization.

Table 2 shows the average execution time of each benchmark, while each benchmark was run 10 times. EmbeddedCaffeineMark 3.0 cannot be measured individually, so the whole execution time of the benchmark is presented. In our experiment, each test is run for approximately the same length of time.

To ensure that our experimental results are impartial to diverse hardware features, performed, another experiment with EmbeddedCaffineMark 3.0 on Galaxy S4, and showed the results in Fig. 10. In Fig. 10, as expected, similar results to those of Fig. 9 were achieved, and we can see that the benchmark scores are also the best when we use both the JIT compilation and static optimization; normalized performance acceleration relative to the baseline is increased by up to 3.17 times on average. Moreover, interpreting with static optimization achieves performance improvement by up to 2.87 times on average. Therefore, when an application is statically optimized, the JIT-compiled performance is 2.94 times faster on average than that of no static optimization. This experimental result clearly shows that our system can improve the performance of Android applications regardless of the type of hardware.

2. Comparison with Other Optimization Schemes

We compared the performance of our static Dalvik bytecode optimization scheme with the Java bytecode level optimizer [17] and another complementary compilation scheme for Android applications [10].

Soot [17] is a Java optimization framework developed at McGill University. It can be used to optimize and analyze class files at a Java bytecode level [34]. For fair comparison, we statically optimized Java bytecodes of benchmarks, such as Benchmark Pi and EmbeddedCaffeineMark 3.0, by using the latest version of Soot with highest optimization options, enabling intra-procedural and whole program optimization.

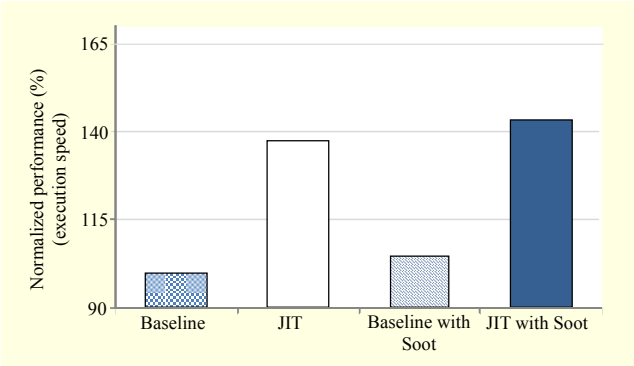

Fig. 11. Normalized performance improvement of Benchmark Pi for Soot.

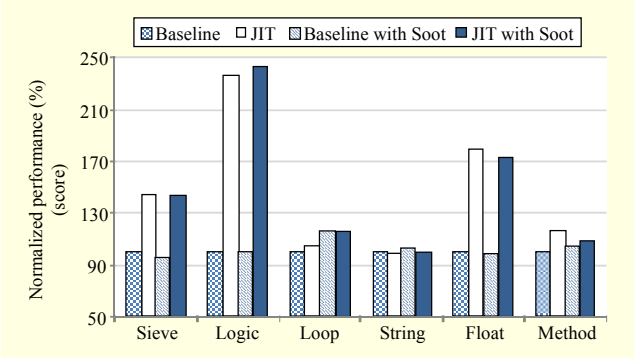

Fig. 12. Normalized performance improvement of EmbeddedCaffeine Mark 3.0 for Soot.

The optimized Java bytecodes are subsequently transformed to Dalvik bytecodes and run on the same machine as that described in Section III-1. We measured the score for the benchmarks and used DVM interpretation as the baseline, measuring the normalized performance improvement.

Figure 11 shows that normalized performance of Benchmark Pi for Soot is improved, where we used execution speed as a performance measure. We observe that the JIT-compiled performance is 44% faster than the baseline performance. Figure 12 also shows that normalized performance of EmbeddedCaffeineMark 3.0 for Soot is improved and we can observe that the JIT-compiled performance is 48% faster than the baseline performance on average. As shown in Figs. 11 and 12, Soot does not outperform DVM dramatically, because it performs only simple optimizations; that is, copy propagation, constant propagation, constant folding, dead assignment elimination, unconditional branch folding, unused local elimination, load store optimization, and peephole optimization.

Icing [10] is a well-known complementary compilation technique for DVM. Unfortunately, we could not directly compare our results with those of Icing, because the Icing project is not accessible to the general public. Instead, we indirectly measured the performance of our optimization scheme against that of Icing by referring to the experimental results in [10]. We observe that our optimization scheme outperforms Icing for the same benchmark applications; whereas, Icing outperforms Dalvik JIT by 2.83 times, our system outperforms Dalvik JIT by 4.34 times.

