Bytecode-to-C Ahead-of-Time Compilation for Android Dalvik Virtual Machine*

Hyeong-Seok Oh Ji Hwan Yeo Soo-Mook Moon
Department of Electrical and Computer Engineering
Seoul National University
Seoul, Korea
{oracle, jhyeo, smoon}@altair.snu.ac.kr

Abstract—**Android employs Java for programming its apps which is executed by its own virtual machine called the Dalvik VM (DVM). One problem of the DVM is its performance. Its just-in-time compiler (JITC) cannot generate high-performance code due to its trace-based compilation with short traces and modest optimizations, compared to JVM's method-based compilation with ample optimziations. This paper proposes a bytecode-to-C ahead-of-time compilation (AOTC) for the DVM to accelerate pre-installed apps. We translated the bytecode of some of the hot methods used by these apps to C code, which is then compiled together with the DVM source code. AOTC-generated code works with the existing Android zygote mechanism, with corrects garbage collection and exception handling. Due to off-line, method-based compilation using existing compiler with full optimizations and Java-specific optimizations, AOTC can generate quality code while obviating runtime compilation overhead. For benchmarks, AOTC can improve the performance by 10% to 500%. When we compare this result with the recently-introduced ART, which also performs ahead-of-time compilation, our AOTC performs better**.

I. Introduction

Android is a platform including OS, framework middleware, apps for mobile device such as smart phones or tablets. Android is based on the Linux kernel, supplemented with middleware and libraries written in C/C++, yet the Android app itself is written in Java and run with the Android framework and Java-compatible libraries. So, Android can enjoy the full benefit of Java such as platform independence, security, and rich APIs. Android employs its own virtual machine to execute Java applications, called the Dalvik virtual machine (DVM) [1]. DVM has its own register-based bytecode and calling convention.

For higher performance, DVM performs bytecode optimization during app installation time called *dexopt*, which performs static linking, called *quickening* [1,2]. More importantly, DVM employs *just-in-time compiler* (JITC), which compiles the bytecode to machine code at runtime so as to execute the machine code instead of interpreting the bytecode [3]. The unit of compilation is trace, which is a hot path in a method such that if some path is known to be hot during interpretation, it is compiled to machine code. This is for reducing the memory overhead of compiling a whole method. Unfortunately, DVM's trace-based JITC cannot generate high-performance code because the length of the trace is too short, with modest optimizations performed [3].

Recently, Android introduced the ART (Android RunTime) in its 4.4 version (KitKat), which can replace the exsiting DVM [4]. ART can compile the framework classes and pre-installed apps in advance during the first booting of the Android device. Also, ART can compile downloaded apps during their installation although it will take longer. In this way, ART can remove the runtime compilation overhead of the JITC. Unfortunately, ART generates code with weak method-based optimizations, so the code quality still has a problem.

In this paper, we propose a different approach to compilation-in-advance to complement DVM JITC or ART. For pre-installed apps, we propose bytecode-to-C ahead-of-time compilation (AOTC). We translated the bytecode of some of the hot methods in these apps and framework to C code, which is then compiled together with the DVM source code. Due to off-line, method-based compilation using existing compiler with full optimizations and Java-specific optimizations, AOTC can generate quality code while obviating runtime compilation overhead and installation-time overhead.

We could successfully design and implement the proposed AOTC in the Android version 4.4 (KitKat), and make it work for benchmarks. We show the performance of AOTC compared to DVM JITC and ART. Our results indicate that the proposed AOTC is competitive.

II. Android and Dalvik VM

A. Android Execution Model

Android employs zygote for efficient launching of an app process [1]. The zygote process starts when the Android device boots. The zygote process creates the DVM and initializes it. Especially, some of the important framework classes are loaded to the DVM and initialized by the initialization methods. Initialization may include the object creation for the static variables.

When an app starts, a process forked from the zygote process is created where a new DVM is available with loaded and initialized classes. These loaded classes are not supposed to be modified during execution, so the app process run would be efficient. Also, app processes forked from the zygote will share the heap of the zygote process which has the loaded class objects and initializied objects. Each app will run on this process forked from zygote.

