IEIE Transactions on Smart Processing and Computing, vol. 4, no. 3, June 2015
http://dx.doi.org/10.5573/IEIESPC.2015.4.3.169

DEX2C: Translation of Dalvik Bytecodes into C Code and its Interface in a Dalvik VM

Minseong Kim[1], Youngsun Han[2], Myeongjin Cho[1], Chanhyun Park[1], and Seon Wook Kim[1,*]

[1] Department of Electrical and Computer Engineering, Korea University / Seoul, Korea
{kissofgod, linux, yasutaxi, seon}@korea.ac.kr
[2] Department of Electronics Engineering, Kyungil University / Gyeongsan, Korea youngsun@kiu.ac.kr

* Corresponding Author: Seon Wook Kim

Received March 27, 2015; Revised April 9, 2015; Accepted April 17, 2015; Published June 30, 2015

** Short Paper*

** Extended from Conference: Preliminary results of this paper were presented at the IEIE ICEIC in January 2015. This paper has been accepted by the editorial board through the regular review process that confirms the original contribution.*

***Abstract*:** Dalvik is a virtual machine (VM) that is designed to run Java-based Android applications. A trace-based just-in-time (JIT) compilation technique is currently employed to improve performance of the Dalvik VM. However, due to runtime compilation overhead, the trace-based JIT compiler provides only a few simple optimizations. Moreover, because each trace contains only a few instructions, the trace-based JIT compiler inherently exploits fewer optimization and parallelization opportunities than a method-based JIT compiler that compiles method-by-method. So we propose a new method-based JIT compiler, named DEX2C, in order to improve performance by finding more opportunities for both optimization and parallelization in Android applications. We employ C code as an intermediate product in order to find more optimization opportunities by using the GNU C Compiler (GCC), and we will detect parallelism by using the Intel C/C++ parallel compiler and the AESOP compiler in our future work. In this paper, we introduce our DEX2C compiler, which dynamically translates Dalvik bytecodes (DEX) into C code with method granularity. We also describe a new method-based JIT interface in the Dalvik VM for the DEX2C compiler. Our experiment results show that our compiler and its interface achieve significant performance improvement by up to 15.2 times and 3.7 times on average, in Element Benchmark, and up to 2.8 times for FFT in Smartbench.

***Keywords*:** Dalvik, Bytecodes, JIT compiler, DEX2C compiler

1. Introduction

Dalvik [1] is a virtual machine (VM) that executes Android applications under the Android runtime environment. Since Dalvik VM's runtime performance is intrinsically restricted due to interpretation overhead, a trace-based just-in-time (JIT) compiler [2, 3] has been employed to improve performance by dynamically compiling hot traces into native code and directly executing them. However, the trace-based JIT compiler is basically designed to apply very restricted optimizations, such as constant propagation, redundant load/store elimination, and so on, to each of the hot traces in order to alleviate dynamic compilation overhead. Moreover, because each trace contains only a few instructions, the trace-based JIT compiler has few opportunities for optimization and parallelization against a method-based JIT compiler that handles all the instructions in a method.

In this paper, in order to improve the performance of applications under the Dalvik VM, we propose a new method-based JIT compiler named DEX2C. Also, we employ C as an intermediate product of the method-based JIT compilation in order to employ additional optimizations of the GNU C Compiler (GCC), and we will find parallelism in the C code by using the Intel C/C++ compiler [4] and the AESOP compiler [5] in our future work. Through this approach, we are able to know which optimizations are needed and which codes are supposed to

be parallelized in Android applications.

Our performance evaluation shows that the DEX2C compiler achieves significant performance improvement of up to 15.2 times and 3.7 times, on average, in Element Benchmark. Also we achieve performance improvement of up to 2.8 times under Smartbench.

Our paper is organized as follows. Section 2 describes the DEX2C compiler infrastructure, and Section 3 evaluates the performance of our DEX2C architecture. Finally, we offer a conclusion in Section 4.

2. DEX2C Compiler Infrastructure

In this section, we introduce both the overall architecture of our DEX2C compiler and its interface. Also we describe the detailed structure of the compiler and the interface.

2.1 Overall Architecture

Fig. 1 shows the overall architecture of the DEX2C compiler infrastructure and its overall compilation flow. The infrastructure is composed of the following two compilers: the DEX2C compiler for translating the DEX code of a method into C, and GCC for generating an executable object file with the generated C code. When one of the hot-methods, detected in the profiling phase, is first invoked by the Dalvik VM, the DEX2C compiler performs DEX-to-C translation. After the translation completes, GCC compiles the generated C code into an object file. Finally, the Dalvik VM dynamically links the object file and executes the target method, the native code of the object file, instead of interpreting its DEX code.

