University of Edinburgh
School of Informatics

An LLVM based compiler from Objective-C to Dalvik Virtual Machine

4th Year Project Report
Computer Science and Mathematics

Stanislav Manilov

January 24, 2013

Abstract: The rapidly growing mobile app market of today, and the need of developers for cross-platform tools justify the creation of a system that can translate iPhone applications to Android ones. Such a system would need to have an Objective-C to Dalvik binary compiler, an Android version of the Cocoa API, and an interface translating module. The goal of this project is to produce a compiler, using LLVM as a foundation. A successful implementation would produce applications that are comparable in size and performance with Java written equivalents. At the time of writing, the system is 60% complete, with the plan of having it 100% complete within two weeks and using the rest of the semester to write the dissertation.

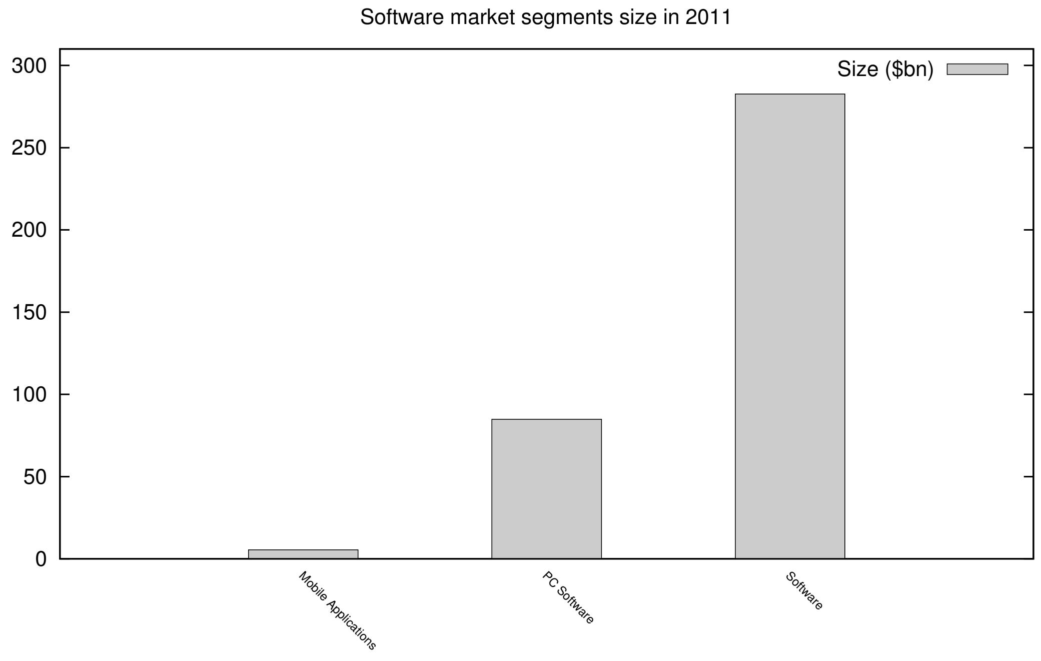
Contents

1. Introduction

1.1 Background

Today we observe a rapidly growing mobile applications market [3] that induces equally fast growing demand for high-quality and reliable development tools. In 2011 the market generated a revenue of \$5.5 bn [3], which is estimated to be 6.5 % of the value of the desktop software market (please, see Figure 1.1). This illustrates the significance of the mobile applications market.

Figure 1.1: Software market size in 2011 [3] [2] [1].