B. Dalvik VM (DVM)

The DVM is a register-based machine where computations are performed using virtual registers included in the DVM. So, the DVM has its own bytecode instruction set architecture

* This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2011-0026481)

978-3-9815370-4-8/DATE15/©2015 EDAA

different from that of the JVM [5]. The Dalvik bytecode is obtained not by compiling the Java source code but by translating the JVM bytecode. That is, the Dalvik's executable called the *dex* file is generated by translating the JVM class file using a tool called the *dx*. During the translation, the JVM bytecode instructions are compiled to the Dalvik bytecode instructions with optimizations. Figure 1 (a) shows an example Java source code and Figure 1 (b) shows the corresponding DVM bytecode.

(a) Java code	(b) Bytecode

class Math {
  public int add(int a, int b) {
    int c = a+b;
    System.out.println(c)
    return c;
  }
}
	// argument: v2(this object), v3, v4
	// virtual register: v0 ~ v4
	0 add-int v0, v3, v4
	2 sget-object v1, field#0
	4 invoke-virtual {v1, v0}, method#3
	=> 4 invoke-virtual-quick {v1, v0}, #29
	(rewritten)
	7 return v0

Figure 1. Java program code and Dalvik bytecode

The bytecode is executed by the DVM interpreter. Each Java thread is assigned an interpreter stack where a stack frame is allocated for each method invoked. The stack frame includes the status information for the method and the virtual register slots. These virtual registers depicted by v0~v4 in Figure 1 (b), for example, are used for computation. The status information is used for method invocation/return, garbage collection, and exception handling.

When a method is called, a stack frame for the callee is pushed on the interpreter stack. The bytecode PC of the caller is saved on the caller's frame. Argument passing is made by copying the virtual registers of the caller frame specified in the function call to the virtual registers of the callee frame (by convention, the last virtual registers in the callee frame are used as arguments). When the callee method returns, the return value in the virtual register of the callee frame is copied to the virtual register of the caller frame. The same call interface is used when the JITC is employed, so JITC-generated code does the same job.

C. Dexopt and JITC in the Dalvik VM

For performance acceleration, *dexopt* and JITC are used. When an app is installed or when pre-installed framework is first used, *dexopt* is applied to the dex file to generate an optimized dex file. The most important optimization in *dexopt* is quickening based on static linking. Those bytecodes that access a field of an object or make a call using a virtual method table include an index in the constant pool (CP) where the field name or the method name are saved as a string. Based on the string, the offset in the object or the offset in the virtual method table should be obtained to execute those bytecodes, which is called CP resolution. The idea of *dexopt* is performing the CP resolution in advance to replace the CP index number in the bytecode by the offset, so that the CP resolution can be omitted during execution. Figure 1 (b) illustrates quickening for the virtual method call, which replaces the CP index *method#3* by an offset in the virtual method table *#29*.

Trace-based JITC is used to translate the bytecode generated by *dexopt* to machine code at runtime. Initially, interpreter is used to execute the bytecode, but when a hot trace is found, the trace is compiled to machine code. The length of the trace is too short to generate efficient code, though. For example, virtual registers cannot be mapped to physical registers, so the access to a virtual register is handled by a load or a store to the virtual register slots in the stack, which would be inefficient. Because interpreted code and JITC-generated code would co-exist in a method, even if virtual registers were mapped to physical registers within a trace, they would be spilled to the virtual registers in the stack before leaving a trace for correct interpretation in out-of-trace. This motivates our ahead-of-time compilation (AOTC) for Android.

III. Design and Implementation of AOTC

A. AOTC Architecture

We propose ahead-of-time compilation for Android to remove the runtime compilation overhead and to generate higher performance code than JITC using method-based compilation. We take the approach of bytecode-to-C (b-to-C) for simpler AOTC [6-8]. Figure 2 shows the architecture of our AOTC.

