A Method-Based Ahead-of-Time Compiler for Android Applications

Chih-Sheng Wang
National Tsing Hua University
Location City: Hsinchu
Location Country: Taiwan
alextrax@sslab.cs.nthu.edu.tw

Guillermo A. Perez
National Tsing Hua University
Location City: Hsinchu
Location Country: Taiwan
gaperez64@sslab.cs.nthu.edu.tw

Yeh-Ching Chung*
National Tsing Hua University
Location City: Hsinchu
Location Country: Taiwan
ychung@cs.nthu.edu.tw

Wei-Chung Hsu
National Chiao Tung University
Location City: Hsinchu
Location Country: Taiwan
hsu@cs.nctu.edu.tw

Wei-Kuan Shih
National Tsing Hua University
Location City: Hsinchu
Location Country: Taiwan
wshih@cs.nthu.edu.tw

Hong-Rong Hsu
MediaTek Inc.
Location City: Hsinchu
Location Country: Taiwan
hong-rong.hsu@mediatek.com

ABSTRACT

The execution environment of Android system is based on a virtual machine called Dalvik virtual machine (DVM) in which the execution of an application program is in interpret-mode. To reduce the interpretation overhead of DVM, Google has included a trace-based just-in-time compiler (JITC) in the latest version of Android. Due to limited resources and the requirement for reasonable response time, the JITC is unable to apply deep optimizations to generate high quality code. In this paper, we propose a method-based ahead-of-time compiler (AOTC), called Icing, to speed up the execution of Android applications without the modification of any components of Android framework. The main idea of Icing is to convert the hot methods of an application program from DEX code to C code and uses the GCC compiler to translate the C code to the corresponding native code. With the Java Native Interface (JNI) library, the translated native code can be called by DVM. Both AOTC and JITC have their strength and weakness. In order to combine the strength and avoid the weakness of AOTC and JITC, in Icing, we have proposed a cost model to determine whether a method should be handled by AOTC or JITC during profiling. To evaluate the performance of Icing, four benchmarks used by Google JITC are used as test cases. The performance results show that, with Icing, the execution time of an application is two to three times faster than that without JITC, and 25% to 110% faster than that with JITC.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors - *Compilers, Optimization, Parsing*

General Terms

Performance, Design, Experimentation, Languages

* The corresponding author

CASES'11, October 9–14, 2011, Taipei, Taiwan.

Keywords

Ahead-of-time compiler, just-in-time compiler, reverse engineering, Android, Dalvik bytecode, static profiling

1. INTRODUCTION

Android [1] is an open source and customizable mobile platform introduced by the Open Handset Alliance (OHA), established by Google and other companies in 2007. It is based on the Linux Kernel 2.6 and is accompanied with the Dalvik virtual machine (DVM) to support interoperability among different mobile devices. The instruction set of DVM is register-based and the virtual machine code is called DEX bytecode. To execute an Android program on DVM, the Android program is first compiled into Java bytecode by the javac compiler [12]. The Java bytecode is then converted to DEX bytecode by a Java-to-DEX translation tool called DX [13]. Overall, the execution mechanism of DVM is similar to the Java virtual machine (JVM).

Although the register-based DEX bytecode is easier to interpret than stack-based Java bytecode, the interpretation overhead is still significant. To minimize the interpretation overhead, JITC and AOTC are two commonly used optimization methods. For the JITC method, it can take advantage of runtime profiling (usually more representative than static profiling), but may suffer from compile time, power consumption and memory usage. On the other hand, the AOTC method is less constrained by memory usage, power consumption and time to compile/optimize. It can apply deep analyses and optimizations to generate high quality code.

In Android 2.2, a trace-based JITC has been provided in DVM. Overall, the JITC compiled code is 2 to 5 times faster than the original DEX bytecode according to Google's experiments on the mobile device Nexus One. However, there still is room for performance improvement because the Dalvik JITC, so far only, exploits some simple optimizations such as load/store elimination, redundant null-check elimination, and so on.

AOTCs, in general, can be divided into two categories: standalone-mode and mixed-mode. A standalone-mode AOTC compiles the whole application, including Java libraries, into native codes. A mixed-mode AOTC compiles only hot methods (methods that consume a significant fraction of execution time) into native codes, and relies on the virtual machine to handle non-

translated code. On resource constrained platforms, such as the mobile devices, the standalone-mode AOTC approach may not be suitable since the native code generated often exceeds many MBs. Using the mixed-mode AOTC approach, only a small portion of time consuming codes are translated into native codes. The rest of the codes and libraries can be leveraged from the VM environment. This imposes a much smaller memory requirement than the standalone-mode AOTC approach. In the mixed-mode AOTC approach, AOTC and JITC can also collaborate with each other to get better performance.

In this paper, we have implemented a mixed-mode AOTC, called Icing, for Android applications. The goal of Icing is to reduce interpretation overhead of DVM and to exploit more aggressive optimizations than JITC to improve the performance of Android applications. The main idea of Icing is to identify and compile time consuming methods of an Android application into native codes, and allow such native codes to be called by the DVM via JNI at run time. How to select hot methods for native code translation and how to translate a DEX bytecode to an efficient native code are the two main challenges of Icing.

The selection of hot methods for native code translation consists of two steps. In the first step, a static proofing technique is used to calculate the execution frequency of each method of an Android application. Those with high execution frequency are selected as candidate hot methods. The downside of using JNI as the interface is its high overhead. For example, for the native code to access information from the DVM side, such as field references, the access time can be 2 to 10 times slower than directly accessing from the DVM side. Therefore, it is imperative that Icing must carefully select methods to compile in order to minimize the communications between DVM and the native code. In the second step, we propose a cost model to determine whether it is worth to translate a candidate method to native code in terms of the execution time gained by native execution and the number of JNI calls. If the translation of a candidate hot method to native code can speed up the execution of an Android application, the candidate hot method will be translated to native code.