A main challenge for mobile app developers is producing and maintaining cross-platform applications. Although there are already plenty of tools that address writing such applications (please, see Table 1.1), there are three main problems with them:

- all of them require learning additional languages (e.g. HTML, JavaScript, or Lua),

Table 1.1: Comparison of popular cross-platform mobile tools [5]

Name	Required Knowledge	Cost	Since
Sencha Touch	HTML, CSS, JavaScript	Free	November 2010
jQuery Mobile	HTML, CSS, jQuery	Free	October 2010
Tiggzi	HTML, CSS, JavaScript	\$0 - \$180 pm	Unknown
AppMakr	HTML, CSS	\$79 pm	January 2010
iBuildApp	HTML, CSS	\$10 pm	2010
Widgetbox	HTML, CSS	\$25 - \$100 pm	October 2010
foneFrame	HTML, CSS, JavaScript	\$0 - \$59	Unknown
phoneGap	HTML, CSS, JavaScript	Free	August 2008
Corona	Lua	\$200 - \$350 py	December 2009

- most of them violate the Apple SDK Agreement [6] [4] by producing apps originating from disallowed languages, and
- none of them can deal with automatic porting of already existing code that was originally written for a specific platform.

These points justify the creation of a compiler that can take the source code of an iPhone application written in Objective-C and produce an equivalent Android application. However, there are a couple of major issues with this idea: firstly, the libraries used for iPhone development - Cocoa API - are proprietary and thus can not be compiled for Android, and secondly, interface guidelines for the two platforms differ and thus interfaces can not be literally translated. Fortunately, there is a community effort to solve the first problem - the GNUStep project. And while the interface guidelines differ, they are specific enough to make the detection and translation of idioms possible.

This discussion outlines two of the three major components that a system of the required type must have: a version of the Cocoa API, compiled for the Android platform, and an interface translation module. The third necessary component is, of course, a compiler core that can translate the actual program logic. As building the whole system is an ambitious task for an honours project, it was decided that the effort would be concentrated on the program logic component. If successful, this project can be built upon, adding the additional critical components for creating a complete, automatic porting facility.

1.2 Goals

As outlined in the background section, the goal of this project is to build a system, using which, an Objective-C code can be compiled to a Dalvik executable and can be ran on an Android emulator or an actual Android based smart phone. Essential properties of the system are:

- correctness: the produced program should be working as described by the source code;
- speed: the time to build a typical program should be acceptable (in the order of minutes at worst).

In addition to these, it is desirable that the resulting system has the following properties:

- performance of produced code: the produced program should be reasonably fast (same order of magnitude), when compared to an identical program, written in Java and built using the standard tool chain (Android Development Kit);
- size of produced code: the size of the produced program should be reasonably small, when compared to an identical program, written in Java and built using the standard tool chain.

The system should be available as a command-line tool. Different options should be available to specify properties (e.g. name or entry point) of the program to be built. Building a graphical user interface or an interactive system is not in the scope of this project.

The Objective-C programs that the compiler must be able to handle are command-line programs that do not depend on external libraries.

2. Approach

2.1 Design

Before the commencing of the work on the project, it was agreed that the best approach is to base the system on the LLVM compiler infrastructure. There are multiple advantages that come with building upon LLVM:

- Firstly, LLVM comes with clang - a complete frontend for languages based on the C programming language. This includes Objective-C, so basing Objective-Droid on LLVM would remove the need of producing front-end components like a parser or a lexer.
- Secondly, LLVM comes with a powerful middle end that contains a wide range of optimizations [7]. Including them in Objective-Droid would be highly beneficial to the quality of the output, while at the same time, it would be infeasible and unnecessary to duplicate all the work of implementing these optimizations.
- Thirdly, using an established compiler infrastructure as a foundation would provide an interface that developers are already used to, at no additional effort. This would reduce the learning curve for users of Objective-Droid and let them feel more confident about using the tool.
- Fourthly, building a backend for LLVM by the guidelines would allow it to be contributed to the project, thus all, or parts of Objective-Droid can be given back to the compiler developer community easier than if it is being developed from scratch.