2.2 DEX2C Compiler Structure

The DEX2C compiler performs the following sequence: the compiler frontend analyzes method information, such as method signature, local variables, DEX code, and so on. After the analysis, the compiler frontend builds an intermediate representation (IR) that

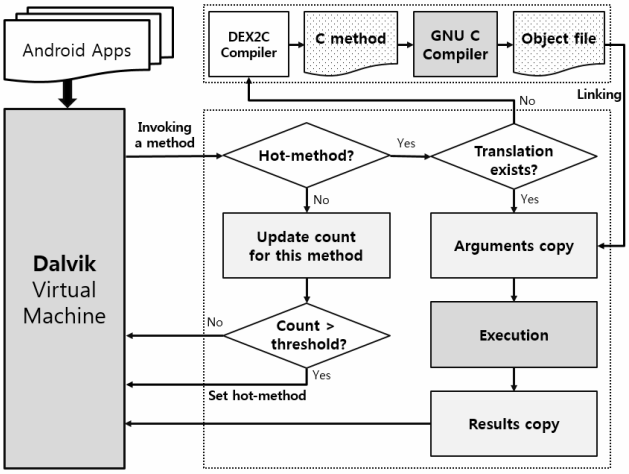

Fig. 1. Overall architecture of the DEX2C compiler.

basically contains both a symbol table and a control flow graph with separated basic blocks by branch instructions from DEX code. Before building the symbol table, the compiler frontend makes a local variable map that contains pairs of virtual register numbers and data types of the register using information on local variables in the DEX code. Since the DEX code is originally register-based, resolving data types of registers is essential for DEX-to-C translation. Unfortunately, since the data types of the temporary registers used in DEX code are not evidently specified, the compiler frontend performs a global liveness analysis in order to find out the data types. In the global liveness analysis, we employ Def-Use chains. Therefore, unknown data types of the temporary registers are ultimately determined with local variable information and type-specific instructions such as *mul-double*, *int-to-double*, and so on. After the symbol table is constructed with both local and temporary variables, the backend of the DEX2C compiler traverses each basic block in the control flow graph and translates its DEX code into a C IR. The C IR is designed to be nearly equivalent to C language syntax so that it is able to be directly converted to C. Finally, the backend emits an epilog, such as a C-style method signature, including return type, method name, and parameters. It also emits definitions of the variables in the symbol table and the C code directly from the C IR.

2.3 Method-based JIT Interface

In order to execute hot-methods as native code on the Dalvik VM, we implement a method-based JIT interface, as shown in Fig. 1. When a method is invoked, we examine whether it is a hot-method or not. If not, a counter associated with invoking the method is increased. At the next visit, if the counter value meets a predefined threshold, the method is set as a hot-method and is going to take a hot-method path at the next invocation. For the path, we provide two different ways: one updates a counter and examines the counter value with the pre-defined threshold; the other specifies a method to be translated by using a configuration file. In other words, if we already know which methods are hot-methods in applications, we can directly specify the methods to be translated in the configuration file.

When a method takes the hot-method path, we check whether a translation of the method already exists or not. If not, the method is translated into C by the DEX2C compiler. Next, the generated C code is compiled into an object file by the GNU C compiler and linked with runtime libraries [6] that include routines for function calls from the generated C code to the original application source code in the Dalvik VM.

Finally, method arguments are copied from Dalvik's argument stack into the native codes' argument registers and stack. In Fig. 2, Dalvik has method arguments on its stack, whereas the native code has its arguments in registers and a stack. For example, if the number of employed argument registers is four and the number of method arguments is less than four, GCC uses registers for those arguments from the beginning of the registers. If the number of method arguments exceeds four, native code