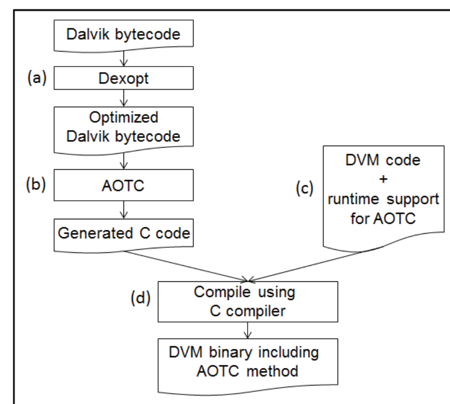

Figure 2. Build Dalvik VM with AOTC

We perform all the compilations and optimizations in Figure 2 at a server, not at the client device. For the AOTC target methods, we generate optimized bytecode using *dexopt* in Figure 2 (a). The bytecode is translated to C code in Figure 2 (b). The DVM source code is updated to use the AOTC methods in Figure 2 (c). Finally, the translated C code is compiled together with the DVM source code to build the new DVM executable in Figure 2 (d). The executable is installed in the Android client device.

Those Android framework classes that include AOTC target methods are loaded in the zygote process. In the zygote process, the method type of these AOTC methods is marked as AOTC methods and their native code is linked properly, so when a new process is forked from the zygote process for an app execution, no runtime linking overhead is needed for the new process.

On the other hand, the app classes that include AOTC target methods are not loaded in the zygote process. So the linking and marking of method type is done when the corresponding class is loaded by the forked process when the app is executed.

One advantage of our AOTC approach compared to JITC is that we can reduce the memory overhead due to sharing of the native code of hot AOTC methods in the zygote process. That is, the native code for an AOTC method of the framework class exists in the memory of the zygote process, shared by all the forked processes without any duplication. On the other hand, the DVM JITC is not invoked for the zygote process, so there

will be no native code in the zygote heap. If the forked processes uses JITC to compile the traces for the same, hot methods of a framework class, their native code would be duplicated in the heap of each forked process.

Our AOTC compiles chosen Java methods in the framework and app classes, and when an AOTC method is invoked during the app execution, the method type is checked in the method table and the corresponding linked native code will be executed.

B. Dexopt and Code Generation

We perform AOTC after performing *dexopt* so the quickened bytecode is translated to C code. In this way, we do not have to generate C code that accesses the CP and perform CP resolution, for the bytecode which can be quickened with static linking.

For C code generation, we read the bytecode, generate the intermediate representation (IR), and build the control flow graph (CFG) of the IR. We also analyze the CFG and perform Java-specific optimizations for the IR. We then traverse each IR one by one and generate the corresponding C code. Figure 3 shows the C code for the example bytecode in Figure 1 (b).

```
01 union Type32bit {int in; float fl; Object* ref;}; // 1 slot register
02 union Type64bit {long lo; double do;};            // 2 slots register
03
04 int AOTC_Math_add(Env* ee, Object* arg1, int arg2, int arg3) {
05   /*----- prologue ------*/
06   Type32bit v0_reg, v1_reg, v2_reg, v3_reg, v4_reg;
07   v2_reg.ref = arg1;                              // this object
08   v3_reg.in = arg2; v4_reg.in = arg3;             // 1st, 2nd argument
09   int result;                                     // return value
10   Frame* frame = getFramePointer(ee);
11   /*----- method body -----*/
12   v0_reg.in = v3_reg.in + v4_reg.in;   // 0 add-int v0, v3, v4
13   // frame->slot[1] = v2_reg.ref;    // save reference (optimized)
14   frame->GCMap = 0x00000000; // save GC-map
15   v1_reg.ref = resolveAndGetField(ee, 0); // 2 sget-object v1, field#0
16   frame->slot[0] = v1_reg.ref;       // save reference
17   // 4 invoke-virtual-quick {v1, v0}, #29
18   if(v1_reg.ref == NULL) {           // null check
19     setNullException(ee);            // throw and set exception
20     goto Lexception;
21   }
22   frame->GCMap = 0x00000001; // save GC-map (v1_reg saved)
23   Method* callee_method = v1_reg.ref->vmethod[29];
24   pushStackFrame(ee, callee_method);
25   callee_method->code(ee, v1_reg.ref, v0_reg.in);   // call println()
26   if(exceptionOccur(ee)) goto Lexception;           // exception  check
27   result = v0_reg.in;                // 7 return v0
28   goto Lreturn;
29   /*----- epilogue -----*/
30   Lreturn:
31     return result;
32   Lexception:
33     return 0;
34 }
```

