國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

使用 LLVM 實作 Ahead-of-Time 編譯技術 於 Android NDK

Implement Ahead-of-Time Compilation via LLVM on Android NDK

谷汶翰

Wen-Han Ku

指導教授：廖世偉 博士

Advisor: Shih-Wei Liao, Ph.D.

中華民國 101 年 7 月

July 2012

Acknowledgements

I offer the deepest respect to my advisor, Dr. Shih-Wei Liao, who always provides lots of support on my graduate time, pays patience on my questions and encourages me all the time even though he is busy. Besides, I’m very thankful for Dr. Cheng-Wei Chen and the team members who I meet in my internship. They bring me about lots of impacts in compiler domain and help me learn many skills on practical projects. I would like to thank every member in our lab as well. Everyone has his own talent so that we can learn from each other.

Last but not least, I appreciate all of those who gave me advices and helped me in any respect during the completion of the thesis.

中文摘要

Android 作業系統近年來在智慧型手機及平板電腦上大放異彩，使用者開發的應用程式數量也愈來愈多。開發者可以選擇單純使用 Java 語言，或額外選用與機器相關的原生語言，例如 C/C++，透過 NDK 開發套件，與前者在程式運行時互相合作。由於原生語言與機器高度相關，因此在開發者本機端就必須先決定好要運行在哪一種裝置上，並且將產生出來的動態連結庫放入 APK 裡面。此作法不只限制了可以運行的裝置種類，也讓開發流程變得有不可移植性。

在本篇論文中，我們透過 LLVM 這個編譯器基礎設施的中間表示式，位元碼 (bitcode)，作為一種可移植的發布格式。在開發者本機端，我們透過工具將原生語言轉換為 bitcode，包進 APK 裡面，待下載到裝置後，利用 LLVM 的元件編譯連結成原生的動態連結庫，這個動作是在運行之前做完的，不影響執行時的效率。目前已經將 NDK 現有的範例程式，例如 Hello-jni、Bitmap-plasma、Native-plasma 等，以及其他較複雜龐大的範例，例如 Quake、Quake 2、Quake 3 等第三方的程式，成功運行在現有 Android 裝置上不同的平台，例如 X86、ARM 上面。藉由這個實驗，我們對於傳統的編譯流程有了創新，並且也讓 Android 因為 NDK 編譯機制而造成的碎片化問題得到了解決。

關鍵字：Android、NDK、動態連結庫、LLVM、可移植性、碎片化

Abstract

Android operating system is more popular on mobile phones and tablets in these years. Correspondingly, developer's applications are more and more. Developers can choose Java language, or native language, like C/C++, which can be embedded into Android packge (APK) by NDK, cooperating with the former. Since native language is highly machine-dependent, developers need to decide which targets they want to ship, and put the compiled DSO into APK. This approach limits the number of kinds of devices, and makes the development process not portable.

In this paper, we exploit the intermediate representation of LLVM infrastructure, bitcode, as a portable public format. On the host side, we compile native language source codes to a LLVM bitcode and put it into APK. On the target side, we use LLVM components to compile the bitcode into a DSO, which is done before runtime without affecting runtime performance. Currently we have successfully run the existing samples of NDK, like Hello-jni, Bitmap-plasma, and other third-party samples, like Quake, Quake 2, Quake 3, on the existing various Android platforms, like X86 and ARM. Through this experiment, we innovate on the traditional compilation flow, and solve the fragmentation problem caused by NDK compilation mechanism on Android.

Keywords: Android, NDK, DSO, LLVM, Portability, Fragmentation

Contents

List of Figures

List of Tables

Chapter 1 Introduction

To deploy the application to Android devices, we can write native-language source codes, such as C, C++, assembly, and use the NDK toolchains to build Dynamic Shared Object (DSO) to improve runtime performance of the program [1]. Besides, Android NDK makes it possible for developers to easily reuse legacy code written in C/C++ language [2].

However, we need to specify one or more targets on host before we publish the Android Package (APK) into devices. This is a native language compilation constraint which cannot be avoided. The more targets we want to ship, the bigger package we will have since every target needs one individual DSO. It is such a big burden for current Android ecosystem since below reasons:

1. Fragmentation. Each device could be different targets, and each target could have different ISA and features. There will be more device diversity since Android market share is increasing. There could be some devices cannot run the program. For example, if we don't consider the MIPS's platform and generate its DSO while compiling, the devices using MIPS architecture cannot install this APK.
2. Pool optimization. As noted above, there are many features in each target could be enabled. We should fully optimize the source code according to the target features. For instance, on ARM devices, some can use NEON but some cannot, we should handle that carefully in order to gain good optimization as far as possible. In current solution, if we need to specify the compilation parameter more detail, we need generate different DSO even for one target.

Like the example above, NDK support “armeabi” and “armeabi-v7a” for ARM platforms. This approach doesn’t make sense since Android will run on many platforms in future. We cannot specify all the features on development timing.

3. Non portability. We need to choose targets we want to deploy on before putting the APK into devices. This also implies that if some devices using other architectures, we have to rebuild and republish the APK.

In order to solve these problems, we have to defer target-related compilation steps to device end and keep on-host compilation portable. We cannot deploy a machine-dependent DSO in the APK. Instead, we use LLVM IR.

LLVM is a compiler infrastructure [3]. It was originally an abbreviation of “Low-Level Virtual Machine”, but now it is a brand of one compiler team. It has many components and is able to compile from source code to machine code currently through its intermediate representation (IR), which can be represented in different forms, like memory buffer, a human-readable file or a human-unreadable bitcode [4]. We exploit the last as we want. Although the official bitcode is not portable, we can restrict some functionality and use some workarounds to make it as portable as we need. The details will be explained in later chapters.