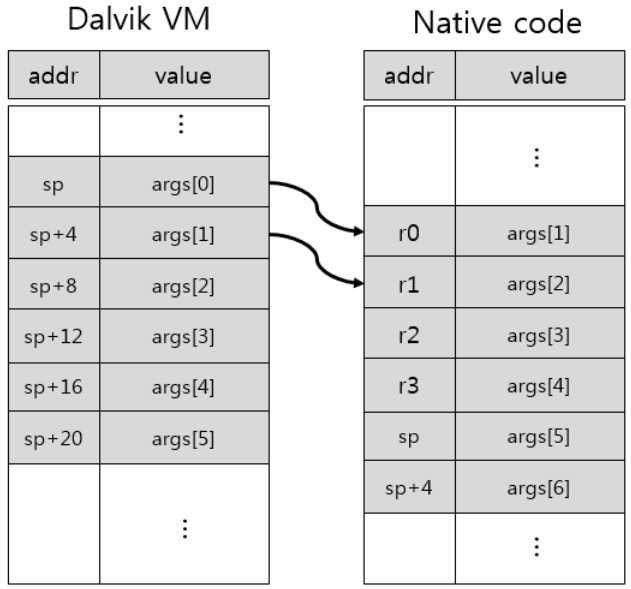

Fig. 2. Argument passing from Dalvik VM to native code.

uses the registers for the first four arguments and the stack for the rest of arguments. Therefore, the method-based JIT interface needs to adjust for the difference between the Dalvik's argument stack and the native code's argument area. Finally, the target method is executed by updating a program counter to the linking address. Also, the return value must be transferred from return registers, *r0* and *r1*, into *retval* registers of the Dalvik VM.

3. Performance Evaluation

In this section, we present the experimental setup for performance evaluation and the performance results of the DEX2C compiler and its interface.

3.1 Experimental Setup

The performance of our DEX2C compiler was tested using Element Benchmark [7] and Smartbench [8] on Nexus 7 [9]. Element Benchmark includes *addition*, *subtraction*, *multiplication*, and *division*. Smartbench contains *Linpack*, *FFT*, *MonteCarlo*, *LU* and *Mandelbrot*. In order to analyze both the performance improvement by executing native code and the overhead due to the method-based JIT interface, we translated only one method of each benchmark into C, which occupies most of the execution time but different function call counts. The generated C code from the benchmarks was compiled into object files by using GCC 4.6.0 with -O2 optimization. Also, in the performance evaluation, dynamic compilation overhead of the DEX2C compiler was ignored by employing pre-compiled object files.

3.2 Evaluation Results

Fig. 3 shows the speedup of our tested benchmarks by using our proposed scheme. We use the existing Dalvik VM with the conventional trace-based JIT compilation as a baseline and normalized all performance results against the baseline.

There are two factors that determined overall performance. One is execution time of the translated method. As explained in Section 3, the method that has the longest execution time was translated into C. Therefore, the higher ratio of the execution time of the translated method to the total execution time implies higher speedup. The other is interaction between bytecodes and the translated C code. As mentioned in Section 2.3, we handled a function call from bytecodes to generated C code by copying arguments and results between Dalvik's area and the native codes' area. Therefore, more frequent interaction between them incurs higher overhead during execution.

As shown in Fig. 3, we could achieve improvement of 3.2 times the speedup in *addition*, 3.1 times in *subtraction*, and 15.2 times in *multiplication*. Different from the other benchmarks, *division* achieved only 1.2 times the speedup. Those optimization opportunities are reduced because the

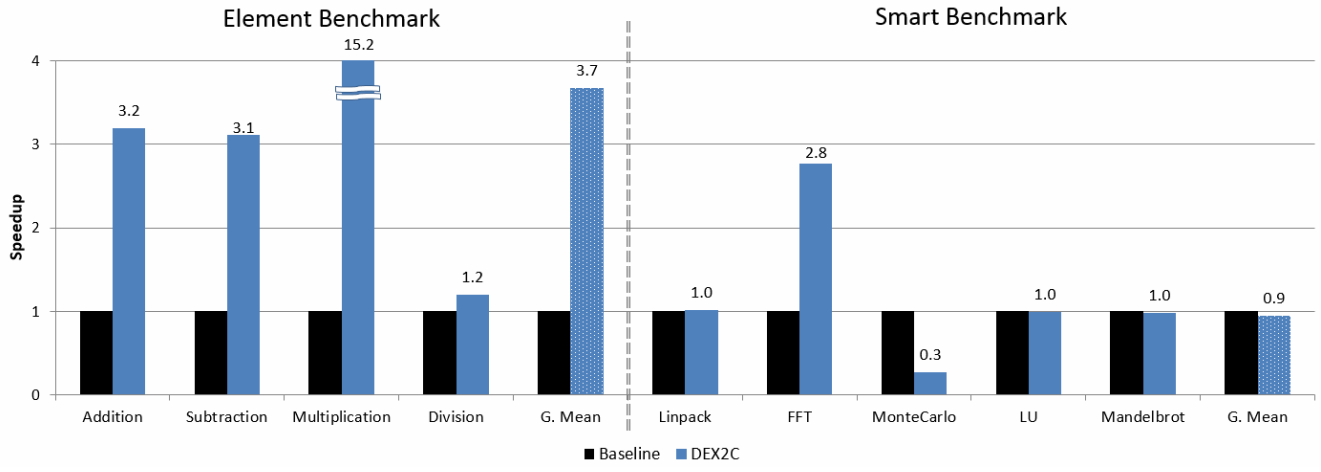

Fig. 3. The speedup of Element Benchmark and Smartbench.

