An LLVM-based Just-In-Time Compiler for the Dalvik Virtual Machine

Author: Matthew Shepherd
Exam Number: 3585047
Supervisor: Hugh Leather

4th of April 2013

Abstract

This project involves the implementation of a new just-in-time compiler for Android's Dalvik virtual machine. A translator was implemented to convert the Dalvik bytecode into LLVM bitcode. LLVM was then used to perform optimisations on the bitcode before it was compiled to native code. A 'trampolining' generator was also implemented to generate assembly code to invoke the LLVM-compiled code and to deal with the result. The aim of the project was to investigate the possibility of improving just-in-time compiled code by performing more aggressive optimisations. The results of experiments show a 7% decrease in execution time in code generated by the new compiler compared to the code generated by the existing just-in-time compiler.

Contents

Chapter 1

Introduction

Android is a popular mobile operating system developed by Google. Currently Android holds a large market share, with some sites claiming a 68% market share [4]. One of Android's key features is its Dalvik Virtual Machine, a virtual machine used to process user applications. By using a virtual machine to handle user applications, Android can be built to run on a wide variety of platforms and hardware. Android is also an open source operating system, allowing phone manufacturers to develop and ship devices running the latest version of the operating system free of charge.

This project looks at a specific part of the Android operating system, Dalvik's Just-In-Time Compiler. With version 2.2 of Android (Froyo) Dalvik came bundled with a Just-In-Time Compiler. Google boasted speed boosts of between 2-5x for CPU-bound tasks [1] compared to the same tasks being run on the interpreter by Android version 2.1. The goal of this project was to develop a new JIT Compiler built using the LLVM infrastructure to provide more aggressive optimisations to code. The benefits of more efficient code include faster execution time, smaller memory footprint, better response times and for mobile devices this can result in longer battery life. As a result of this project, a 7% decrease in execution time was observed in an experiment.

1.1 Project Description

Android's process virtual machine, Dalvik, is one of the most important parts of Android. Android is commonly used on mobile phones, tablet computers and netbooks. Android applications have their code stored in the Dalvik executable formtat (.dex), which is then executed on the virtual machine. Recently, Google added a Just-In-Time (JIT) Compiler to the virtual machine. The JIT compiler is a piece of software that converts the program code to run on the device natively, rather than to be run within a virtual machine.

LLVM (formerly the Low Level Virtual Machine) is a compiler infrastructure. It most common usage is as a robust toolset for building compilers. It currently supports many frontends for programming languages and supports many architectures in its back end. It also includes the LLVM IR (Intermediate Representation), a low level representation used as an intermediary between the frontend and backend. As a result it can be used to generate compilers for existing languages to new architectures, compilers for new languages to existing architectures and anything inbetween.

In this project a new just-in-time compiler for Dalvik will be developed using LLVM. Code that is to be just-in-time compiled will be translated into a format that LLVM can work with. LLVM will be used to optimise the translated code before LLVM compiles it into native code.

1.2 Aims & Goals

The primary goal of this project was to develop a just-in-time compiler for Android using LLVM. The new just-in-time compiler will be able to accept a Dalvik trace and compile it for native exection. By using LLVM, traces can receive a higher level of optimisation before compilation and improve the performance of the trace.

Chapter 2

Background

This chapter describes terminology used in this report and describes technology used. Information on general terminolgy is covered as well as more in-depth information about the Android operating system and the LLVM compiler infrastructure.

2.0.1 Just-in-time Compilation

Just-in-time (JIT) compilation is a technique used to provide performance benefits to interpreted programs. Just-in-time compilers work by compiling sections of the interpreted program into native machine code, allowing these sections of code to be executed much faster as native code. Two types of just-in-time compilation are trace-based (TB) and method-based(MB). Method-based JIT works by JIT compiling whole methods, normally selected based on their frequency of execution. One main issue with MB JIT is that because whole methods are compiled, some time may be wasted compiling parts of the method which may rarely be executed. For example:

```
1  function doSomething(Parameter)
2  {
3      Value result;
4          if Parameter is Type 1:
5                  result = Simple;
6          otherwise:
7                  [A very complex computation, many lines of code]
8                  result = Complex;
9          return result;
10 }
```

Figure 2.1: Example to highlight an issue with method-based JIT compilers

This function may be called frequently, with the 'Simple' path taken 99% of the time. This method may be JIT compiled but the performance benefit may be minimal compared to the amount of time spent compiling the 'Complex' side of the method.