For the translation of a DEX bytecode to an efficient native code, a DEX bytecode is first translated to C code. The C code is then translated to native code using GCC compiler. A set of techniques have been proposed to handle some translation challenges such as information loss during code conversion and variable type recovery from mapping a low level typeless register to a high level C variable with a data type. To minimize the impact of information loss, Icing provides annotations to the immediate forms during the DEX-to-C conversion process. For variable type recovery, Icing supports two approaches, one with variable renaming and the other using the C union to allow different types housed in one memory location. In order to further reduce the overhead of JNI operations, we have also integrated three optimizations: ahead-of-time constant pool resolution, caching the information of method/field ID's obtained from DVM for quick references in the native code, and cloning methods in native side to avoid context changing overhead.

To evaluate the performance of Icing, four benchmarks used by Google JITC are used as test cases. They are CaffeineMark 3.0 [17], Checkers game [24], Linpack [23], and the BenchmarkPi [22]. The performance results show that, with Icing, the execution time of an application is two to three times faster than that without JITC, and 25% to 110% faster than that with JITC.

The rest of this paper is organized as follows. In Section 2, the overview of the Icing framework is given. The static profiling model is presented in section 3. In Section 4, the detail of DEX-to-C translation and what optimizations are employed in the Icing framework are described. Experimental results are shown in section 5. In Section 6, the related work is given.

2. OVERVIEW

2.1 Icing Components

The Icing framework is composed of three components, a static profiling model, an Icing AOTC, and a bridge library. The static profiling model is used to find out which methods are suitable for AOTC and which are better to be handled by the interpreter or JITC. The entire profiling process is conducted at static time. The Icing AOTC is used to translate selected hot methods into C code, translate the C code to native code, and link the native code with the bridge library. The bridge library is implemented with JNI, and contains APIs to handle operations of accessing DVM's resources from the native side.

2.2 The Execution Flow of Icing

Figure 1 illustrates the execution flow of Icing. At the beginning, the static profiling model is used to help identifying hot methods of a DEX bytecode. Methods of an Android application will be divided into two sets, AOTC and JITC lists. For methods in the AOTC list, their DEX bytecodes are fed into the Icing AOTC as input. After compilation, Icing modifies the original DEX bytecode by replacing the original hot method's bytecode with a native header, so that each method that invokes the hot method will call the native code generated by Icing. At this stage, the package with the modified DEX code and the native code generated by AOTC is built into a new application (.apk). Finally, the newly optimized application can be executed on the DVM and can switch between the native side and the DVM with the help of the bridge library.

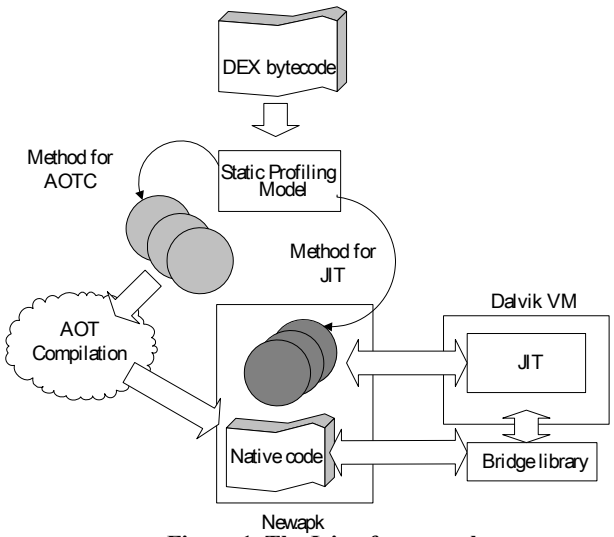

Figure 1. The Icing framework

3. The Static Profiling Model

Unlike GCJ [2] and Toba [4], Icing does not implement the entire Java/Android core libraries as native code for the code size issue. Icing instead exploits the JNI to support the invocation of core libraries and the accessing of fields from the DVM side. Therefore, the goal of the static profiling model is to classify methods of an Android application into AOTC and JITC lists.

3.1 The Profiling Method

In Icing, we use Traceview [28], a profiling tool provided by Google, to collect the run-time information of methods of an Android application. The information collected includes the call graph, the execution time of each method, the frequency of invocations, and the execution time percentage of each child method. Since the Android application is only executed once by the Traceview, Icing uses a static profiling approach to collect run-time information of methods of an Android application. The drawback of the static profiling approach is that the input data of an application may change from one run to the other. It is hard to guarantee that the statically collected profile information can cover all the running circumstances at runtime. To minimize the impact from inaccuracy introduced by the static profiling approach, we use the Monkey [25] program to generate potential user behaviors as much as possible at static time. The Monkey [25] is a program provided by Google. It generates random streams of user events such as clicks, touches, or gestures. Some experimental results on smart phones show that the Monkey can almost cover all paths for some embedded applications and enhance the quality of the static profiling approach.

3.2 Detecting Hot Methods

Since we prefer not to compile the core libraries, the opportunity for improvement resides in user defined methods that are hot. The main idea of the hot method detecting mechanism is to collect and count the occupancy of the user-defined methods in the entire execution-flow, and add the methods whose occupancy exceed a predefined threshold to the candidate AOTC list. The counting flow is organized into the following steps:

- Step 1: Select one of the most time-consuming methods according to profiled data and check the self-code execution rate of this method. If the self-code execution rate of the selected method is over a pre-defined threshold, e.g. over 80%, push the selected method into the candidate AOTC list. If it is not large enough, the occupancy of the selected method is equal to its self-code execution rate and go to step 2.
- Step 2: Check all user-defined child methods of the selected method and add their execution rates on user-defined codes to the occupancy of the select method recursively.
- Step 3: If the occupancy of the selected method is over a pre-defined threshold, push the selected method in to the candidate AOTC list.

Through this hot method detecting mechanism, a method which spends a lot of time on the core library will have a low occupancy, and is not considered a candidate for Icing.