IV. Related Work

Improving the performance of Android applications has received a lot of interest. Here, we discuss some studies that are most related to our work.

1. Complementary Compilation with DVM

Icing [10] converts hot methods in the Dalvik bytecode to C codes using the GCC compiler. The translated native codes are executed by DVM through calling the Java Native Interface. Lim and others [11] generate a hybrid DEX file that includes selective ahead-of-time compilation information based on the profiling information of hot methods, and then compile and execute the hybrid DEX file. BLOAT [16] eliminates common access expressions by performing partial redundancy elimination based on type-based alias analysis. It employs optimization between java class files at a bytecode-to-bytecode level. However, since BLOAT mainly focuses on exploiting intra-procedural analysis and optimization, the performance gain may be limited. As shown in Section III, the compilation with our static optimization and DVM JIT surpassed that of previous studies.

2. Modification in DVM

In Swift [4], the authors propose a lightweight JIT compiler for the ARM architecture, which is among the most popular on the Android platform. They use a simplified register allocation process for JIT based on the fact that in most Java methods eleven physical registers in the ARM core are sufficient to fulfill the virtual registers in DVM. Absar and Shekhar [12] improve the array bounds check optimization algorithm in DVM JIT so that it can handle indices such as affine functions of iterators, loop invariants, and literals. Modifying DVM causes maintenance problems since DVM is under active development. AccelDroid [13] is the HW/SW co-designed system for accelerating Android applications. To boost the power and performance efficiency of Android applications, a Dalvik bytecode–aware processor is implemented by co-designing hardware and software; this is so that Dalvik bytecodes can be executed directly using dynamic binary translation on the special processor. ART runtime [9] performs ahead-of-time compilation from Dalvik bytecode to binary

during an application installation, and then the generated machine code is executed when the application is run. However, since ART runtime is still an ongoing project, the speed-up of performance is not dramatic — under 1.8 times on EmbeddedCaffeinemark 3.0 [9]. In conclusion, our static optimization can achieve higher performance of applications with less engineering overhead than those of DVM modification techniques.

V. Conclusion

In this paper, we proposed a static Dalvik bytecode optimization system for the performance improvement of Android applications. We found that Android applications, which are compiled by the Java compiler and transformed by the Dx tool, still offer considerable opportunities for further optimization with static compilation. We resolve the performance shortage by adopting static optimization with DVM JIT as a complementary compilation technique of DVM. We exploited a mature compiler infrastructure, LLVM, to enhance the code quality for Android applications. Moreover, we proposed techniques to close the gap between the high-level Dalvik bytecode and the low-level LLVM IR code, and to optimize the LLVM IR code conforming to language information of Dalvik bytecode. Our experimental results show that the static Dalvik optimization system with DVM JIT surpasses interpretation in DVM by 6.08 times and DVM JIT by 4.34 times.

Our optimization framework can also be applied for Android application executing in ART runtime. As mentioned in Section IV, ART runtime compiles Dalvik bytecode to native code using an on-device code translation tool in advance, and then directly executes the precompiled native code to boost the execution performance at runtime. However, this scheme inevitably requires considerable installation time and more memory space than DVM. We expect that the installation time and the amount of memory usage can be reduced by compiling the optimized Dalvik bytecode, which is generated by our framework, at code translation process in ART runtime. In addition, we believe that our aggressive optimization scheme can also give positive effects on code optimization in ART runtime, which remains as a future work.

References

[1] M.J. Cho et al., "AndroScope: An Insightful Performance Analyzer for All Software Layers of the Android-Based Systems," *ETRI J.*, vol. 35, no. 2, Apr. 2013, pp. 259–269.

[2] D. Bornstein, "Dalvik Virtual Machine Internals," presented at the Google I/O Developer Conf., San Francisco, CA, USA, 2008.

[3] Y. Shi et al., "Virtual Machine Showdown: Stack versus Registers," *ACM Trans. Archit. Code Optimization*, vol. 4, no. 4, Jan. 2008, pp. 1–36.

[4] Y. Zhang et al., "Swift: A Register-Based JIT Compiler for Embedded JVMs," *Proc. ACM SIGPLAN/SIGOPS Conf. Virtual Execution Environment*, London, UK, Mar. 3–4, 2012, pp. 63–74.