Figure 3. Translaed C code for the example in Figure 1

Each virtual register used in the bytecode is translated to a C variable (e.g., *v0_reg* in Figure 3), which has a union type of int, float, and Object reference (*Type32bit*). The DVM as a typed VM can use a virtual register for holding different-type values at different parts of the bytecode program, so declaring a virtual register as a union-type C variable allows using the C variable properly depending on the type used. For bytecodes that use two virtual register slots for long and double type values, those corresponding C variables are declared as a different union-type (*Type64bit*).

One argument of the translated C function is declared as an environment variable (*Env* ee*) which depicts the execution thread. The environment variable is used to access the interpreter stack for argument passing, exception handling, and garbage collection.

The C function is composed of three parts: prologue, body, and epilogue. In prolog, C local variables corresponding to the virtual registers are declared (*v0_reg ~ v4_reg*) as well as temporary variables (*frame*, *result*). Arguments are copied to virtual register C variables (*v2_reg ~ v4_reg*). In the epilog, C code for returning is generated. In the method body, C code for bytecode is generated. CP resolution which was not handled in *dexopt* is handled by generating C code that invokes the VM functions, as in *dexopt*. We also generated C code for garbage collection and exception handling, which will be explained more in detail in Sections D and E.

C. AOTC Method Call

As in the interpreter (or JITC) mode, when a method is invoked in an AOTC method, a stack frame is first pushed on the call stack. If the callee method is also an AOTC method (we can check the type in the method table), the corresponding native code is invoked directly. If the callee method is not an AOTC method, the interpreter routine is invoked, which will execute the method by the interpreter or if some traces were already compiled (which can be checked in the trace table), the native code of the trace will be executed.

Argument passing to an AOTC method is important. When the C code for a method is translated, the argument is defined as in a regular C function such that the arguments and the environment variable (*ee*) are defined as regular C function arguments (see Figure 3: line #04). At the function prologue, the arguments are copied to virtual register variables. And, when an AOTC method calls another AOTC method, the C function call includes the the environment variable and the C virtual register variables.

When the C functions of an AOTC method are compiled by the C compiler, the native code will use four *physical registers* for argument passing and the remaining arguments will be passed using the *native stack* in the case of the ARM CPU. This AOTC-to-AOTC calls would perform best with this standard C call interface. However, there can also be interpreter-to-AOTC calls, which requires reconciling the call interface between the two since the interpreter passes arguments using the interpreter call stack.

We employed a mixed call interface. We make AOTC use physical registers for argument passing, and use the interpreter call stack instead of the native stack if there are additional arguments. That is, when we generate the C code for a method call in an AOTC method, the environment variable and the maximum three 32-bit arguments as regular C arguments, as shown in Figure 4 (a). Additional arguments are assigned to the interpreter stack of the callee. Interpreted method passes arguments using the interpreter stack assign as shown in Figure 4 (b). In the prologue of an AOTC method, the regular C arguments and the arguments in the interpreter stack are assigned to virtual register C variables as shown in Figure 4 (c). The return value is handled as in the C interface.

AOTC-to-AOTC call can work efficiently since the four arguments can be passed using physical registers, and the native

