저작자표시-비영리-변경금지 2.0 대한민국

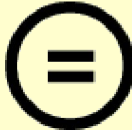
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학박사 학위논문

Hybrid Java Compilation of Just-in-Time Compilation and Ahead-of-Time Compilation for Embedded Systems

내장형 시스템에서 Just-in-Time 및 Ahead-of-Time 컴파일을 활용한 하이브리드 자바 컴파일

2015 년 8 월

서울대학교 대학원

전기컴퓨터공학부

오 형 석

Hybrid Java Compilation of Just-in-Time Compilation and Ahead-of-Time Compilation for Embedded Systems

지도 교수 문 수 묵

이 논문을 공학박사 학위논문으로 제출함
2015 년 7월

서울대학교 대학원
전기컴퓨터공학부
오 형 석

오형석의 공학박사 학위논문을 인준함
2015 년 7월

위 원 장 ________ 백 윤 흥 ________ (인)

부위원장 ________ 문 수 묵 ________ (인)

위 원 ________ 이 혁 재 ________ (인)

위 원 ________ 이 재 진 ________ (인)

위 원 ________ 정 동 헌 ________ (인)

Abstract

Hybrid Java Compilation of Just-in-Time Compilation and Ahead-of-Time Compilation for Embedded Systems

Hyeong-Seok Oh
School of Electrical Engineering and Computer Science
The Graduate School
Seoul National University

Many embedded Java software platforms execute two types of Java classes: those installed statically on the client device and those downloaded dynamically from service providers at runtime. For higher performance, it would be desirable to compile static Java classes by ahead-of-time compiler (AOTC) and to handle dynamically downloaded classes by just-in-time compiler (JITC), providing a hybrid compilation environment. We propose a hybrid Java compilation approach and performs an initial case study with a hybrid environment, which is constructed simply by merging an existing AOTC and a JITC for the same JVM. Contrary to our expectations, the hybrid environment does not deliver a performance, in-between of full-JITC's and full-AOTC's. In fact, its performance is even lower than full-JITC's for many benchmarks. We analyzed the result and found that a naive merge of JITC and AOTC may result in inefficiencies, especially due to calls between JITC methods and AOTC methods. We also observed that

the distribution of JITC methods and AOTC methods is also important, and experimented with various distributions to understand when a hybrid environment can deliver a desired performance.

The Android Java is to be executed by the Dalvik virtual machine (VM), which is quite different from the traditional Java VM such as Oracle's HotSpot VM. That is, Dalvik employs register-based bytecode while HotSpot employs stack-based bytecode, requiring a different way of interpretation. Also, Dalvik uses trace-based just-in-time compilation (JITC), while HotSpot uses method-based JITC. Therefore, it is questioned how the Dalvik VM performs compared the HotSpot VM. Unfortunately, there has been little comparative evaluation of both VMs, so the performance of the Dalvik VM is not well understood. More importantly, it is also not well understood how the performance of the Dalvik VM affects the overall performance of the Android applications (apps). We make an attempt to evaluate the Dalvik VM. We install both VMs on the same board and compare the performance using EEMBC benchmark. In the JITC mode, Dakvik is slower than HotSpot by more than 2.9 times and its generated code size is not smaller than HotSpot's due to its worse code quality and trace-chaining code. We also investigated how real Android apps are different from Java benchmarks, to understand why the slow Dalvik VM does not affect the performance of the Android apps seriously.

We 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 correct 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 65%. When we compare with the recently-introduced ART, which also performs ahead-of-time compilation, our AOTC performs better.

We cannot AOTC all middleware and framework methods in DTV and android device for hybrid compilation. By case study on DTV, we found that we need to adopt AOTC enough methods and reduce method call overhead. We propose AOTC method selection heuristic using method call chain. We select hot methods and call chain methods using profile data. Our heuristic based on method call chain get better performance than other heuristics.

Keywords : Hybrid compilation, JITC, AOTC, Java, Dalvik
Student Number : 2008-30228

Contents

Chapter 1 Introduction 1
1.1 The need of hybrid compilation 1
1.2 Outline of the Dissertation 2

Chapter 2 Hybrid Compilation for Java Virtual Machine 3
2.1 The Approach of Hybrid Compilation 3
2.2 The JITC and AOTC..6
2.2.1 JVM and the Interpreter 7
2.2.2 The JITC..8
2.2.3 The AOTC 9
2.3 Hybrid Compilation Environment...11
2.4 Analysis of the Hybrid Environment 14
2.4.1 Call Behavior of Benchmarks 14
2.4.2 Call Overhead 15
2.4.3 Application Methods and Library Methods 18
2.4.4 Improving hybrid performance..20
2.4.4.1 Reducing the JITC-to-AOTC call overhead.............20
2.4.4.2 Performance impact of the distribution of JITC methods and AOTC methods...21

Chapter 3 Evaluation of Dalvik Virtual Machine 23
3.1 Android Platform 23
3.2 Java VM and Dalvik VM...24
3.2.1 Bytecode ISA 25
3.2.3 Just-in-Time Compilation (JITC) 27
3.3 Experimental Results 32
3.3.1 Experimental Environment 33
3.3.2 Interpreter Performance...34
3.3.3 JITC Performance 37
3.3.4 Trace Extension..43
3.4 Behavior of Real Android Apps..44

Chapter 4 Ahead-of-Time Compilation for Dalvik Virtual Machine51
4.1 Android and Dalvik VM Execution..51
4.1.1 Android Execution Model 51
4.1.2 Dalvik VM 51
4.1.3 Dexopt and JITC in the Dalvik VM....................................53
4.2 AOTC Architecture..54
4.3 Design and Implementation of AOTC 56
4.3.1 Dexopt and Code Generation...56
4.3.2 C Code Generation 56

4.3.3 AOTC Method Call 58
4.3.4 Garbage Collection 61
4.3.5 Exception Handling 62
4.3.6 AOTC Method Linking 63
4.4 AOTC Code Optimization 64
4.4.1 Method Inlining 64
4.4.2 Spill Optimization 64
4.4.3 Elimination of Redundant Code 65
4.5 Experimental Result 65
4.5.1 Experimental Environment 66
4.5.2 AOTC Target Methods 66
4.5.3 Performance Impact of AOTC 67
4.5.4 DVM AOTC vs. ART 68

Chapter 5 Selecting Ahead-of-Time Compilation Target Methods for Hybrid Compilation 70
5.1 Hybrid Compilation on DTV 70
5.2 Hybrid Compilation on Android Device 72
5.3 AOTC for Hybrid Compilation 74
5.3.1 AOTC Target Methods 74
5.3.2 Case Study: Selecting on DTV 75
5.4 Method Selection Using Call Chain 77
5.5 Experimental Result 77
5.5.1 Experimental Environment 78
5.5.2 Performance Impact 79

Chapter 6 Related Works 81

Chapter 7 Conclusion 84

Bibliography 86

List of Table

Table 3-1 Redundancy ratio of the Dalvik JITC 40
Table 3-2 Android applications experimented 46
Table 5-1 Number of selected methods and different methods with call chain heuristic 80

List of Figure

Figure 2-1 Performance of the full-JITC, the full-AOTC, and the hybrid with AOTC and JITC, compared to the interpreter.............5
Figure 2-2 AOTC method call from interpreter in MIPS.............13
Figure 2-3 Call behavior of each benchmark 15
Figure 2-4 Call count distribution among different type of calls .16
Figure 2-5 The call overhead of invokestatic and invokevirtual .17
Figure 2-6 Performance of hybrid environment, faster hybrid environment, and full-JITC 17
Figure 2-7 Performance of another hybrid environment that AOTC applications and JITC libraries..19
Figure 2-8 Performance of mixed argument passing in AOTC....20
Figure 2-9 Performance of hybrid compilation with different T values for the EEMBC benchmarks 22
Figure 3-1 Java program, Java bytecode, and Dalvik bytecode ..25
Figure 3-2 Dalvik VM JITC process...28
Figure 3-3 An example CFG of BBs and CFG of traces 30
Figure 3-4 Machine code generation by DVM and JVM JITCs....32
Figure 3-5 Interpreter performance ratio 35
Figure 3-6 Static bytecode size and bytecode count ratio...........36
Figure 3-7 Dynamic bytecode count ratio.....................................36
Figure 3-8 Dynamic bytecode size ratio 37
Figure 3-9 VM performance with JIT compiler 38
Figure 3-10 Compiled bytecode size ratio 39
Figure 3-11 Generated machine code size ratio...........................39
Figure 3-12 Generated machine instruction count ratio 40
Figure 3-13 Generated instruction count per bytecode's 1 byte 41
Figure 3-14 Generated machine code size per bytecode's 1 byte 42
Figure 3-15 Compile time ratio..43
Figure 3-16 Compilation overhead over the total running time...43
Figure 3-17 Performance impact of expanding the trace............44
Figure 4-1 Java program code and Dalvik bytecode 52
Figure 4-2 Interpreter stack and argument passing in Dalvik VM 53
Figure 4-3 Quickening result by dexopt.......................................53
Figure 4-4 Build Dalvik VM with AOTC..54
Figure 4-5 Sharing zygote AOTC methods by app processes.....56
Figure 4-6. Translated C code for the example in Figure 4-1 ...57
Figure 4-7 Method call using a mixed call interface 60
Figure 4-8 Class loading and AOTC method linking.....................64
Figure 4-9 Dalvik AOTC performance in benchmark 67

Figure 4-10 Performance impact of method inlining 68
Figure 4-11 Performance impact of optimizations........................68
Figure 4-12 Comparison of Dalvik AOTC with ART 69
Figure 5-1 The architecture of DTV hybrid execution................70
Figure 5-2 Hybrid using hot methods AOTC performance on DTV 76
Figure 5-3 Method call graph and method selection 77
Figure 5-4 Hybrid performance in DTV xlet application 79
Figure 5-5 Hybrid performance in android app loading................79
Figure 5-6 JITC-to-AOTC and AOTC-to-JITC call count in android app loading 81

Chapter 1 Introduction

1.1 The need of hybrid compilation

Many embedded devices support service downloading system for device user by software platform. For example, smartphone have the features as a cell phone and support variable service using apps installed by device user. These downloading systems are used on tablet PC, digital TV, blu-ray discs.

Java is a popular software platform for these downloading service platforms. This is mainly due to its advantage in platform independence, security, and rich APIs for software development by using Java Virtual Machine (JVM). That is, the use of a virtual machine allows a consistent runtime e environment for diverse client devices that have different CPUs, OS, and hardware components. Moreover, Java has little security issues such that it is extremely difficult for malicious Java code to break down a whole system. Finally, it is much easier to develop software with Java due to its sufficient, mature APIs and its robust language features such as exception handling and garbage collection.

JVM executes Java' s compiled executable called the bytecode. The bytecode is a stack-based instruction set which can be executed by an interpreter on any platform without porting the original source code. Since this software-based execution is obviously much slower than hardware-based execution, compilation techniques for translating bytecode into machine code have been used, such as just-in-time compilers (JITC) [1] and ahead-of-time compilers (AOTC) [2-5]. JITC performs an online translation on the client device at runtime, while AOTC performs an offline translation on the server before runtime and the translated machine code is installed on the client device.

Generally, AOTC is more advantageous in embedded Java systems since it obviates the runtime translation overhead of JITC, which would waste the limited computing power and runtime memory of embedded systems, and may affect the real time

behavior of the client devices. On the other hand, download classes dynamically at runtime cannot be handled by AOTC and thus, should be executed by the interpreter. However, the performance benefit achieved by AOTC can be easily offset by such interpretive execution. So it would be desirable to employ JITC as well to handle dynamically loaded classes for complementing AOTC. That is, we need a hybrid compilation for downloading system.

We actually constructed such a hybrid compilation by merging an AOTC [8,9,15] and a JITC [14], each of which takes the most generally-accepted approach of compilation. And we adjust this hybrid compilation for commercial digital TV (DTV) and android device. We can adopt AOTC for statically installed Java class in client device. However, there are too many Java methods device for AOTC. This makes binary size which generated by AOTC too huge. And sometimes we cannot AOTC all methods by the limitation of build environment. So we need to select AOTC target methods efficiently.

1.2 Outline of the Dissertation

The rest of this paper is organized as follows. Chapter 2 introduces Java platform and analysis hybrid compilation by naïve merging of JITC and AOTC. In chapter 3, we evaluate Dalvik virtual machine by comparing with Java Virtual Machine and analysis execution of real apps. In chapter 4, AOTC technique for Dalvik virtual machine is proposed and evaluate. In chapter 5, we suggest heuristic for AOTC target method selection based on method call chain. A summary is follows in chapter 6.

Chapter 2 Hybrid Compilation for Java Virtual Machine

2.1. The Approach of Hybrid Compilation

Our proposed hybrid compilation environment targets an embedded Java software platform which can download classes at runtime. For example, a software platform for digital TVs (DTV) is typically composed of two components: a Java middleware called OCAP or ACAP (and Java system classes) which are statically installed on the DTV set-top box, and Java classes called xlets which are dynamically downloaded thru the cable line or the antenna. Also, a software platform for mobile phones is composed of the MIDP middleware on the phone and midlets downloaded via the wireless network. Blu-ray disks consist of the BD-J middleware on the BD player and xlets on the BD titles. We believe these dual-component systems will be a mainstream trend for embedded Java software architecture.

Another trend of these dual-component systems is that both the Java middleware and the downloaded classes become more complex and substantial. The initial downloaded Java classes were mainly for displaying idle screen images or for delivering simple contents, but now more substantial Java classes such as games or interactive information that take a longer execution time are being downloaded. In order to reduce the network bandwidth (wired or wireless) for downloading, the Java middleware also gets more substantial to absorb the size and the complexity of downloaded classes. In mobile phones, for example, the first MIDP middleware provided libraries for user interfaces only, yet its successor middleware called the JTWI provided an integrated library for the music players and the SMS as well. Now a more substantial middleware called the MSA with more features is being introduced.

For achieving high performance on these substantial, dual-component systems, it would be desirable to employ hybrid

acceleration such that the Java middleware is compiled by AOTC while the downloaded classes are handled by JITC. However, our point is that a naïve merge of an existing AOTC and a JITC would not lead to a performance level that we would normally expect from a hybrid environment. In order to motivate readers, we actually constructed a hybrid environment by merging an AOTC and a JITC, which we developed independently for the same JVM, and then we experimented with it as follows.

Both the AOTC and the JITC targets Oracle' s CDC VM (CVM). The AOTC takes a bytecode-to-C approach such that the bytecode is translated into C code, which is then compiled with the CVM source code using a GNU C compiler. The JITC uses adaptive compilation method, where Java methods are initially executed by the CVM interpreter until they are determined to be hot spots, and then are compiled into native code.

After merging both compilers, we could compile some methods by the AOTC before runtime and compile some methods by the JITC at runtime (which we call AOTC methods and JITC methods, respectively). Calls between JITC methods and AOTC methods are handled appropriately by executing additional code to meet the calling conventions between them; we do not use the JNI (Java native interface) for interoperation of the translated C code and Java code since JNI would be too slow. The details of the JITC, the AOTC, and the hybrid environment will be described in detail in section 2.2 and 2.3.

Based on this hybrid environment, we experimented with a dual-component Java system using conventional Java programs, by compiling the Java system library classes with the AOTC, while handling regular application classes with the JITC. This is assuming that the system classes correspond to the middleware on the client device, while the application classes correspond to downloaded classes from the service provider. We thought that this experimental setup is a reasonable simulation of our target embedded environment since the interaction between system classes and application classes would exhibit a similar behavior to

the interaction between middleware classes and downloaded classes (unfortunately, this is not exactly true, as we will see later).

We compare the performance of this hybrid environment (hybrid) with the performances of the full-AOTC and the full-JITC environments, where both library classes and application classes are handled solely by the AOTC and by the JITC, respectively. In this way, we can evaluate the constructed hybrid environment.

Our experimental environment is as follows. The experiments were performed with the AOTC and the JITC implemented on CVM reference implementation (RI). Our CPU is a MIPS-based SoC called AMD Xilleon which is popularly employed in Digital TVs. The MIPS CPU has a clock speed of 300MHz and has a 16KB I-cache/16KB D-cache, with a 128MB main memory. The OS is an Embedded Linux. The benchmarks are SPECjvm98 (except for javac) and EEMBC.

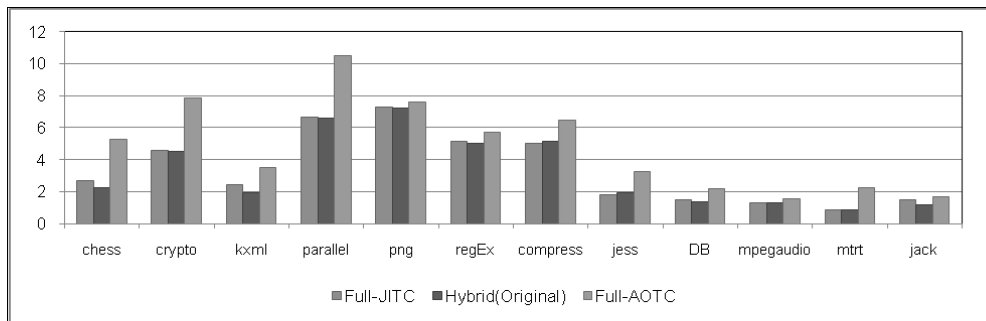


Figure 2-1 Performance of the full-JITC, the full-AOTC, and the hybrid with AOTC and JITC, compared to the interpreter (1.0x)

Figure 2-1 shows the performance ratio of the full-JITC, the hybrid, and the full-AOTC, compared to the performance of interpreter execution (full-interpreter) as 1.0x for each benchmark. The performance of the full-AOTC, which is an average of 4.0x over the interpreter, is consistently higher than the performance of the full-JITC, which is an average of 2.7x over the interpreter, as we expected. The problem is the performance of the hybrid. Contrary to our expectation, its performance is not positioned in-between of the full-JITC' s and the full-AOTC' s, and even lower

than the full-JITC' s in many benchmarks. This is somewhat surprising since we expected a performance at least better than the full-JITC' s, by handling library methods using the more powerful AOTC while handling others by the same JITC used in the full-JITC environment.

We may speculate on the reasons for this performance anomaly as follows. One possibility would be the call overhead between AOTC methods and JITC (interpreter) methods due to additional code executed for meeting the calling conventions. Or, it might be related to JVM features such as garbage collection (GC) or exception handling (EH), which should work correctly even in the hybrid environment (e.g., an exception raised by an AOTC method can be handled by a JITC method, or GC can occur when the call stack is mixed with AOTC methods and JITC methods). This might cause additional overhead in the hybrid environment for their correct operation. Finally, it might be related to the characteristics of library classes such that their performance would not increase when they compiled with AOTC, compared to when compiled with JITC.

In order to find out the major reason(s) for the performance anomaly of our hybrid environment, we first need to understand how the AOTC, the JITC, the interpreter, and the hybrid environment are constructed. We will describe them in the next two sections, especially focusing on the issues raised above.

2.2. The JITC and AOTC

Although our AOTC and JITC target the same CVM, they have been developed independently without any consideration of hybrid execution (in fact, no JITC or AOTC has been developed considering hybrid execution, as far as we know, and it is not clear at this point how to build such a hybrid-execution-aware JITC and AOTC, as will be discussed later). On the other hand, each compiler was developed as reasonably and generally as possible for its best performance benefit, so although the experimental results in section

2.1 were obtained with our specific implementation, a similar result is likely to be expected with other implementations. This section describes both compilers, especially focusing on their calling conventions, optimizations, and how GC and EH are handled. We start with an overview of the JVM and the CVM interpreter.

2.2.1 JVM and the Interpreter

These become local variables of the callee, and are followed by the callee’s other local variables, method/frame information of the callee, and the callee’s operand stack. When a method returns, the callee’s stack frame is popped and the return value is copied from the callee’s operand stack to the top of the caller’s operand stack. As a GC-based, object-oriented language, garbage objects are reclaimed automatically. GC requires tracing all reachable objects from the root set to reclaim all unreachable objects. The root set is composed of operand stack slots and local variables of all methods in the call stack (Java stack) and static variables, whose types are object references.