Trace-based JIT compilation takes a different approach to decided what code to compile. A standard trace-based JIT compiler works with traces, which are a linear sequence of instructions that always occur in the same order. This technique can build traces that can span over several methods, or be contained in a simple loop. A TB JIT is better equipped to determine which parts of the program are being executed the most frequently, as a single trace can encompass multiple method calls and describe a frequently followed code path.

```
1  function main()
2  {
3          done = false
4          counter = 0
5          while( !done )
6                   counter = increment(counter, 1)
7                  power = pow(counter, 2)
8                  if(power > 1000000)
9                          done = true
10     return 0;
11
12 }
```

Figure 2.2: Example program

```
1  function pow(base, index)
2  {
3          return base^index
4  }
```

Figure 2.3: Implementation of *pow* for the example

```
1  function increment(value, amount)
2  {
3          return value + amount
4  }
```

Figure 2.4: Implementation of *increment* for the example

For example, if this code was JIT compiled with a method-based JIT, only the pow and increment functions could be JIT compiled (assuming the main method is only called once), but a trace-based JIT compiled could potentially produce (with optimisations):

For example, if a method-based JIT compiler was used on the above code it may only JIT compile the *pow* and *increment* methods (depending on how methods are selected). A trace-based jit might be able to select:

```
1 trace:
2 counter = counter + 1
3 power = counter^2
4 if(power < 1000000)
5         goto trace
```

Figure 2.5: What the trace might look like

As its trace, inlining the function calls and producing a JIT-compilable trace that covers the most intensive sections of the program. This is only a theoretical example that shows the potential usefulness of trace-based JIT compilers.

2.0.2 Process Virtual Machine

A process virtual machine is a type of virtual machine designed for execution of a single process. A virtual machine is, in essence, a simulated computer. There exist different types of virtual machine, for example, there are virtual machines designed for running a 'client' operating system within a 'host' operating system. The type of virtual machines referenced in this report are Process Virtual Machines, which simulate a simple processor using an area of memory as its general purpose registers.

2.1 Android

Android is a popular mobile operating system developed by Google. Android is an open source linux-based operating system for smartphones, tablets and other mobile computers.

2.1.1 Dalvik

Dalvik is Android's process virtual machine. It is used to execute Android applications using an interpreter. With the release of Android version 2.2, Froyo, Dalvik had a simple just-in-time compiler built in. The performance benefits of this just-in-time compiler allows cpu-bound code to execute 2-5x faster than if the code was interpreted [1].

Dalvik simulates the registers of the process by using an array of memory to store all the registers needed by by application it is currently executing.

2.1.2 Java

Java is a high-level object-oriented programming language. Java compiles to a bytecode representation designed for execution on the Java Virtual Machine. Java is used as the primary development language for Android applications.

Applications developed for the Android platform are generally written in java. In order to compile an application to a format suitable for installation in Android, the application goes through several steps. The first step involves compiling the application using the Java Compiler. This produces Java class files from the source. As the Java Virtual Machine is a stack-based virtual

machine, some conversion must be performed on the Java bytecode in order to make it suitable for execution on a register-based virtual machine. This conversion is done automatically by the dex tool in the toolchain.