Once we have the “portable” bitcode, we can split the compilation into two parts, the host and the device. In the former, we only compile the source code into one bitcode, and enclose it into the APK. We don’t need lots of prebuilt toolchains anymore. Only one step the developers need to do is using Clang [5] to emit LLVM bitcode for the only one phony “Android” target.

On device end, there are many efforts need to do. While we install the APK, we have to use LLVM components to compile the bitcode into the native object file, which will be linked to a DSO successively. Since we postpone some compile and linking stages until the time before running, this approach is well-known as Ahead-of-Time (AOT) [6].

The rest of the paper is organized as follows. Chapter 2 reviews the existing NDK approach. Chapter 3 explains our efforts in detail. Chapter 4 mentions some related works. Chapter 5 points out the next policy in future we want to do. Finally, Chapter 6 summarizes all we did and make a short conclusion.

Chapter 2 Reviews of Android NDK

Android has NDK since 2009. It is a JNI wrapper and has many native components for different targets. Developers who want to gain more performance or need the native side functionality support can make a good use on it [7]. NDK can support “armeabi”, “armeabi-v7a”, “x86” and “mips” architectures until May 2012. It provides a simple way to implement native library into APK to publish on Android devices [8]. We discuss compilation process of NDK and its defect below.

2.1 Compilation Process

NDK has different prebuilt toolchains for different targets. It also prepares different header files and native libraries for different Android platform. NDK developers only need to write a simple Android.mk, choose targets in Application.mk, and then the target-specific toolchain will generate corresponding native static libraries or shared libraries we specified.

As Figure 1 shown, NDK is a pre-pass of SDK, it generates DSOs and SDK toolchain will package them into one APK. Developers don’t need to worry the details of cross-compiling, such like header files, libraries, and complicated compilation process. On device, PackageManager will unpack the APK and copy the ABI-matching DSO into the private application data directory. Users cannot download APKs from Google Market which doesn’t contain the ABI matched.

This approach is a general traditional way to compile native libraries. However, it causes some problems on Android ecosystem. First of all, users expect they can download what they want definitely. For example, users who hold MIPS Android machines cannot download Angry-Birds before NDK r8 (May 2012) since some functions are written in native languages and NDK toolchains don't support MIPS ABI [9].

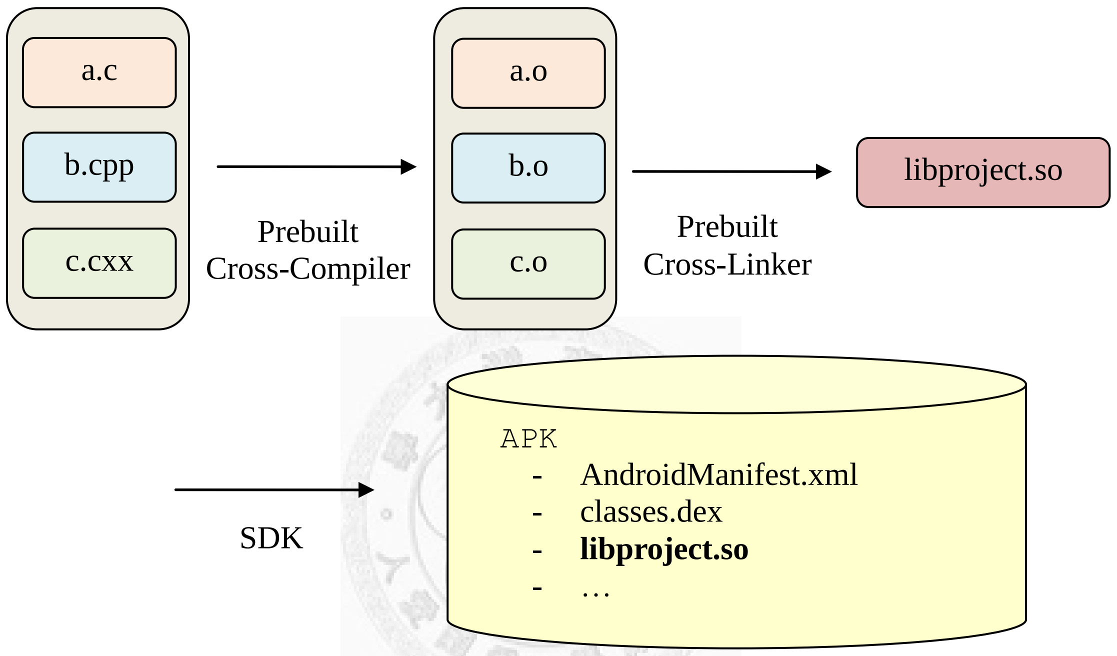

Figure 1 NDK Compilation Process on Host

NDK will cause fragmentation problem on Android platforms. It cannot highlight inherent portability of Android platform, or break that advantage even. Another problem is that official team cannot enumerate all targets and subtargets for all devices using Android in time. This will make fragmentation even worse. NDK developers need to update their projects periodically because Google supports more targets in each revision. They have to rebuild and republish them and notify all users to upgrade the APKs. This is not a good permanently solution on native DSO support. Developers need to solve "toolchains hell" at each update. For example, they might encounter include-path search problem from NDK r4 to r5 [10].

In conclusion, to end users, NDK mechanism will break their anticipation; to developers, NDK will make them spend more meaningless time on development.

2.2 Binary Translation

Suppose we want to achieve our portability goal without changing too many components, we might use a small tool on device and let it translate the binary on demand. Before introducing our approach, we figure out an intuitive solution by binary translation [11]. First we choose one target as "public" platform, for instance, ARM. We install a binary translator for other platforms on devices and only publish the public-version DSO in APKs. When loading the native libraries, Android will make a dynamic binary translation on the DSO and generate native machine codes to execute [12]. However, this makes another big problem that we break the native-language performance demand from NDK developers. There is a paper showing that the slowdown compared to native execution is with an average of 2.02 on a general-purpose dynamic binary translation [13].

