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

March 18, 2013

Abstract: 6 pages per section.

Contents

1. Introduction

1.1 Motivation

1.2 Goals

1.3 Overview

2. Background

2.1 Objective-C

Objective-C was originally developped between 1981 and 1983 by Brad Cox and Tom Love as a superset of the C programming language that was strongly inspired by Smalltalk and incorporates its object oriented model[1]. The creators of the language concentrated on building an extension that relies on run-time libraries and dynamic binding to improve the ease of integration of different components, rather than inventing a complex powerful software manufacturing system from scratch, which is the approach taken with C++.

The Objective-C language implements the object-oriented paradigm in a different way than the more widely used and successful C++ language. The former's object oriented model is based on Smalltalk, as previously mentioned, while that part of the latter is modelled after the aproach taken in the Simula language. As a result of this, even though the two languages are good at tackling similar domains of problems, their core principles are different. Objective-C was designed with simplicity in mind - the language architects aimed at adding the absolute minimum of additional features that would enable the programmers to write object-oriented programs. Backward compatibility with C and understandability were also main objectives of the project. In contrast, Bjarne Stroutstrup (the creator of C++) wanted to create a language for maximum efficiency that allowed him and fellow collegues to "express program organisation as could be done in Simula"[1]. The focus was on power and speed, and as a result C++ turned out to be more complex and harder to understand than Objective-C. However, in the long run it also turned out to be more popular, the reason for which the Objective-C creators find in the fact that C++ was given away for free by AT&T, rather than in actual benefits of the language. The compiler and support libraries of Objective-C were the only revenue sources of Stepstone (the company of Love and Cox), so they couldn't afford to open source the project, and thus it remained mostly unpopular.

The first notable recognition of the language was when it was chosen as the native language for the NeXTStep 1.0 operating system in 1989 - the first mature operating system of NeXT Computer. It is hard to find the reasons for choosing Objective-C over C++ in that setting. A propaganda NeXTStep manual from 1992[11] claims that NeXT chose Objective-C for its ease of learning and because of its support for dynamic binding, arguing that the latter allows for more seemless updates of the system, but this can be achieved equally easy by using shared/dynamically linked libraries. The author speculates that a major part

of the decision was the need to differentiate from other OS vendors of the time and create a NeXT specific developer community and user base. Whatever the reason, the result remains and it is a popularisation of the Objective-C language.

NeXT later acquired the rights over Objective-C in 1995 and became the main advocator of the language. The company itself was acquired by Apple Inc. in 1997 and the NeXTStep operating system eventually evolved into what became known as Mac OS X over the period of four years. This way Objective-C propagated its way into the Mac computers of today. Native Mac apps are developed using the proprietary Cocoa API, which is written in Objective-C. A mobile version of this API - Cocoa Touch - is what is used for developing applications for the iPhone, iPod, and iPad.

Since programming in the past couple of decades has not been restricted to a selected few, but rather an essential skill sometimes compared to literacy in its importance, NeXT/Apple have recognised that they need to provide the general public (rather than only Mac users) with tools that allow creation of Objective-C programs if they want to make Objective-C a competative and popular language.

The first such initiative was OpenStep which was an object-oriented API for non-NeXTStep operating systems. The GNUStep project is an open source implementation of Cocoa that is based on OpenStep. Later, a team of NeXT/Apple engineers, lead by Steve Naroff developed the GCC Objective-C frontend. However, it relied on run-time libraries, which were not open sourced, so this effort remained useless to the general public. Finally, Apple invested in building the Objective-C frontend to Clang in 2007, which enabled the author to undertake this project.

Thanks to the existence of Clang, the reader need not understand the minute details of the Objective-C language as it is translated by this LLVM frontend to LLVM bytecode.

2.2 LLVM and Clang

2.2.1 History

LLVM was initially concieved as "a compilation strategy" that was designed to provide "effective multi-stage optimisation and more effective profile-driven optimisation, and to do so without changes to the traditional build process or programmer intervention"[9]. It was started as the Masters Thesis project of Chris Lattner under the supervision of Professor Vikram Adve in the year of 2000 at the University of Illinois at Urbana-Champaign. The main goals were to lay the foundation of a modern modular compiler, that has similar interface to

its contemporaries and predecessors. The paper was published in 2003, and the LLVM infrastructure was open sourced, subsequently being widely adopted by the community and researchers[8]. In 2005, Apple hired Lattner to work at their compilation team and during the following seven years he advanced in positions to reach that of "Director and Architect, Developer Tools Department" in January 2013. Now he is responsible for the management of the entire Developer Tools department at Apple[cite http://www.nondot.org/sabre/Resume.html]. Apart from an example of impressive personal development, this is important for the evolution of LLVM, as during his career at Apple Lattner has mainly been working on LLVM and related tools, including Clang.