An example is given in Figure 2 to show the occupancy calculation of a method. In Figure 2, we assume that method A is one of the most time-consuming methods according to profiled data. Since the self-code execution rate of method A is 50% and is not over 80%, the occupancy of method A is 50% and we proceed to Step 2 to check the user-defined child methods of method A. In this example, it is method A_1. We need to add the execution rate on user-defined code of method A_1 to the occupancy of method A. Since method A_1 has a user-defined child method A_1_1, the execution rate on user-defined code of method A_1 is equal to the execution rate of method A_1 times the sum of the self-code execution rate of method A_1 and the execution rate on user-defined child method of method A_1, which is 30% * (50% + the execution rate on user-defined code of method A_1_1). Since method A_1_1 has no user-defined child method, the execution rate on user-defined code of method A_1_1 is 35% * 70%. The final occupancy of method A is 50% + 30% * (50% +35% * (70%)) = 72.35%, which indicates that the execution-flow started from method A spent 72.35% of its execution time on the user-defined code.

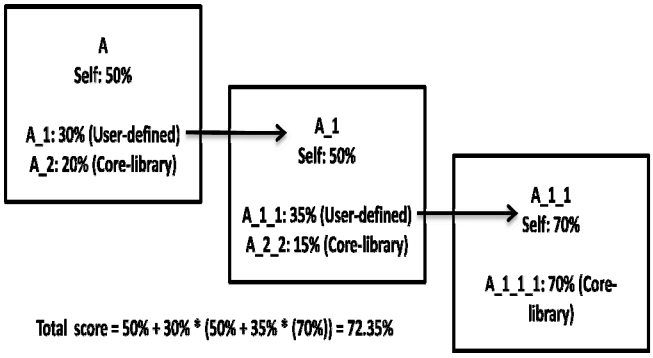

Figure 2. An example of the profiled data used in Icing

3.3 Avoid Invocation Overhead

Besides the hot method detecting mechanism, we need to avoid compiling methods which contains frequent JNI invocations by the native code generated from Icing. A simple mechanism is proposed based on the ratio of the number of JNI invocations of a method to the execution time of that method. If the ratio exceeds a threshold, we exclude the method from the candidate AOTC list. This mechanism helps us to minimize the impact of frequent JNI invocation overhead. However, if the threshold is not set appropriately, the performance will go down. The threshold is currently set based on the feedback from a set of experiments. For example, if the number of JNI invocations exceeds 700 times per second, it is unwise to convert this method to native code. In the future, we will set the value adaptively based on the prediction of the performance gap between code generated by Icing and JITC.

3.4 The Cost model

Based on the concept described above, we define a cost model to identify which methods are suitable for AOTC and which are good for JITC. Figure 3 shows how the cost model works. First, we calculate each method's occupancy with the approach provided in section 3.2. Those methods whose occupancy does not exceed the threshold are put to JITC list and other methods are in the candidate AOTC list. For each method in the candidate AOTC list, we then check if the ratio of the number of JNI invocations of a method to the execution time of that method exceeds the predefined threshold. If so, then it is in the JITC list. Otherwise, it is in the AOTC list.

4. COMPILATION AND OPTIMIZATION

In this section, we discuss how to translate the DEX bytecode to native code and what additional optimizations are implemented in the Icing framework.

4.1 DEX-to-C Convertion

The main work of the Icing AOTC is to translate DEX bytecode to C code. The work is based on the COINS compiler infrastructure [27]. The translation steps are shown in Figure 4.

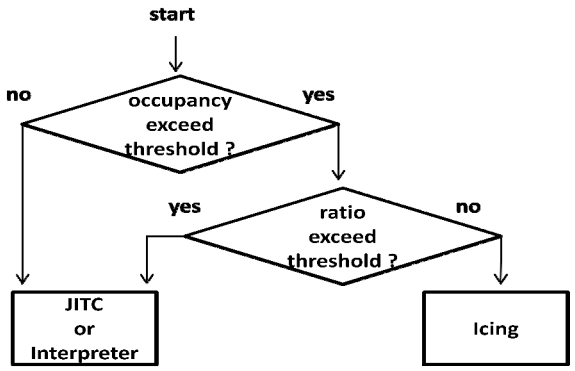

Figure 3. The Cost model to classify methods

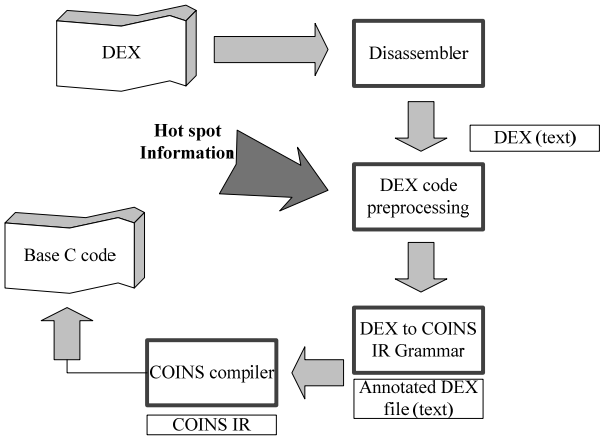

Figure 4. The DEX-to-C compilation flow of Icing

4.1.1 DEX code Preprocessing

In this preprocessing, the DEX binary is disassembled into a human-readable file by using *backsmali* and *dexdump*. *Smali/backsmali* [14] is an open source tool which serves as the assembler and disassembler for DEX bytecode. Dexdump is a DEX disassembler provided by Google. We employed both tools in order to obtain all the necessary information needed for converting DEX bytecode to C code. The JNI headers, which play the role of bridging DVM and native code, are generated as a side product. The main purpose of this phase is to generate a better prepared DEX code and parse it into the COINS's IR. Two additional works have been included in this step. The first is to chain the hot methods together to form cycles of native calls. This can reduce the associated overhead of procedure calls. The second one is to insert annotations to provide information for the upcoming compilation steps.

Chaining the hot methods can be done by modifying the indirect-jump bytecode instructions. We first analyzed the bytecode statically and collected child method information of the hot methods. If the child method is a user-defined method, we patch the branch instruction in the parent with a direct jump to the compiled native child method. This is to avoid unnecessarily switching back and forth between native mode and DVM mode. More details will be provided in the optimization section.