[5] R. Hutcherson, *Compiler Optimizations: Can You Count on Compilers to Optimize Your Java Code*, Java World, 2000. Accessed Jan. 20, 2014. http://www.javaworld.com/javaworld/jw-03-2000/jw-03-javaperf_4.html?page=1

[6] P. Haggar, *Java Bytecode: Understanding Bytecode Makes You a Better Programmer*, IBM Developer Works, 2001. Accessed Jan. 20, 2014. http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode/

[7] K, Venugopal, G. Manjunath, and V. Krishnan, "sEc: A Portable Interpreter Optimizing Technique for Embedded Java Virtual Machine," *Java Virtual Mach. Res. Technol. Symp.*, San Francisco, CA, USA, Aug. 1–2, 2002, pp. 127–138.

[8] B. Cheng and B. Buzbee, "A JIT Compiler for Android's Dalvik VM," presented at the Google I/O Developer Conf., San Francisco, CA, USA, 2010.

[9] B. Carlstrom, A. Ghuloum, and I. Rogers, "The ART Runtime," presented at the Google I/O Developer Conf., San Francisco, CA, USA, 2014.

[10] C.-S. Wang et al., "A Method-Based Ahead-of-Time Compiler for Android Applications," *Proc. Int. Conf. Compiler, Archit. Synthesis Embedded Syst.*, Taipei, Taiwan, Oct. 9–14, 2011, pp. 15–24.

[11] Y.-K. Lim et al., "A Selective Ahead-of-Time Compiler on Android Device," *Int. Conf. Inf. Sci. Appl.*, Suwon, Rep. of Korea, May 23–25, 2012, pp. 1–6.

[12] J. Absar and D. Shekhar, "Eliminating Partially-Redundant Array-Bounds Check in the Android Dalvik JIT Compiler," *Proc. Int. Conf. Principles Practice Programming Java*, Kongens Lyngby, Denmark, Aug. 24–26, 2011, pp. 121–128.

[13] C. Wang, Y. Wu, and M. Cintra, "Acceldroid: Co-designed Acceleration of Android Bytecode," *IEEE/ACM Int. Symp. Code Generation Optimization*, Shenzhen, China, Feb. 23–27, 2013, pp. 1–10.

[14] *GCJ - The GNU Compiler for the Java Programming Language*. Accessed Jan. 20, 2014. http://gcc.gnu.org/java/

[15] *DragonEgg - Using LLVM as a GCC Backend*. Accessed Jan. 20, 2014. http://dragonegg.llvm.org/

[16] A.L. Hosking et al., "Partial Redundancy Elimination for Access Path Expressions," *Software: Practice and Experience*, vol. 31, no. 6, May 2001, pp. 577–600.

[17] R. Vallee-Rai et al., "Soot: A Java Bytecode Optimization Framework," *Conf. Center Adv. Studies Collaborative Res.*, Toronto, Canada, 2010, pp. 214–224.

[18] N. Geoffray et al., "VMKit: A Substrate for Managed Runtime

Environments," *Proc. ACM SIGPLAN/SIGOPS Conf. Virtual Execution Environment*, Pittsburgh, PA, USA, Mar. 17–19, 2010, pp. 51–62.
[19] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," *Int. Symp. Code Generation Optimization*, Palo Alto, CA, USA, Mar. 20–24, 2004, pp. 75–86.
[20] *The LLVM Compiler Infrastructure*. Accessed Jan. 20, 2014. http://llvm.org/
[21] *GCC- The GNU Compiler Collection*. Accessed Jan. 20, 2014. http://gcc.gnu.org/
[22] *Google Android Dx Tool*. Accessed Jan. 20, 2014. http://wing-linux.sourceforge.net/guide/developing/tools/othertools.html
[23] Security Engineering Research Group, "Analysis of Dalvik Virtual Machine and Class Path Library," Institute of Management Sciences, Peshawar, Pakistan, Tech. Rep., Nov. 2009.
[24] *LLVM Language Reference Manual*. Accessed Jan. 20, 2014. http://llvm.org/docs/LangRef.html
[25] *Extensible Metadata in LLVM IR*. Accessed Jan. 20, 2014. http://blog.llvm.org/2010/04/extensible-metadata-in-llvm-ir.html
[26] J. Holewinski, "PTX Back-End: GPU Programming with LLVM," presented at the LLVM Developer's Meeting, San Jose, CA, USA, Nov. 8, 2011.
[27] *Extending LLVM: Adding Instructions*, *Intrinsics*, *Types*, *etc*. Accessed Jan. 20, 2014. http://llvm.org/docs/ExtendingLLVM.html
[28] *LLVM's Analysis and Transform Passes*. Accessed Jan. 20, 2014. http://llvm.org/docs/Passes.html
[29] *The LLVM Target-Independent Code Generator*. Accessed Jan. 20, 2014. http://www.llvm.org/docs/CodeGenerator.html
[30] A. Korobeynikov, "Tutorial: Building a Backend in 24 Hours," presented at the LLVM Developer's Meeting, Cupertino, CA, USA, 2009.
[31] *Benchmark Pi – The Android Benchmarking Tool*. Accessed Jan. 20, 2014. http://androidbenchmark.com/
[32] *The Embedded CaffeineMark*, Pendragon Software Corporation. Accessed Jan. 20, 2014. http://www.benchmarkhq.ru/cm30/info.html
[33] *Galaxy Nexus*, Samsung Electronics. Accessed Jan. 20, 2014. http://www.samsung.com/sec/consumer/mobile-phone/mobile-phone/skt/SHW-M420STSSC.