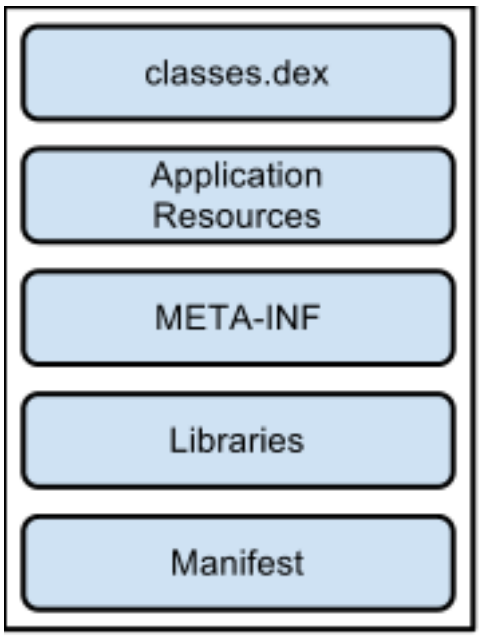

Figure 2.6: APK Components

Finally, the APK builder builds an APK from the Dalvik bytecode, project resources, and any 3rd party libraries the application depends upon. Once the package has been built, it is signed as debug or release before the package is zipaligned [2]. Zipalign performs optimisations on the package. Uncompressed data such as images are aligned to 4-byte boundaries, this improves the efficiency when loading the application into memory.

2.1.3 Dalvik Bytecode

In this report the term bytecode is used to reference the Dalvik bytecode that comprises an Android application. The bytecode is the executable code contained within an APK, Android's application package format. Dalvik bytecode is a register-based bytecode, in comparison to JVM's stack-based bytcode.

2.1.4 Dalvik Interpreter

Dalvik uses a threaded interpreter with two handler tables. A handler table is a collection of 64 byte aligned handlers. A handler is a block of native code used to execute a certain instruction type. The main handler table is composed of the handlers used to execute each instruction type. For example, the opcode *move* has a numerical value of 1, which points to the second entry in the handler table (the first being at position 0). The interpreter branches to the correct position in the table to handle the bytecode instruction. When Dalvik is started it uses the 'Standard' handler

to execute bytecode. The second handler is used when the interpreter is in tracing mode. Each instruction is executed one at a time by incrementing the Program Counter (PC), and using the current handler table to jump to the assembly needed to execute the instruction. Each instruction is also checked to see if it is a trace head.A trace head is regarded as any bytecode instruction that is a branch target or the start of a method. If the current instruction is indeed a trace head, its threshold counter is updated and the instruction is handled as normal. Eventually a trace head counter will reach its threshold, and the virtual machine switches from interpretation mode into tracing mode. Figure 2.7 shows a flowchart of the operating of the interpreter.