Annotation to the DEX code is a way to keep important information around for building the COINS's IR and later translating IR to C code. Table 1 lists the annotations we have used. In Table 1, the ".descriptor" gives hints about a method's parameter type and return type to help constructing a method's parameters and return variable from virtual registers to C variables. For example, ".descriptor %(I)I non-static %" indicates that the non-static function which is going to be compiled has one integer parameter and returns an integer value. The ".arrayType" informs if an array is a class field or local variable. The ".line" records a bytecode's line number. The "offset/index" records the offset resolved by the *dexopt* tool for optimizing the call-back operations. For example, "+iget-quick v3, v6, [obj+0014] I" represents that we want to get an integer field from object v6, store it to v3, and the offset of this field is "0014".

Table 1. An example of annotations

Annotation kind	Example
.descriptor	.descriptor %(I)I non-static %
.arrayType	.arrayType %[[D% 1
.line	.line 43
offset/index	+iget-quick v3, v6, [obj+0014] I

4.1.2 DEX-to-C type recovery issues

An ideal AOTC should translate DEX code directly to native code. However, building such a compiler, especially with comprehensive optimizations, is a time consuming and daunting task. One compromise is to leverage some existing compiler infrastructures, such as the GCC and the LLVM compilers. Therefore, we have determined to take the path of translating DEX code to C code, and take advantage of GCC's mature optimizations. There are some issues in the DEX-to-C translation process.

```
+iget-object-quick v4, v6, [obj+0010] Ljava/lang/String;
...
mul-int/lit8 v4, v1, #int 65 // #41
...
```

Figure 5. Virtual registers of DEX bytecode

```
int v4;
…
void * Ljava_lang_stringBuffer_v4;
…
```

Figure 6. Example code of variable renaming

The first one is that Java is an object-oriented language and provides various features to support object types, such as instance creation, static/virtual method invocation, field accessing, etc. Since those features are usually done through JNI, we have implemented them as part of our bridge library to cover these features in the translated C code.

The second one is to handle function overloading in Java. We use the method renaming approach to rename each method to a unique name in C. For example, the method "int execute(void*)" in class "com.android.cm3.StringAtom" is renamed to "int com_android_cm3_StringAtom_execute_VSTAR (void * v6)" by the Icing AOTC in the translated C code.

```
typedef union _JValue {
    jboolean     z;
    jbyte        b;
    jchar        c;
    jshort       s;
    jint         i;
    jlong        j;
    jfloat       f;
    jdouble      d;
    void*        l;
} _JValue;…
…
v4.l        =        (hir___ADDR(_dvmGetFieldObject))
( v6,(hir_t_int  )16);
 …
v4.i = hir___MULT(v1.i,(hir_t_int  )65);
```

Figure 7. Example code of union variables

The third one is that the bytecode instructions of Dalvik virtual machine are register-based, which means all computations are handled at the register level by using almost unlimited numbers of virtual registers. So there is a type recovery issue similar to what happens during traditional assembly-to-C translations. When mapping a virtual register to the C code, we usually map a register to a unique variable name. However, a virtual register is basically typeless. Like a general purpose register, it can store any data. The DEX bytecode, as shown in Figure 5, illustrates the issue involved. From Figure 5, we can see that register v4 was used to store the java.lang.String object by the instruction "+iget-object-quick", but was loaded with an integer value later by the instruction "mul-int/lit8". What type should be set when virtual register v4 is converted to a C variable?

One commonly used solution for type recovery is renaming, which applies live-range analysis to split one virtual register into multiple sub-live-ranges, and each sub-live-range can be mapped to a different variable with its own type. However, this will drastically increase the number of variables used, and requires complex analyses. With the help of type information annotations from the preprocessing of DEX code, we can map each virtual register to a union variable that includes eight primitive types, and one pointer for object types. By doing so, we reduce the complexity of type recovery and the number of variables required in C code. Figures 6 and 7 give the example codes of the two mechanisms.

4.1.3 Code generation

After the preprocessing shown in Figure 4, the DEX code is parsed into the High-level Intermediate Representation (HIR) of the COINS compiler infrastructure, and the corresponding C code is generated with the support of the COINS's HIR-to-C translator. Eventually, the C code is compiled into a shared library with the arm-gcc-4.4.0 by the Android NDK. Furthermore, we developed some APIs to handle the accesses to the VM's resources from the native side, and packaged them with the bridge library. The package is compiled with the native code later. These APIs perform operations such as method invocation, field accessing, instance creation and so on.

4.2 Optimizations

With JNI, Java code is able to call functions implemented with other languages. However, JNI suffers from time and space overhead just like other mechanisms of supporting interoperability. The causes of JNI overhead can be divided into two categories, call-out and call-back operations. The call-out operations are used by the VM to invoke a native function through JNI. The overheads of the call-out operations include argument passing, native initialization, returning from native code, etc. Such overheads can sometimes be reduced via dynamic inlining optimizations.

The call-back operations are used by the native code to access VM's resources. Compared with the call-out operations, the call-back operations involve more significant overhead since they need to access the VM's resources from the native code. The call-back operations may suffer from the indirect-jump overhead caused by referencing the JNI environment variable. In addition, when a call-back operation is executed, it takes significant amount of time to resolve the offset or index in the constant pool and then performs a context-switch. Table 2 shows the comparison of the execution time of a call-back operation to the time of an equivalent operation in the VM side. In Table 2, each operation performs a thousand times. As we can see, the performance gap is so wide that some optimizations are called for. A native call inlining mechanism was presented by Levon Stepanian *et al*. [20] to eliminate the JNI call-out and call-back overheads. Here, without the necessity of modifying Android's DVM, we focus on optimizing the call-back overhead to improve the quality of generated native code. We introduce three optimizations that are applied in Icing. They are ahead of time resolution, caching, and method cloning.

Table 2. The time of 1000 call-back operations

	Native	Java
Non-static field	**127.2 ms**	**11.8 ms**
Static field	**108.6 ms**	**6.2 ms**
Non-static method	**42.8 ms**	**2.0 ms**
Static method	**124.5 ms**	**31.1 ms**

4.2.1 Ahead of time resolution

To avoid slow field accessing from the native side, some types of instructions such as field accessing and method invocation are organized with a reference in symbolic forms. The reference is converted to offset/index later. This process is called constant pool resolution. In order to eliminate the overhead of constant pool resolution during run-time, we get the variables' offset in the constant pool ahead of time. The *dexopt* [15], a tool provided by Android, makes this job easier. The DEX bytecode after optimization by this tool is called ODEX. Some constant pool referencing instructions in ODEX are replaced by a quicker version, which reference the constant pool by means of offsets. With the information provided by ODEX, we can significantly speed up call-back operations in the translated native code. However, this does not include static references because the offsets of static references are related to the address from where the class is loaded.