stack is not used when the C compiler generates code, which obviates maintaining both the native stack and the interpreter stack. For interpreter-to-AOTC calls, there still exists overhead for copying from the interpreter stack to the physical registers, but there is no memory-to-memory copy (from the interpreter stack to the native stack) as previously but only memory-to-register copy.

```
(a) caller (AOTC method)
Frame* thisframe = getFramePointer(ee);
…
copyArgument(thisFrame, v3_reg.in, v4_reg.in);
callee_method = v0_reg.ref->vtable[index];
if(callee_method.type = AOTC_METHOD) {
   callee_method->code(ee, v0_reg.ref, v1_reg.in, v2_reg.in);
} else { INTERPRETER_CALL_ROUTINE; }
(b) caller (interpreted method)
Assign arguments from caller frame to callee frame
callee_method = obj->method_table[index];
JUMP_TO_BRIDGE_FUNCTION(ee, callee_method);
{
// implemented by a separate assembly function
  Copy from callee frame to the registers
  Call callee_method->code
  Read return value from register and assign to caller frame
}
Copy return value from callee frame to caller frame
(c) callee (AOTC method)
int callee_method(Env* ee, Object* a0, int a1, int a2)
{
  Frame* frame = getFramePointer(ee);
  v0_reg.ref = a0; v1_reg.in = a1; v2_reg.in = a2;
  v3_reg.in = frame[0]; v4_reg.in = frame[1];
  …
  return result;
}
```

Figure 4. Method call using a mixed call interface (callee has 5 arguments)

D. Garbage Collection

DVM should reclaim garbage objects in heap automatically. To find garbage objects, garbage collector should trace all reachable heap-allocated, live objects from program variables and reclaim them. In the DVM, the garbage collector can find live objects by tracing all reachable objects from virtual registers in the interpreter stack for each execution thread. So, the root set is the virtual registers that have object reference [9].

Since our AOTC translates virtual registers into C variables, it is difficult to know where the C compiler places those C variables in the final machine code. We propose a method to support GC by generating C code that saves reference variables in a stack frame when a reference-type variable is updated. To save references in the stack frame, we allocate a reference save slot in the stack frame for each C variable which can possibly have a reference (s*lot[0]* for *v1_reg* in Figure 3: line #16).

To generate C code for GC, we need to know GC-point, which means a point in the program where GC can possibly occur. Examples of GC-point are memory allocation request, method call, field access, loop backedge, or synchronization point. The DVM can start GC only when every thread waits at one of its GC points since the GC cannot find all reachable objects otherwise. So we generate the check code at some GC points if there can be pending GC request from other threads.

GC needs a data structure describing the location of each root at the GC-point, which is called the GC-map. We generate GC-map for each GC-point. If reference save slots are less than 32, we save GC-map value directly to stack frame (*frame->gcMap* in figure 3: line #22). Otherwise, we generate a GC-map table for the method and generate C code at each GC-point that saves the address of GC-map table entry, so that the garbage collector can access the table.

E. Exception Handling

Java provides try and catch blocks for exception handling; if an exception occurs in a try block, it will be caught by an appropriate catch block depending on the exception type [10]. One issue is that the exception try block and the catch block might be located in different methods on the call stack.

When an exception occurs in interpreter, it searches methods in the interpreter stack backward to find one that has an exception-handler [11]. If exception occurs in the JITC code, it returns to interpreter and finds a handler in the same way.

Our AOTC uses a simple solution based on exception checks. When an exception occurs in an AOTC method, the method sets exception in environment variable (i.e., *setNullException(ee)* in Figure 3: line #19). If there is a catch block, we jump to the catch block and check the handler type. If there is no appropriate exception handler in the method, the method returns to the caller using by a jump (goto Lexception in Figure 3: line #20).

If the caller is an AOTC method, we check whether an exception occurred in the callee right after a method call; if so, we try to find an exception handler in the caller method. If there is no appropriate exception handler, the method also returns and this process is repeated until an appropriate exception handler is found. This means that we need to add an exception check code after every method call (i.e., *if (exceptionOccur(ee)) {…}* in Figure 3: line #26). If the caller is an interpreted method, it checks the environment variable whether an exception occurred in the callee; if so, find an exception handle by searching the interpreter stack. If it could not find an exception handler until meeting an AOTC method in interpreter stack, it makes a return to the caller AOTC method.

F. AOTC Method Linking

The DVM binary code coexists with the native code for the AOTC methods. As we described in section A, the native code for the AOTC method is shared by all forked app processes from the zygote. This is possible after the linking process is done for the AOTC methods. We link the native code of the AOTC method at the method table entry of the DVM.