The CVM requires all threads to wait at their GC-point before it performs GC [6], which is a point in the program where GC can possibly occur. Examples of GC-points include memory allocation requests, method calls, loop backedges, etc. So, a thread should check if there is any pending GC request whenever they pass through a GC-point, and wait there if there is one. When all threads wait at their GC-point, the CVM can start GC by first computing the root set. For this computation, GC needs a data structure describing the location of each root at the GC-point, which is called a GC-map. When GC occurs during interpretation, the interpreter is supposed to analyze the bytecode for each method in the call stack and to compute the GC-map at each GC-point in the method (it saves the GC-maps at the method block for their reuse when GC occurs again). When GC occurs, this GC-map is consulted to decide which stack slots and local variables in the call stack are reference-typed thus being included in the root set. Reference-type static variables

are already included in the root set.

Java supports EH such that when an error occurs in a try block, the error is caught and handled by one of subsequent catch blocks associated with the try block. One problem is that the exception throwing try block and the exception-handling catch block might be located in different methods on the call stack, so if no catch block in the method where the exception is thrown can handle the exception, the CVM searches backward through the call stack to find a catch block which can handle it. This mechanism is called stack unwinding [7] and is performed by the exception handler routine included in the CVM interpreter [6].

2.2.2. The JITC

Our JITC uses adaptive compilation, where a method is initially executed by the CVM interpreter until it is determined as a hot spot method. Then, the method is compiled into native code, which then resides in the memory and is re-used whenever the method is called again thereafter. Our JITC performs many traditional optimizations for the compiled method including method inlining. The operand stack slots and the local variables are allocated to registers, with copies corresponding to pushes and pops being coalesced aggressively.

As to the JITC calling convention, all machine registers mapped to the operand stack locations and the local variables at the time of a method call are first spilled to the Java stack (to their mapped locations) before the call is made. Consequently, the Java stack is maintained exactly the same as in the case of interpreter execution during method calls. This is for simplifying argument passing for calls between interpreted methods and JITC methods. Moreover, GC and EH can be handled more easily with this calling convention, as will be explained shortly.

As to GC, unlike in the interpreted methods, there is no GC-time computation of the GC-map in the JITC methods; instead the JITC itself computes and saves a GC-map at each GC-point during

translation by checking which Java stack locations (not registers) have a reference at that GC-point. This is so since all registers mapped to the Java stack locations are also spilled to the Java stack at a GC-point if there is a pending GC request (exactly as in method calls; actually, a method call itself is a GC-point). So the GC-map in JITC methods includes only Java stack locations as in the interpreted method and the Java stack is maintained the same when GC occurs.

As to EH, the exception handling routine in the CVM interpreter is supposed to handle exceptions even for JITC methods such that if an exception occurs in a JITC method, it will jump to the handling routine which will perform stack unwinding. When a catch block is found in a JITC method, the bytecode of the catch block will be executed by the interpreter, so there is no need to compile the catch block by the JITC. This is fine since an exception would be an "exceptional" event, so the performance advantage of executing compiled catch blocks would be little. This interpreter-based execution of catch blocks requires the Java stack to be maintained exactly the same as in the interpreter execution, so registers are also spilled to the Java stack when an exception occurs even in a JITC method.

2.2.3 The AOTC

Our AOTC translates the bytecode of classes into C code, all of which are then compiled and linked together with the CVM source code using gcc to generate a new CVM executable [17]. We took this particular approach of AOTC rather than other alternatives considering a few aspects, as explained below.

We took the approach of bytecode-to-C [5] rather than bytecode-to-native [4], since we can resort to an existing compiler for native code generation, which allows a faster time-to-market and a better portability. In addition, we can generate high-quality code by using full optimizations of gcc, which would be more reliable and powerful than our own optimizer. In fact, most AOTCs

take the approach of bytecode-to-C [2,3,16,17], including commercial ones such as Jamaica [18], IBM WebSphere Real-time VM [19], PERC [20], and Fiji [21], so we believe that anyone who wants to build a hybrid environment is likely to employ the bytecode-to-C AOTC. Our AOTC also performs some Java-specific optimizations that gcc cannot handle, such as elimination of redundant null pointer checks or array bound checks [15].

We statically compile and link every translated C code with the CVM source code, instead of compiling each C code separately and loading its machine code to the CVM dynamically at runtime. This allows method calls or field accesses to different classes to be resolved at translation time, obviating runtime resolution. Also, inlining between different classes is much easier.

In our AOTC, each local variable and operand stack slot is translated into a C variable with a type name attached. For example, a reference-type operand stack slot 0 is translated to s0_ref. We then translate each bytecode to a corresponding C statement, while keeping track of the operand stack pointer. For example, aload_1 which pushes a reference-type local variable 1 onto the stack is translated into a C statement s0_ref=l1_ref; if the current stack pointer is zero when this bytecode is translated.

The calling convention of our AOTC follows the format of a regular C function call. That is, a method call in the bytecode is translated into a C function call whose name is composed of the Java class name and the method name (similar to a JNI method naming convention). The argument list consists of an environment variable for capturing the CVM state, followed by regular C variables corresponding to the argument stack locations at the time of the call. Such a C function call will be compiled and optimized by gcc and arguments will be passed via registers or the C stack, so AOTC calls will be much faster than JITC calls or interpreter calls. Our AOTC also performs inlining for some method calls.

As to GC, since our AOTC translates stack slots or local variables that have root references into C variables, it is difficult to know where gcc will place those variables in the final machine code. So

the AOTC cannot make a GC-map. Our solution is generating additional C code that saves references in the Java stack frame whenever a reference-type C variable is updated such that when GC occurs, all Java stack slots of AOTC methods constitute a root set8. For this purpose, a Java stack frame is still allocated and extended during the execution of AOTC methods, although the machine code of AOTC methods (including calls) is based only on the C stack and the registers. In order to reduce the runtime overhead caused by the additional C code, we perform optimizations to reduce the number of reference saves in the stack frame and the number of stack extensions.

There is one more issue in GC with the AOTC. Since the CVM employs a moving GC algorithm, objects can be moved during GC. CVM GC is supposed to update the addresses of moved objects for those references saved in the Java stack frame, but not the updated reference C variables. So after GC, we need to copy the addresses from the Java stack frame back to the reference C variables, which require additional C statements.

As to EH, when an exception occurs in an AOTC method, the environment variable will be set appropriately and the control will transfer to a catch block if the method has one that can handle it. If there is no catch block, the method simply returns to the caller. In the caller, we check if an exception occurred in the callee and if so, we try to find an appropriate catch block in the caller. If there is no catch block, then the method also returns and this process repeats until a catch block is found. This means that we need to add an exception check code right after every method call, which would certainly be an overhead. However, it is a simple check and merged with the GC check for copying references back, so the overhead is not serious [9].

2.3. Hybrid Compilation Environment

Previous section described the AOTC and the JITC, each of which has been developed efficiently for its own performance advantage.

This section describes how we merged them to build a hybrid environment. We address how method calls, GC, and EH are made across different environments. Since the interpreter and the JITC have already worked collaboratively together for adaptive compilation, the merge across AOTC and JITC (interpreter) would be the primary issue of a hybrid environment.

When there is a call between a JITC method and an AOTC method, some reconciliation process is needed to meet the calling convention between them, especially for parameter/return value passing and the stack management. Since an AOTC method passes parameters using registers and the C-stack, while a JITC method passes parameters using the Java stack, appropriate conversion between them is required.

When a JITC method calls an AOTC method, a stack frame for the AOTC method is first pushed on the Java stack and is marked as an AOTC frame. Then, parameters in the Java stack of the JITC method are copied into registers (and into the C-stack if there are more than four parameters in MIPS). Finally, a jump is made to the AOTC method entry. This process is depicted in Figure 2-2 and is performed by an assembly function. When an AOTC method returns back to a JITC method, a similar process is needed to copy the return value in a register of the AOTC method to the Java stack of the JITC method.

If there are updates of reference C variables in the AOTC method, the Java stack is extended to accommodate those references for preparation of GC, as explained in Section 2.2.3 (depicted as a reference stack in Figure 2-2). If the AOTC method calls another AOTC method which also updates reference C variables, the Java stack frame is extended again. Consequently, a single stack frame is shared among consecutively-called AOTC methods and is extended for saving references whenever necessary.

When an AOTC method calls a JITC method, a new stack frame is pushed on the Java stack (marked as a JITC frame), and the parameters that are in C variables are copied to the operand stack and a jump is made to the JITC method.

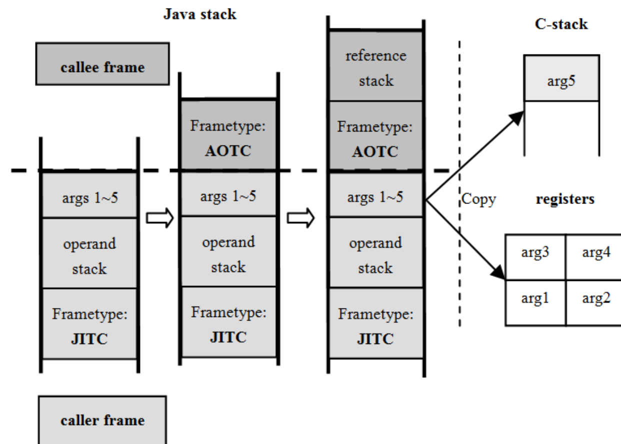


Figure 2-2 AOTC method call from interpreter in MIPS

When GC occurs in a hybrid environment, all the root references are guaranteed to be located in the Java stack, but identifying Java stack slots that contain references depends on the type of the stack frame. If it is an AOTC frame, all stack locations will have references since we saved only references there. If it is a JITC method or an interpreter method, we check with the GC-map at the GC-point to tell the stack slots which have references.

Handling exception across AOTC and JITC methods is relatively simple, which is done with exception checks for both AOTC and JITC methods. When a JITC method calls an AOTC method, there should be an exception check added right after the call, as we did in an AOTC method. If an exception occurs somewhere in the call chain at an AOTC method and if it is not caught before returning to this JITC method, it will be checked at the added exception check code. Then, the control is transferred to the exception handling routine of the interpreter, which performs stack unwinding starting from this JITC method.

When an AOTC method calls a JITC method, we add a check code right after the call as usual. If an exception occurs somewhere in the call chain at a JITC method, the exception handling routine of the interpreter will perform stack unwinding. It checks for each method on the call stack, one by one, if there is a catch block who can handle the exception. It can also tell if a method on the call

stack is a JITC method or an AOTC method using the frame type, so when the caller AOTC method is eventually met during stack unwinding, the interpreter will simply make a call return, which will transfer the control back to the AOTC method as if the JITC method returns. Then, the exception check code is executed and the normal AOTC exception handling mechanism based on the exception check proceeds.

As one can notice easily, supporting EH or GC correctly in the hybrid environment causes a relatively little overhead since they are essentially the same as in AOTC and JITC. On the other hand, method calls may cause some overhead due to different calling conventions. We will analyze method calls in the following section.

2.4. Analysis of the Hybrid Environment

Previous section described our hybrid environment, which we think is a reasonable merge of high-performance JITC and AOTC. In this section, we analyze our performance results in Section 2 to understand the root causes of its performance anomaly.

2.4.1. Call Behavior of Benchmarks

We first examine the call behavior of our benchmarks. Figure 2-3 depicts the distribution of calls and execution time between application methods and library methods in each benchmark. The call distribution varies widely from benchmark to benchmark. Since most calls in crypto, png, regEx, compress, jess, mpegaudio, and mtrt are application method calls, compiling library methods by AOTC in the hybrid would not improve performance much compared to the full-JITC. However, most calls in parallel, DB, and jack and around half of the calls in chess and kxml are library calls, although the library execution time takes a less portion. So we should expect some reasonable performance improvement at least for these benchmarks with the hybrid compared to the full-JITC. Unfortunately, the hybrid led to worse performance in these benchmarks as well as the first set of benchmarks, as we saw

previously in Figure 2-1.

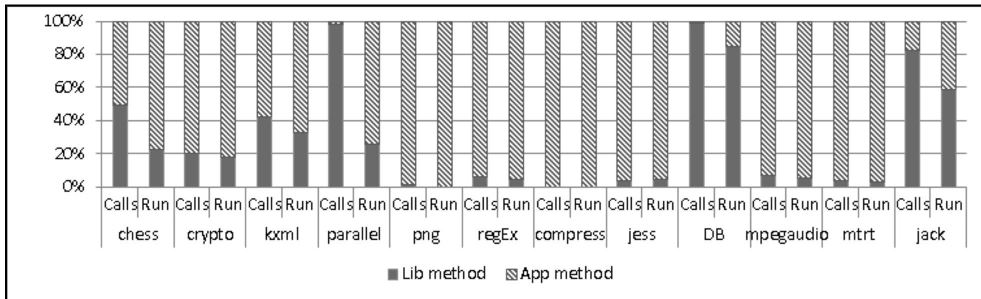


Figure 2-3. Call behavior of each benchmark

Generally, our benchmarks tend to spend more time in applications rather than in libraries, which would not exactly be the case in the middleware and downloaded classes. As the middleware gets more substantial, much of the execution time should be spent in the middleware, as explained in Section 2.1. However, even when the libraries are dominant as in db and jack, Figure 2-1 shows that the hybrid performs worse than the full-JITC. In fact, even when the libraries are not dominant, there should be, in theory, no performance degradation compared to the full-JITC. We suspect the call overhead as one reason, as analyzed below.

2.4.2. Call Overhead

In the hybrid environment, we classified the call types. We measured how many calls are from JITC methods to AOTC methods, from AOTC to AOTC methods, from JITC to JITC methods, and others (e.g., JNI methods calls) which is shown in Figure 2-4. Those benchmarks which include many library calls have many JITC-to-AOTC calls, as expected.

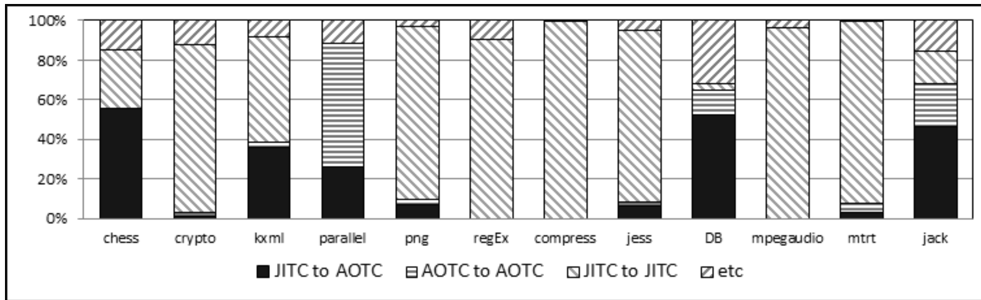


Figure 2-4 Call count distribution among different type of calls

We then measured the call overhead from a JITC method to an AOTC method (J-to-A), which is supposed to occur frequently in our hybrid environment, compared to the call overhead of a JITC method to a JITC method (J-to-J). For this evaluation we made a simple Java method which makes a return. We compiled this method with our AOTC and made a JITC method to call it five million times with a variable number of arguments. We measured the running time and then isolated the loop and argument pushing overhead in order to identify the J-to-A call overhead only. Then, we compiled this method with the JITC and measured the J-to-J call overhead similarly. Finally, we compiled all methods with the AOTC and measured the A-to-A call overhead. We experimented both with a static method call and a virtual method call.

Figure 2-5 depicts the call overhead (in micro seconds) of a single J-to-A call, J-to-J call, and A-to-A call for the static method call and the instance method call. It shows that the call overhead of J-to-A is around 2.5 times to that of J-to-J. It is also shown that the J-to-A call overhead increases as the number of arguments increases, due to argument copying.

We also estimated the total J-to-A call overhead during execution, by multiplying a single J-to-A call overhead and the J-to-A call counts. Its ratio to the whole running time is found to be significant for chess (20%), kxml (23%), db (19%), jack (16%), and parallel (7%) which have many J-to-A calls.

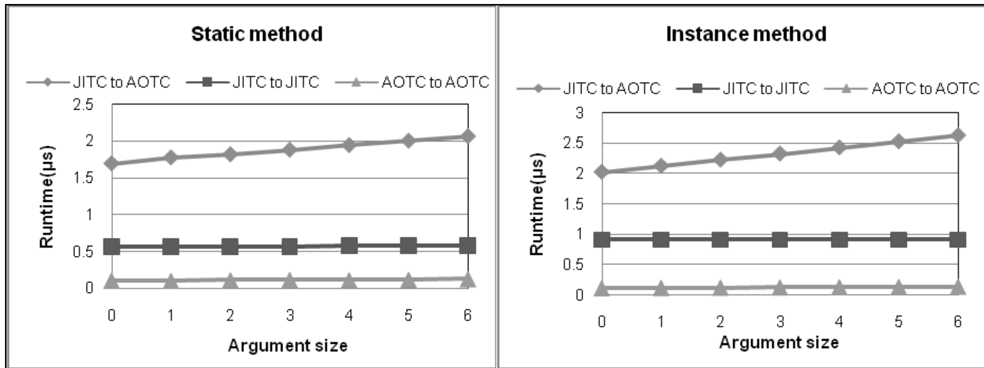


Figure 2-5 The call overhead of invokestatic and invokevirtual

These results indicate that J-to-A calls are much slower than J-to-J calls and they take a significant portion of running time when they are frequent. We estimated the execution time of some hypothetical "faster" hybrid environment when the J-to-A call overhead is replaced by the J-to-J call overhead (i.e., for each J-to-A call, compute the overhead difference from a J-to-J call with the same number of arguments, multiply it by the call count, and subtract the result from the execution time of the original hybrid environment). Figure 2-6 shows the performance of such a faster hybrid environment, compared to those of the original hybrid and the full-JITC. The faster hybrid outperforms the original hybrid tangibly in chess, kxml, parallel, DB, and jack, which have a higher ratio of the total J-to-A call overhead to the running time, and even outperforms the full-JITC in parallel and DB. Consequently, it appears that the J-to-A call overhead significantly contributes to the performance degradation of the original hybrid.

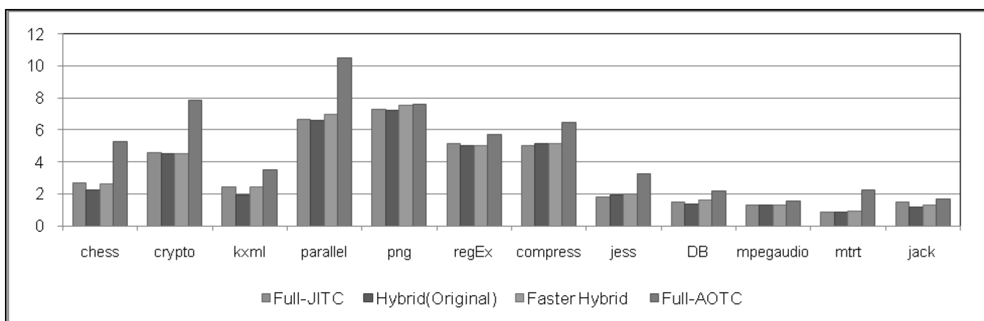


Figure 2-6 Performance of hybrid environment, faster hybrid environment, and full-JITC

The J–to–A call overhead is primarily due to interfacing the AOTC calling convention based on the C stack and the JITC calling convention based on the Java stack. It might be argued that this interface problem would not occur if we perform AOTC using the JITC module, with the full–fledged optimizations that are missing in JITC enabled. This JITC–based AOTC would certainly obviate the interface problem for J–to–A calls, yet there is one important issue in our hybrid environment. That is, even for every A–to–A call, the caller first needs to spill registers to the Java stack so as to keep the Java stack exactly the same as in interpreter execution, as we did with JITC methods. This will certainly slow down A–to–A calls, which are supposed to occur frequently with a substantial Java middleware, and would affect the performance negatively. Another minor issue is that we cannot use the approach of bytecode–to–C anymore, losing many of its advantages described in Section 2.2.3, such as faster time–to–market, portability, and powerful and reliable optimizations with an existing compiler.

2.4.3 Application Methods and Library Methods

We also tried with an opposite hybrid environment where we compiled application methods by the AOTC and handled library methods by the JITC. This is for understanding the characteristics of library methods and application methods in terms of their profitability achievable by AOTC compared to by JITC.