4.2.2 Caching

For the static field accessing and static method invocation, there are no short cut solutions provided by ODEX, such as the ahead of time resolution. Therefore, we implement a caching mechanism for improving the performance of static call-backs. To access the VM's resource in a standard way through JNI the class of the object is obtained first, then the method/field ID is found by name comparison, finally, the call-back operation is performed with this ID.

To reduce this referencing overhead, the ID is cached at the native side. A hash table is used to record the method/field ID based on the method name or field name. Whenever a static call-back occurs, the hash table is looked up for the cached ID. If there's a match, the ID found is applied during static referencing. Otherwise referencing is carried out as normal and the newly found ID is put into the hash table for later use. Although this mechanism comes with some time and space overhead, the locality of ID reference allows a high hit rate in cache and yields six times performance improvement over the original solution.

4.2.3 Method cloning

In order to minimize context switch overhead between the native side and the DVM side, we compile all the child methods of those candidate methods. Furthermore, we patch the invoke instruction to direct branches so as to keep the execution context in native side as long as possible. Basically, this optimization greatly improves the performance of method invocation in native side. However, there is a downside to this approach: the total code size is increased due to code duplication for each child method in native side. As shown in Figure 8, method B invokes method C and native method D by using JNI originally, and this mechanism brings about significant overhead if the invocations happen frequently. To reduce the number of JNI invocations, we compile method C into a native version and replace the invoke instruction of method B with a direct jump to the compiled method C and native method D, as shown in the right hand side of Figure 8. Furthermore, we do not remove the Java version of method C. By doing that, the number of context switches between DVM and native side will be greatly reduced at both sides.

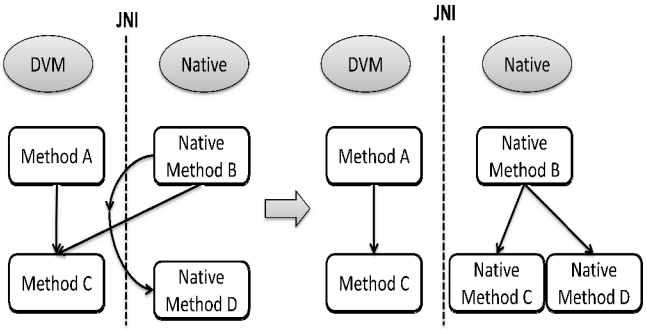

Figure 8. Method cloning

5. EXPERIMENTAL RESULTS

To evaluate the performance of Icing, four Android applications, CaffeineMark 3.0 [17], Checkers game [24], Linpack [23], and the BenchmarkPi [22], are used as benchmarks. These four programs can be downloaded from Android Market and have been used to test the performance of Dalvik JITC by Google. Our experimental environment is based on the HTC G1 mobile device running on Android 2.2. The measurements are based on four configurations, original, JIT, Icing, and Icing+JIT. For the original configuration, benchmarks are executed without the help of JITC and Icing. For the JITC configuration, benchmarks are executed with JITC. For the Icing configuration, benchmarks are executed with Icing. For the Icing+JIT configuration, benchmarks are executed with Icing and JITC.

5.1 Performance of CaffeineMark 3.0

CaffeineMark 3.0 is one of the well adopted benchmarks for measuring the performance of Java. It contains a series of tests, and the scores generally represent the number of Java instructions executed per second. Table 3 gives a description of the tests used in CaffeineMark 3.0. Figure 9 shows the performance results of CaffeineMark 3.0. Note that in this run, the profiling model in Icing has not been applied yet. As shown in Figure 9, the scores of Icing and Icing-plus-JIT, even without the profiling model, are much better than the original runs and JITC runs, in most test cases except for the String test. Overall, the Icing+JIT run has the best performance. The performance anomaly of the String test is due to the frequent JNI invocations.

Table 3. Description of each CaffeineMark 3.0 item

Item	Description
Sieve	The classic sieve of Eratosthenes finds prime numbers.
Loop	The loop test uses sorting and sequence generation as to measure compiler optimization of loops.
Logic	Tests the speed with which the virtual machine executes decision-making instructions.
Method	The Method test executes recursive function calls to see how well the VM handles method calls.
Float	Simulates a 3D rotation of objects around a point
String	Operation of basic string

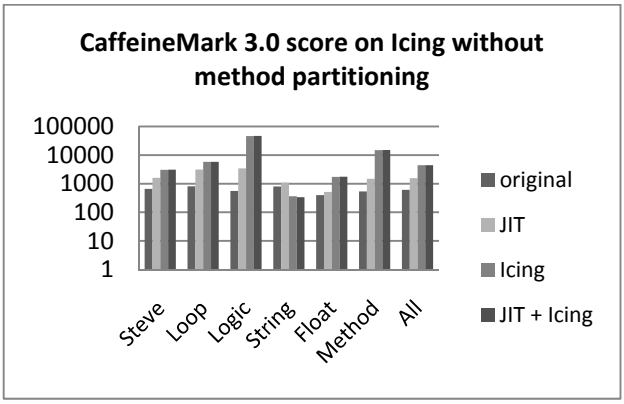

Figure 9. CaffeineMark 3.0 scores without the cost model

Table 4 shows the profiling information of the top 8 hot methods. The "Method" column represents the name of each method. The "Weight" column means the occupancy calculated. The "Calls" column shows the number of JNI call-back operations. The "Time" column gives the execution time of each method. According to the profile information in Table 4, the method SieveAtom.execute performs no JNI call-back invocations in

7.469956 seconds. However, the method StringAtom.execute performs 5061 JNI call-back operations in 4.56556 seconds. Overall, the method String.execute has the largest ratio of the number of JNI invocations to the execution time of String.execute. The JNI call-back operation overhead degrades the performance of this test. Besides, the JNI operations here has to pass complex object arguments such as strings or arrays which results in expensive copy operations from native side to the DVM side. By applying the cost model mentioned in Section 3, we can successfully avoid to translate the method StringAtom.execute to native code. Figure 10 shows the scores after applying the profiling model for CaffeineMark 3.0. From Figure 10, we can see that the perofrmance of Icing is now as good as JITC for the String test. We also observe that the score of CaffeineMark 3.0 is 7.3 times higher than that without JITC, and 2.83 times higher than that with JITC.