Jeehong Kim received his BS and MS degrees in electronic engineering from the Department of Electronic Engineering, Kwangwoon University, Seoul, Rep. of Korea, in 2008 and 2010, respectively, and his PhD degree in computer engineering from the Department of Mobile Systems Engineering, Sungkyunkwan University, Suwon, Rep. of Korea, in 2015. Since 2015, he has been a senior software engineer at Samsung Electronics, Suwon, Rep. of Korea. His research interests include embedded systems and system securities.

Inhyeok Kim received his BS and MS degrees in computer engineering from the Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon, Rep. of Korea, in 2006 and 2010, respectively, and since 2010, he has been a PhD student with the Department of Electrical and Computer Engineering, Sungkyunkwan University. His research interests include system software and UI/UX platforms.

Changwoo Min received his BS and MS degrees in computer science from the Department of Computer Science, Soongsil University, Seoul, Rep. of Korea, in 1996 and 1998, respectively, and his PhD degree in computer engineering from the Department of Mobile Systems Engineering, Sungkyunkwan University, Suwon, Rep. of Korea, in 2014. From 1998 to 2005, he was a research engineer with the Ubiquitous Computing Laboratory of IBM, Seoul, Rep. of Korea. From 2005 to 2014, he was a principal software engineer at Samsung Electronics, Suwon, Rep. of Korea. Since 2014, he has been working as a postdoctoral researcher at Georgia Institute of Technology, Atlanta, USA. His research interests include embedded systems, storage systems, and operating systems.

Hyung Kook Jun received his BS degree in computer engineering from the Department of Computer Science and Engineering, Sungkyunkwan University, Suwon, Rep. of Korea, in 1999 and his MS degree in computer engineering from the Department of Electrical and Computer Engineering, Sungkyunkwan University, in 2001. Since 2001, he has been a senior researcher with the Cyber-Physical Systems Research Team, ETRI. His research interests include CPS, embedded systems, communication middleware, and multimedia systems.

Soo Hyung Lee received his BS and MS degrees in electronic engineering from the Department of Electronic Engineering, Hanyang University, Seoul, Rep. of Korea, in 1991 and 1993, respectively, and his PhD degree in computer engineering from the Department of Computer Engineering, Chungnam National University, Daejeon, Rep. of Korea, in 2012. In August 1993, he joined the Network Design Laboratory of DACOM corporation, Seoul, Rep. of Korea. Since October 2000, he has been a principal member of the engineering staff of the Cyber-Physical Systems Research Team, ETRI. His research interests include IT converging systems, CPS, Smart Factory, and network security.

Won-Tae Kim received his BS, MS, and PhD degrees from the Department of Electronic Engineering, Hanyang University, Seoul, Rep. of Korea in 1994, 1996, and 2000, respectively. He established a venture company named Rostic Technologies, Inc. in 2001 and worked as CTO from 2001 to 2005. He joined ETRI in 2005 and has led in Cyber-Physical Systems (CPS) research team, SW Contents Technology Research Lab, ETRI, since 2010. Now, he is a professor at Korea University of Technology and Education, Cheonan, Rep. of Korea. His research interests include CPS, IoT networking, and Neuromorphic computing.

Young Ik Eom received his BS, MS, and PhD degrees in computer science from the Department of Computer Science, Seoul National University, Rep. of Korea, in 1983, 1985, and 1991, respectively. He was a visiting scholar with the Department of Information and Computer Science, University of California, Irvine, USA, from September 2000 to August 2001. Since 1993, he has been a professor at Sungkyunkwan University, Suwon, Rep. of Korea. His research interests include system software, operating systems, virtualization, cloud systems, and system securities.