Figure 2–7 shows the performance of a new hybrid environment compared to the full–JITC and the full–AOTC. This time this hybrid environment exhibits a performance level in–between of the full–JITC's and the full–AOTC's for crypto, parallel, compress, jess, mpegaudio, and mtrt, where application methods are dominant (application method calls in parallel are scarce, yet the execution time of application methods takes more than 70%, as seen in Figure 2–3). For other benchmarks where application methods are not dominant, the hybrid performance is still lower than the full–JITC's. This also appears to be due to the call overhead from

AOTC methods to JITC methods, which is even higher than JITC-to-AOTC calls.

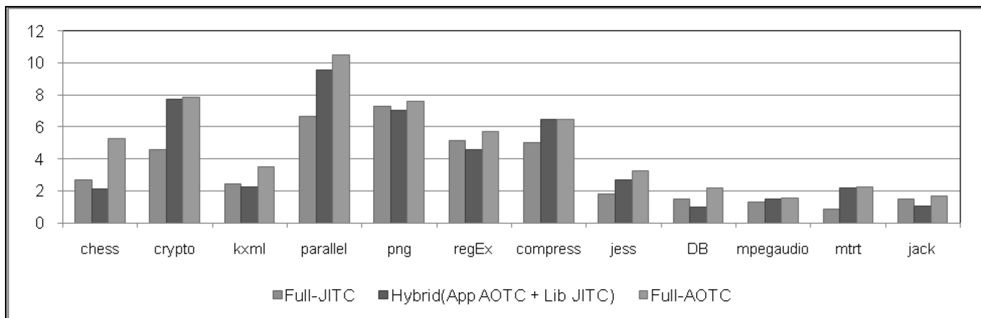


Figure 2-7 Performance of another hybrid environment that AOTC applications and JITC libraries

It is questioned why the hybrid environment can improve the performance of the full-JITC when compiling applications by AOTC for application-dominant benchmarks, while it cannot when compiling libraries by AOTC for library-dominant benchmarks. One possible reason is that hot application methods often include computation loops, which will be better optimized when compiled by AOTC than by JITC, since AOTC includes more powerful optimizations. On the other hand, hot library methods tend to have no computation loops but are called many times, so the benefit of compiling them by AOTC is easily offset by the J-to-A call overhead. For example, the hot library methods in db are called frequently but they either have no loops or have a loop which calls many methods, so the benefit of AOTC for them would be small.

Consequently, the characteristics of methods are also important in deciding whether their performance could be improved when compiled with AOTC compared to when compiled with JITC. This can be used effectively in choosing AOTC candidates. In fact, the real characteristics of the middleware and downloaded classes might be different from those of the library and the application used in our experiment, so we need to investigate the real cases further.

2.4.4. Improving hybrid performance

Previous section analyzed the performance problems of the hybrid environment. In this section, we investigate how we could possibly achieve the desired hybrid performance. We first describe how to reduce the JITC-to-AOTC call overhead. We then explore the performance impact of the distribution of JITC methods and AOTC methods

2.4.4.1. Reducing the JITC-to-AOTC call overhead

The analysis result in the previous section indicates that the call from JITC methods to AOTC methods is problematic since its overhead is higher than other type of calls. One solution would be reducing the JITC-to-AOTC call overhead itself. The overhead of copying arguments from the operand stack of the JITC method to the registers and the C stack of the AOTC method by an assembly routine appears to be substantial. We can reduce this overhead by allowing the callee to access the caller' s operand stack directly for retrieving the argument. This can increase the overhead of AOTC-to-AOTC calls slightly, though, since an AOTC method should first check if its caller is an AOTC method or a JITC method. In fact, our AOTC is designed to maximize its performance only, including the AOTC-to-AOTC calls, so it would be reasonable to slow down AOTC-to-AOTC calls slightly to increase the overall performance of a hybrid environment.

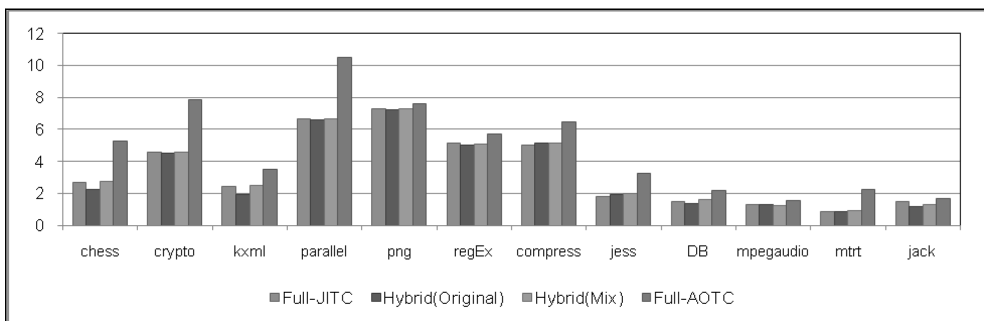


Figure 2-8 Performance of mixed argument passing in AOTC

We actually implemented this idea and experimented with it. Figure 2-8 shows the performance of the new calling convention,

hybrid(Mix), compared to the original hybrid, hybrid(Original), the full−JITC, and the full−AOTC. For those benchmarks whose JITC−to−AOTC calls are frequent such as chess, kxml, DB, and jack (where there were improvements in Figure 2−6), the hybrid(Mix) improves the performance tangibly, outperforming the full−JITC in some benchmarks. Comparing this graph with Figure 2−1, we can see that most of the anomaly results are gone away, placing the hybrid performance at least equal to the full−JITC performance in most benchmarks. However, we still could not see any hybrid performance that is definitely better than the full−JITC's.

2.4.4.2. Performance impact of the distribution of JITC methods and AOTC methods

Although the call overhead optimization in the previous section could remove the performance degradation of the hybrid environment, placing its performance closer to the full−AOTC's would be dependent upon the distribution of execution time among JITC methods and AOTC methods. That is, if we spend more time in the AOTC methods than in the JITC methods, the benefit of AOTC over JITC will take effect, leading to a higher performance than the full−JITC's. In embedded Java platforms, this means that the middleware is well−designed such that downloaded classes are implemented mainly by calls to the middleware rather than by their own computations, which allows spending more time in the middleware than in the downloaded classes. In this section, we want to explore the impact of the distribution of JITC methods and AOTC methods on the performance of the hybrid environment.

For this experiment, we compiled the library methods by the AOTC as previously. Then, we compile additional application methods by the AOTC, depending on their call depths from the main method. That is, we measure the minimum call depth of each method (e.g., if a method has a call depth of three for a call chain and four for a different call chain, its minimum call depth is three), and if it is higher than a given threshold T, we compile it by the

AOTC. The remaining application methods will be compiled by the JITC as usual. Consequently, a lower T value will make more methods to be compiled by the AOTC. Since we have the estimated execution time profile of each method, we can sum up the distribution of JITC methods and AOTC methods, and we can understand the relationship between the hybrid performance and the distribution.

Figure 2-9 shows the hybrid performance of the EEMBC benchmark with a diverse T value. We experimented with T=2, 4, 8, 12, and 16 (we use the calling convention of Section 2.4.1). When T=8, for example, we perform AOTC for those application methods whose minimum call depth is higher than or equal to 8, in addition to the library methods. For each T value, each graph also includes the proportion of the estimated execution time of AOTC methods to the estimated execution time of all (AOTC+JITC) methods. As T decreases, more methods are compiled by the AOTC (the proportion of AOTC methods comes closer to 1), boosting the hybrid performance closer to the AOTC performance (we could observe a similar results for the SPECjvm98 benchmarks). These graphs indicate that the distribution of AOTC methods and JITC methods affect our hybrid performance seriously, meaning that the hybrid compilation can be effective only when enough running time is spent in the middleware, which are compiled by the AOTC.

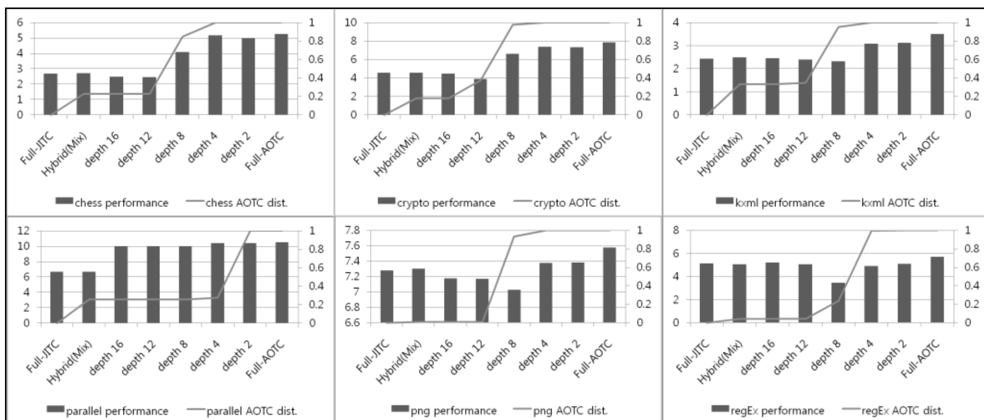


Figure 2-9 Performance of hybrid compilation with different T values for the EEMBC benchmarks

Chapter 3 Evaluation of Dalvik Virtual Machine

3.1 Android Platform

Android is a mobile OS for smart phones or tablets. Android is based on the Linux kernel, supplemented with middleware and libraries written in C/C++, yet the Android application (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, added security, and ease of app development. Android employs its own virtual machine (VM) to execute Java applications, called the Dalvik VM [23].

The Dalvik VM (DVM) differs from the conventional Java VM (JVM). First, the DVM has its own, register-based bytecode rather than the JVM's stack-based bytecode [24]. This can, in theory, lead to more efficient interpretation due to fewer bytecode instructions fetched for interpretation, reducing the fetch overhead which is crucial to the interpreter performance. Secondly, the DVM employs trace-based just-in-time compilation (JITC) [25], while high-performance JVMs including the HotSpot JVM, use method-based JITC [26]. The motivation for the trace-based JITC is that it would reduce the memory overhead of the mobile devices and compilation time, without affecting the performance benefit of JITC since only hot paths within a method will be compiled.

Obviously, it is interesting to compare the DVM and the JVM in terms of the performance, the memory overhead, and the compilation overhead to check if the DVM meets its promised goals. Also this is useful for understanding how the DVM performs for Android apps. Unfortunately, there has been little comparative evaluation of both VMs, nor any evaluation of the DVM for Android apps.

In this paper, we attempt to evaluate the DVM. We first evaluate it compared to the HotSpot JVM on the same, experimental tablet board using the embedded Java benchmarks. We also investigate how the DVM is affecting the performance of Android apps by

analyzing the behavior of several popular apps. As far as we know, our work is the first attempt to evaluate the DVM comprehensively and its performance issues for real apps.

3.2 Java VM and Dalvik VM

In this section, we describe the Dalvik VM compared to the HotSpot JVM, including the bytecode instruction set architecture (ISA), the interpreter, the JITC, and the VM components such as garbage collection and exception handling.

3.2.1 Bytecode ISA

The Java VM (JVM) is a typed stack machine. Each thread of execution has its own Java stack where a new stack frame is pushed when a method is invoked and is popped when it returns. A stack frame includes method/frame information, local variables, and the operand stack. All computations are performed on the operand stack after loading local variables, and computation results are saved back to the local variables, so there are many pushes and pops between the operand stack and local variables.

The Dalvik VM (DVM) is a register-based machine where computations are performed using virtual registers included in the VM. A stack frame of each method includes a register file, which have general-purpose registers as well as special registers. General-purpose registers include those mapped to local variables/ arguments, and temporaries. Special registers are for saving the caller method’ s PC, the pointers to the caller’ s register file and the method data, etc, when a method invocation is made, so the caller’ s execution state can be restored when the method returns.

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 produced 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, which are then converted to the static single

assignment form for optimization [27]. Many unnecessary bytecode instructions, especially those corresponding to the pushes and pops in the JVM are removed via dead code elimination, common subexpression elimination, loop invariant code motion, etc. This also leads to compact usage of registers. The number of registers needed for each method is determined, and they are allocated to the stack when each method is invoked.

(a) Source code

```
public static int addition(){
  int result = 1;
 for(int i=1;i<10000;i++)
   result = result + i;
 return result;
}
```

(b) Java bytecode

```
0x00  iconst_1
0x01  istore_0
0x02  iconst_1
0x03  istore_1
0x04  iload_1
0x05  sipush #10000
0x08  if_icmpge <0x15>
0x0b  iload_0
0x0c  iload_1
0x0d  iadd
0x0e  istore_0
0x0f  iinc 1 #1
0x12  goto <0x04>
0x15  iload_0
0x16  ireturn
```

(3) Dalvik bytecode

```
0x00  const/4 v0,#int 1
0x01  move v1,v0
0x02  const/16 v2,#int 10000
0x04  if-ge v0,v2,<0x0a>
0x06  add-int/2addr v1,v0
0x07  add-int/lit8 v0,v0,#int 1
0x09  goto <0002>
0x0a  return v1
```

Figure 3-1 Java program, Java bytecode, and Dalvik bytecode

Figure 3-1 shows an example of the Java source code, the JVM bytecode, and the Dalvik bytecode. For the Java loop (bold-faced), translation and optimization generate 5 Dalvik bytecode instructions from 9 JVM bytecode instructions. Three virtual register are required, so they are allocated to v0~v2.

3.2.2 Interpretation

Since the DVM has a "fatter" instruction set than the JVM, it is expected to interpret fewer bytecode instructions, yet need to do

more work for interpreting each bytecode instruction. It is questioned how this difference would affect the interpretation performance. According to the evaluation study of stack-vs-register-based bytecode [28], a register-based ISA leads to a better performance than a stack-based ISA because it can obviate many register moves corresponding to pushes and pops in a stack-based ISA, reducing the number of interpreted instructions by 46%. In fact, one dominant bottleneck in the traditional fetch-and-switch interpretation is the overhead of fetching the next bytecode because it often requires executing a couple of branches, some of which have a high overhead. So, reducing the number of interpreted instructions would reduce the fetch overhead, improving the performance. However, both the DVM and the JVM interpreters employ indirect threading to reduce the fetch overhead such that instead of jumping to the loop header for switching to the handling routine after fetching the next instruction, threading allows a direct jump to the handling routine with the help of a dispatch table, which obviates a couple of branches in the fetch-and-switch interpretation. So, we need to check if the DVM's interpretation is really faster than the JVM's.

One thing to note is that the DVM provides both the assembly version and the C version of the interpreter, while the JVM which we experiment with provides only the C version. So, we experiment with both versions of the DVM interpreter to compare against the JVM interpreter. The assembly interpreter is faster because the handling routine of each bytecode is implemented manually by at most 16 ARM instructions so that its starting address is located sequentially at the boundary of 16 ARM instructions; after fetching the bytecode, we can find the starting address of its handling routine fast via the opcode multiplied by 16, which is then added to the pointer of the handling routine area, instead of loading from the dispatch table.

Another thing to note is that the DVM has the dexopt, an install-time optimizer. When we run a dex file for the first time, the dexopt optimizes it and saves the optimized file in the system, which will be

used thereafter in the next runs without the dexopt process again. The optimization in the dexopt includes quickening based on static linking and generation of some additional data for efficient VM execution. Quickening patches the bytecode for reducing constant pool access. Some of these optimizations are performed during the class loading time in the case of the JVM.

3.2.3 Just-in-Time Compilation (JITC)

Generally, one of Java's advantages as a mobile software platform is platform independence, achieved by using the bytecode that can be executed by the interpreter on any platform without porting. Since this software-based execution is much slower than hardware-based execution, just-in-time compilation (JITC) for translating bytecode into machine code at runtime has been used in the JVM [1]. The DVM is not an exception and it also employs JITC. However, there are some differences in the JITC techniques used in the two VMs.

Basically, both VMs employ the same approach of adaptive compilation: the bytecode is executed first by the interpreter, and when a unit of bytecode is detected as a hot spot it is compiled to machine code. The machine code is saved at the code cache in the VM, which is executed thereafter [29]. The differences include the unit of JITC and the compilation techniques used during JITC.

The unit of JITC in the JVM is a method such that a method is interpreted during initial invocations but when it is determined to be hot, it is compiled. The hotspot detection is based on a cost-benefit model where the cost is the compilation overhead while the benefit is the performance advantage. A hot method is identified by estimating its runtime during interpretation since a long-running method is likely to be hot. The runtime estimation of a method is made based on its invocation count and the loop iteration counts. If the estimated value is larger than a threshold, the method is compiled [30].

The unit of JITC in the DVM is a trace, which is a fragment of hot

execution paths that can span beyond a basic block. Instead of compiling the whole method, only hot traces in the method are compiled. This can, in theory, reduce both the amount of machine code generated and the compilation time, hopefully without affecting the performance. This would be desirable for mobile devices with a limited memory and a real-time requirement. Traces have been used for dynamic optimization such as Dynamo [40] and for JavaScript JITC such as TraceMonkey [42]. These traces span multiple basic blocks, even across function call boundaries. Unfortunately, the DVM traces are much shorter because they do not span beyond a branch or a method call, which limits the code optimization opportunities. The DVM JITC process is depicted in Figure 3-2.

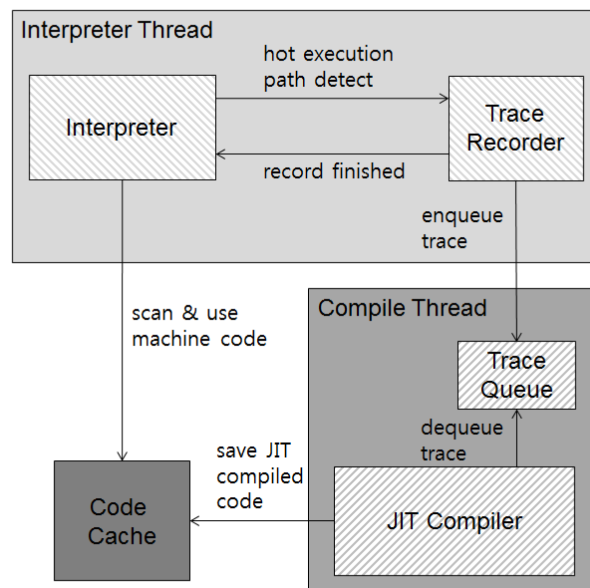


Figure 3-2 Dalvik VM JITC process

There are two threads in the DVM: the main thread and the JITC thread. The main thread is responsible for interpreting the bytecode, detecting the traces, handing the traces over to the JITC thread via the trace queue, executing the JITC-generated trace if it is available in the code cache when the trace is to be interpreted. The JITC thread is responsible for compiling the traces in the trace queue and placing them in the code cache.

The trace detection in the main thread works as follows. Initially, the bytecode is interpreted. A counter is maintained for each entry point of a trace and is incremented when the trace is entered. The target bytecode of a jump (branch, method invoke and goto instructions) or the next bytecode interpreted after exiting from a trace can be an entry point. When the counter value of an entry point exceeds a threshold (40 in our environment), trace recording starts such that the bytecode is interpreted and at the same time it is recorded in the trace buffer. The trace recording stops when a branch or a method invocation is interpreted or when the number of bytecode in the trace buffer exceeds a pre-defined value (100). After finding a trace, it is enqueued to the trace queue for compilation by the JITC thread.

Figure 3-3 (a) shows a control flow graph (CFG) of a loop composed of seven basic blocks (BBs). Since a jump target can be an entry point of a trace, all of basic blocks (BB1 ~ BB7) can be an entry point. If all of their counters exceed the threshold during the execution of the loop, for example, we would obtain a CFG of traces in Figure 3-3 (b). The trace from BB1 can include only itself because there is a branch at the end of BB1. Similarly, the trace from BB2 spans thru BB4 but stops there due to the branch at its end. It should be noted that BB4 is also included in the trace starting from BB3 because it is a join point but a trace does not stop at a join point (there can also be a trace composed of BB4 only although it is not reachable). This means that BB4 is duplicated in the JITC-generated code. BB7 is also duplicated in the traces starting from BB5 and BB6, respectively. Consequently, trace-based compilation is involved with some duplication of code.