Table 4. Profiling information of CaffeineMark 3.0

Method	Weight	Calls	Time (s)
SieveAtom.execute	23.9	0	7.469956
LoopAtom.execute	8.5	0	2.659588
LogicAtom.execute	12.0	0	3.764250
StringAtom.execute	1.5	5061	4.56556
FloatAtom.execute	11.4	2665	3.576081
MethodAtom.execute	21.7	0	7.2172
MethodAtom.notInlineableSeries	11.0	0	3.461429
MethodAtom.arithmeticSeries	10.5	0	3.264694

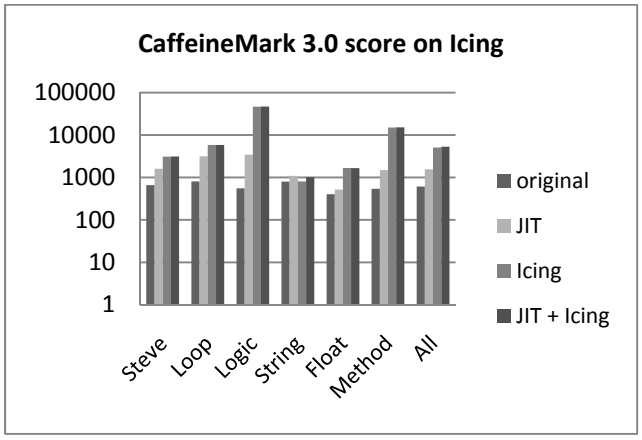

Figure 10. CaffeineMark 3.0 scores with the cost model

5.2 Performance of Linpack

The Linpack benchmark has been used for many years to test floating point computation. It measures a computer's performance by solving a linear equation $Ax = b$. A great improvement introduced by Icing can be revealed in Figure 11. From Figure 11, we can see that the Icing+JIT run has the best performance. We also observe that the execution of Linpack is 2 times faster than that without JITC, and 1.25 times faster than that with JITC. The results are a little bit discouraging. The reason is that the JNI issue comes out again, which can be observed from the last column. The matgen is a method of the Linpack application, which performs the JNI call-back invocations 80004 times in 5.336383 seconds. Therefore, we got essentially no benefit from compiling the matgen method ahead of time. Note that the current experimental platform HTC G1's ARM-based Qualcomm MSM7201 CPU does not have VFP (Vector Floating Point), so all the floating point operations of Linpack are based on software emulation. This significantly limits the potential performance contribution of applying aggressive compiler optimizations such as instruction scheduling and loop transformations. The speedup of Linpack from Icing can be far better if the underlying hardware has a floating point unit.

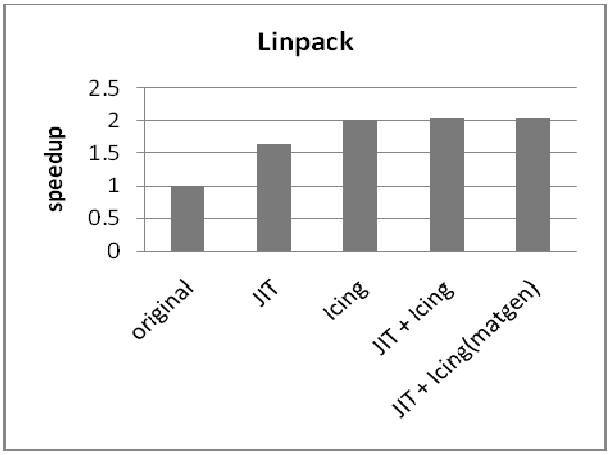

Figure 11. Performance comparison of Linpack

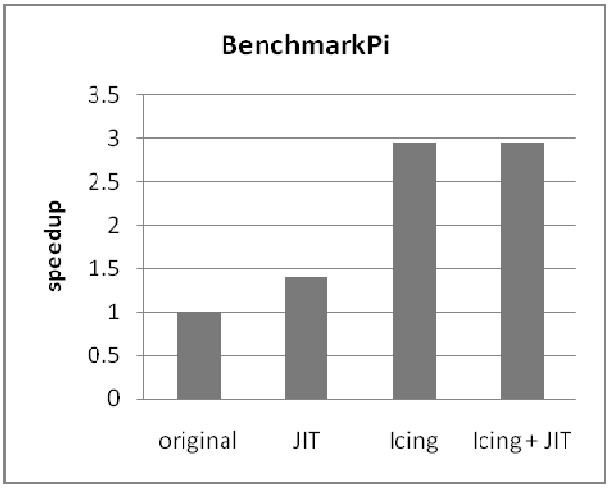

Figure 12. Performance comparison of BenchmarkPi

5.3 Performance of BenchmarkPi

The BenchmarkPi is another popular application on the Android Market. It can test a device by calculating the valued of Pi, and is a very useful tool to test the performance of a CPU. In this benchmark, Icing compiled the method that is responsible for the main workload of BenchmarkPi. The performance results are shown in Figure 12. From Figure 12, we can see that the Icing+JIT run has the best performance. We also observe that the execution of BenchmarkPi is 2.9 times faster than that without JITC, and 2.1 times faster than that with JITC.

5.4 Performance of Checkers

Checkers is a famous game. It is CPU intensive and is one of Google's favorite benchmarks to test the performance of Dalvik JITC. As the name implies, Checkers is a game playing on an 8x8 board of checkers against the computer. Whenever the computer finishes thinking of the next step to go, it shows a number of potential future moves on the bottom right corner. In other words, the faster your computer is, the more challenging the game will be for a human. The performance results are shown in Figure 13. From Figure 13, we can see that the Icing+JIT run has the best performance. We also observe that the execution of Checker is 2.61 times faster than that without JITC, and 1.67 times faster than that with JITC.