Our AOTC generates an *AOTC table* that has a pointer to the generated native code, based on the class name and the method table offset for each AOTC method. The method table offset is obtained in the static linking of *dexopt*. This table is used for linking the native code for AOTC methods, as follows.

After the framework class is loaded in the zygote process, it uses the AOTC table to find AOTC methods in the loaded classes. For each AOTC method, it sets the method type in the method table and links to the native code. And we modify the class loading module to link app AOTC methods based on the AOTC table in each app processes' DVM.

IV. AOTC Code Optimizations

A. Method Inlining

The call interface to push/pop the stack frame and to pass

the argument/return value is an overhead. And Java as an object-oriented language includes many calls of short methods. So the call overhead takes a significant portion of the total execution time. Most C compilers perform function inlining. However, function inlining of the C compiler is not enough since it cannot inline large methods or virtual methods. Therefore, we perform inlining in the context of the AOTC.

We perform inlining for the static method easily since the callee is known at translate time. We also try to inline virtual methods based on static linking in the *dexopt*, but we add a check code to confirm whether the inlined method is right method to call.

B. Spill Optimization

As we mentioned, our AOTC saves a reference variable to the reference save slot (i.e., spilled) whenever it is updated for GC. However, if a reference variable is not live at any GC point, we do not need to save it. So our AOTC performs live analysis at each GC-point and generate C code for saving the reference variables for only live references. This can also reduce the reference save slot.

C. Elimination of Redundant Code

The C compiler can remove redundant code, yet AOTC can remove additionally based on Java specific features.

Java requires checking whether an object pointer is NULL before referencing it. Therefore, our AOTC adds a check code before each object reference. Obviously, many of these checks are redundant or unnecessary. The C compiler optimization may eliminate some. However, the AOTC can understand the control flow information of the original Java program better than the C compiler because C compiler would read the check code as a regular branch. And the AOTC can know that *this* pointer in virtual method can never be null because the caller method checked it already. So our AOTC finds redundant null checks or null checks for *this* pointer, and remove them.

All of the redundant copy bytecodes are removed by the bytecode compiler, *dx*. Our AOTC generates new copy operations for method inlining because argument passing are replaced to copy operations, yet most of these copies can be removed by the C compiler. One problem is that copies of the reference-type arguments would generate unnecessary save operations (spill) and save slots for GC. To reduce unnecessary reference save slots and save operations, our AOTC perform copy propagation for these references, which removes such copies and reference save slot allocation.

V. Experimental Result

We experimented with Android KitKat (version 4.4.1) for which we implemented a bytecode-to-C AOTC. The experiments were performed on a tablet called *Nexus 7*. It has a 1.9GHz quad-core ARM Cortex-A15 CPU with 2GB memory.

The C compiler we used is the gcc in the Android tool chain (arm-linux-androideabi-gcc 4.7). It compiled the DVM source code with –Os optimization by default. The –Os optimization focuses on the binary size. However, it is more important to optimize performance than binary size of AOTC, so we used the optimization level of -O3 for our DVM AOTC.

For the Java benchmark, we used the EEMBC GrinderBench. We translated EEMBC from a Java executable to a DVM executable using translation tool *dx* because EEMBC is a JVM benchmark. We executed the EEMBC in the DVM directly using android command line because the translation result is not an android app. It does not use zygote process forking mechanism.

For the android app benchmark, we used the AnTuTu, Quadrant, Linpack, and Benchmark PI. Some benchmark items spend most of running time in the native code. So we ignore these benchmark items. As a result, we used only the UI in AnTuTu and the CPU in Quadrant.

We cannot AOTC all of framework methods and benchmark app methods. In our experimental environment, android platform have around 90,000 Java methods in framework. To choose candidates of AOTC, we used profiled information.

We modified the *Traceview* tool included in the Android for method profiling to collect larger profile data. We choosed AOTC candidate method based on the portion of it running time. The sum of the running time of the chosen AOTC methods covers the 80% of the total running time in all Java methods, except for native methods (such as JNI and native platform methods).

We first measured the DVM AOTC performance. Figure 5 shows the performance of the DVM AOTC performance compared to that of DVM JITC as a basis.