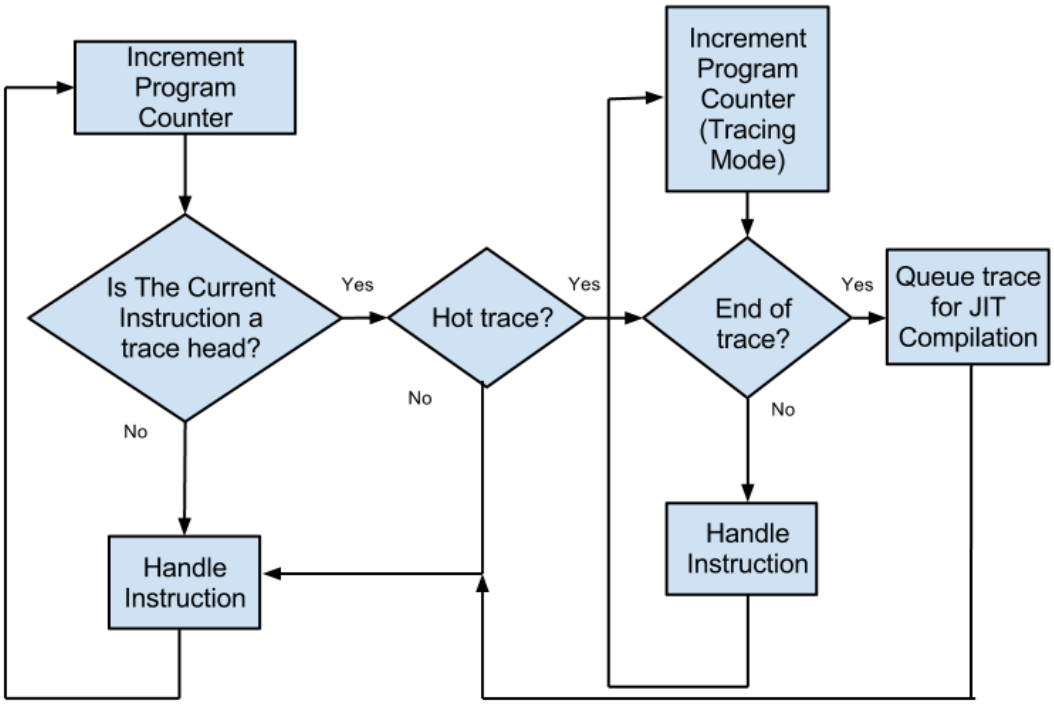

Figure 2.7: Flow chart for Interpretation and Tracing of a bytecode program

Trace Building

During tracing, the second handler is used to help build the trace. Once the PC has been incremented, the interpreter jumps to the tracing handler instead. If the instruction is a trace head, its PC is stored in a table to record the number of instructions reached from that position, the tracing handler then begins to build the trace. The tracing hanlder then jumps back to the standard handler to execute the instruction. For each subsequent bytecode instruction the instruction count in the table entry is incremented and the instruction is added to the trace. Trace building is terminated by certain instructions such as a conditional branch, method invoke, a switch, a return or if the trace is too large.

2.1.5 Dalvik Just-In-Time Compiler

The Dalvik JIT compiler is responsible for compiling traces constructed by the interpreter into native machine code. The Dalvik JIT uses two intermediate representations to produce code ready for compilation. The first representation used is the Middle Intermediate Representation (MIR). MIR closely resembles the Dalvik bytecode, and is essentially a decoded version of the original bytecode. The MIR is then converted into the Lower Intermediate Representation (LIR) which is closer to an assembly format, but not for a specific platform. During the conversions, simple optimisations are conducted on the code. When all the code has been converted to LIR, it is passed to the platform specific code generator to compile and install the JIT compiled trace.