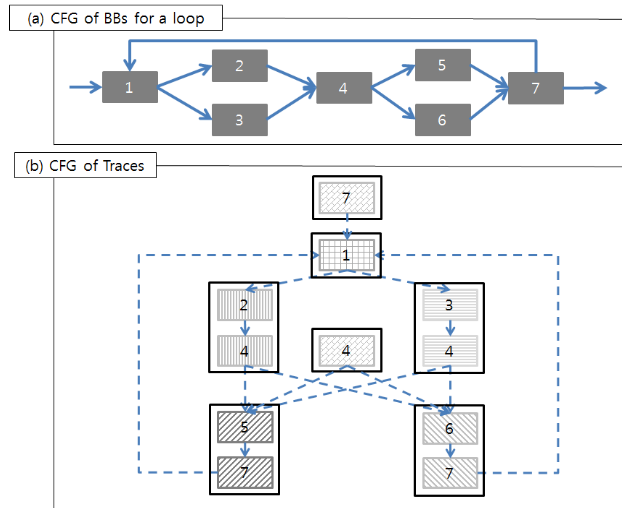


Figure 3–3 An example (a) CFG of BBs and (b) CFG of traces

Control transfer from one trace to another trace is made via a mechanism called chaining. At the end of each trace, a tiny code and data space called a chaining cell is added for each target of the trace (a single cell if the trace ends with a method call and two cells if it ends with a branch). In Figure 3–3, for example, two chaining cells for BB2 and BB3 are made at the end of the BB1 trace. The chaining cell includes a jump instruction to a VM internal function which handles the chaining, followed by a data space to cache the address of the target trace. When the execution of a trace ends, we first jump to the chaining cell for the next target, where we make another jump to the internal function. The function first checks if the data space in the chaining cell has an address, and if so, we make a jump to the address to transfer to the corresponding trace. Otherwise, the function checks if a trace starting from the next target is already available. If so, we cache the address of the trace at the data space and make a jump to the trace. If not, we jump to the interpreter to interpret the target. If the trace is too short, the space overhead of chaining cells would be nontrivial because there would be many chaining cells.

Now we discuss the performance issues with the DVM traces. One problem is the preciseness of hot trace detection. Since every jump target can be a trace entry point, there are too many of those

during the execution of a program, requiring a huge table of counters. To reduce the space and performance overhead, DVM uses a fixed-size (2048) byte array of counters, indexed based on the bytecode address of trace entry point. This can make different entry points share the same index to maintain their counters, leading to imprecise detection of hot traces.

Another performance issue is code quality. Unlike the JVM JITC which compiles a whole method at once possibly with other inlined methods, the DVM JITC compiles fractions of code that are too small. This can lead to fewer opportunities for code optimization, although the code generation is faster. For example, the DVM JITC uses simple load/store elimination, null check elimination, scheduling, and loop invariant code motion, which would not be effective if their target code is too short. Also, if there are too many short traces, the overhead of chaining cells will be substantial. This can lead to a poor quality of generated code, affecting the performance as well as the code size negatively.

The third issue is register allocation. Since the DVM employs the register-based bytecode, register allocation would be more straightforward than the JVM with the stack-based bytecode. However, the DVM JITC cannot map and allocate the physical registers to the virtual registers globally, because of the trace-based JITC. That is, there would exist JITC-generated code and interpreted code within a method, and whenever there is a transition between them, copying (load/store) between the physical registers and the virtual registers saved in the stack frame would be required, if the virtual registers were fixedly mapped to physical registers in the JITC-generated code. This copying overhead also exist in method-based JITC when there is a call between JITC-generated method and interpreted method, but it would be much higher in trace-based JITC because the transition would be much more frequent. So, instead of fixed mapping, register accesses in the DVM JITC-generated code are translated to loads and stores to virtual registers in the stack frame. For example, v0=v0+v1 requires two loads from v0 and v1 and a store to v0 after the

addition. Redundant loads in a trace are eliminated via optimization, but such elimination opportunities would be low due to small traces, as mentioned above. For the JVM JITC, physical registers are used and allocated more efficiently due to better optimizations with the method−based JITC.

(a) Dalvik JITC-generated code

```
ldr  r0, [r5, #4]   // add-int/2addr v1,v0
ldr  r1, [r5, #0]
adds r0, r0, r1
adds r1, r1, #1     // add-int/lit8 v0,v0,(#1)
str  r0, [r5, #4]   // goto
str  r1, [r5, #0]
ldr  r2, [rpc, #48] // const v2,(#10000)
ldr  r3, [r5, #0]   // if-ge v0,v2
cmp  r3, r2
str  r2, [r5, #8]
bge  L1
b    L2
```

(b) JVM JITC-generated code

```
ldr  v8, [PC, #+0]     // sipush #10000
cmp  v4, v8 lsl #0     // if_icmpge
bge  L1
add  v3, v3, v4 lsl #0  // iadd
str  v3, [rJFP, #-8]
add  v4, v4, #1        // iinc
str  v4, [rJFP, #-4]
b    L2               // goto
```

Figure 3−4 Machine code generation by DVM and JVM JITCs

Figure 3−4 shows an ARM assembly code generated by the DVM JITC and the JVM JITC for the example loop in Figure 3−1. For the DVM JITC, a trace entry is the target of the branch (if−ge), which is add−int/2addr. The trace spans across the backward branch up to the branch (if−ge). The ARM code generated for the trace shows that loads are needed to load the virtual registers of the source operands and stores are needed to save the computed results to the virtual registers. If there are redundant loads, they are eliminated. Compared to the DVM JITC−generated code with 12 instructions, the JVM JITC generates a much better code with 8 instructions due to better reuse of registers and optimizations.

3.3 Experimental Results

The previous section described the DVM with its interpreter and JITC, compared to the JVM’ s. In this section we evaluate the two VMs on the same mobile platform for the Java benchmarks.

3.3.1 Experimental Environment

We experimented with a tablet PC where both Android and Linux OS can be installed. It has an ARM Cortex-A9 Quad Core CPU (Exynos4412) with a 1 GB memory [34].

We use the Android platform 4.1.2 Jellybean which includes the JITC. The Android platform is based on Linux version 3.0.51 for core system services such as security, memory management, process management, network, and driver model. The system C library (libc) in Android is a BSD-derived implementation of the standard C system library, tuned for embedded Linux devices [37].

We use the Java VM called the PhoneME Advanced, which is an open-source JavaME (Micro Edition) project [35]. Its JITC is based on the HotSpot technology. The PhoneME includes the CDC (Connected Device Configuration) implementation of a JVM [36], so it is appropriate for the mobile environment and suitable to compare against Android. We build the PhoneME using gcc-4.7.1 and installed on the Linux 3.0.51 with glibc 2.15. So, the Linux for the JVM and the Android Linux have different implementations of the C libraries, yet this would not affect the performance evaluation much, especially when we experiment with the Java benchmarks because the running time would be dominantly spent in the VM (see Figure 3-19).

For Java benchmark, we use the EEMBC GrinderBench [38]. Unlike the SPECjvm benchmark for the JavaSE or the DaCapo benchmark for the server environment, the EEMBC benchmark is for the JavaME embedded environment, thus more suitable for our evaluation.

For those six programs in EEMBC (Chess, Crypto, kXML, Parallel, PNG, RegEx), both the DVM and the JVM use common core Java class libraries, but their implementation is slightly different. That is, some library methods are implemented by a Java method in one VM, but are implemented by a native (JNI) method in the other VM. For java.lang.String class, for example, the JVM implements 10 methods

including hashCode() and indexOf() by native methods, while the DVM implements only 7 of them by a native method. Fortunately, we found that only cold methods in kXML and RegEx use them, so the impact caused by the different implementation of those library methods would not be significant but should be taken into consideration.

There is another issue to be considered when comparing the DVM JITC and the JVM JITC because they generate different ARM ISAs. The DVM JITC generates Thumb2 code while the PhoneME JITC generates the ARM code (there is no PhoneME JITC published for Thumb2 and no Dalvik JITC published for ARM). Thumb2 has 16-bit instructions as well as 32-bit instructions. It is aimed at improving the performance of the Thumb which has only 16-bit instructions, thus suffering seriously from performance degradation even though it reduces code size significantly [32]. Thumb2 makes a compromise between the code size and the performance with 32-bit instructions. It is known that the Thumb2 code generated by the JVM JITC achieves a code size reduction of 15% and a performance degradation of 6%, compared to the ARM [33]. Therefore, we need to take this difference into consideration when comparing the two JITCs in terms of the performance and the code size.

3.3.2 Interpreter Performance

We first measure the interpreter performance. Figure 3-5 shows the performance of the two DVM interpreters (assembly version and C version) compared to that of the JVM interpreter as a basis. The assembly version of the DVM interpreter is 55% faster than the JVM interpreter, yet the C version is only 12% faster. Since the JVM interpreter is also C-based, the 12% improvement would be more appropriate. This indicates that there is no big difference between the two interpreters, or more precisely, between the register-based bytecode and the stack-based bytecode for interpretation. This difference is much lower than the one in the stack-vs-register study [28], and the reason appears to be related

to the dynamic size of executed bytecode, as discussed below.

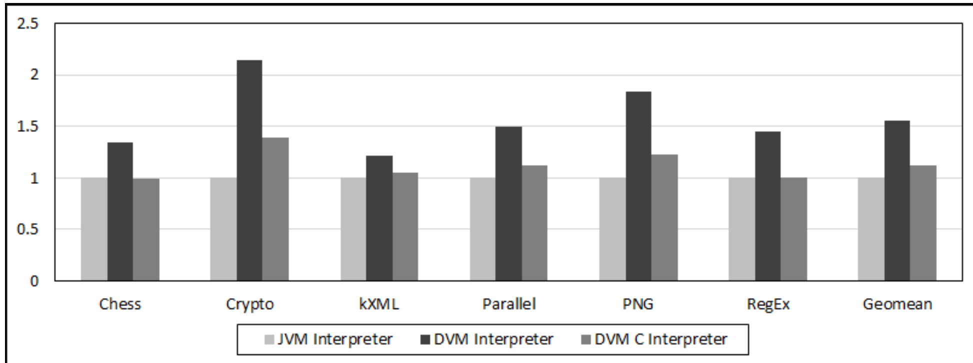


Figure 3−5 Interpreter performance ratio

To understand the difference between the two VMs, we compared the bytecode size and count. We first show the static result. Figure 3−6 (a) and (b) show the static bytecode code size and the bytecode count for the two VMs, respectively, for the benchmarks in total (left) (for static result, we show the total size/count because there are many sharing of the same classes among benchmarks). The graph shows that there is little difference in the static bytecode size, while the DVM bytecode count is 70% smaller than the JVM count. Actually, the DVM includes a "super" bytecode instruction called fill−array−data, which initializes an array with a single instruction, often used in class initializations. A total of 189 instructions exist in EEMBC, whose size (37KB) take 33% of the static bytecode size in Figure 3−6 (a), so they are substantial. The JVM does not include an equivalent instruction, so it requires additional instructions for the same job. If we exclude those methods that include the super bytecode, the DVM bytecode code size becomes 33% larger than the JVM's and the DVM bytecode count is 32% smaller, as shown in Figure 3−6 (right).

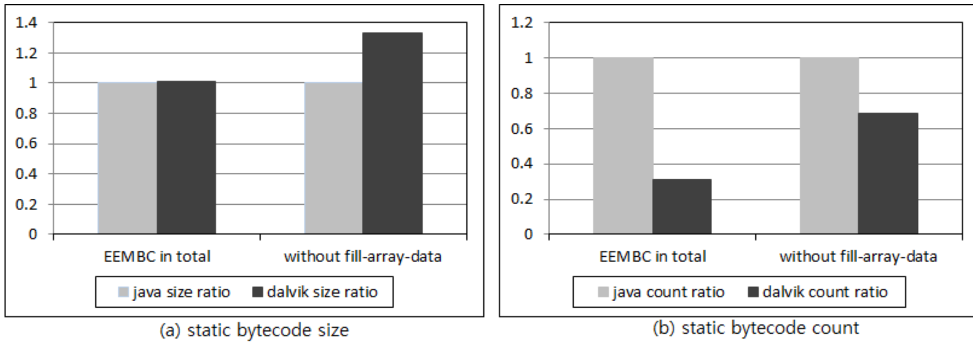


Figure 3-6 Static (a) bytecode size and (b) bytecode count ratio

The stack-vs-register study shows that statically its register-based bytecode has 26% larger bytecode size and 44% smaller count than its stack-based bytecode [28]. This means that the DVM has a larger number of fatter bytecode instructions than the study.

Now we compare the dynamic result. Figure 3-7 compares the dynamic bytecode count of the two VMs for each benchmark score. As expected, the DVM bytecode count is 43% smaller than the JVM bytecode count, which is similar to the 46% smaller count in the stack-vs-register study. This would reduce the fetch overhead, leading to the performance improvement compared to the JVM.

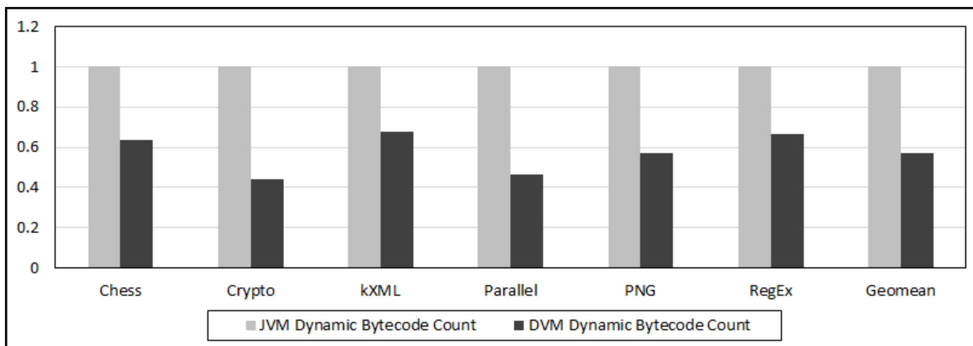


Figure 3-7 Dynamic bytecode count ratio

Figure 3-8 compares the dynamic size of the executed bytecode, which shows that the DVM bytecode size is 52% larger than the JVM. This means that DVM requires a 52% larger program than the JVM for achieving the same job. Since the fill-array-data is executed mostly in the class initialization only, its impact on the

dynamic code size is little. Actually, this result is much higher than the dynamic code size result of 25% in the stack-vs-register study [28], which appears to be due to the larger bytecodes of the DVM. If we compute the average bytes for a dynamic bytecode instruction from Figure 3-7 and 3-8, it is 3.69 bytes for the DVM and 1.34 bytes for the JVM.

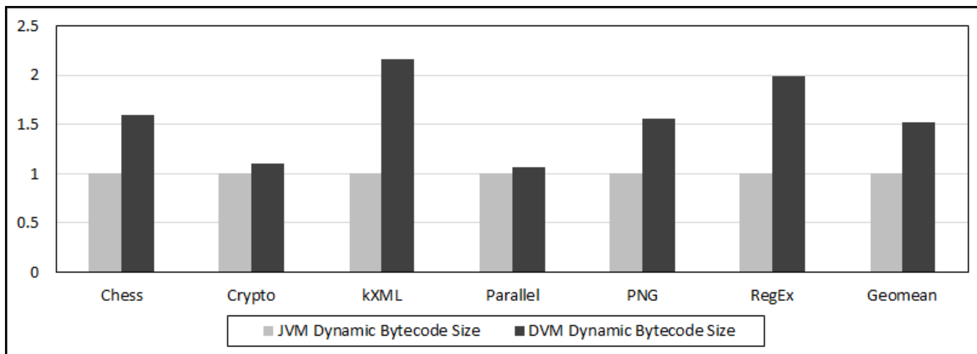


Figure 3-8 Dynamic bytecode size ratio

We think that this difference made the DVM interpretation perform worse than the interpretation in [28]. That is, the study claims that there are two factors that contribute to faster interpretation than the JVM. One is the smaller dynamic count which reduces the fetch overhead of branches. This would be the same in the DVM bytecode. The other is the reduced memory accesses due to fewer reads/writes of operands in the register-based bytecode, which can outweigh the increase of memory accesses for fetching larger instructions. Since the DVM bytecode is much fatter than their bytecode, the memory access for fetching instructions is not well outweighed by the reduced operand reads/writes, making the interpretation perform worse than [28].

3.3.3 JITC Performance

We enabled the JITC for each VM and measured the performance. Figure 3-9 shows the speedup of the two JITCs compared to the JVM interpreter performance as a basis. Unlike the interpreter performance, the JVM JITC shows 8.7x speedup while the DVM

JITC shows only 3.9x speedup, achieving only half of the JVM performance. This is a big difference even if we take the 6% performance difference between the ARM code and the Thumb2 code into consideration.

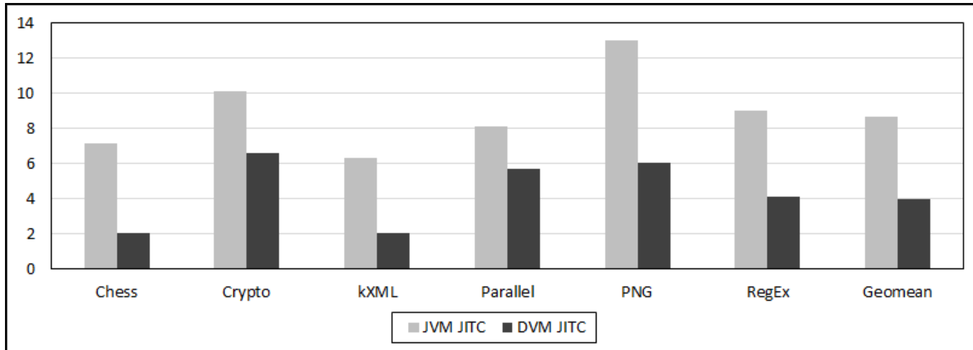


Figure 3-9 VM performance with JIT compiler

There can be a few possible reasons for the worse performance. One is the small trace size which limits efficient code generation, producing a worse quality code. In fact, we found that a trace includes only three bytecode instructions, on average. Also, the trace-based JITC might compile too small sections compared to the method-based JITC, reducing the portion of native execution. Maybe hot trace detection might not be precise enough.

We measured the amount of bytecode compiled during the execution of the benchmark programs on the two VMs and Figure 3-10 shows their ratio. Although the DVM executes a larger amount of bytecode than the JVM as we saw in Figure 3-8, it compiles a smaller amount of bytecode than the JVM, which is reasonable because only hot traces are compiled rather than the whole methods. One exception is RegEx which uses some String class methods, and they are native methods in the JVM, thus not being compiled, while they are bytecode methods in the DVM, thus being compiled. Generally, this means that the portion of program compiled by the JITC is 20~30% smaller in the DVM.

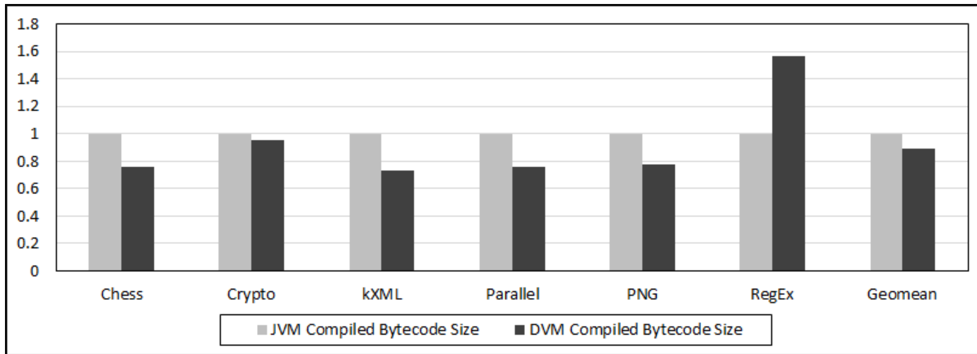


Figure 3-10 Compiled bytecode size ratio

We also measured the amount of machine code generated during execution for the two VMs, and Figure 3-11 shows the ratio. Contrary to the claim that the trace-based compilation can reduce the memory pressure by compiling only traces, the DVM JITC generates a 70% more machine code than the JVM JITC. If the DVM JITC were generating the ARM code instead of the Thumb2 code, the difference would be even higher. This means that the DVM causes a higher pressure for the code cache.