How about using a static AOT binary translator? PackageManager will translate public-version machines codes to native machine codes, use a tiny link editor to generate a new native-ABI DSO and install it. The public-version DSO will be dropped after installation time. This is a better solution since it doesn't need the additional run time cost. Nevertheless, AOT binary translator has the same problem as dynamic binary translator. It needs to guarantee the translator and the link editor can generate the best code quality comparable with the original one. That is, it needs powerful optimizations. We cannot compromise the native code efficiency for the portability problem. Besides, this usually needs static-profiling to help generate the better code quality [14]. This also might include problem below.

2.2.1 Target-Specific Problems

Without considering the runtime performance problem, we still have difficulty on translating public-version DSO to native-version one. For example, ARM binary usually mix ARM code, Thumb code and literal pool. Its code section not only contains the data, but also contains more than one kind of code [15].

ARM relocatable object files use *$a*, *$t* and *$d* local symbols to point to the different position of ARM code, Thumb code and literal pool respectively, since they are needed by static linker. However, ARM stripped DSO binary doesn't use these symbols so that we cannot know information until running. All target-specific problems like this cause the goal hard to achieve.

We should use a fundamental solution to overthrow the original NDK approach. Portability is a big issue so that it is worth to make significant changes. We need a complete compiler tool which can generate the best code quality before runtime, rather than using merely a tiny link editor or some binary translator. We choose LLVM since it is a good modular compiler infrastructure. The details will be explained in next chapter.

Chapter 3 Portable NDK

All problems in current NDK is introduced since that it compiles native language to machine codes on host compilation process. It ties the output tightly with native machine environments. This will not benefit Android ecosystem since Android always run on different platforms. We need another way to work around this problem. We choose one other format to spread to improve portability and still keep the runtime performance by running on machine level as original one.

We take LLVM bitcode as our new format. It provides a new phony ISA to loose coupling between high level portability and native machine environments [16]. LLVM bitcode is the key person of LLVM compiler infrastructure. As its intermediate representation, it is planned to encode both features of high-level languages and low-level information. For example, LLVM IR encodes struct, pointer, array, exception handling and with type properties although it is a linear low-level IR [17]. On the other hand, LLVM has developed for many years. It is a mature open-source organization and has steady improvements every day. We combine its power on Android system to make both stronger.

However, LLVM IR may be not portable at all [18]. It has some properties that are determined in frontend translation time. For example, C/C++ language has a keyword, “sizeof”. Programmers can use sizeof(something) to get how many bytes it occupies. It will become a simple magic number in the generated bitcode. The number may change between different targets. Different targets mean different sizes, alignments, ABIs from the same high-level value. LLVM specified this as one item of FAQ [19]. We need to customize the bitcode to make it more portable, which is regarded as “Android bitcode”.

3.1 Android Bitcode

We attempt to address the ecosystem issue due to the tremendous diversity in Android market. We believe any vendor wants to make sure all Android applications can run well across all its processor family. On the other side, Consumers want all applications can work seamlessly across Android devices. However, we already got too much non-portable native code that works on only one particular CPU type. Android bitcode, aka ABC, has two design goals:

1. Volume. Enable as much ecosystem as possible. This is not just for any company but is for making the world better and non-fragmenting.
2. Platform. We provide platform-level solution and prove that it can work on current leading architectures.

Besides, Android bitcode has one design principle: Constraning. The most constraining is the most portable. We point out all specifications as detail as possible. Vendors can adapt to their ABIs because of no information loss.

ABC defines many rules of ABI-level. It uses little endian since much more devices now are little-endian. ABC maps each language-level type into fixed size and alignment. For example, “char” is signed, mapped to “i8” of LLVM IR, one-byte alignment, and “long long” is mapped to “i64” and 8-byte alignment. ABC follows ILP32, which means sizeof(int), sizeof(long) and sizeof(void*) are equally 32-bit. ABC does not need to specify target-specific calling convention since LLVM materialize it on code-generation phase, not bitcode level.

ABC wraps LLVM bitcode into a simple wrapper structure. This wrapper contains ABC header and indicates the offset and size of the embedded bitcode file. ABC header encodes magic number, version, and something information we need. On devices, there will be a bitcode compiler which is responsible for lowering ABC into specific native machine code. ABC defines many details but we don’t mention all of them in this paper. Fortunately, this rules work for most cases we test so far.

3.2 Portable Support Library

Some header files is highly system-dependent, for instance, struct stat. Its implementation is different by X86 and ARM. We cannot handle this by bitcode level only. Fortunately, the amount of these is not too many. We collect them and provide a shim layer to split programmer-level from system-level. We re-define “stat” to “stat_portable”, and re-implement it distinctively in different target library to adapt it to the origin one. This corresponding target library, as a shim layer, is named “libportable”. This library is responsible for translating each portable API into target-specific implementation and will be installed to different targets to make sure our approach works. It is necessary by dynamic linker on devices.

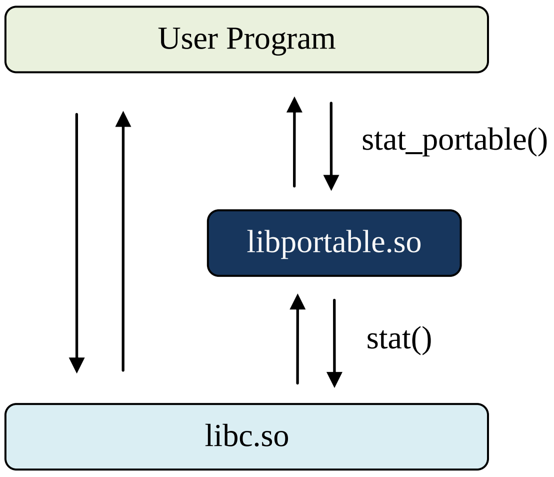

Figure 2 Target-dependent Portable Support Library

3.3 Host End

We take a look of Clang here, which is a LLVM frontend translating C/C++/Obj-C into LLVM IR. Clang follows LLVM guideline that it is modular, flexible and easy to reuse its components like Diagnostic, Preprocessor, Parser, ASTContext and ASTConsumer and etc [20]. We can easily customize its compilation steps.