2.1.6 Trace Chaining

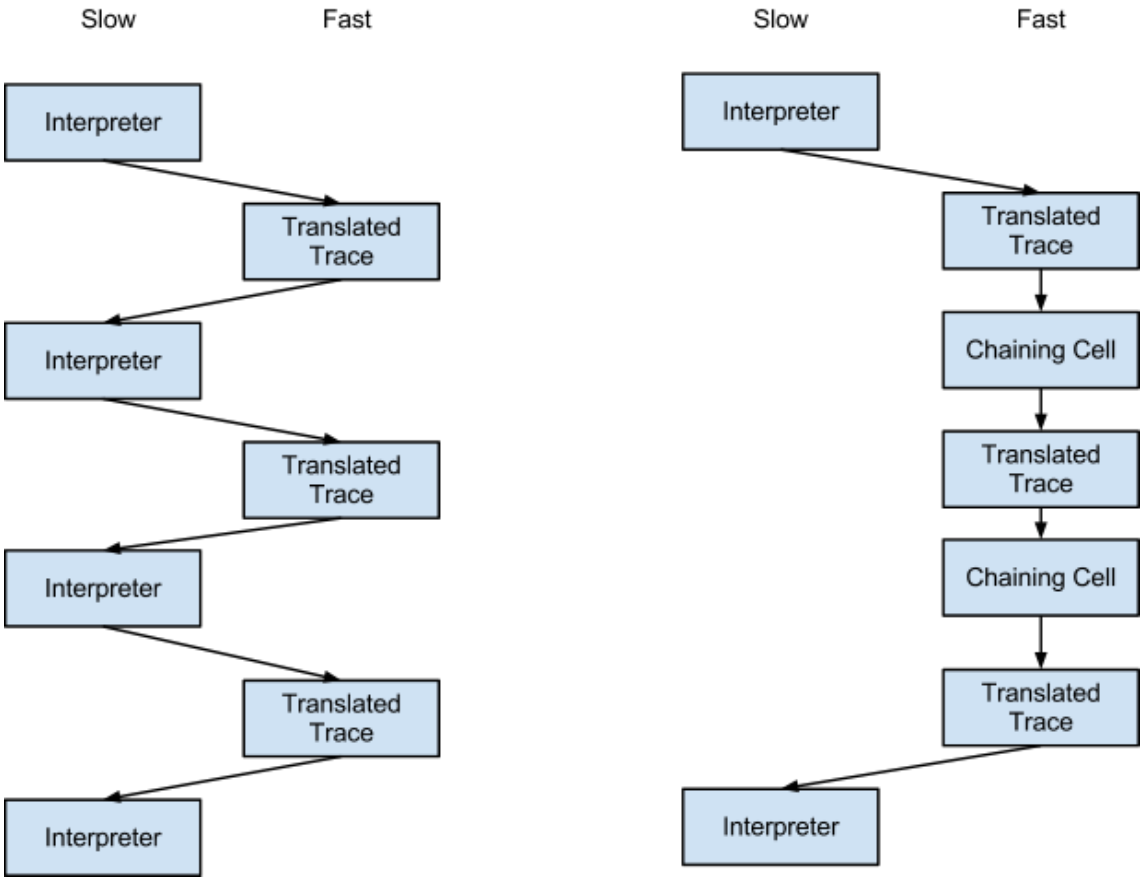

Figure 2.8: Benefit of Trace Chaining

Trace chaining is a technique used to link compiled traces together, allowing a series of JIT compiled traces to be executed without returning to the interpreter. Figure 2.8 shows what would happen if chaining cells were not used. Figure 2.9 shows what a typical chaining cell looks like. The first line of the chaining cell acts as a placeholder, and by default branches the 4th line. The 4th line loads the address of the JITToInterp handler, where r6 is the thread pointer and the literal offset

is the location of the handler inside the thread structure. A branch and link is performed to jump to the handler while saving the current address to the link register.

```
1 b L1
2 <padding>
3 L1:
4 ldr r0, [r6, #100]
5 blx r0
```

Figure 2.9: Typical Chaining Cell

JitToInterp Handler

The purpose of the JitToInterp handler is to perform one of two jobs. It uses the program counter to check if there exists a compiled trace for the next instruction. If not, then it performs a transition back to the interpreter. However, if there exists a compiled trace, then it attempts to chain the traces.

As a branch and link was performed to reach the handler, the address of the previous branch instruction is now stored in the link register. The handler uses this address to determine the address of the first instruction of the cell, the *b L1* instruction. The handler directly modifies this instruction to replace its target with a new address. The new address will be the address of the next trace. By performing this modification, the next time the trace is executed it will be able to branch straight to the next trace without having to use the handler or returning to the interpreter. Figure 2.10 gives an example of what the modified cell would look like.

```
1 b 0x20003042
2 <padding>
3 L1:
4 ldr r0, [r6, #100]
5 blx r0
```

Figure 2.10: Typical Modified Chaining Cell

2.2 LLVM

This section describes LLVM, its features and its role in this project.

2.2.1 About

LLVM, formerly known as the Low Level Virtual Machine, is a compiler toolset and framework. It is commonly used for the construction of compilers as there exist frontends for all major programming languages and backends to support many platforms. LLVM also boasts a robust Intermediate Representation (IR) and a powerful set of optimisations that can be performed on its IR.

2.2.2 Bitcode LLVM IR

The Intermediate Representation (IR) used by LLVM is also known as bitcode. The bitcode is a low-level representation that adheres to Single Static Assignment, meaning that a variable can only be assigned once. The use of SSA simplifies the language and makes it easier to perform some optimisations on and helps simplify some stages of compilation.

2.2.3 Optimisations

LLVM has a large variety of optimisations available to be performed on its bitcode. These optimisations range from removing 'dead' code, combining instructions, simplifying loops, function inlining etc. These optimisations are performed individually, but some are designed to complement one another.