2.2.2 Structure

There are four stages of compilation included in LLVM: pre-linking compilation, link-time optimisations, run-time optimisations, and offline optimisations[9]. The first two stages are common in traditional compilers and include static analysis and classical optimisations. After stage two, the program is compiled to machine code, but high-level type information is also included in the output. This is one of the key novelties in LLVM, as this information is later used to perform the run-time optimisations and offline optimisations.

During execution of an LLVM compiled program, profile information is gathered from the usage statistics. This is different to the usual approach of profiling, as the usage information is actual, user specific data, rather than synthetic data that is fabricated by the developer. This way, the pitfall of profiling for the wrong data is avoided and the increase of performance is higher. In addition to this, the whole profiling process is automated and the overhead of effort is removed.

Other activities that take place during execution of the program are run-time optimisations. These are optimisations that can be found in standard Just-In-Time (JIT) compilers, but also optimisations that use the profiling data to make common cases in the code faster, for the price of slowing down less common control flow paths. If there are profile driven optimisations that are too expensive to perform, they are sheduled for the offline optimiser.

The offline optimisations in LLVM are more aggressive dynamic optimisations that are too expensive to run alongisde the program, so they are performed when the system is idle. They are particularly useful for programs in which the majority of time of the execution is spent over a large part of the code, rather than having a particular "hot" spot. In such cases the profiler would detect optimisation opportunities, but it would not be beneficial to actually perform them in run-time, so they are delayed to when the offline optimiser can step in.

2.2.3 Language

The instruction set that LLVM uses provides an infinite amount of typed virtual registers in Static Single Assignment (SSA) form, which hold values of primitive types. The primitive types include integers, floating point numbers, and pointers.

Memory is accessed via the store and load operations and is partitioned in global area, stack, and heap. Stack variables are allocated using the alloca instruction and are freed automatically when the function returns. Heap variables are allocated using malloc and need to be freed explicitly using the free instruction. In this way, the memory management in the LLVM language is similar to the C programming language.

The arithmetic and logical operations are in three-address form. The available arithmetic operations are: add, sub, mul, div, and rem. The names are self explanatory and correspond to the usual operations in arithmetic. Div and rem are whole number division and remainder functions. The available logical operations are: not, and, or, xor, shl, and shr. Additionally there is a collection of comparison instructions that produces a boolean result: seteq, setne, setlt, setgt, setle, and setge. Respectively, they stand for: equal, not equal, less than, greater than, less than or equal, and greater than or equal. Other instructions can take 0, 3, or a variable number of arguments. Examples are the call instructions and the SSA phi instruction.

All instructions in the LLVM instruction set are polymorphic. This means that they can operate on different types and do not need to have special versions for each of the possible types. The types of the operands also define the semantics of the instruction.

2.2.4 Building a new backend

Because of the high modularity of the architecture of LLVM it is possible to build new backends and plug them in the whole system. A backend is basically a module that takes the LLVM internal representation and translates the program to a specific target native code. Since Clang includes an Objective-C frontend for LLVM, the author concentrated on developing only a backend that produces Dalvik bytecode from LLVM internal representation.

There are different ways to do this, depending on the amount of the recommended structure that the developer wants to incorporate in the backend. Ideally, it should extend a large number of LLVM classes and provide target specific details[10]. This way more of the previously discussed LLVM features can be enabled. However, if the aim is simply to get a working prototype, it is sufficient

to extend only one class (see section Implementation) and then to register the backend.

Also, work in LLVM is done in passes of different modularity (block, function, or module), and the developer that is extending the functionality can decide which one (or more) of those they want to implement. However, when writing a backend it is obvoius that the whole program needs to be processed, so writing a module (the largest granularity) pass is sufficient.

2.3 Android and Dalvik

2.3.1 Android

Android was revealed on the 5th November 2007 by Google, when they introduced the Open Handset Alliance (OHA)[3]. The alliance comprises of several dozens of telecommunication companies and their aim is to produce more open and customisable mobile phones. The major product to achieve this is Android - an open source mobile operating system based on Linux.

Historically, the development of the Android operating system starts around 2003 at a company called Android Inc[2]. The project is developed under high secrecy and little is known about its existence before it was introduced in 2007, apart from the fact that the company engaged with it was acquired by Google in 2005.

Android entered a market that was dominated by conservative vendors: Apple, Microsoft, and RIM. Their philosophies don't reserve a place for the idea of open-source software and this is what was different about the new operating system. During the following five years the community of programmers developing Android apps grew rapidly, producing around 700 000 apps for the platform[7]. Today Android is the most popular mobile operating system, owning three quarters of the market share[6].