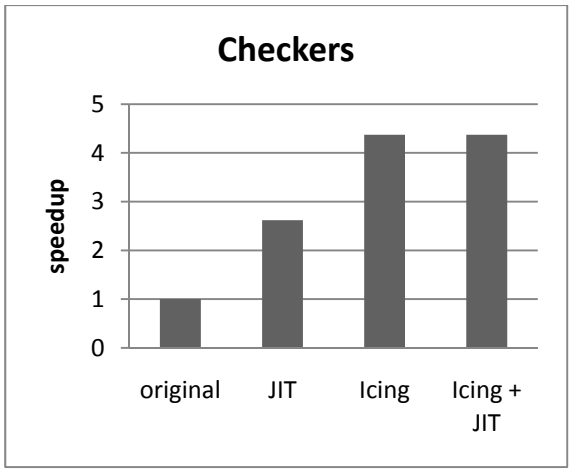

Figure 13. Performance comparison of Checkers

5.5 Comparisons of Icing and GCJ

It is worth comparing Icing (a mixed-mode AOTC) and GCJ (a standalone-mode AOTC) in terms of code size and performance of codes they generated. Since Icing is running on DVM and GCJ is running on JVM, we use CaffeineMark 3.0 that has both Android and Java versions as the benchmark for the comparison. Table 5 shows the the code size of CaffeineMark 3.0 with and without the optimization of Icing and GCJ. From Table 5, we can see that the code sizes of CaffeineMark 3.0 before and after the optimization of Icing are 17 KB and 69 KB, respectively. However, the code sizes of CaffeineMark 3.0 before and after the optimization of GCJ are 13 KB and 44.1 MB, respectively. Even after we dynamically link the generated code with the GCJ's shared library, it still requires 125 KB for the object code and 31.3 MB for the GCJ's library.

Figure 14 shows the perofrmance of CaffeineMark 3.0 with and without GCJ. The experimental environment is Java HotSpot(TM) Client VM [26] running on PC, and the performance comparison is given when GCJ is applied with and without static linking. The last bar "GCJ_static_opt" indicates that all optimizations of GCJ are applied on the generated code. From Figure 14, we can observe that the performance of the last bar (with all GCJ optimizations) can be even worse than the "JVM (enable JIT)" bar in some cases. Also the performance advantage of GCJ over JIT is not clear in most cases.

Table 5. Code sizes produced by Icing and GCJ

	CaffeineMark 3.0	
Method	Original	Optimized
Icing	17 KB	69 KB
GCJ (static)	13 KB	44.1MB
GCJ (dynamic)	13 KB	31.425MB

Overall, the comparion descibed above indicate that a mixed-mode AOTC like Icing that only compiles hot methods to native codes is a more desirable approach for mobile devices compared to a standalone-mode AOTC like GCJ in terms of code size and performance of codes they generated. Also, keeping native code running at the native side and reduce the number of JNI invocations are very important for Icing. With the optimizations and the profiling mechanism we proposed, Icing has achieved a great improvement for Android applications.

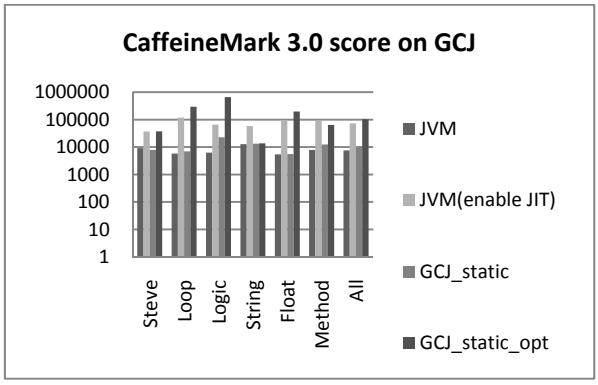

Figure 14. Performance of CaffeineMark 3.0 using GCJ

6. RELATED WORK

Previous AOTCs for Java virtual machine can be divided into two classes: standalone-mode and mixed-mode. Standalone-mode AOTCs, like Toba [4] and GCJ [2], translate the whole application to native code as a standalone executable. On the other hand, mixed-mode AOTCs only compile the hot spots and interact with VM, such as Harissa [5], TurboJ [6] and our work. Despite being able to use the AOTC to translate bytecode into native code directly, we take the path to generate C code first and then compile the C code with existing compilers to exploit the advantages of mature machine independent and machine dependent optimizations. . This approach has been inspired by A. Varma et al. [7] and G. B. Muller et al. [5], who built Harissa and were influential in the early discussions of our AOTC development. Muller also suggested using class hierarchical analysis [8], and transforming virtual calls into non-virtual calls for optimization.

On the aspect of profiling, Chandra Krintz et al. [9] [10] proposed a mechanism to reduce the JITC's run-time compilation overhead by adding annotations at static time. Apart from this, Krintz et al. combined the on-line and off-line profile information to apply different level of optimizations. Sunghyun rt al. [11] implemented a client-AOTC to reduce the translation and memory overhead of JITC by storing the native code generated by JITC in the permanent storage. They also proposed an approach to deal with the constant resolution issue when the native code is reused

in later runs. This early binding idea exists in the Icing too. However, our AOTC is based on server side in order to employ the more complex compilation frameworks to generate better quality of native code.

Levon Stepanian et al. [20] proposed a mechanism to reduce the JNI overhead, which is to inline the native calls with the help of JIT compiler's inlining optimization. Therefore, the native code will be executed in the JVM's context. This may effectively mitigate the negative impact of expensive JNI calls. The dex2jar [19] and undx [21] are works under developing which translate DEX bytecode back to Java bytecode, that is, map register-based instructions back to stack-based. With these tools, further optimizations and analysis can be done at Java level by the Soot [18] framework.

7. SUMMARY and CONCLUSIONS

AOTC can effectively avoid the interpretation overhead in a JVM when a JIT compiler is not available, or avoid the translation overhead of the JIT Compiler. Furthermore, AOTC can apply more aggressive optimizations to significantly improve the quality of generated code. In this paper, we show that a mixed-mode AOTC is more suitable for embedded devices. The generated native code can cooperate with the virtual machine via the JNI interface.