The code size in RegEx is especially higher, and this is due to its big hot methods where hot traces are uniformly distributed all over the BBs in the methods, so the trace-based compilation works similarly to the method-based compilation, generating even larger code than in other benchmarks. Also, the inlined methods by the JVM JITC are smaller than in other benchmarks, which lead to generation of the smaller code in the JVM JITC.

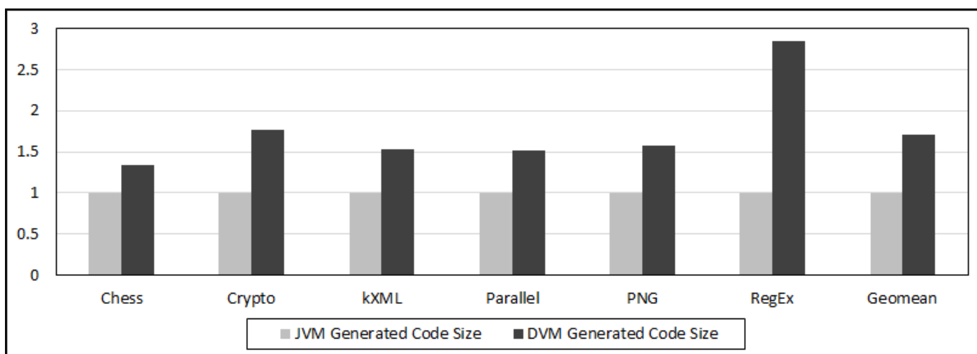


Figure 3-11 Generated machine code size ratio

The machine code size in Figure 3−11 includes constant data embedded in the code as well as instructions, so we measured the instruction count separately as shown in Figure 3−12, which is higher than in Figure 3−11, due to the removal of constant data.

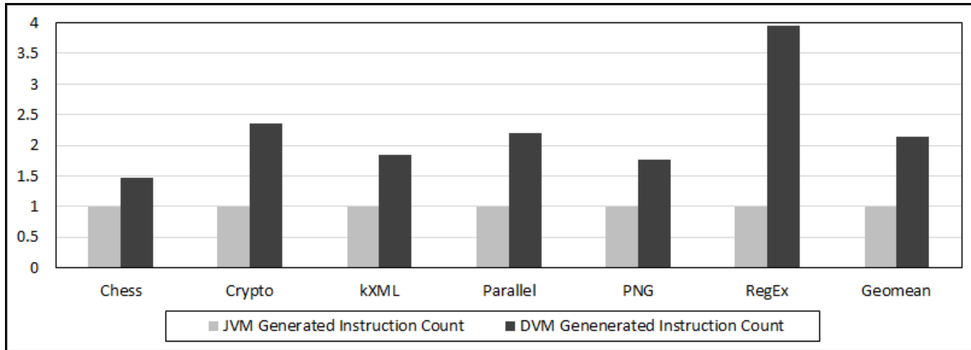


Figure 3−12 Generated machine instruction count ratio

One thing that causes the larger code size in Figure 3−11 is the redundant compilation of bytecode at join points. We measured for those compiled bytecodes how many times a DVM bytecode instruction is compiled redundantly, and Table 3−1 shows the "redundancy" ratio. It shows that a bytecode instruction is compiled 1.27 times, on average. Those benchmarks with a high ratio include big switch statements (Chess) or repeated if−else statements (Crypto, Parallel, PNG). The redundancy ratio for the JVM would be one if we ignore duplications caused by inlining.

Table 3−1 Redundancy ratio of the Dalvik JITC

	Chess	Crypto	kXML	Parallel	PNG	RegEx	Avg.
Ratio	1.26	1.56	1.14	1.27	1.23	1.20	1.27

One implication of the DVM JITC that compiles a smaller amount of bytecode (Figure 3−10) but generates a larger amount of machine code (Figure 3−11) is that its code quality would be worse. The code quality is difficult to measure in our environment due to the difference of both the bytecode ISA and the machine code ISA. We estimated it by computing the number of machine instructions generated from one byte of the bytecode for each VM, which is

obtained by the machine instruction count in Figure 3-12 divided by the compiled bytecode size in Figure 3-10. The result is shown in Figure 3-13. The JVM JITC generates an average of 1.24 ARM instructions per one byte of the JVM bytecode. The DVM JITC generates an average of 3.02 Thumb2 instructions per one byte of the DVM bytecode, which is equivalent to 0.57 JVM bytecode as we can estimate from Figure 3-7. So, The DVM JITC would generate an average of 5.30 (=3.02/0.57) Thumb2 instructions per one byte of the JVM bytecode (we guess ARM instruction count is similar to the Thumb2 instruction count even if the code size differs by 6%). Consequently, it appears that the DVM JITC generates at least 4.3 times larger machine code than the JVM JITC, affecting both the performance and the memory negatively.

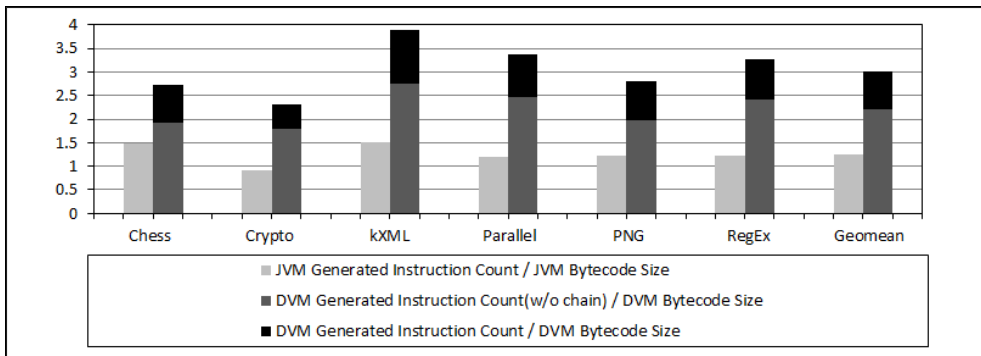


Figure 3-13 Generated instruction count per bytecode's 1 byte

If we compare the machine code size, not the instruction count, generated from one byte of the bytecode, the result is in Figure 3-14, which is similar to Figure 3-13, yet the constant data as well as instructions are included.

There are two factors that contribute to the generation of larger machine code from the same size of bytecode. One is poor code quality caused by inefficient code generation and optimization for the trace. The other is the additional code for chaining cells. We found that a trace is compiled to around 130 bytes of machine code on average, followed by 32~36 bytes of the chaining cell code, which is equivalent to 32~36 instructions, followed by 10 instructions of the chaining cell code. We separated the code

overhead of the chaining cells in Figure 3−13 and 3−14 (marked by the top black portion in the DVM bar), which is substantial. If we exclude the overhead of chaining cells, the graph would show the impact of poor code quality (the remaining portion of the DVM bar), which is an average of 3.86 (=2.20/0.57) Thumb2 instructions per one byte of the JVM bytecode. So among the 4.3 times larger machine code generated by the DVM JITC than the JVM JITC, 3.1 times larger machine code is caused by poorer code generation and the remaining is caused by the chaining cells.

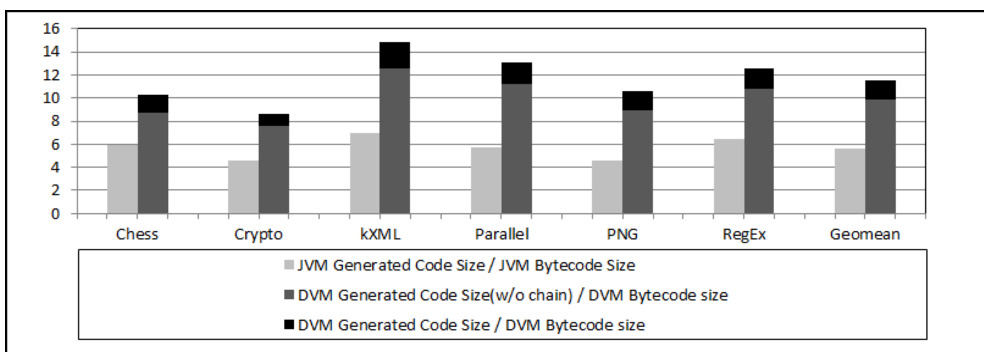


Figure 3−14 Generated machine code size per bytecode's 1 byte

We also measured the compilation time for both JITCs, which is depicted in Figure 3−15. The compilation time of the DVM JITC is 3.7 times longer than the JVM JITC, and it shows a similar behavior to the generated instruction counts in Figure 3−12. Figure 3−16 shows the DVM compilation overhead over the running time, which is 2.7% and is also higher than the JVM overhead of 1.6%.

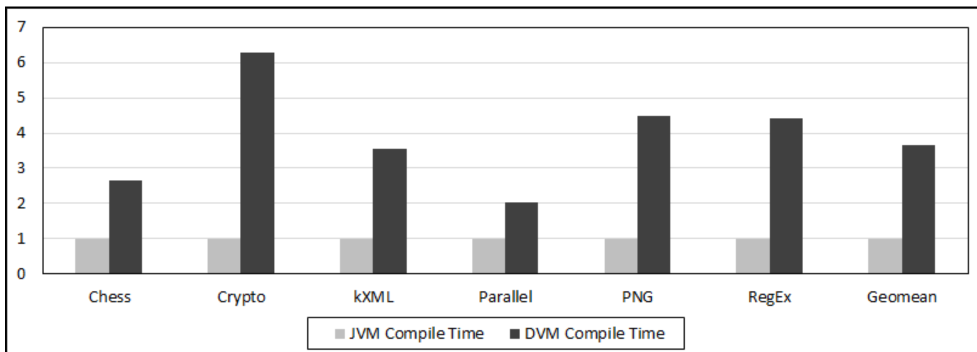


Figure 3−15 Compile time ratio

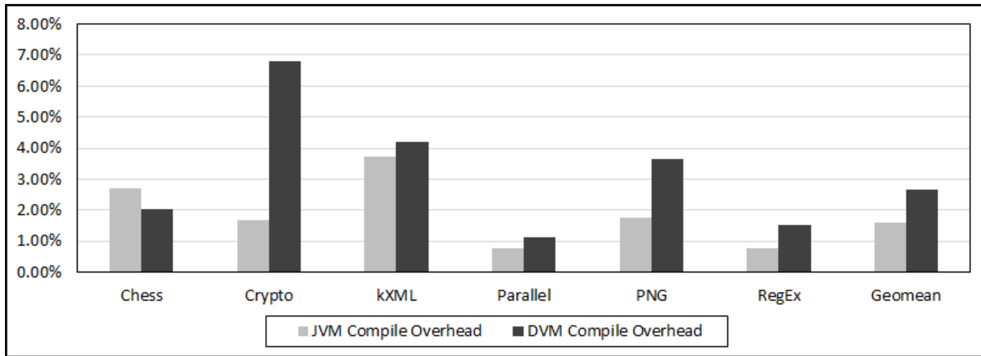


Figure 3-16 Compilation overhead over the total running time

3.3.4 Trace Extension

It would be questioned if we can improve the performance of the DVM JITC by increasing the trace size, because it will increase the code optimization opportunities in the trace. We actually modified the tracing algorithm DVM JITC so that the trace continues to span beyond a branch and stops only at a method invocation or when the number of bytecode exceeds a threshold.

Unfortunately, the performance would not improve but even degrade slightly, as shown in Figure 3-17 (the second bar). We found that the existing optimizations are not applied any significantly more, while the redundant compilation increases seriously (the size of compiled bytecode and generated machine code in Figure 3-10 and 3-11 increases by 12% and 5%, respectively). We believe the increase of the compilation overhead degraded the performance.

We added additional optimizations after the trace extension such as loop detection and loop-invariant code motion (LICM). And we perform additional benchmark test, Linpack benchmark and Quadrant (CPU) to identify LICM optimization. Then, the performance increases as shown in Figure 3-17 (the third bar) where on average, the performance decreased by 12% and 3% compared to the original JITC and the trace-extension-only, respectively. And the performance increases 2% in Linpack becnhmark and 27% in Quadrant compared to the original JITC.

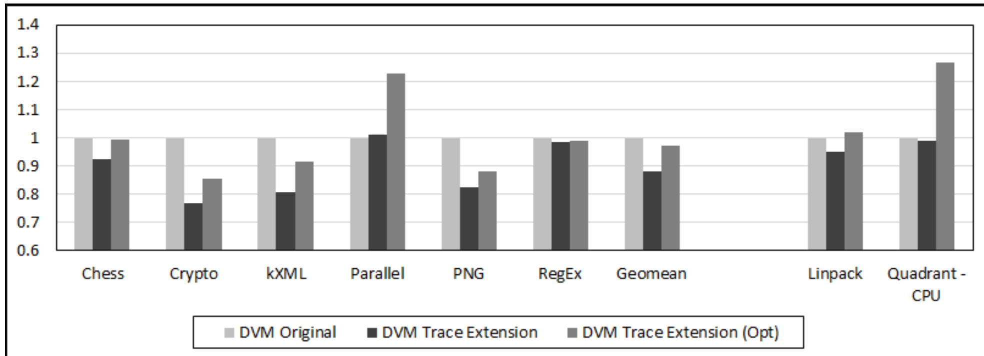


Figure 3-17 Performance impact of expanding the trace

We strongly feel that it is not easy to improve the DVM JITC performance significantly (by as much as the JVM performance), with its current form of trace-based compilation. In fact, some of the major performance benefit of the JVM JITC comes from inlining, so stopping at the call boundary for traces would be a major bottleneck for performance enhancement.

One thing to note in JB is that a preliminary implementation of a naïve method-based JITC is included in its source code, yet disabled in the current release. We think that they also noted the same performance issues of trace-based JITC raised here and proposed a method-based JITC for the future DVMs.

3.4 Behavior of Real Android Apps

The previous section showed that the DVM is much slower than the JVM, even with no advantage of memory overhead. Then, it might be questioned how come the Android apps are running with such a big performance disadvantage of the DVM.

In fact, Java execution even on the JVM would be slower than native execution, such as the one used in the iOS apps, for example. To estimate the performance implications of different approaches to mobile applications such as iOS apps (native) and Android apps (Java) from programming language perspectives, we implement the Java CaffeineMark [44] with C++ and run on the same board in the previous section with the best optimizations enabled. Figure 3-18 shows the performances of C++, Java (JVM), and Java (Dalvik),

with C++ as a basis of 1.0. We found that the JVM and the DVM are at least five times and ten times slower than the C++ version, respectively. This means that the same application would run at least six times slower on the Android platform compared to the iOS platform (assuming that C++ performs similarly to Objective-C). Actually, there is some unconfirmed observation that the same application tends to be slightly slower as an Android app than as an iPhone app. However, there is no big complaint on the performance of Android apps. In this section, we want to investigate why there is little performance issue on Android, by analyzing the behavior of real Android apps compared to benchmarks, especially their execution profiles.

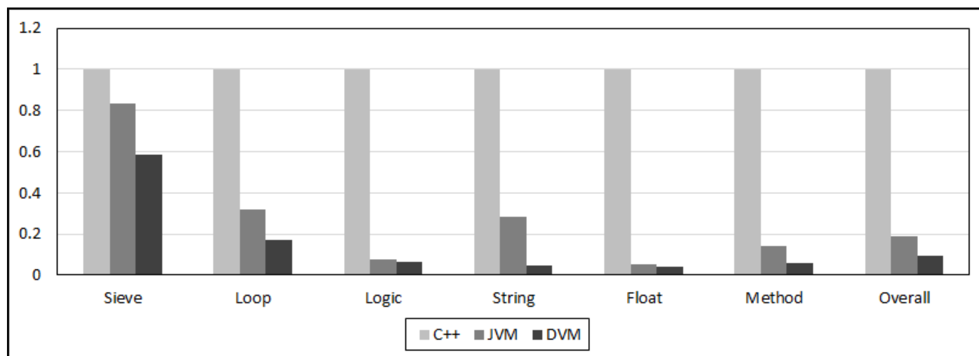


Figure 3-18 Performance of C++, JVM and DVM

Basically, Android apps are not entirely run on the DVM because JNI methods, libraries, and kernels are run in native. So, if the Java portion running on the DVM is not dominant, slow Java execution on the DVM would not affect the overall app performance seriously. Based on this idea, we analyze the runtime profile of six popular Android apps listed in Table 3-2, which cover diverse categories of applications. We ran them until waiting user first input (loading), and ran with user input based on definite scenario (running).

Table 3-2 Android applications experimented

Applications	Category	Running Details
TempleRun	Game	Play for 20 seconds
DoodleJump	Game	Play and restart game for 20 seconds
Evernote	Productivity	Make new note and delete it
Twitter	SNS	Click tweets and pictures
Astro File Manager	File Navigator	Search file system
Google Map	Navigation	Navigate and enable satellite map

We use the OProfile tool included in the Android-Linux for profiling. Using the hardware counters, OProfile can measure the runtime data including cycle counts, which are used to report the runtime portions of native and DVM. OProfile can ignore the idle time during execution (e.g., waiting for user inputs), so it does not affect the profile data.

OProfile reports the time spent in the DVM (interpreter and JITC code) and the time spent in the native (kernel+library and native app). Figure 3-19 shows the profile for those six Android apps for loading, and Figure 3-20 for running with user input. EEMBC benchmark results are also shown for comparison. For the benchmarks, the DVM portion is more dominant than the native portion, and the JITC portion is also more dominant than the interpreter portion. For the Android apps, however, the native portion is much more dominant, and among the DVM portions, interpreter portion is more substantial.. TempleRun appears to be implemented mainly by C/C++ using JNI, so a shared objects (libunity.so and libmono.so) are included in the app file. We depict its portion in Figure 3-19 and 3-20 by "native app". Execution of other JNI methods would be included in library+kernel. Consequently, the graph shows that for Android apps, the Java portion, especially the JITC portion, is much smaller, so even if the JITC or the interpreter does not perform well, its impact is limited.

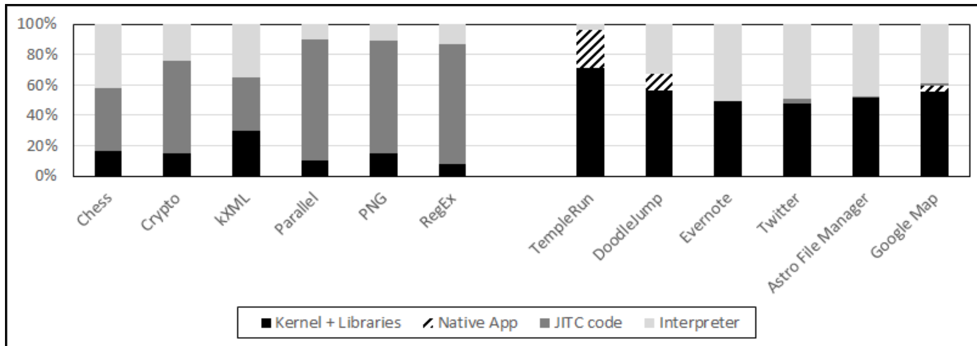


Figure 3-19 Profile of EEMBC and Android Apps for loading

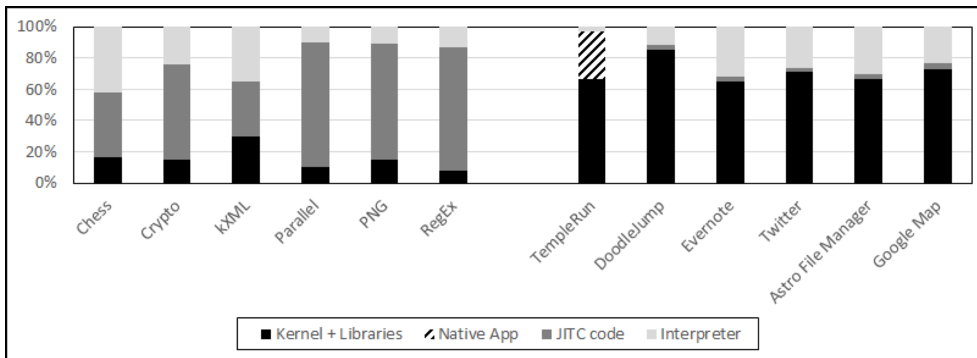


Figure 3-20 Profile of EEMBC and Android Apps for running

We also analyzed why the JITC portion in the apps is much smaller than in the benchmarks in app loading. Figure 3-21 shows the average iteration count of a loop per second for the benchmarks and the apps. A benchmark loop iterates tens/hundreds of thousands of times, while an app loop iterates only hundreads of times at most. Figure 3-22 shows the average call count of a method made per second during execution. Again, a benchmark method is called tens of thousands of times per second while an app method is called hundreds of times per second. Consequently, the apps have much colder loops and colder methods, compared to the benchmarks. Even if these colder spots are compiled by the JITC, they will not be executed frequently, minimizing the JITC portion. Actually, if this is the typical behavior of the Android apps, JVM would also have a lower JITC portion, and its high-performance JITC would not be helpful in improving the Android apps, although we could not prove this by running them on the JVM. Even the difference between the Android Java and the iOS native would get

narrower.