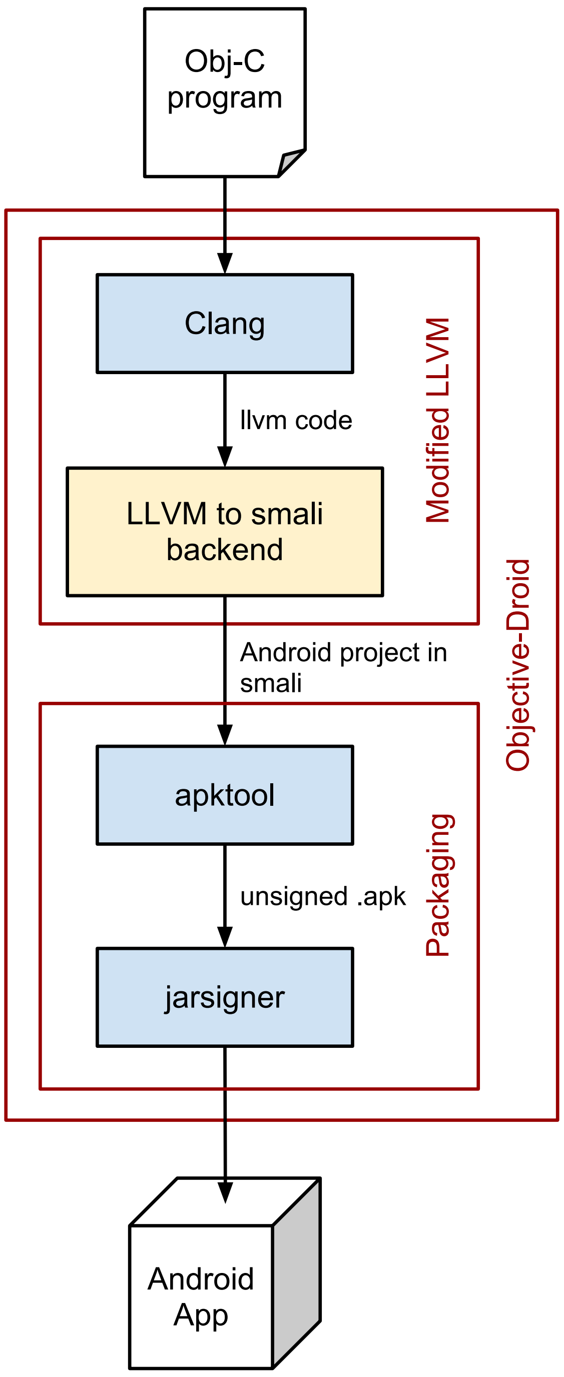
Thus, the first essential component of the Objective-Droid system is the LLVM backend. The target code was decided to be smali code, as there are already assembling tools for it that produce Dalvik binaries.

The second essential component is a packager that given the smali code assembles it, packages it, and signs the package. This would allow the package to be installed and tested on an emulator or a phone.

The final component of the Objective-Droid system is a wrapper program or a script that automates the whole process of compiling, packaging, and possibly installing the program on a phone. This script would be useful for testing, but is not mandatory.

For a summary of the design, please see Figure 2.1.

Figure 2.1: Overall design of the Objective-Droid system.



2.2 Implementation

2.2.1 File structure

As apparent from the design outline, the most integral part of the project is building the LLVM backend. This required familiarising with the conventions that are accepted within the LLVM community for undertaking such work, and understanding the general structure of a backend.

An LLVM backend needs to be implemented as a subclass of the TargetMachine class [8]. Depending on the features of the target architecture, the implementation needs to provide a suite of files. E.g. if MIPS is the target architecture, the implementation needs to provide files that describe: the instructions, the registers, the interface for JIT compilation, etc. On the other hand, if the target is not a real architecture, but rather a high-level language, the implementation can be as simple as a single class. This is the case of the backend that produces C++ code.

The initial approach taken when developing the Objective-Droid backend was to follow the general structure and provide separate classes for the different functions of the backend. However, this was decided to be overly complicated, especially when compared to the approach of the C++ backend. It was argued that the code that the backend has to produce - smali code - is not actual machine code, so it does not need many of the classes that a code for a real architecture would require (e.g. an assembler definition).

Thus, for the main part of the project the backend was developed in a single monolithic class. Still, attempting to use the full target class hierarchy was not a waste of time, as it provided an insight of how the LLVM project works - an insight that was helpful later.

The decision to write the whole backend as a single class provoked the danger of bad maintainability. Thus, during development, effort was put towards writing simpler and cleaner code at first, and then optimising the common parts of it. Also, comments were put as documentation, as general comments, and as specific comments explaining complicated code. The overall result was readable code that is easy to understand for a newcomer to the project.