We develop a tool, named llvm-ndk-cc, on the basis of clang libraries. We use llvm-ndk-cc to generate bitcodes on demand. Currently, Clang is sufficient to meet our requirements, but we trust that we still need to deploy llvm-ndk-cc for the long-term demands since C family languages are such different for target-specific values.

After generating each bitcode, we need to link them into one integrated bitcode. This link will do nothing except collect them all into one. We use llvm-ld, which is an official tool built with LLVM. It will encode the libraries needed into the bitcode.

3.4 Device End

On device end, we need to compile ABC to native DSO. It needs not only a bitcode compiler but also a static linker since LLVM has no integrated linker yet until now. We compile ABC to native object files by the bitcode compiler and link it with some necessary libraries, like libc, libstdc++, libportable and generate a DSO file. All these actions occur at installation time and will not harm any runtime performance. Android has libbcc to compile LLVM bitcodes since RenderScript [21] is already developed, and we can continue to enhance and use it. On the other hand, we use MCLinker [22] to support translate native object files to native DSO. MCLinker is a full-fledged, system linker for mobile devices. It is fast and small with low memory footprint so that we can use it on devices. We only need few modifications to integrate MCLinker into libbcc.

We show our compilation process in the following Figure 3. It is clear that we defer backend code-generation phase to device to gain more portability.

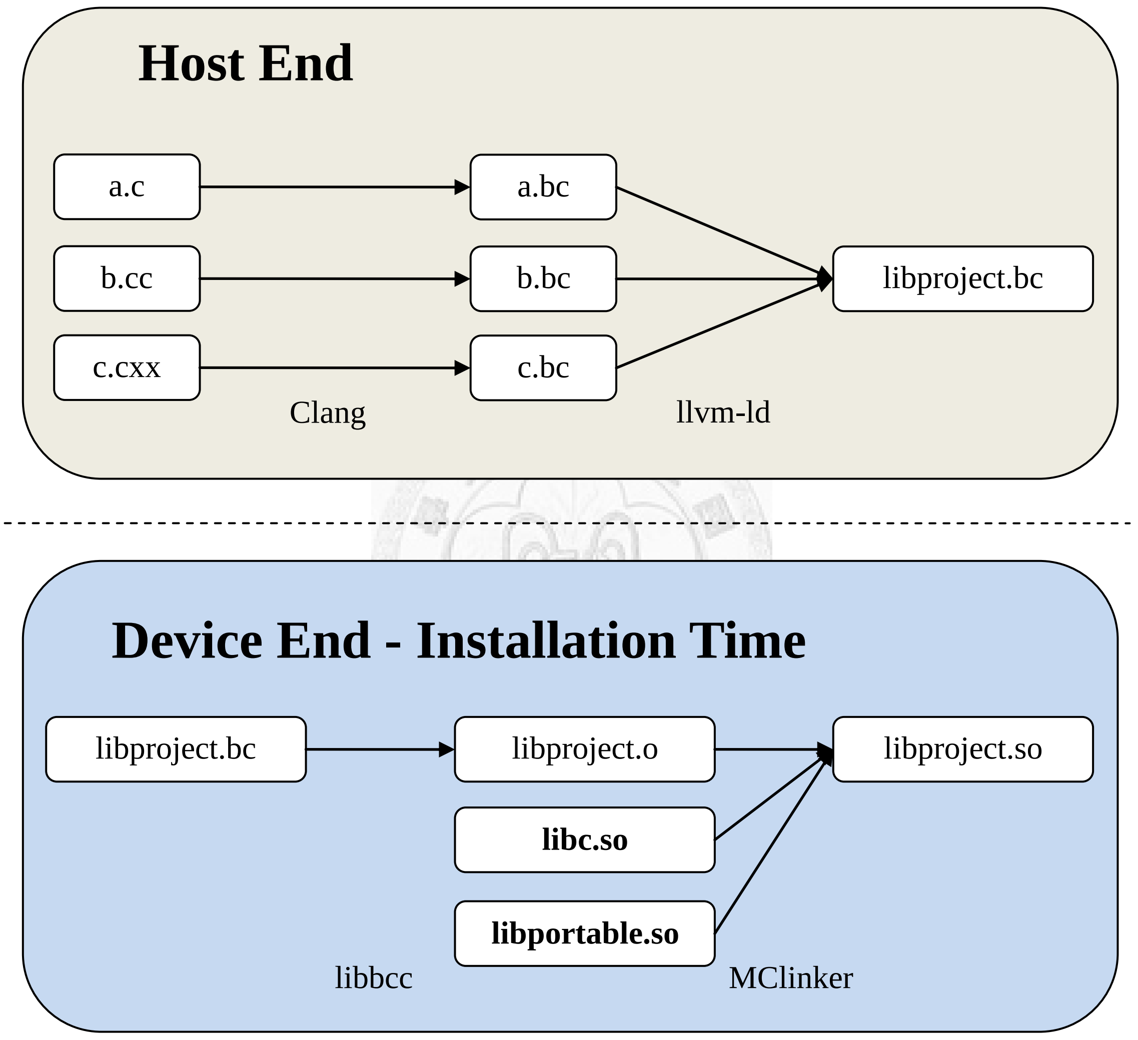

Figure 3 Portable NDK Compilation Process

3.5 Experimental Result

We have designed and implemented Portable-NDK architecture. We take some samples from NDK, like Hello-jni, Bitmap-plasma, Native-plasma, and some third-party Quake, Quake 2, Quake 3 to check our progress. Those are successfully run on different Android devices for ARM, X86 and MIPS.