We have built an AOTC for the Dalvik virtual machine on the Android platform, called Icing. Instead of building a compiler from scratch, we leverage the comprehensive optimizations in GCC and its high portability. We convert DEX bytecode to C code and then compile C code into native code. A few challenging issues such as handling information loss due to low-level to high-level conversions and type recovery when converting DEX's virtual registers to C variables have been addressed by Icing. We have successfully built a prototype ahead-of-time compiler in a short time.

Icing translated native code is not always better than JIT compiled code. This is due to high overhead associated with JNI call-out and call-back operations. Icing has carefully minimized such overhead with several optimizations such as ahead-of-time resolution, caching method/field IDs, and method cloning. Furthermore, Icing leverages on the existing profiling mechanism to further determine which methods should be compiled by Icing or JITC. With the above optimizations, Icing has achieved a much better performance than the current JITC in DVM on four benchmarks from the Android Market.

Since the current JITC in DVM on Android 2.3 will continue to enhance, we may soon have JITC with adaptive optimizations to generate more competitive code. We believe an effective collaboration between JITC and AOTC can deliver the best cost performance and the best power-performance. Therefore, we will continue to search ways to improve Icing and enable more effective collaboration between AOTC and future JITC.

8. ACKNOWLEDGMENTS

This research was supported in part by MediaTek Inc., R.O.C., under Grant 099H19EA. We are also grateful to the anonymous reviewers for many helpful comments.

9. REFERENCES

[1] Google Android - An Open Handset Alliance Project, 2008. http://code.google.com/android/

[2] GCJ - The GNU Compiler for the Java Programming Language, from http://gcc.gnu.org/java/

[3] Sun Java Native Interface, 1997, from http://java.sun.com/j2se/1.3/docs/guide/jni/

[4] Proebsting, T. A., Townsend, G., Bridges, P., Hartman, J. H., Newsham, T. and Watterson, S. A. Toba: Java For Applications: A Way Ahead of Time (WAT) Compiler. University of Arizona, 1997.

[5] Muller, G., B, Moura, r., Bellard, F. and Consel, C. Harissa: A flexible and efficient java environment mixing bytecode and compiled code. In Proceedings of the Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 3 (Portland, Oregon, 1997). USENIX Association.

[6] Weiss, M., Fran, Ferri, o. d., Delsart, B., Fabre, C., Hirsch, F., Johnson, E. A., Joloboff, V., Roy, F., Siebert, F. and Spengler, X. TurboJ, a Java Bytecode-to-Native Compiler. In Proceedings of the Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (1998). Springer-Verlag.

[7] Varma, A. and Bhattacharyya, S. S. Java-through-C Compilation: An Enabling Technology for Java in Embedded Systems. In Proceedings of the Proceedings of the conference on Design, automation and test in Europe - Volume 3 (2004). IEEE Computer Society.

[8] Dean, J., Grove, D. and Chambers, C. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In Proceedings of the Proceedings of the 9th European Conference on Object-Oriented Programming (1995). Springer-Verlag.

[9] Krintz, C. Coupling on-line and off-line profile information to improve program performance. In Proceedings of the Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (San Francisco, California, 2003). IEEE Computer Society.

[10] Krintz, C. and Calder, B. Using annotations to reduce dynamic optimization time. In Proceedings of the Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation (Snowbird, Utah, United States, 2001). ACM.

[11] Hong, S., Kim, J.-C., Shin, J. W., Moon, S.-M., Oh, H.-S., Lee, J. and Choi, H.-K. Java client ahead-of-time compiler for embedded systems. (San Diego, California, USA, 2007). LCTES'07. ACM.

[12] Sun javac - Java programming language compiler, 2002, from http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/javac.html

[13] Google Android Dx tool, 2007, from http://developer.android.com/intl/zh-TW/guide/developing/tools/othertools.html#dx

[14] JesusFreke smali/baksmali: An assembler for Android's dex format 2009), from http://code.google.com/p/smali/

[15] Google Dalvik Optimization and Verification With dexopt, 2008, from http://android.git.kernel.org/?p=platform/dalvik.git;a=blob_plain;f=docs/dexopt.html;hb=master

[16] Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M. R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E. W., Reitmaier, R., Bebenita, M., Chang, M. and Franz, M. Trace-based just-in-

time type specialization for dynamic languages. In Proceedings of the Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (Dublin, Ireland, 2009). ACM.

[17] Pendragon CaffeineMark1997,

From http://www.benchmarkhq.ru/cm30/

[18] Vall, R., e-Rai, Co, P., Gagnon, E., Hendren, L., Lam, P. and Sundaresan, V. Soot - a Java bytecode optimization framework. In Proceedings of the Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research (Mississauga, Ontario, Canada, 1999). IBM Press.

[19] dex2jar: Translate android DEX bytecode back to Java bytecode, from http://code.google.com/p/dex2jar

[20] Stepanoan, L., Brown, A. D., Kielstra, A., Koblents, G., and Stoodley, K. Inlining Java native calls at runtime. In Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments. (Chicago, IL, USA, 2005) VEE'05. ACM.

[21] UNDX: Translate android DEX bytecode back to Java bytecode, from http://illegalaccess.org/undx.html

[22] BenchmarkPi – The Android benchmark tool, from

http://androidbenchmark.com/

[23] Linpack for Android,

from http://www.greenecomputing.com/apps/linpack/

[24] Checkers game, from http://aartbik.blogspot.com/

[25] Monkey, from

http://developer.android.com/intl/zh-TW/guide/developing/tools/monkey.html

[26] Java HotSpot Client and Server Virtual Machines, from

http://download.oracle.com/javase/1.3/docs/guide/performance/hotspot.html

[27] Sassa, M., Nakaya, T., Kohama, M., Fukuoka, T. and Takahashi, M.: Static Single Assignment Form in the COINS Compiler Infrastructure, SSGRR 2003w - International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, e-Medicine, and Mobile Technologies on the Internet, L'Aquila, Italy, No. 54 (Jan. 2003).

[28] Google Traceview profiling tool, from

http://developer.android.com/guide/developing/debugging/debugging-tracing.html