division operation is supported by built-in libraries of Dalvik and GCC. The overhead due to the function call from bytecodes to the generated C code is almost nothing, because each translated method in Element Benchmark invokes once over the entire execution phase. Also, we could achieve remarkable performance improvement of 2.8 times in *FFT* only in Smartbench. *Linpack*, *LU*, and *Mandelbrot* showed almost the same performance as the conventional Dalvik VM. Unfortunately, *MonteCarlo* represents considerable slowdown.

In Element Benchmark, a method that is translated into C takes 95% of the total execution time in each benchmark. As a result, we achieve a significant performance improvement of 3.7 times on average. Similarly, in *FFT*, the *transform_internal* method that occupies 95% of the total execution time is translated into C. In addition, because there is no function call in *FFT*, there is no function call overhead. Therefore, we could achieve improvement of 2.8 times the performance. However, in *Linpack*, *LU*, and *Mandelbrot* benchmarks, the translated method occupies about 70% of the total execution time. In addition, interaction occurs occasionally. Therefore, the performance improvement from executing native code and the performance slowdown from its interface overhead are countervailed. As a result, we achieved almost the same performance as baseline.

In *MonteCarlo*, the *integrate* method occupying only 44% time of the total execution time was translated into C. In addition, a function call that causes performance slowdown appears frequently. Consequently, we observed a slowdown of 0.3 times the performance. To summarize, in Smartbench, we get somewhat different performance compared to Element Benchmark because of the characteristics of the benchmarks.

4. Conclusion

To the best of our knowledge, our work is the first attempt at implementing a method-based JIT compiler and an interface that supports DEX-to-C translation and its execution framework. The performance results show that our DEX2C compiler and its interface achieve reasonable performance improvement against the existing trace-based JIT compiler under the Dalvik VM.

In future work, in order to find out what optimization can be applied to Android applications, we are going to evaluate performance with various optimization options of the GCC using our DEX2C infrastructure. We will also study how to parallelize Android applications with parallelism opportunities found by the Intel C/C++ compiler and the AESOP compiler through the generated C code of the DEX2C compiler.

References

[1] Google, “Dalvik VM Internals,” https://sites.google.com/site/io/dalvik-vm-internals, 2008. Article (CrossRef Link)

[2] B. Cheng and B. Buzbee. “A JIT compiler for Android’s Dalvik VM,” http://dl.google.com/googleio/2010/android-jit-compiler-androids-dalvik-vm.pdf, 2010. Article (CrossRef Link)

[3] B. Stefan, “Analysis of the Android Architecture,” http://os.itec.kit.edu/downloads/sa_2010_braehler-stefan_android-architecture.pdf, 2010. Article (CrossRef Link)

[4] Intel, “Intel C and C++ Compilers,” *https://software.intel.com/en-us/c-compilers*, 2014. Article (CrossRef Link)

[5] A. Kotha, T. Creech and R. Barua, “AESOP: The Autoparallelizing Computer for Shared Memory Computers,” http://aesop.ece.umd.edu, 2013. Article (CrossRef Link)

[6] Y. Han, S. Kim, H. Kim, S. J. Hwang and S.W. Kim, “Code Generation and Optimization for Java-to-C Compilers,” Emerging Directions in Embedded and Ubiquitous Computing Lecture Notes in Computer Science Volume 4097, 2006, pp 785-794. Article (CrossRef Link)

[7] K. Kodama, “Element Benchmark,” https://sites.google.com/site/elementbenchmark, 2012. Article (CrossRef Link)

[8] 123 Smartmobile, “Smartbench 2012,” http://123smartmobile.com, 2012. Article (CrossRef Link)

[9] Google, “Galaxy nexus,” http://www.google.com/nexus/, 2014. Article (CrossRef Link)

[10] J. Dongarra, P. Luszczek, A. Petitet, “The LINPACK benchmark: Past, present, and future,” Concurrency and Computation: Practice and Experience Volume 15, 2003, pp 803-820. Article (CrossRef Link)