For the DVM JITC, we also compared how the traces are compiled and reused. Figure 3-23 shows the average execution count of a trace per second. On average, a benchmark trace is executed 1600 times more than an app trace after it is compiled, which is something expected. What is not expected is Figure 3-24, which shows the average number of traces compiled per second. On average, DVM compiles similar traces in the apps than in the benchmarks (TempleRun has far fewer traces due to its native app). In fact, the total number of traces compiled for apps is much higher than that for benchmarks, and it tends to increase as we increase the running time of apps. This means that apps are likely to generate more traces than benchmarks, yet app traces are executed far fewer times than benchmark traces, perhaps even not enough to justify the JITC overhead. So, the apps are likely to suffer more from JITC overhead.

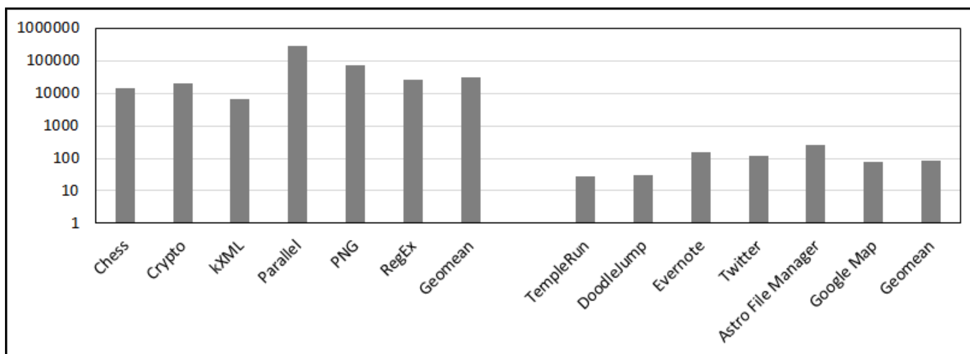


Figure 3-21 Average iteration count of a loop per sec. (log scale)

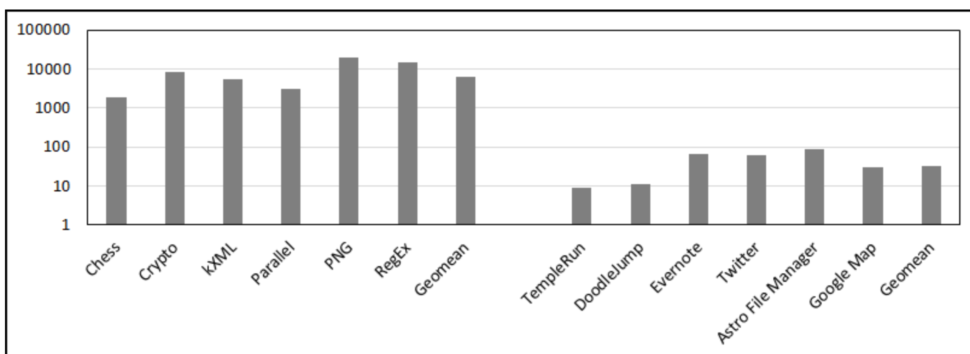


Figure 3-22 Average call count of a method per sec. (log scale)

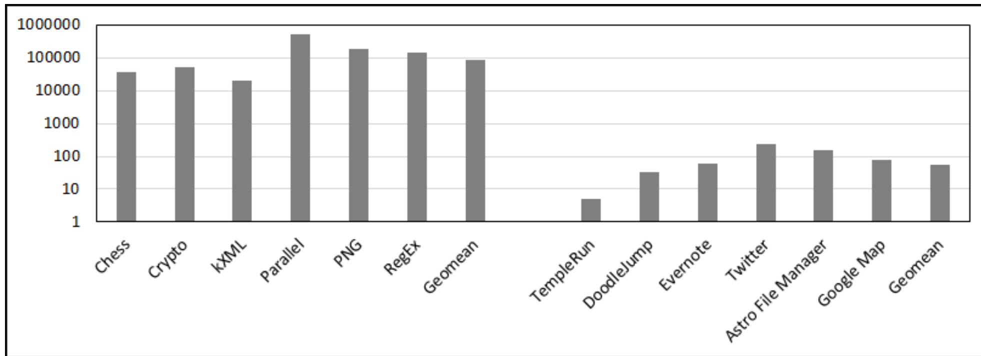


Figure 3-23 Average execution count of a trace per sec. (log scale)

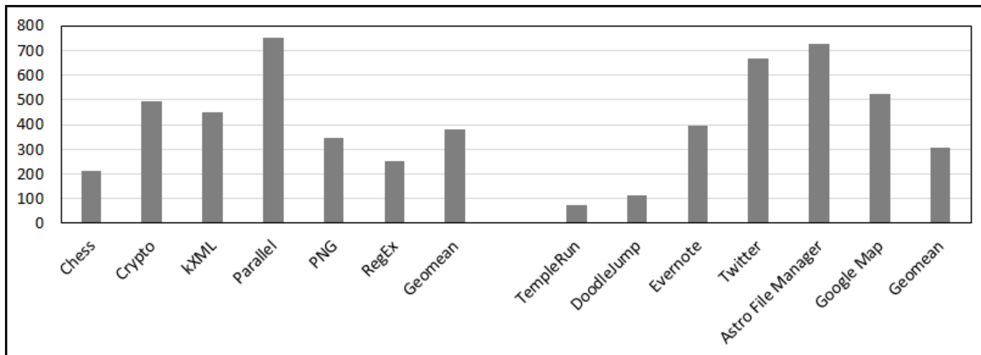


Figure 3-24 Number of traces compiled per sec

These app behaviors indicate that increasing the Android DVM performance requires improving the JITC portion as well as the JITC itself. This would need faster detection of "warm" spots in addition to hot spots. Also, apps tend to suffer more from the JITC overhead due to the generation of more traces, so ahead-of-time compilation to reduce the JITC overhead would be useful.

Chapter 4 Ahead-of-Time Compilation for Dalvik Virtual Machine

4.1 Android and Dalvik VM Execution

In this section, we describe the android application execution model and the architecture of the Dalvik VM.

4.1.1 Android Execution Model

Android employs zygote for efficient launching of an app process [23]. 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 initization 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 initizied objects. Each app will run on this process forked from zygote.

4.1.2 Dalvik VM

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 different from that of the JVM [46]. 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. Bytecode optimization based on the static single assignment (SSA) form is performed. Figure 4-1 (a)

shows an example Java source code and Figure 4-1 (b) shows the corresponding DVM bytecode.

(a) Java code

```
class Math {
  public int add(int a, int b) {
    int c = a+ b;
    System.out.println(c)
return c;
}
}
```

(b) Bytecode

```
// 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
7 return v0
```

Figure 4-1 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~v3 in Figure 4-1 (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. This is illustrated in Figure 4-2. The same call interface is used when the JITC is employed, so JITC-generated code does the same job.

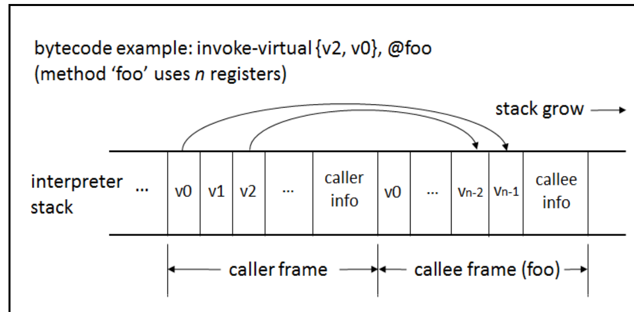


Figure 4-2 Interpreter stack and argument passing in Dalvik VM

4.1.3 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 4-3 illustrates quickening for the virtual method call in Figure 4-1 (b), which replaces the CP index method#5 by an offset in the virtual method table #29.

```
……
4 invoke-virtual {v1, v0}, method#3
=> 4 invoke-virtual-quick {v1, v0}, #29
    (rewritten)
……
```

Figure 4-3 Quickening result by dexopt

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. This is so because interpreted code and JITC-generated code would co-exist in a method, so even if virtual registers were mapping 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.

4.2 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. Figure 4-4 shows the architecture of our AOTC

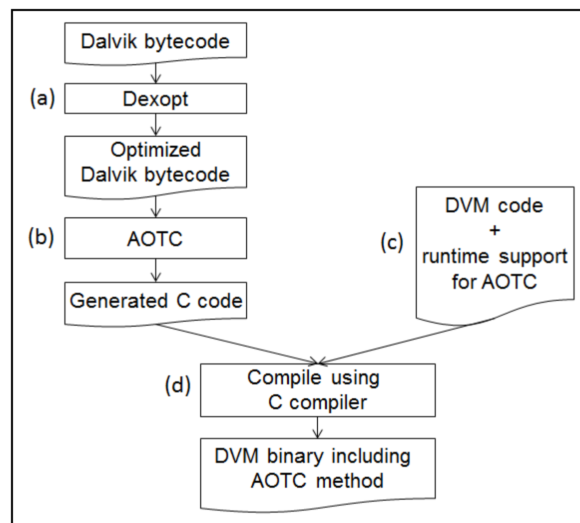


Figure 4-4 Build Dalvik VM with AOTC

We perform all the compilations and optimizations in Figure 4-4 at a server, not at the client device. For the AOTC target methods, we generate optimized bytecode using dexopt in Figure 4-4 (a). The bytecode is translated to C code in Figure 4-4 (b). The DVM source code is updated to use the AOTC methods in Figure 4-4 (c).

Finally, the translated C code is compiled together with the DVM source code to build the new DVM executable in Figure 4-4 (d). The executable is installed in the Android client device.

Those 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.

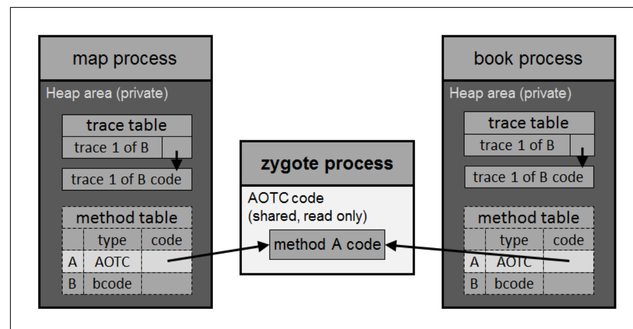


Figure 4-5 Sharing zygote AOTC methods by app processes

Figure 4-5 illustrates the sharing of the native code for the

AOTC method A where only one native code exists in the zygote memory. For the method B not included in the zygote, JITC is used to compile traces, which are duplicated in the heap of the map and the book processes.

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.

4.3 Design and Implementation of AOTC

This section describes the details of our AOTC design and implementation.

4.3.1 Dexopt and Code Generation

We perform AOTC for selected methods in the framework classes and the app classes for the pre-installed apps, because the space overhead will be huge if we AOTC all methods. So we profile app execution to choose hot methods only.

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 those bytecode which cannot be quickened, we should generate C code for CP resolution and linking as well.

4.3.2 C Code Generation

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, as will be described in Section 4.4. We then traverse each IR one by one and generate the corresponding C code. Figure 4-6 shows the C code for the example bytecode in Figure 4-1 (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 4-6. Translated C code for the example in Figure 4-1

Each virtual register used in the bytecode is translated to a C variable (e.g., v0_reg in Figure 4-6), 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 those bytecode 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 ~ v3_reg) as well as temporary variables (result). Arguments are copied to virtual register C variables (v2_reg, v3_reg). In the epilog, C code for returning is generated. In the method body, C code for bytecode is generated. Arithmetic, compare, and branch operations are easily translated to the C code. CP resolution that 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 Section 4.3.4 and 4.3.5

4.3.3 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 4-6). 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 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 interpreter pass arguments using the interpreter call stack. One idea is copying the arguments in the interpreter stack to the four registers and the native stack of the callee, via a separate routine. This is involved with an overhead of iterating thru the argument list, testing their types, and copying the argument data based on their types. Another idea is for the AOTC to follow the argument passing of the interpreter such that the caller AOTC method copy arguments from the virtual register variables to the call stack and the callee AOTC method copy arguments from the call stack to the virtual register variables, which would slow down AOTC-to-AOTC calls seriously.

We employed a mixed call interface that can work efficiently for both AOTC-to-AOTC and interpreter-to-AOTC calls. We make AOTC use physical registers for argument passing, but 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 4-7 (a). Additional arguments are assigned to the interpreter stack of the callee. Interpreted method passes arguments using the interpreter stack assign Figure 4-7 (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 4-7 (c). The return value is handled as in the C interface.

(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 4-7 Method call using a mixed call 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 copies.

AOTC-to-interpreter calls also exist, which are involved with higher overhead, so we need to reduce such calls by choosing the AOTC methods properly based on the profiling. When we generate C code for AOTC-to-interpreter, it save all parameter to

interpreter call stack and call interpreter routine. When the interpreter operate return bytecode, it check caller method type and return to caller method if it is AOTC method.

4.3.4 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, 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 in the interpreter stack that have object reference.

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 reference save slot in the stack frame for each C variable which can possibly have a reference (slot[0] for v0_reg.ref, slot[1] for v2_reg.ref in Figure 4-6).

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 calls, field access, loop backedge, or synchronization points. The DVM can start GC only when every thread waits at one of its GC points since otherwise GC cannot find all reachable objects. So we generate the check code at the GC point if there is any pending GC request from other threads. In Figure 4-6, we generated the check code at the loop backedge (i.e., if GC_Occur(ee){…}).

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 4-6). 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.

4.3.5 Exception Handling

Java provides try blocks 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. One issue is that the exception try block and the catch block might be located in different methods on the call stack. So we need to find the catch block (the exception handler) when an exception occurs.

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

Our AOTC use a simple solution based on exception checks. When an exception occurs in an AOTC method, the method set exception in environment variable (i.e., setNullExcepetion(ee) in Figure 4-6). 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 return to the caller using by a jump (goto LEpilogue_exception in Figure 4-6).

If the caller is an AOTC method, right after a method call we check whether an exception occurred in the callee; 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 repeats 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 4-6).

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. So, exception handling is handled by collaboration of the AOTC methods and the interpreted methods.

4.3.6 AOTC Method Linking

The DVM binary code coexists with the native code for the AOTC methods. As we described in Section 4.1.1, the native code for the AOTC method is shared by all forked processes from the zygote. This is possible after the linking process is done for the AOTC methods; otherwise the interpreter cannot find the native code for AOTC method, and the AOTC method cannot find the correct callee method for a virtual method call. So, we link the native code of the AOTC method at the method table entry of the DVM.

Our AOTC generates an AOTC method 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.

As we described in Section 4.1.1, some important framework classes are loaded in the zygote process. When the DVM loads a class, a class object with a method table is generated and initialized. We modify the DVM to load those framework classes as well which include the AOTC methods. After the framework class is loaded, the zygote process uses the AOTC method 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. When an app process is forked by zygote process, a framework AOTC method is linked naturally in this way. An app AOTC method is not linked yet. We modify the class loading module in the DVM to link app AOTC methods based on the AOTC method table. Figure 4-8 illustrates this linking process for the framework AOTC methods.

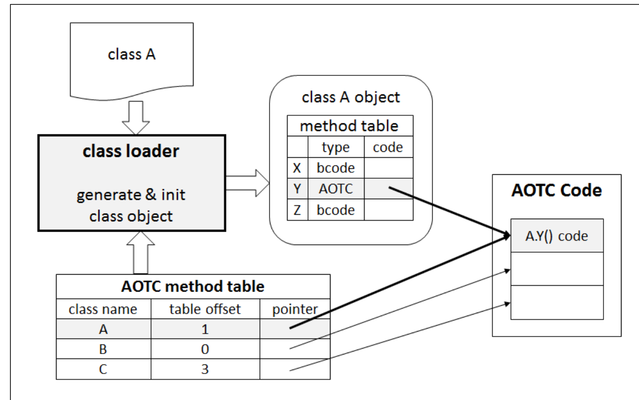


Figure 4-8 Class loading and AOTC method linking

4.4 AOTC Code Optimization

One of the most important benefits of using bytecode-to-C AOTC is that we can use the optimization of the existing C compiler. However, C compiler is not enough since it cannot do Java-specific optimizations, which should be handled by the AOTC. In this section, we discuss such optimization opportunities.

4.4.1 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 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 inlning 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.

4.4.2 Spill Optimization

As we mentioned in Section 4.4, 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.

4.4.3 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, so we need to remove them. The existing 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 checks it already. So our AOTC finds redundant null checks or null checks for this pointer, and remove them.

All of redundant copy bytecodes are removed by the bytecode compiler, dx. Our AOTC generate 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.

4.5 Experimental Result

The previous section described DVM AOTC design and optimization issues. This section evaluates our AOTC.

4.5.1 Experimental Environment

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. 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.

4.5.2 AOTC Target Methods

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 chose 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).

4.5.3 Performance Impact of AOTC

We first measured the DVM AOTC performance. Figure 4-9 shows the performance of the DVM AOTC performance compared to that of DVM JITC as a basis.

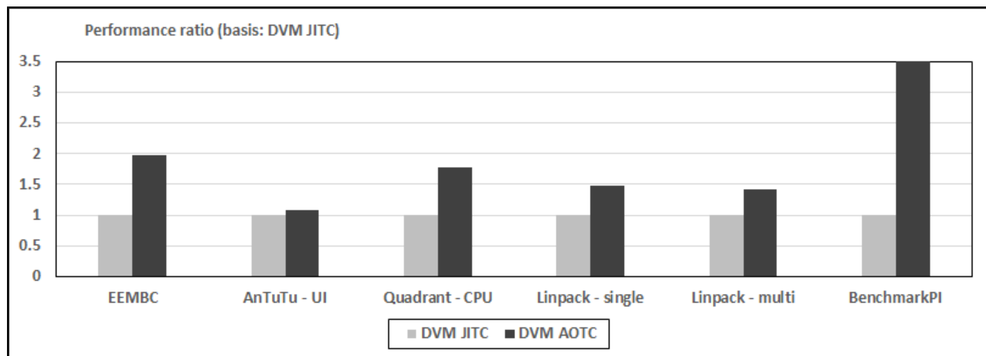


Figure 4-9 Dalvik AOTC performance in benchmark

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 4-10 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.

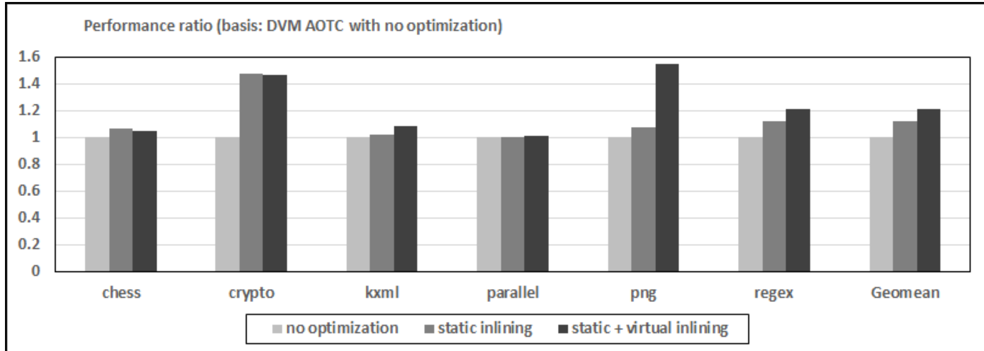


Figure 4-10 Performance impact of method inlining

We also evaluated the impact of redundant code elimination and spill optimization. Figure 4-11 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.