2.2.4 JIT Compiler

LLVM also includes an Execution Engine and JIT compiler. The Execution Engine can be used to interpret a bitcode, but it can also be used to JIT compile a bitcode function. When a bitcode function is JIT compiled, a pointer to the function is returned and can be called.

Chapter 3

Related Work

3.1 Introduction

This chapter examines other works to improve the performance of bytecode process virtual machines. Techniques used include ahead-of-time (AOT) compilation, method-based JIT compilation, and trace-based JIT compilation.

3.1.1 Hybrid Just-In-Time Compiler [5]

This article describes a method-based JIT compiler for Android built using LLVM. This project expands on the existing JIT compilation capabilities by adding a method-based JIT compiler that selects methods based on the number of times they are invoked. When the number of times a method has been invoked reaches the threshold, the method is submitted for compilation. By using both JIT compiler types in conjunction, the original performance of Dalvik's JIT compiler is preserved while providing the benefit of a method-based JIT compiler in cases where it can provide a full performance benefit.

This hybrid JIT compiler relates to the current project through its use of LLVM to optimise and compile Dalvik bytecode. It uses a similar technique to produce bitcode from Dalvik bytecode, but instead of using a trace provided by the interpreter, it works with an entire method's bytecode.

THe main issue with this work is that it adds a method-based compiler. The project might have had an even greater performance benefit if it JIT compiled traces instead of only working with methods. Implementation would have involved creating a bytecode to bitcode translator, and that work may have been put to better use as a trace compiler.

3.1.2 A method-based ahead-of-time compiler for Android applications [6]

This article describes Icing, a method-based ahead-of-time compiler for Android. It is designed to improve performance of Android applications by profiling and converting methods of an application from their original DEX format into C, before compiling these methods with GCC to be invoked by the Java Native Interface (JNI). The stated performance benefit of this project is between 25% to 110% for hot methods.

This ahead-of-time compiler project relates to the current project through its goal of producing fast, native code from the Dalvik bytecode. This technique has the benefit of offloading the optimisation and compilation of the methods to the computer that compiles the application from source. However, as some of the code is now native, the compiled code is locked to a specific platform and would not run on another platform such as x86.

One of the main issues with this technique is that the profiler may not select the best methods for compilation, it may end up selecting methods that will have no real performance benefit or negatively impact performance. The Java Native Interface can have huge overheads when dealing with certain structures, for example accessing a java array within native code requires the array to be copied before the method starts and for the array to be copied back upon completion of the native code. Overall this may result is poorer performance compared to the method being left as bytecode.

3.1.3 Adaptive Multi-Level Compilation in a Trace-based Java JIT Compiler [3]

This article describes a dynamic trace-based JIT compiler that alters its criteria based on the execution of the current program. When a program is started small traces are JIT compiled with low levels of optimisations to prevent slow-startups. During execution hot paths are identified and long traces built to encapsulate these hot paths. These traces are then compiled with high levels of optimisations. Longer traces are carefully selected using a control flow engine called TTgraph. The stated performance improvements this project provide a program can be up to 58% at peak performance, and 22% on average. This work is based on JIT compiling JVM traces, and so is working with a stack-based bytecode. This JIT compilation technique appears to work well for JVM, but the same performance benefits might not carry over to a register-based bytecode.

3.1.4 Swift: A Register-based JIT Compiler for Embedded JVMs [7]

This article describes a project that implements a register-based just-in-time compiler for Java Virtual Machines. A study determined that most java methods compiled into a register-based bytecode can be completed using 11 physical registers. By mapping virtual registers to physical registers, the rest of the Java bytecode can be translated to native machine code. Swift is implemented to handle the Dalvik bytecode format DEX, as DEX is a register-based bytecode and is the official bytecode format for Android application. Swift claims a 3.13x speedup over the interpreter on certain benchmarks, and a speedup of 1.42x over another Andoird JIT compiler known as JITC-Droid.

3.1.5 Summary

The different techniques used by these related works show that there are multiple ways of improving a bytecode programs performance with some form of compilation. While just-in-time compilation is the focus of most of these papers, the Icing project shows that ahead-of-time compilation is also a viable tactic.