In addition to advocating open-source, Android has the advantage that it allows hardware manufactures to concentrate on improving the devices and not worry about the software that runs on them. This way it is very useful for new companies that don't have the financial or technical resource to develop their own operating system. As a result, Android is even employed in devices it was not originally intended for, like TV sets and game consoles[12][4].

2.3.2 Dalvik

Dalvik is the virtual machine (VM) that runs on Android devices. Since Android applications are written in Java, Dalvik is similar to a Java Virtual Machine (JVM), but has several key differences.

First of all, Dalvik is register based, compared to the stack based JVM. This means that arguments for the instructions reside in virtual registers (a total of 65K of them) and not on a stack. Thus, the push and pop instructions that are used to manipulate the stack in order to execute stack based operations can be removed, resulting in less instructions, and thus smaller programs - important improvement for mobile applications.

Secondly, Dalvik does not use the defaut Java GUI frameworks (i.e. AWT or Swing) but rather relies on a dedicated one. However, open source frameworks have been successfully ported to run on Android, notably the popular Qt framework, so it is still possible to write applications which are independent of the VM used.

Most importantly though, due to the subtle differences between the Dalvik bytecode and the Java bytecode, Dalvik has its own file format - .dex. The .dex files are usually compiled from JAVA .class files using the dx tool from the Android SDK. However, the different file format further distantiates the two virtual machines and makes them incompatible in the strict sense.

2.3.3 Smali

Smali is an assembler for the dex format, the syntax for which is loosly based on Jasmin's syntax - an assembly code for Java[5].

Smali code, although being assembly code for a virtual machine, shares a lot with real RISC architectures. Arithmetic and logical operations are in three-address mode[13] which is the same as LLVM's. An important thing to note is that there are no boolean comparison operators. In order to compare numbers one has to use either the ternary comparison operators (returning -1, 0, or 1 respectively for less than, equal, or greater than results) that can not take short (single word) integer values, or the family of conditional branches if-eq, if-ne, if-lt, if-gt, if-le, and if-ge. The difference of the two approaches is examined in the implementation section.

Another dissimilarity from LLVM's internal representation is that arithmetic and logical operations in smali are not polymorphic, e.g. there are four versions of add: add-int, add-long, add-float, and add-double. There are also versions for when one of the operands is a literal, and versions taking only two registers.

This flexibility allows for selecting the minimum amount of instructions in each context and thus achieving the desired reduction of program size.

Finally, there are two other operation classes relevant to this project: move and const. There are 13 variants of move, each specific for a different situation, and the behaviour is generally moving the contents of one register to another. The variants of const are 11, and they are intended to move different kinds of literal values to a register.

3. Design and Implementation

3.1 Design

3.2 Implementation

4. Evaluation

4.1 Correctness

4.2 Performance

5. Related Work

5.1 LLJVM

5.2 C to Java compilers

6. Conclusions

Bibliography

[1] F Biancuzzi and S Warden. *Masterminds of Programming*. 2009.

[2] Bloomberg Businessweek. Google Buys Android for Its Mobile Arsenal, 2005.

[3] Brian DeLacey. Google Calling: Inside Android, the gPhone SDK. http://onlamp.com/pub/a/onlamp/2007/11/12/google-calling-inside-the-gphone-sdk.html, 2007.

[4] Darrell Etherington. OUYA Android Gaming Console To Start Shipping To Backers March 28. http://techcrunch.com/2013/02/28/ouya-android-gaming-console-to-start-shipping-to-backers-march-28, 2013.

[5] Google Project Hosting. smali About Page. http://code.google.com/p/smali/, 2013.

[6] IDC. Android Marks Fourth Anniversary Since Launch with 75.0% Market Share in Third Quarter, According to IDC. http://www.idc.com/getdoc.jsp?containerId=prUS23771812#.UUZ7veNMjqg, 2012.

[7] Zak Islam. Google Play Matches Apple's iOS With 700,000 Apps. 2012.

[8] Chris Arthur Lattner. LLVM Related Publications.

[9] Chris Arthur Lattner. *LLVM : An Infrastructure for Multi-stage Optimization*. PhD thesis, 2002.

[10] LLVM Project. Writing an LLVM Compiler Backend. http://llvm.org/docs/WritingAnLLVMBackend.html, 2013.

[11] NeXT Corporation. Object-Oriented Applications Development With NeXTstep. Technical report, 1991.

[12] Telecompaper. Lenovo unveils smart TV running Android 4.0. http://www.telecompaper.com/news/lenovo-unveils-smart-tv-running-android-40–848497, 2012.

[13] The Android Open Source Project. Bytecode for the Dalvik VM. http://source.android.com/tech/dalvik/dalvik-bytecode.html, 2007.