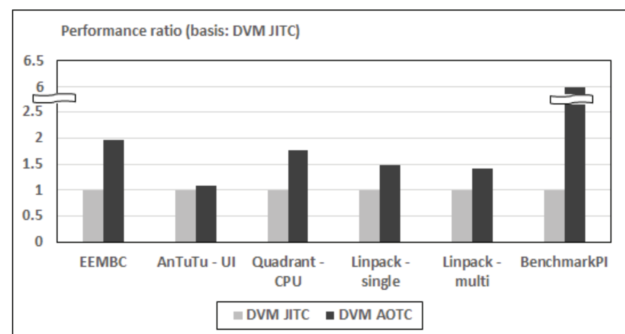

Figure 5. Dalvik AOTC performance in benchmark

The DVM AOTC is 8% faster than DVM JITC for AnTuTu-UI. It shows a 97% speedup in EEMBC, while showing 77% and 41% speedup for Quadrant and Linpack, respectively. BenchmarkPI is 6 times faster with AOTC, and since it has one hot method which is large and complex, DVM AOTC generates much more optimized native code than the DVM JITC does.

We analyzed the performance of DVM AOTC more using the EEMBC Grinderbench. The EEMBC Grinderbench consists of 6 programs: *Chess*, *Crypto*, *kXML*, *Parallel*, *PNG*, and *RegEx*. For these experiments, all of the called library and app Java method are compiled.

We evaluated the impact of optimizations. We first evaluated the performance of the method inlining. In order to isolate the performance impact of other optimizations, we experimented with only method inlining for static method turn on and for with virtual method, compared to that all optimization turned off as a basis. Figure 6 shows that our static method inlining leads to a performance improvement of an average of 12%, and we achieved 21% performance improvement with virtual method inlining. *Chess* and *Crypto* with virtual method inlining show some slowdown compared to the static method inlining only. We found that this is due to the overhead to compare inlined virtual method and real callee method because we failed to use the inlined method.

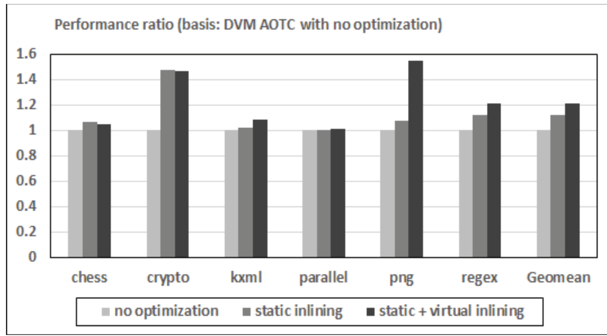

Figure 6. Performance impact of method inlining

We also evaluated the impact of redundant code elimination and spill optimziation. Figure 7 shows the performance impact of these optimizations, with method inlining turned on as a basis. It shows that the performance impact of both optimization is an average of 8%. When we disabled the redundant code elimination, the performance improvement is dropped to 5%. When we disable the spill optimization, the performance improvement is reduced to 2%, so it has a higher impact.

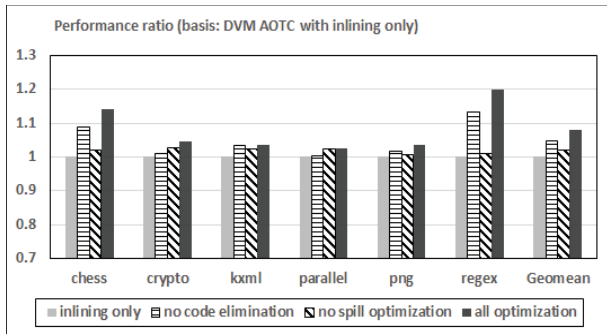

Figure 7. Performance impact of optimizations

We attempt to evaluate our DVM AOTC compared to ART. Figure 8 shows the performance comparison of DVM AOTC and ART based on DVM JITC. The graph shows that DVM AOTC has better performance than ART AOTC an average of 44%.