Developers who use NDK mean they emphasize runtime performance since C/C++ is run on native environment. In order to show we can keep runtime performance comparable to the original approach, we measure Frame-per-Second (FPS) of Native-plasma since it has instrumentation code inside. This indicates we won't do anything which might change their behavior.

We exploit NDK r8 versus LLVM 3.1 on Galaxy Nexus [23] which is developed by a partnership between Google and Samsung. We use the same optimization level "-O2" and other target-dependent options. The FPS results are 19.2 and 19.3 which are very close. This sample proves that we propose a new solution which can solve portability problem without sacrificing runtime performance.

Chapter 4 Related Works

We have introduced our solution for portability problem deeply. Now we mention some other papers about portability improvement or AOT compilation. Moreover, we discuss some minor issues about our solution.

4.1 Google Native Client

Google has designed and implemented a sandbox, named “Native Client” on its browser [24]. Native Client aims to give browser-based applications the computational performance of native applications without compromising safety. Like our project, it exploits LLVM bitcodes to spread and compile it to native machine codes. Native Client has been proposed for many years and has joined since Chrome v10. It is a good demonstration that LLVM bitcodes is a suitable format to help improve portability in practice.

4.2 Method-Based AOT on DEX File

The well-known tradeoff of Java languages’ portability is the inefficiency of its basic execution model. We have noticed that we can use JIT compilation [25] technique to speed up runtime performance [26], but that is still not good enough. In the past year, there is one paper which implemented AOT compilation for Android applications [27]. It proposed a tool, Icing, which uses static profiling on DEX bytecode to find hot method and exploits existing cross-compiler to translate it to native machine code. DEX bytecode is to Dalvik VM what Java bytecode is to JVM [28]. Icing is a mixed-mode AOT compiler, i.e., it only compiles some host methods instead of the whole file. It uses some tools and infrastructures, like smali [29], dexdump, COINS [30], arm-gcc-4.4.0, to prevent reinventing the wheels.

That paper demonstrates a good AOT example. It improves runtime performance two to three times faster than that without JIT compilation, and 25% to 110% faster than that with JIT compilation. However, our case has a little difference from it. Android NDK uses C/C++, not Java, i.e., that approach doesn't consider portability problem on language level. Our solution focuses on Android NDK and put the portability and fragmentation problem at the most important position. We can simultaneously keep the runtime performance and achieve more portability by only one-family compiler infrastructure, LLVM. This has more good impact to Android ecosystem.

4.3 Debugging Support

In this paper, we change the compilation process, and this will cause something needs more efforts, for example, debugging. Here is the comparison of the original debugging support and the new one.

4.3.1 NDK Debugging Mechanism

On the original way, there are two DSOs on host. One is stripped, under the "*libs*" directory, without debugging section, and will be installed into devices. The other is not stripped, generated under the "*obj*" directory, and it can be used by GDB to get debug information. Besides, it needs prebuilt cross-compiled gdb client on host, and a target-dependent gdbserver binary on device. When user wants to debug his application, NDK will use gdb client to attach the application process and the gdbserver. Once it needs debug information, it will get what it want from the un-stripped DSO on host.

This way works since there is an un-stripped DSO on host and GDB can make use of it, but our approach will not generate any DSO on host. We need some efforts on this issue.

4.3.2 Portable-NDK Debugging Mechanism

We need different prebuilt binaries, like gdb and gdbserver, on different targets, so we need additional directive describing which target we are debugging now. That is, debugging is not target-independent. For example, we use a file noted “DEBUG_TARGET=arm”, this will let GDB know it should use the ARM-specific toolchain.

Recall that bitcodes will be translated to DSOs on installation time. When we run GDB, it can pull the binary from device to the “*obj*” directory and get the debug information. This also means our LLVM toolchain needs to generate the DSO with debug section. This only costs a little extra storage space.

Moreover, we can embed some metadata in the bitcode, noting that we expect it is a debug version or a release one. Our LLVM toolchain can extract this field to decide whether this file needs debugging sections and which optimization level.

There are many target-specific works need to be handled. Debugging mechanism is a good example worth to investigate. Fortunately, we can use some workaround to handle them.

4.4 Compilation Time Speedup

In addition to run-time performance, the most important thing we care is the AOT compilation time. User can only endure few seconds when downloading the APK, so how we can improve the compilation time on restricted devices becomes a challenging problem. We demonstrate some variants on our approach in below.

4.4.1 Option Tweaking

Here is the time passes result of compiling Quake on Motorola Xoom, by default optimization level, “-O2", shown on Figure 4. Motorola Xoom is the first tablet to be sold with Android 3.0 honeycomb. We choose it as our experimental device since it has dual core. We will make good use of that feature.

LLVM spends lots of time on code generation phase and loop strength reduction optimization. There is a short-term solution that we can tune the option, i.e. use fast instruction instead of traditional DAG instruction selection, use fast register allocation instead of greedy register allocation, remove the loop strength reduction optimization… and etc. We can shrink these most-cost passes to save more compilation time. However, this will harm run time performance severely even though we can make such 30% improvement on compilation time. We should use other solutions instead.