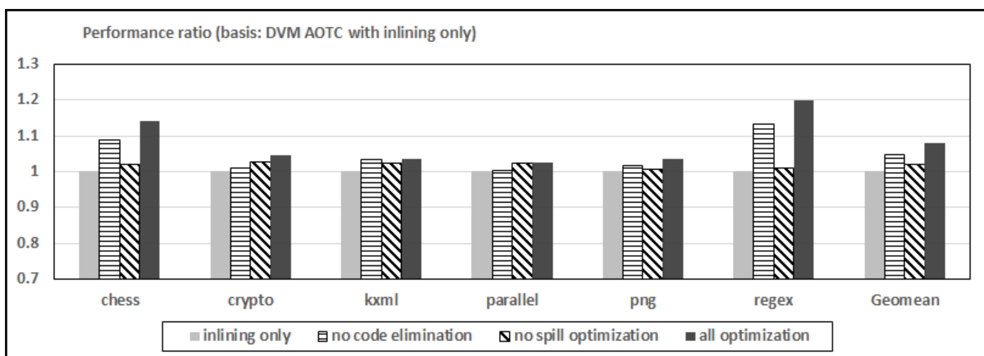


Figure 4-11 Performance impact of optimizations

4.5.4 DVM AOTC vs. ART

Android introduced the ART (Android RunTime) in its 4.4 version (KitKat), which can replace the existing DVM [45]. 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 generate code with weak method-based optimizations, so the code quality still has a problem.

We attempt to evaluate our DVM AOTC compared to ART. Figure 4-12 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%.

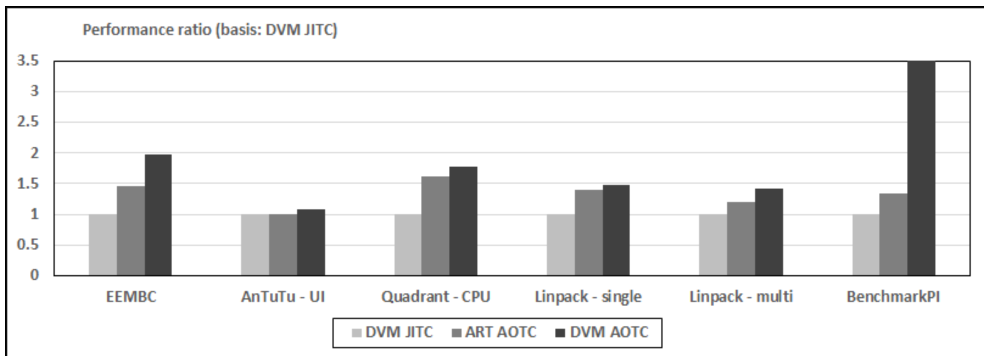


Figure 4-12 Comparison of Dalvik AOTC with ART

Chapter 5 Selecting Ahead-of-Time Compilation Target Methods for Hybrid Compilation

5.1 Hybrid Compilation on DTV

Our target DTV Java platform is the ACAP, a standard terrestrial DTV and IPTV platform in Korea. It is similar to other Java-based standard platforms (OCAP for cable TVs and DVB-MHP for satellite TVs). The ACAP middleware on the DTV set-top box is primarily based on GEM (globally executable MHP), the common Java-based application environment across MHP-based platforms.

For data broadcasting in the ACAP, DTVs receive an xlet application. Each channel has a different xlet application, so if the channel is switched, a new xlet application is downloaded. When the DTV is turned on, the JVM starts and a Java program called an application manager is initiated. Then, the xlet application for the current channel will be executed with ACAP middleware.

Our target DTV platform employed an open source version of Oracle's Connected Device Configuration (CDC) JVM, called PhoneME Advanced. We constructed hybrid compilation on DTV by implementing AOTC for PhoneME Advanced. Generally, AOTC generate better code than JITC by offline translation. But we cannot AOTC xlet classes because of downloaded applications. So we can think that handle middleware classes by AOTC and xlet classes by JITC.

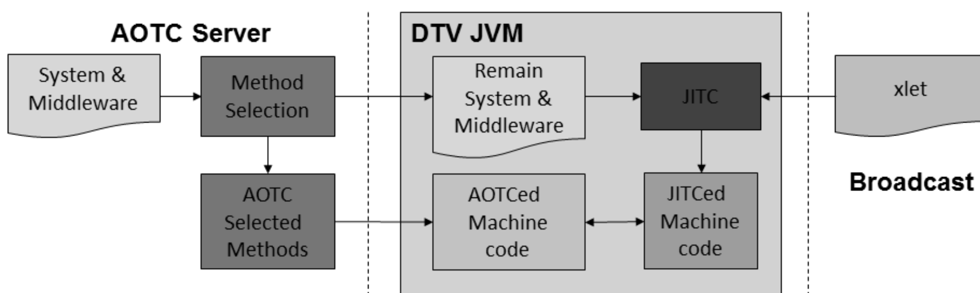


Figure 5-1 The architecture of DTV hybrid execution

The Java VM is a typed stack machine. Each thread of execution has its own Java stack where a new stack frame is pushed when a method is invoked and is popped when it returns. A stack frame includes method/frame information, local variables, and the operand stack. Interpreter operates all computations on the operand stack and saves temporary results in the local variables, so there are many pushes and pops.

Our target JVM, PhoneME Advanced includes an adaptive JITC based on Oracle' s HotSpot technology, where a Java method is first executed by the interpreter but when it is detected as a hot spot, it is compiled to machine code and saved at the code cache in the JVM, which is executed thereafter. To find hot spot methods, interpreter collect additional information, such as method invoke count and taken number of backward branch.

Interpreter and JITC have same calling convention. When a method call is made, actual parameters are already pushed on the operand stack by the caller. These become local variables of the callee, and are followed by the callee' s other local variables, method/frame information of the callee, and the callee' s operand stack. When a method returns, the callee' s stack frame is popped and the return value is copied from the callee' s operand stack to the top of the caller' s operand stack.

JITC use optimization techniques for generating machine code in runtime. The interpreter collects runtime information to support optimization such as method inlining.

We employed a bytecode-to-C server-AOTC. The bytecode-to-C server-AOTC translates the bytecode into the C code with java-specific optimizations, which is then compiled using gcc in server. Machine code generated by AOTC is merged in JVM binary.

In our AOTC, each local variable and operand stack slot is translated into a C variable with a type name attached. For example, a reference-type operand stack slot 0 is translated to s0_ref. We then translate each bytecode to a corresponding C statement, while keeping track of the operand stack pointer. For example, aload_1 which pushes a reference-type local variable 1 onto the stack is

translated into a C statement s0_ref=l1_ref; if the current stack pointer is zero when this bytecode is translated.

The calling convention of our AOTC basically follows the format of a regular C function call. That is, a method call in the bytecode is translated into a C function call whose name is composed of the Java class name and the method name (similar to a JNI method naming convention). The argument list consists of an environment variable for capturing the JVM state, followed by regular C variables corresponding to the argument stack locations at the time of the call. Such a C function call will be compiled and optimized by gcc and arguments will be passed via registers or the C stack, so AOTC calls will be much faster than JITC calls or interpreter calls.

AOTC call interface has good performance for calling other AOTC method. But it has overhead in method call between JITC and AOTC method, because of additional operation. To improve performance in method calls, we generate wrapper function to copy arguments and return value for each AOTC method. When JITC method call AOTC method, JVM uses wrapper function. If JVM use bridge function for common JITC to AOTC call, bridge function should check argument size and return value type of callee method. But wrapper function can reduce overhead to check feature of callee method.

AOTC basically uses optimization of C compiler for generated machine code quality. But C compiler cannot optimize Java specific optimization, such as null check elimination. So we implement Java specific optimization and use in C code generation. And AOTC uses profiling data for method inlining because AOTC cannot use runtime data.

5.2 Hybrid Compilation on Android Device

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. Android employs its own virtual machine to execute Java applications, called the Dalvik virtual machine (DVM).

We employ hybrid compilation by merging DVM with our AOTC for AOTC. In Android platform, we can adopt AOTC for framework and pre-installed apps. Because user downloaded app cannot be handled our AOTC, it should be handled by DVM interpreter and JITC.

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 different from that of the JVM. 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.

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. 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. But the length of the trace is too short to generate efficient code.

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.

We employed a bytecode-to-C server-AOTC. The unit of compilation is method. The bytecode is translated to C code, and translated C code is compiled together with the DVM source code to

build the new DVM executable. The executable is installed in the Android client device.

Each virtual register used in the bytecode is translated to a C variable. We also perform Java specific optimization, such as method inlining, and null check elimination.

For calling convention, 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. Additional arguments are assigned to the interpreter stack of the callee.

5.3 AOTC for Hybrid Compilation

5.3.1 AOTC Target Methods

We can think that we adopt AOTC for all methods installed in device easily. Generally, AOTC increases space overhead because AOTC translate Java methods' to machine code from bytecode, virtual ISA. But number of target methods for real device are huge. For example, there are over 20000 middleware methods in our DTV environment.

Our bytecode-to-C AOTC compile and link all the translated C code together with the VM source code into a single static executable. This can get efficiencies when there are many calls and accesses across compilation units or to the VM functions since they resolved and linked at binary generation. We can compile each of translated C code and load dynamically, but it will suffer performance down because it should do resolving and linking at runtime. On the other hand, our build environment for DTV has size limitation to compile and link C function translated from Java method. So we cannot AOTC all pre-installed methods in our C compiler and linker. As a result, we should select AOTC target methods in pre-installed Java methods.

5.3.2 Case Study: Selecting on DTV

We need to analysis hybrid compilation to find heuristic AOTC method selection. As on case study, we select some middleware Java method and adopt AOTC experimentally.

Our target DTV set-top box includes a 333MHz MIPS CPU with a 128MB memory. Its software platform has the Oracle's PhoneME Advanced MR2 version with advanced common application platform (ACAP) middleware, running on the Linux with kernel 2.6.12. We use real xlet application in Korea.

There are three terrestrial TV stations in Korea, each of which broadcasts a different xlet application. We designate them as A, B, and C in this paper. A and B xlets have news, weather, traffic, and stock menu items, while C xlets have news and weather only (other menu items of C xlets are excluded due to the difficulty of measuring the running time). We are primarily interested in the running time of displaying the chosen information on the TV screen when each menu item is selected using the remote control.

To select AOTC method, we use JVM JITC in DTV. At first, we used original JVM in DTV which is not merged with AOTC (full-JITC) to execute xlet applications. Then we got profile data which include JIT compiled method list full-JITC in xlet execution. We define these methods as 'hot methods'. For hybrid compilation, we adopt AOTC to hot methods except xlet application methods. We can compare hybrid compilation based on hot methods AOTC with full-JITC.

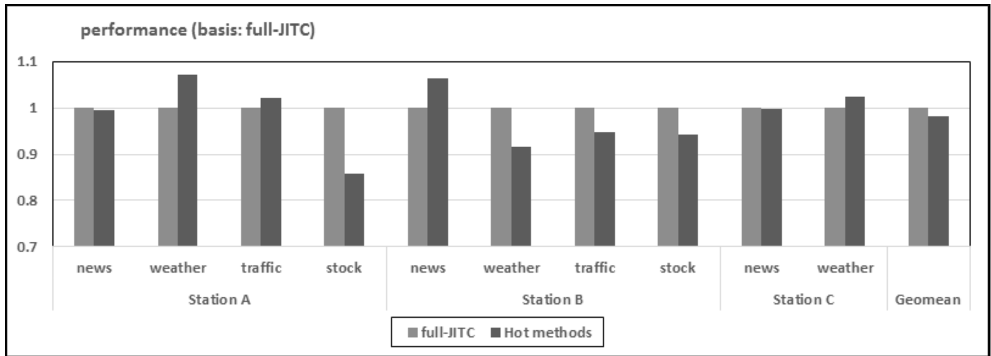


Figure 5-2 Hybrid using hot methods AOTC performance on DTV

Figure 5-2 shows performance of hybrid compilation and full-JITC when the DTV run xlet application. In spite of hybrid compilation got better performance in some cases, we found that hybrid compilation got less performance than full-JITC in most of test items. This result is due to hybrid compilation has call overhead between JITC and AOTC method, and performance gain from AOTC is not enough. As additional experiment, we found that the ratio of call count of AOTC method is less than 40% in total call count.

JITC and AOTC uses different calling convention. In AOTC, call interface uses native stack such as general C function. But in interpreter and JITC uses interpreter stack to pass arguments and return value. So in hybrid compilation which occur method call between JITC and AOTC frequently need additional operation to matching calling convention.

When a JITC method calls an AOTC method, a stack frame for the AOTC method is first pushed on the interpreter stack and is marked as an AOTC frame. Then, parameters in the interpreter stack of the JITC method are copied into registers (and into the C-stack if there needs). Finally, a jump is made to the AOTC method entry. When an AOTC method calls a JITC method, a new stack frame is pushed on the interpreter stack, and the parameters that are in C variables are copied to the operand stack and a jump is made to the JITC method. In our DTV hybrid compilation, JITC to AOTC method call has 3 times overhead and AOTC to JITC method call has 8.5 times overhead than JITC to JITC method call.

In this result, we found that we need to compile enough methods and select to reduce method call overhead.

5.4 Method Selection Using Call Chain

To improve performance, we need to adopt AOTC more methods. First, we can select methods that have method call count over certain threshold. Or we can find more hot methods by modifying JITC threshold in full-JITC execution. But these heuristics do not consider method call overhead. To consider method call overhead, we select AOTC method based on method call chain.

At first, we got profile data from full-JITC environment by executing applications. It has basic data such as hot methods, method call count. To get method call chain information, we collected caller-callee relation between called methods and call count between caller-callee methods.

From profile data, we made method call graph that has called method as node, edge as caller-callee relations, and edge weight as call count caller-callee. Then we selected AOTC target method using method call graph.

In method call graph, we select hot methods. Then we select caller methods of hot methods and trace recursively. And we select callee methods of hot methods and trace recursively. But we trace call chain if edge weight has value over threshold. Figure 4-3 shows example of method call graph.

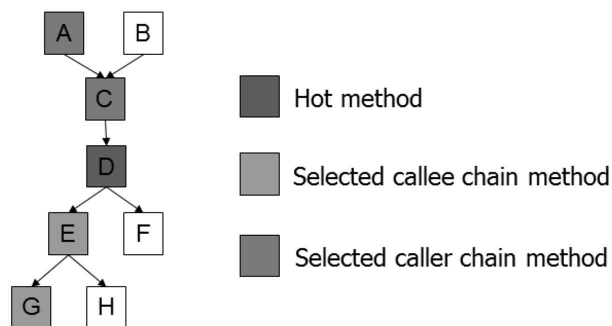


Figure 5-3 Method call graph and method selection

5.5 Experimental Result

5.5.1 Experimental Environment

We build hybrid compilation in DTV and android device. To build hybrid compilation in DTV, we use environment introduced in chapter 3. We experimented with Android KitKat (version 4.4.1) for which we implemented hybrid compilation. The experiments were performed on a tablet called Nexus 7. It has a 1.9GHz quad-core ARM Cortex-A15 CPU with 2GB memory. For the android experiment, we used popular real android app: book, calendar, gmail, weather, clock, twitter, line.

It is not simple to measure the performance in real app because it differs depending on user inputs (events). So we measure real app performance based on the app’s loading time only. We assume that app’s loading is started when the app call the framework method android.app.Activity.onCreate(). This method is called always when an app starts. And, app’s loading is finished when the app call android.app.Activity.onResume(). This method is called when an app is ready to start interactions with the user.

We cannot use hot methods selection heuristic in DVM which is the same as JVM because DVM uses trace-based JITC. We use the Traceview tool included in the Android for method profiling, but it does not work for the benchmark app precisely because it can collect the profile data only with a limited size. So we modify this profile tool to collect larger profile data. We choose hot methods based on the portion of it running time. The sum of the running time of the chosen hot methods covers the 40% portion of the total running time in all Java methods, except for native method (such as JNI and native platform methods).

To compare AOTC target method selection heuristic, we implement other heuristic for AOTC method selection. Using method call count, we select hot methods and methods that have call count over certain threshold. And we get warm methods by modifying JITC threshold in JVM in DTV and get warm methods. In

android device, we select warm methods using profile data. We generate similar binary size for each method selection heuristic.

5.5.2 Performance Impact

We compare performance of hybrid compilation in different method selection heuristic.

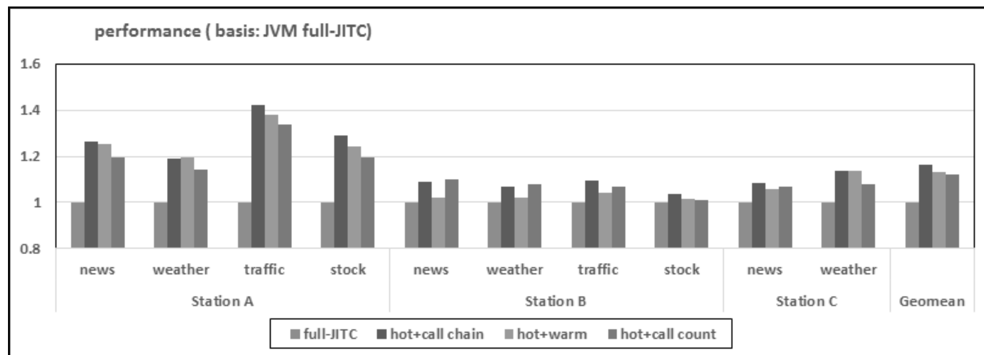


Figure 5-4 Hybrid performance in DTV xlet application

Figure 5-4 shows the performance in DTV xlet application. All heuristics have better performance than full-JITC. Call chain heuristic improves 16% than full-JITC. Call count heuristic improves 12%, and more hot method heuristic improve 13% than full-JITC.

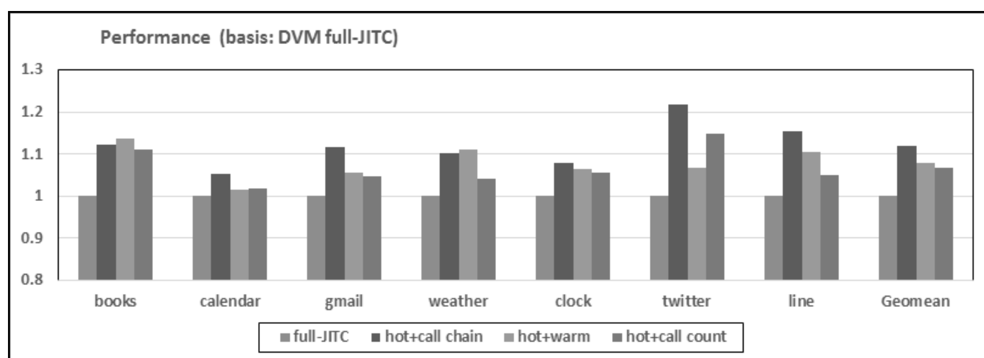


Figure 5-5 Hybrid performance in android app loading

Figure 5-5 shows the performance of hybrid compilation in android app loading compared to that of full-JITC as a basis. Call chain heuristic is 12% faster than full-JITC. More hot method

heuristic improves 8%, and call count heuristic improves 7% than full−JITC. As a result, heuristic based on call chain shows better performance than other heuristics.

We compare compiled method at each heuristic. Table 5−1 shows number of selected methods, and difference compared to call chain heuristic. In DTV, 30% of selected methods in other heuristics are different compared to call chain heuristic. And in android, 45% of selected methods in hot method heuristic, and 67% of selected methods in call count heuristic are different compared to call chain heuristic. Especially in android, call count heuristic have much more selected methods. We select method to have similar space overhead in each heuristics. Because call count heuristic select many small size methods, it select more methods. Basically AOTC generate better quality of machine code than JITC, but small size methods have less opportunity to optimize in AOTC. This becomes one reason that call chain heuristic has better performance than call count heuristic in android in spite of call count heuristic adopt AOTC more methods.