2.2.2 `Target` class

The main class of the implementation inherits the `ModulePass` class of LLVM. The `ModulePass` is typically used for passes that manipulate the whole program, thus it serves the needs of the Objective-Droid backend. This makes the `runOnModule` method to be the entry point of the class. This method is given a description of the program in a `Module` object as an argument and returns `false` on success.

The implementation of the `runOnModule` method simply builds a corresponding Android `Activity` class by iterating over the global variables and functions of the `Module`, and translating them to static fields and to static methods, respectively. The boiler-plate code for the class includes a start up method - `onCreate` - which is a standard way of starting up an Android `Activity`. It invokes the translation of the main function of the supplied Objective-C code, and calls the typical `finish` method that terminates the program.

The functions are translated by iterating over the instructions in them and translating each one in separately. It was found that for most instructions this is trivial, and others were either simply translated to a series of instructions (as in the case of comparison instructions) or emulated by a dedicated external library. This approach would result in a slightly lower performance of the output program, but ensures good performance of compilation and simplicity of the compiler.

3. Progress

3.1 Up to now

At the time of writing, the system is 60% complete. This includes 75% of the instructions being translated, and a prototype of the packaging script being produced.

Two months were spent in researching about LLVM and initialising a separate backend component. The next two months were spent in incremental implementation of instructions, alongside thorough testing to ensure the correctness of each translation. This method proved useful, as it caught defects in implementation, notably in instructions that required to be emulated by a sequence of other instructions (e.g. the 'set on compare' instructions). During this period a code base of benchmark programs emerged and will be used after the project is complete for its evaluation.

3.2 Left to do

The programming work that remains to be done is:

- translating global variables as static fields
- translating the remaining 25% of the instructions
- supporting multiple file compilation
- cleaning up the source code
- adding missing documentation

When this is done, the supporting scripts need to be updated and polished, after which the system will be evaluated against the benchmark code base. Finally, the dissertation would be written up and submitted.

The following milestones are scheduled:

- February 10 / academic week 4 - have a full-featured version of the system
- February 24 / academic week 6 - finish most of the testing and have a foundation for the final report
- March 10 / academic week 8 - finish first draft of the report and submit it for review

- March 24 / academic week 10 - implement feedback from the review and submit a final draft of the report for review
- April 8 / academic week 12 - prepare a draft presentation and submit it for review; submit final report
- April 22 - finalise presentation and submit for review
- week starting April 29 - Present

Bibliography

[1] Business Software Alliance. Software Industry Facts and Figures. http://www.bsa.org/country/public%20policy/~/media/files/policy/security/general/sw_factsfigures.ashx, 2009.

[2] Datamonitor. Software: Global Industry Guide. http://www.datamonitor.com/store/Product/software_global_industry_guide?productid=46A43D30-02A3-4425-BFFB-47FCFD5A3AE5, 2011.

[3] P. Farago. The Great Distribution of Wealth Across iOS and Android Apps. http://blog.flurry.com/bid/88014/The-Great-Distribution-of-Wealth-Across-iOS-and-Android-Apps, 2012.

[4] J. Gruber. New iPhone Developer Agreement Bans the Use of Adobe's Flash-to-iPhone Compiler. http://daringfireball.net/2010/04/iphone_agreement_bans_flash_compiler, 2010.

[5] D. Hay. 10 Solutions for Creating Cross-Platform Mobile Apps. http://sixrevisions.com/mobile/cross-platform-mobile-apps/, 2012.

[6] Apple Inc. iPhone SDK Agreement. http://www.wired.com/images_blogs/gadgetlab/files/iphone-sdk-agreement.pdf, 2010.

[7] LLVM Project. LLVMs Analysis and Transform Passes. http://llvm.org/docs/Passes.html, 2013.

[8] LLVM Project. Writing an LLVM Compiler Backend. llvm.org/docs/WritingAnLLVMBackend.html#Prerequisite, 2013.