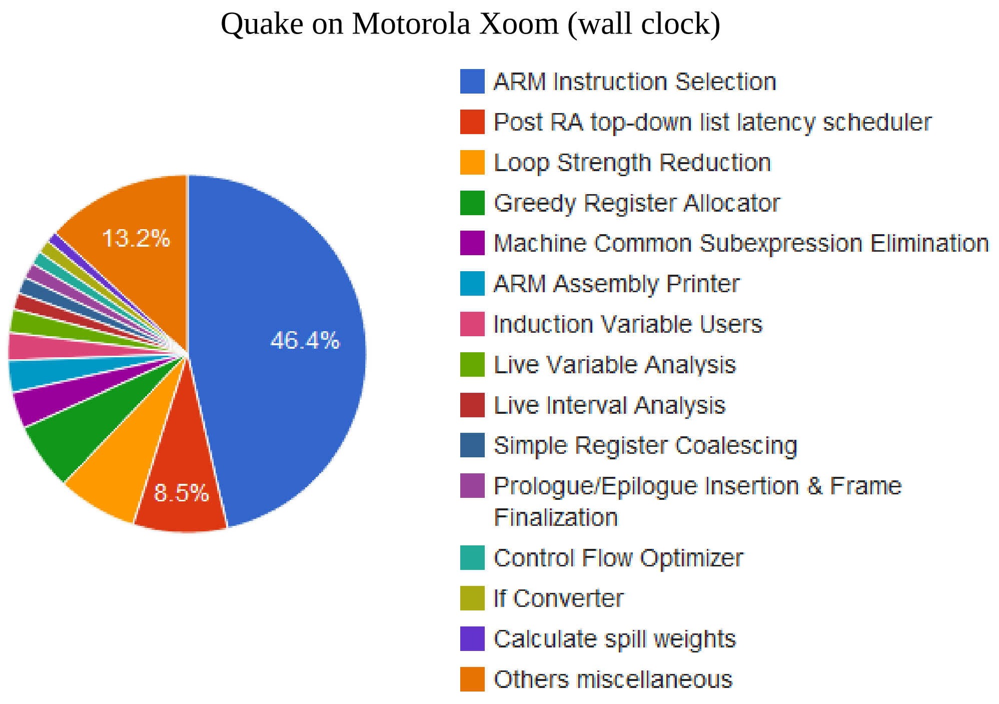

Figure 4 Quake AOT Time Passes Analysis on Motorola Xoom

4.4.2 Divide and Conquer

Some applications have many sub-modules. For example, Quake has 104 separate compilation units. It will generate the same order of amount small bitcodes, and link them into one large bitcode. If we compile the large bitcode on device, it will cause much more time-consuming.

We have measured that our approach spends compilation time much costly than linking time. For instance, Quake needs 21 seconds for compiling but need only 1 to 2 seconds for linking. We can focus on compilation time and divide it into separate small units. One approach is that don't generate the large bitcode. Instead, we put all small bitcodes into APK and keep them all in one container. Since each bitcode is small enough, it won't take too much time to compile. Moreover, we can use parallel multi-processing on multicore. For instance, we run three samples on Motorola Xoom, which has dual-core, to observe the time change. The result is on Figure 5. As we expected, the compilation time can be saved by almost one-half.

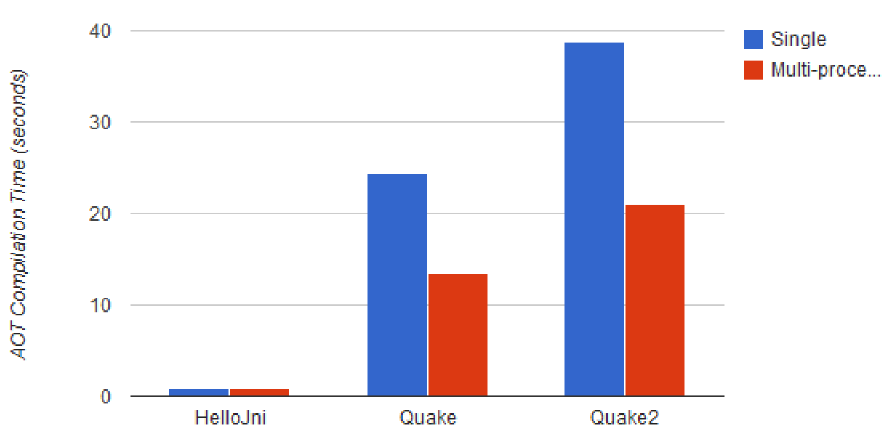

Figure 5 Three Samples Compilation Time on Motorola Xoom

4.4.3 Change Code Generation Policy

LLVM has a large general architecture design for the backend. That makes adding a new backend support is a relatively easy way than other compiler infrastructure. The detail is represented in Figure 6. LLVM will translate its bitcode IR into a SelectionDAG (Directed-Acyclic Graph) for instruction selection phase. It uses code generator generator, like tblgen, to select the low-cost instructions for tiling the DAG via tree pattern-matching. In the meantime, each llvm::Instruction in DAG nodes will become an llvm::MachineInstr which encodes target-specific information inside.

Instruction Scheduler will decide the order from the DAG to a sequence of linear llvm::MachineInstr IR. Register Allocator will perform de-SSA and do the register allocation and assignment. After that, LLVM will translate llvm::MachineInstr into llvm::MCInst for generating output files. The MC (Machine Code) layer is used to represent and process code at the raw machine code level. We can use MCStreamer to generate assembly or object files from MC layer.