Table 5−1 Number of selected methods and different methods with call chain heuristic

	Heuristic	No. of Methods	different methods
DTV	Call Chain	834	-
	Hot methods	816	237
	Call Count	820	254
Android	Call chain	699	-
	Hot methods	725	325
	Call count	1325	891

We evaluate the impact of call overhead. We count JITC−to−AOTC call, AOTC−to−JITC call count by each heuristic in android device. Figure 5−6 shows JITC−to−AOTC and AOTC−to−JITC call count and JITC−to−AOTC call count at each app as basis. At first, we compared call chain heuristic and more hot method heuristic. Call chain heuristic has similar or more total count than

more hot method heuristic. But more hot method heuristic has more AOTC-to-JITC call count. Because AOTC-to-JITC call overhead bigger than JITC-to-AOTC call overhead, performance of call chain heuristic is better than more hot method heuristic. When we compare call chain heuristic and call count heuristic, call count heuristic has over 25% more total count than call chain heuristic.

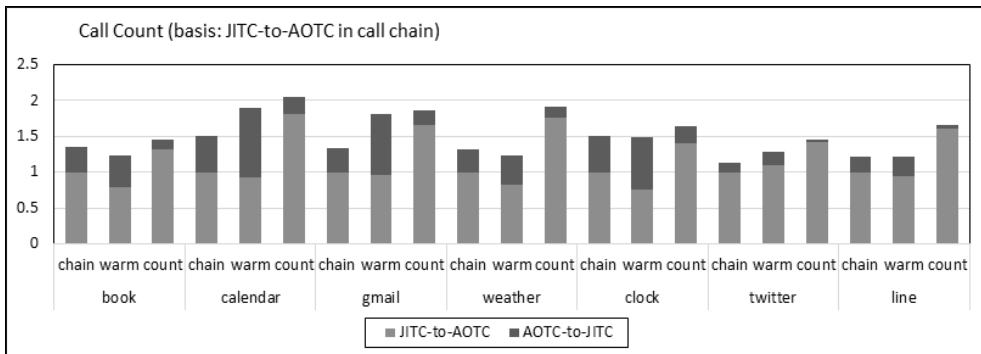


Figure 5-6 JITC-to-AOTC and AOTC-to-JITC call count in android app loading

Chapter 6. Related Works

We can say that existing JITCs and AOTCs already employ some form of a hybrid execution environment. For example, most JITCs are using adaptive compilation where the interpreter is used for hot spot detection. In addition, most AOTCs require the interpreter for supporting dynamic class loading. However, this interpreter execution is simply for supplementing JITC or AOTC, so this is not a genuine form of a hybrid environment.

QuickSilver is a quasi-static compiler developed for the IBM's Jalapeno system for servers [10]. It saves all JITC methods in the files at the end of execution, and loads them directly without JITC when they are used in later execution. Therefore, it employs a form of AOTC, yet it is based on a JITC, not on a separate offline compiler. There are no interface issues between JITC methods and AOTC methods in this hybrid environment because their machine code is generated by the same compiler, unlike our AOTC-JITC hybrid environment. However, the benefit is merely reducing the JITC overhead without any improvement for the AOTCed code quality. Actually, there still is the class loading overhead of constant pool (CP) resolution or building class data structures for the AOTCed classes, unlike our bytecode-to-C AOTC where the translated C code is compiled together with the JVM source, hence no such overhead (already-resolved CP entries and class data structures are romized in the JVM). This also obviates any loading process of the AOTC machine code into the memory, while QuickSilver can suffer from a relocation and CP resolution overhead during the loading process.

There is a commercial JVM that takes a similar approach to QuickSilver. Oracle's PhoneME Advanced has an AOT option which allows compiling a list of pre-chosen methods using its JITC module and saves their machine code in a file on a persistent storage. When the JVM starts officially, it will use the compiled machine code directly without interpretation or JITC, when they are

executed. There are no interface issues between AOT methods and JITC methods since they are based on the same compiler as in QuickSilver. However, unlike QuickSilver, the machine code is generated statically irrespective of program execution, so the AOT-generated code is not exactly the same as the JITC-generated code but worse. For example, the AOT inlining is inefficient since it is not based on the runtime profile information unlike JITC. Moreover, a few code optimizations in JITC are disabled due to the relocation and code patch issues. Fundamentally, JITC would not perform any time-consuming optimizations since the compilation overhead is part of the running time, so a JITC-based AOTC is likely to underperform the bytecode-to-C AOTC, as indicated by the comparison graph of the full-AOTC and the full-JITC in Figure 2-1.

Jikes RVM includes two kinds of compilers: a baseline compiler and a tiered set of optimizing compilers [11]. The baseline compiler translates bytecode into machine code before execution starts, while the optimizing compilers re-compile hot methods with optimizations at runtime. So, the baseline compiler and the optimizing compiler correspond to an AOTC and a JITC, respectively. However, what the baseline compiler generates is machine code corresponding to what the interpreter does, so the relationship between the two compilers is more like our JITC-interpreter, not our JITC-AOTC.

The .NET platform of Common Language Runtime VM also employs a JITC, which translates MSIL (MS intermediate language) into machine code [13]. It is also possible to invoke the JITC offline so as to compile ahead-of-time. This JITC-based AOTC can save only the JITC overhead, as QuickSilver can.

Comparison of stack-based bytecode and register-based bytecode was first made in the context of interpretation by Shi et. al [28]. They converted the stack-based Java bytecode to their own register-based bytecode and compared the bytecode count and size as well as the performance for the SPECjvm98 benchmarks. They reported that the static/dynamic code size increase by

26/25% while the static/dynamic count decreases by 44%/46%. The code size is somewhat smaller than our result, and it appears to be their bytecode format which is leaner than the Dalvik' s.

There is a web page in Oracle where the performances of Java SE and Android are compared [39]. It uses Android 2.2 Froyo and Java SE 1.6 on a beagle board with Cortex-A8 and a Tegra2 board with ARM Coretex-A9. They experiment with Caffeinemark and SCIMark and their result shows that Java SE is 2~3 times faster than Android, which is consistent to our results in EEMBC. However, their evaluation is not complete as in this paper.

As to the trace-based compilation, Dynamo would be the first, full-fledged trace-based system where traces can span beyond branches, join points, and even function calls [40]. It is for dynamic optimization of PA-RISC binaries, not for JITC.

There is a trace-based JITC for embedded Java called the HotPath VM [31]. Unlike the Dalvik JITC, a trace can span beyond branches and method calls until a branch target becomes the trace entry, so a path in a loop even across method calls can be compiled at once as a single trace. A trace-based JITC was proposed for IBM J9 which originally uses a method-based JITC [41], yet its performance is worse than the method-based JITC. This also indicates that a trace-based JITC is not as competitive as a method-based JITC.

A JavaScript engine called TraceMonkey used in the Firefox browser also employ a trace-based JITC [42]. It uses a similar technique for tracing to the HotPath VM. However, a method-based JITC has also been published to replace it [43].

Bytecode-to-native server AOTC for DVM was presented in [47]. 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 [48]. This approach also used profile data for selecting AOTC target methods. But this research is for compile general apps, not android framework and pre-installed apps.

Chapter 7 Conclusion

This paper proposes a hybrid compilation environment with both AOTC and JITC for accelerating this dual-component software architecture.

We performed a case study by merging an existing JITC and AOTC, yet found some performance anomaly with such a hybrid environment. Our analysis shows that the anomaly is primarily due to method calls between JITC methods and AOTC methods which cause serious call overhead. An optimization to reduce the call overhead could reduce the anomaly, but the desired hybrid performance with our environment appears to be achievable only when enough running time is spent on the AOTC methods. Fortunately, the middleware and system classes are dominantly executed in a DTV environment, justifying the proposed hybrid compilation.

Dalvik VM employs register-based bytecode instead of stack-based bytecode used in the conventional JVM, which reduces the fetch overhead, leading to slightly better interpretation performance. It also employs trace-based compilation instead of method-based translation of the JVM, which suffers from worse optimizations and chaining overhead due to too-short traces, leading to much worse JITC performance. Also, Dalvik's trace-based JITC does not reduce the memory overhead nor the compilation time, unlike their claim. We could not observe any significant performance benefit even when we extend the traces and add optimizations. Consequently, we believe Dalvik's trace-based JITC has a severe performance problem in its current form.

Despite Dalvik's serious performance issues, we do not experience any critical problems in running the Android apps. Our analysis of real Android apps indicates that the Dalvik portion in the total running time is not dominant, which allows the slow Dalvik VM not to affect the Android performance seriously. In fact, Android apps lack hot spots unlike benchmarks, requiring a faster warm spot

detection or ahead-of-time compilation, in addition to improving compilation technique itself.

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.

We cannot AOTC all middleware and framework methods in DTV and android device for hybrid compilation. By case study on DTV, we found that we need to adopt AOTC enough methods and reduce method call overhead.

We proposed AOTC method selection heuristic using method call chain. We selected hot methods and call chain methods using profile data. Our heuristic based on method call chain got better performance than other heuristics such as based on method call count or ratio of method runtime.

Bibliography

[1] J. Aycock. A Brief History of Just-in-Time, ACM Comput. Surv. 35 (2003) 97-113.

[2] 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, Proc. USENIX Conf. Object-Oriented Technologies and Systems (Portland, Oregon, 1997), p. 3.

[3] G. Muller, B. Moura, F. Bellard and C. Consel, "Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code", Proc. USENIX Conf. Object-Oriented Technologies and Systems (Portland, Oregon, 1997), p. 1.

[4] M. Weiss, F. Ferrière, B. Delsart, C. Fabre and F. Hirsch, TurboJ, a Java Bytecode-to-Native Compiler, Proc. ACM SIGPLAN Work. Languages, Compilers, and Tools for Embedded Systems (1998), pp. 119-130.

[5] A. Varma and S. S. Bhattacharyya, Java-through-C Compilation: An Enabling Technology for Java in Embedded Systems, Proc. Conf. Design, Automation and Test in Europe (2004), p. 30161.

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

[7] T. Ogasawara, H. Komatsu and T. Nakatani, A study of exception handling and its dynamic optimization in Java, Proc. ACM SIGPLAN conf. Object oriented programming, systems, languages, and applications (USA, Tampa Bay, 2001), pp. 83-95.

[8] D. Jung, S. Bae, J. Lee, S. Moon and J. Park, Supporting Precise Garbage Collection in Java Bytecode-to-C Ahead-of-Time Compiler for Embedded Systems, Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems (Korea, Seoul, 2006), pp. 35-42.

[9] D. Jung, J. Park, S. Bae, J. Lee and S. Moon, Efficient Exception Handling in Java Bytecode-to-C Ahead-of-Time Compiler for

Embedded Systems, Comput Lang Syst Str 34(4) (2008) 170-183.
[10] M. Serrano, R. Bordawekar, S. Midkiff and M. Gupta, Quicksilver: A Quasi-Static Compiler for Java, Proc. ACM Conf. Object-Oriented Programming Systems, Languages, and Applications (2000), pp. 66-82.
[11] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo and V. Sarkar, The Jikes Research Virtual Machine project: Building an open-source research community, IBM Syst. J. 44(2) (2005) 399-417.
[12] R. Wilkes,
http://msdn.microsoft.com/msdnmag/issues/05/04/NGen
[13] UW Dynamic Compilation Project,
http://www.cs.washington.edu/research/projects/unisw/DynComp/www/
[14] S. Lee, S. Moon, S. Kim, Enhanced Hot Spot Detection Heuristics for Embedded Java Just-in-Time Compilers, Proc. ACM SIGPLAN/SIGBED 2008 Conf. Languages, Compilers, and Tools for Embedded Systems (USA, Tucscon, 2008), pp. 13-22.
[15] D. Jung, S. Moon, S. Bae, Design and Optimization of a Java Ahead-of-Time Compiler for Embedded Systems, Proc. Int. Conf. Embedded and Ubiquitous Computing (China, Shanghai, 2008), pp. 169-175.
[16] A. Nilsson and S. Robertz, On Real-Time Performance of Ahead-of-Time Compiled Java, Proc. IEEE Int. Symp. Object-Oriented Real-Time Distributed Computing (2005), pp. 372-381.
[17] A. Armbuster, J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka and J. Vitek, Real-time Java virtual machine with applications in avionics. ACM T. Embed. Comput. S. 7(1) (2007) 1-49.
[18] F. Siebert, Eliminating external fragmentation in a non-moving garbage collector for Java, Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems (USA, San Jose, 2000), pp. 9-17.
[19] M. Fulton and M. Stoodley, Compilation techniques for real-

time Java programs. Proc. Int. Symp. Code Generation and Optimization (2007), pp. 222-231.
[20] T. Schoofs, E. Jenn, S. Leriche, K. Nilsen, L. Gauthier and M. Richard-Foy, Use of PERC Pico in the AIDA avionics platform. Proc. Int. Work. Java Technologies for Real-Time and Embedded Systems (Spain, Madrid, 2009), pp. 169–178.
[21] F. Pizlo, L. Ziarek, E. Blanton, P. Maj and J. Vitek, High-level programming of embedded hard real-time devices. Proc. European Conference on Computer systems (France, Paris, 2010), pp. 69-82.
[22] D. Jung, S. Moon, and H. Oh, Hybrid Java Compilation and Optimization for Digital TV Software Platform, Proc. Int. Symp. Code Generation and Optimization, (Canada, Toronto, 2010), pp. 73-81.
[23] D. Bornstein. Dalvik VM internals, http://sites.google.com/site/io/dalvik-vm-internals
[24] J. Gosling, B. Joy, and G. Steele, The Java Language Specification Reading, Addison-Wesley, 1996.
[25] 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
[26] T. Kotzmann et. al, Design of the Java HotSpot™ client compiler for Java 6. ACM Transactions on Architecture and Code Optimization (TACO), v.5 n.1, pp.1-32, May 2008
[27] Android Open Source Project. .dex - Dalvik executable format. http://source.android.com/tech/dalvik/dex-format.html
[28] Y. Shi, K. Casey, M. A. Ertl, D. Gregg. Virtual machine showdown: Stack versus registers. ACM Transactions on Architecture and Code Optimization (TACO), v.4 n.4, pp.1-36, January 2008
[29] M. Arnold, S.J. Fink, D. Grove, M. Hind and P.F. Sweeney. A Survey of Adaptive Optimization in Virtual Machine. IBM Research Report RC23143, May 2004
[30] Sun Microsystems. CDC Runtime Guide for the Sun Java Connected Device Configuration Application Management System version 1.0. (Page 58),

[31] A. Gal, C. W. Probst, M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. International conference on Virtual execution environments, June 2006.
[32] R. Phelan, Improving ARM code Density and performance: New Thumb extension to the ARM architecture (Thumb-2 white paper), June 2003.
[33] T. R. Halfhill. ARM strengthens Java compilers: New 16-Bit Thumb-2EE Instructions Conserve System Memory. Microprocessor Report, July 2005.
[34] Cortex-A9 Technical Reference Manual, revision: r2p0
[35] PhoneME Advanced, http://wiki.java.net/bin/view/Mobileandembedded/PhoneMEAdvanced
[36] Oracle, Java ME Technology – CDC, http://www.oracle.com/us/technologies/java/cdc-137762.html
[37] Google, What is Android? http://developer.android.com/guide/basics/what-is-android.html
[38] EEMBC GrinderBench Benchmarks. http://www.grinderbench.com
[39] Java SE Embedded Performance Versus Android 2.2, http://blogs.oracle.com/javaseembedded/entry/how_does_android_22s_performance_stack_up_against_java_se_embedded
[40] V. Bala et. al, Dynamo: a transparent dynamic optimization system. ACM conference on Programming language design and implementation (PLDI), pp.1-12, June 2000
[41] P. Wu et. al, Reducing trace selection footprint for large-scale Java applications without performance loss. ACM international conference on object oriented programming systems languages and applications (OOPSLA), Oct. 2011
[42] A. Gal et. al, Trace-based just-in-time type specialization for dynamic languages. ACM conference on Programming language design and implementation (PLDI), June 2009
[43] JagerMonkey project, https://wiki.mozilla.org/JaegerMonkey
[44] CaffineMark 3.0, http://www.benchmarkhq.ru/cm30/
[45] Android Open Source Project, Introducing ART,

http://source.android.com/devices/tech/dalvik/art.html
[46] Android Open Source Project, Bytecode for Dalvik VM, https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html
[47] Y. Lim, S. Parambil, C. Kim, and S. Lee, "A Selective Ahead-of-Time compiler on Android Device", in Proc. of ICISA, 2012.
[48] 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.

초 록

많은 내장형 자바 소프트웨어 플랫폼에는 기기에 미리 설치되어 있거나, 서비스 공급자로부터 다운로드 받는 두 가지 종류의 자바 클래스를 수행한다 성능 향상을 위해, 미리 설치되어 있는 자바 클래스는 선행 컴파일러로, 다운로드 받는 클래스는 즉시 컴파일러로 처리하는 방식이 바람직하다. 이 논문에서는 선행 컴파일러와 즉시 컴파일러로 구성된 하이브리드 자바 컴파일 방식을 제안한다. 하이브리드 컴파일의 효과를 확인하기 위해 일반적으로 사용되는 방식의 선행 컴파일러와 즉시 컴파일러를 합쳤다. 그러나 선행 컴파일러와 즉시 컴파일러를 단순히 합치는 방식으로는 즉시 컴파일러만을 사용한 것보다 나쁜 성능을 보이는 것을 벤치마크 수행 결과에서 확인할 수 있었다. 분석 결과 즉시 컴파일 메소드와 선행 컴파일 메소드간의 호출의 비효율성이 성능에 영향을 주는 것을 찾아내었다. 그리고 하이브리드 컴파일에서 성능을 얻기 위해서는 즉시 컴파일 메소드와 선행 컴파일 메소드의 분포 또한 중요하다는 것을 실험을 통해 확인하였다.

안드로이드 플랫폼에서는 달비 가상 머신을 통해 자바를 수행하는데, 이는 전통적인 자바 가상 머신과 약간 차이가 있다. 차이점을 확인하기 위해, 우리는 달비 가상 머신을 자바 가상 머신인 HotSpot 가상머신과 비교 분석하였다. HotSpot 은 스택 기반 바이트코드를 사용하는 데에 비해 달빅은 레지스터 기반 바이트코드를 사용한다. 또한 HotSpot 은 메소드 기반 즉시 컴파일러를 사용하는 데에 비해 달빅은 추적 기반 즉시 컴파일러를 사용한다. 성능 확인을 위해 동일한 기기에 달빅과 HotSpot 을 활용해 벤치마크를 수행하였다. 수행 결과 달빅에 비해 HotSpot 이 3배 빠른 것을 확인할 수 있었다. 분석결과 달빅 즉시 컴파일러의 코드 질이 떨어지고, 짧은 추적으로 인한 연결 코드의 영향임을 확인할 수 있었다. 또한 실제 앱을 수행한 결과 핫스팟을 특정하기 어렵다는 것을 확인할 수 있었다. 우리는 달빅에 하이브리드 컴파일 구축을 위해 즉시 컴파일러를 구현하였다. 즉시 컴파일러를 거친 코드는 안드로이드의 zygote 방식과 융합되어 동작하며, 쓰레기 처리 동작과 예외 처리 동작을 수행할 수 있다. 메소드 기반 컴파일과 최적화을 활용하여 수행 오버헤드 없이 코드의 질을 높일 수 있다. 벤치마크에서는 달빅 즉시 컴파일러에 비해 높은 성능 향상을 확인할 수 있었다.

안드로이드 기기와 디지털 텔레비전(DTV)에 하이브리드 컴파일을 구축할 때, 모든 시스템 및 미들웨어를 선행 컴파일 할 수 없다는 문제가 있다. DTV 에서의 선행 실험을 통해, 선행 컴파일 메소드에서의 수행비중을 높이면서 호출 오버헤드를 줄일 수 있는 선행 컴파일 메소드 선정이 필요하다는 것을 확인하였다. 우리는 메소드의 호출 연결을 활용하여 컴파일 메소드로 선택하는 휴리스틱을 제안하였다. 우리는 핫메소드와 호출 연결을 프로파일 정보를 활용하여 선택하였다. 메소드 호출 연결을 기반으로 한 휴리스틱을 통해 다른 휴리스틱보다 좋은 성능을 얻었다.

주요어 : 하이브리드 컴파일, 즉시 컴파일러, 선행 컴파일러, 자바, 달빅
학 번 : 2008-30228