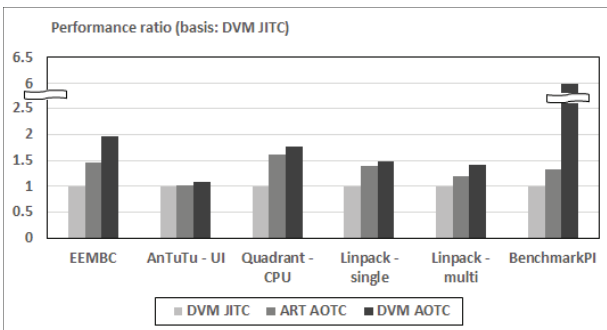

Figure 8. Comparison of Dalvik AOTC with ART

VI. Related Works

There are many previous bytecode-to-C AOTCs for JVM [6-8]. But many of previous JVM researches do not deal with the compilation strategy, call optimization and Java-specific optimizations issues specifically.

Bytecod-to-native server AOTC for DVM was presented in [12]. In this approach, profile data is used for selecting AOTC target methods. They used DVM JITC for code generation and additional optimization, but they get 13% performance gain in benchmarkPI. Another bytecode-to-C AOTC for DVM was proposed in [13]. This approach also used profile data for selecting AOTC target methods. But this research is for compiling general app, not android framework. And this research do not deal with the call optimization.

VII. Summary

DVM employs trace-based JITC instead of method-based JITC, which suffers from worse optimization. To overcome performance problem, this paper proposes a bytecode-to-C AOTC for DVM. We translated the bytecode of some of the hot methods to C code based on profile data. Then we compiled the C code with the DVM source code using a C compiler.

AOTC-Generated native code works with the existing Android zygote mechanism, with correct GC and exception handling. We also described call interface for efficient handling of interpreter-to-AOTC calls as well as AOTC-to-AOTC calls. We also present Java-specific optimization such as method inlining, spill optimization, and unnecessary code elimination. Due to these optimizations, AOTC can generate more optimized code than JITC while obviating runtime compilation overhead.

Our results with benchmarks show that DVM AOTC can improve the performance than DVM JITC tangibly. Comparison with ART shows that our approach appears to perform better than ART for pre-installed app.

References

[1] D. Bornstein. Dalvik VM internals, http://sites.google.com/site/io/dalvik-vm-internals

[2] Sun Microsystems, Porting Guide Connected Device Configuration and Foundation Profile version 1.0.1 Java 2 Platform Micro Edition, 2002.

[3] B. Cheng, B. Buzbee. A JIT Compiler for Android's Dalvik VM. http://www.google.com/events/io/2010/sessions/jit-compiler-androids-dalvik-vm.html

[4] Android Open Source Project, Introducing ART, http://source.android.com/devices/tech/dalvik/art.html

[5] Android Open Source Project, Bytecode for Dalvik VM, https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html

[6] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson, "Toba: java for applications a way ahead of time (WAT) compiler", in *Proc. of* COOTS, June 1997.

[7] G. Muller, B. Moura, F. Bellard, and C. Consel, "Harissa: a flexible and efficient java environment mixing bytecode and compiled code", in *Proc. of* COOTS, 1997.

[8] A. Varma, and S. S. Bhattacharyya, "Java-through-C Compilation: An Enabling Technology for Java in Embedded Systems", in *Proc. of* DATE, 2004.

[9] R. Jones and R. Lins, Garbage Collection Algorithms for Automatic Dynamic Memory Management, JOHN WILLEY & SONS, 1996

[10] J. Gosling, B. Joy, and G. Steele, The Java Language Specification Reading, Addison-Wesley, 1996.

[11] T. Ogasawara, H. Komatsu and T. Nakatani, "A Study of Exception Handling and Its Dynamic Optimization in Java", in *Proc. of* OOPSLA, 2001.

[12] Y. Lim, S. Parambil, C. Kim, and S. Lee, "A Selective Ahead-of-Time compiler on Android Device", in *Proc. of* ICISA, 2012.

[13] C. Wang, G. Perez, Y. Chung, W. Hsu, W. Shih, and H. Hsu, "A method-based ahead-of-time compiler for android application", in *Proc. of* CASES, 2011