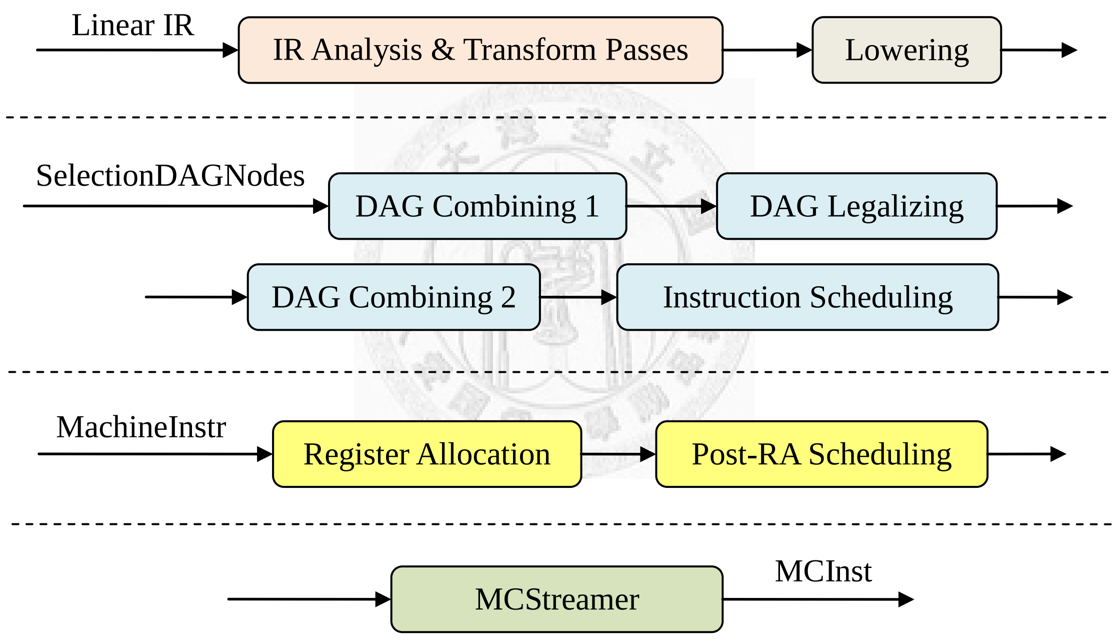

Figure 6 LLVM Code Generation Design

After reviewing the code generation phase of LLVM, we saw lots of transformation since generality and architecture demand. Observe Figure 4 which shows the time passes result on compiling Quake bitcodes. It spends lots of time on Instruction Selection. It is time-consuming that transforming a data structure to another one and using general but large algorithm on code generation phases.

One approach is breaking the code generation design of LLVM. We translate one bitcode IR instruction into several low-level IR ones, and use peephole optimization to translate low-level IRs into machine instructions. It will save more time since it will not need cost on DAG creation, combining and bottom-up DP selection.

However, this is not a good long-term solution because it makes Android cannot reuse LLVM backend components. LLVM has many well-written function and can be easily pluggable. We need to re-design and re-implement the backend phase for the peephole instruction selection phase and the successive register allocation phase. This might be a good short-term solution, but it is not a feasible trade-off for saving compilation time in long term.

We have figured out and implemented some improvements to conquer the cost of AOT compilation time. We enumerate these approaches since they are more easily to implement. However, as we mentioned, these are not a good long-term solution. We will see how to solve it by fundamentally on more high level, i.e., cloud, in next chapter.

Chapter 5 Future Work

The approach we proposed is a good solution to improve the portability problem and fully optimize the DSO. The cost we need to pay is long compilation time. We have mentioned some solutions for this problem in the former chapter, but still cannot solve it fundamentally. They are not so intuitive and effective. While users download the APK, they still need to wait until the compilation process is done. The hardware specification is poorer; the waiting time is longer. Besides, consuming battery power to compile, even though not too much, is still a minor issue.

Google has strong power on cloud computing so that it can make this action more early. After developers publish APKs, Google can scan them and compile bitcodes on the cloud, which generates different target-dependent DSO and packages them into the new corresponding APKs. For example, developer publishes a "portable" APK and Google will translate it to many versions for ARM, X86, MIPS, and so on. Once users want to download, they only need to send their hardware ID hash code to Google and Google will choose the appropriate target-dependent version of APKs to users. The whole process is like Figure 7. This approach makes development process more intuitive and more suitable on all devices in the long term. User won't need to take care about the compilation time anymore. This change is not a pure technical problem and it needs more and more communications overhead. Besides, it will change the whole Android ecosystem so that we cannot guarantee this approach can be accepted. For these reasons, we put this solution in this chapter since we cannot finish it before this paper release.

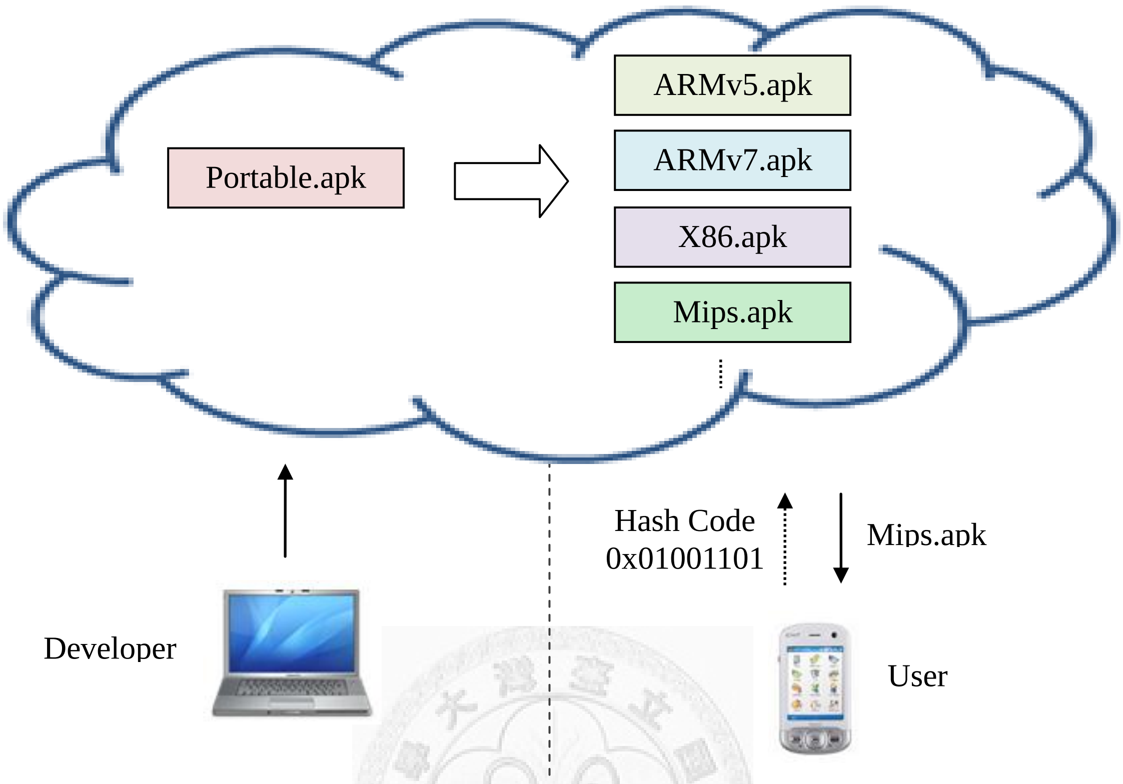

Figure 7 Google Cloud for AOT Compilation

Chapter 6 Conclusion

In this paper, we propose a new approach to make Android application developers who may use native languages through NDK, to generate an Android bitcode instead of a DSO enclosing to APKs. It breaks the native language restriction of machine dependency, makes it portable to publish, and innovates a new compilation process. This approach benefits Android ecosystem and makes it away from fragmentation problem caused by native language development. In addition, we can leverage many target-dependent optimizations and gain better run time performance on this solution. In conclusion, we have made Android ecosystem more portable on application development only in exchange for some additional on-device AOT compilation time.

References

[1] Android NDK, http://developer.android.com/sdk/ndk/index.html

[2] Lee, S. and Jeon, J.W., *Evaluating performance of Android platform using native C for embedded systems*, Control Automation and Systems (ICCAS), IEEE, pp. 1160-1163, 2010

[3] The LLVM compiler infrastructure, http://llvm.org

[4] LLVM Assembly Language Reference Manual, http://llvm.org/docs/LangRef.html

[5] Clang: a C language family frontend of LLVM, http://clang.llvm.org/

[6] AOT compiler, http://en.wikipedia.org/wiki/AOT_compiler

[7] Lin, C.M., Lin, J.H., Dow, C.R. and Wen, C.M., *Benchmark Dalvik and Native Code for Android System*, Innovations in Bio-inspired Computing and Applications, IEEE, pp. 320-323, 2011

[8] Ratabouil, S., *Android Ndk Beginner's Guide*, Packt Publishing, 2011

[9] Daniel Ionescu, *Angry Birds Devs Angry At Android Fragmentation*, http://www.pcworld.com/article/211152/angry_birds_devs_angry_at_android_fragmentation.html, 2010

[10] Oliver, *Build error (include search path?) on ndk r5*, https://groups.google.com/forum/?fromgroups#!topic/android-ndk/FO124L4Y85E, 2010

[11] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson, *Binary translation*, Communications of the ACM, vol 36, pp. 69-81, 1993

[12] Mark Probst, *Dynamic Binary Translation*, UKUUG Linux Developer's

Conference, 2002

[13] Moore Ryan W., Baiocchi Jose A., Childer Bruce R., Davidson Jack W. and Hiser Jason D., *Addressing the challenges of DBT for the ARM architecture*, SIGPLAN Not., vol. 44, num. 7, pp. 147-156, 2009

[14] Chernoff A., Herdeg M., Hookway R., Reeve C., Rubin, N., Tye T., Yadavalli S.B. and Yates J., *A profile-directed binary translator*, IEEE Micro, vol. 18, pp. 56-64, 1998

[15] Chen J.Y., Yang W., Hung T.H., Su H.M., and Hsu W.C., *A static binary translator for efficient migration of ARM-based applications*, ODES-6: 6th Workshop on Optimizations for DSP and Embedded Systems, 2008

[16] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke, *LLVA: A Low-level Virtual Instruction Set Architecture*, Proceeding of the 36th annual ACM/IEEE International Symposium on Microarchitecture, 2003

[17] Chris Lattner and Vikram Adve, *LLVM: A Compilation Framework for Lifelong Program Analysis and Transformations*, Proceeding of the International Symposium on Code Generation and Optimization, 2004

[18] Dan Gohman, *LLVM IR is a compiler IR*, http://comments.gmane.org/gmane.comp.compilers.llvm.devel/43769, 2011

[19] Can I compile C or C++ code to platform-independent LLVM bitcode? http://llvm.org/docs/FAQ.html#can-i-compile-c-or-c-code-to-platform-independent-llvm-bitcode, LLVM FAQ

[20] Christopher Guntli, *Architecture of clang*, HSR – University of Applied Science in Rapperswil, 2011

[21] Komatineni S., MacLean D. and Hashimi S.Y., *Programming 3D Graphics with OpenGL*, Pro Android 3, Springer, pp. 623-691, 2011.

[22] MCLinker: LLVM Linker for Mobile Computing, http://code.google.com/p/mclinker/

[23] Galaxy Nexus, http://en.wikipedia.org/wiki/Galaxy_Nexus

[24] Yee B., Sehr D., Dardyk G., Chen J.B., Muth R., Ormandy T., Okasaka S., Narula N., and Fullagar N., *Native client: A sandbox for portable, untrusted x86 native code*, Security and Privacy, 30th IEEE Symposium on, pp. 79-93, 2009

[25] Just-in-time compilation, http://en.wikipedia.org/wiki/Just-in-time_compilation

[26] Muller, G., Moura, B., Bellard, F. and Consel, C., *Harissa: A flexible and efficient Java environment mixing bytecode and compiled code*, Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems, vol. 16, num 20, pp. 1-20, 1997

[27] Chih-Sheng Wang, Guillermo Perez, Yeh-Ching Chung, Wei-Chung Hsu, Wei-Kuan Shih and Hong-Rong Hsu, *A method-based ahead-of-time compiler for android applications*, Proceeding of the 14th International conference on Compilers, architectures and synthesis for embedded systems, 2011

[28] Bornstein, D., *Dalvik vm internals*, Google I/O Developer Conference, vol. 23, pp. 17-30, 2008

[29] Smali: An assembler/dissembler for Android's dex format, http://code.google.com/p/smali/

[30] COINS Compiler Insfrastructure, http://www.coins-project.org/international/