Chapter 4

Design & Implementation

This chapter describes the design and implementation of the LLVM-based JIT Compiler. It looks at the general design of the implementation and how each component works and integrates into the system as a whole. The programming language used for implementation is C++, the same language used to implement most of Dalvik. Use of C++ was necessary to integrate the new JIT compiler with the existing process virtual machine. The following section inspects the general design of the new JIT compiler. Each component is then described in greater detail in its contained sections.

4.1 Overview

The main task of the new LLVM -based JIT compiler is to accept a trace and produce native machine code that performs the same task as the original trace. As LLVM cannot compile a simple block of bitcode, a trace must be represented as a function. This is achieved by producing a function with a pointer to the registers and translating the bytecode into LLVM bitcode that will perform the same operations. The issue with representing the trace as a function is that the existing system expects to branch straight into the compiled trace. To tackle this inherent incompatibility, a block of native code is generated alongside the function. This block of native code handles the calling of the function and dealing with the transition onto the chaining cells. The returned value of the function identifies a specific chaining cell to follow out of the compiled trace.

Figure 4.1 is a visual representation of what a compiled trace may look like when compiled by Dalvik's JIT compiler and the LLVM-based JIT compiler. The arrows pointing into the blocks of code represent the entry point, and the trailing arrows pointing outwards represent the transition back to the interpreter or into another compiled trace.

4.1.1 Translator

The main purpose of the translator is to produce a suitable bitcode representation of a Dalvik trace. It does this by looping through the trace, adding the necessary bitcode instructions to implement each of the Dalvik instructions. The translator also produces information used for the

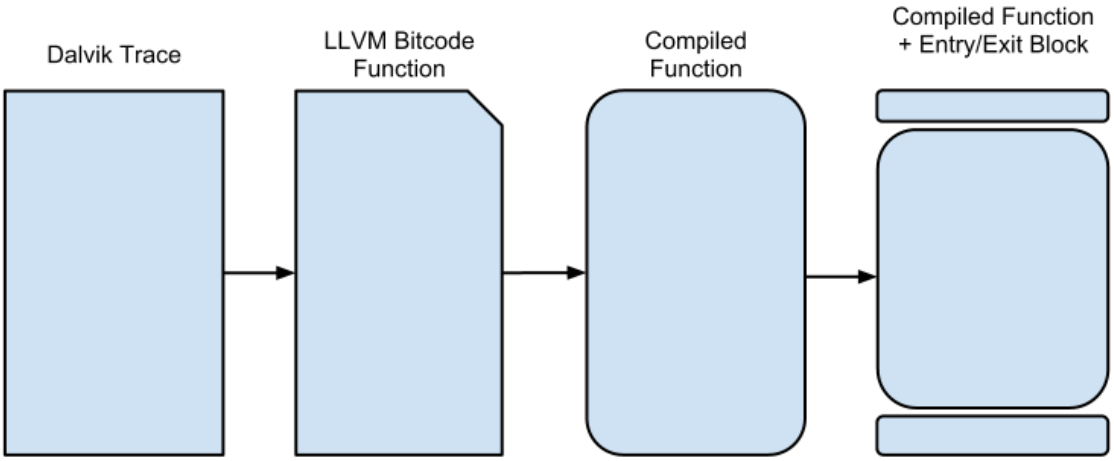

Figure 4.1: The stages of compiling a trace

generation of chaining cells at a later stage.

4.1.2 Optimisations & Compiler

This part of the system works with a complete bitcode function produced by the translator. It verifies the produced bitcode and performs a set of optimisations on the bitcode. Once the optimisations are complete the function is scheduled for compilation by LLVM's JIT compiler. The resulting function pointer is then passed onto the Trampolining code generator.

4.1.3 Trampolining

The trampolining component handles interactions between the compiled function and the rest of the Dalvik virtual machine. It acts as the entry to the compiled trace, calls the JIT function and handles the result of the function. The trampolining block is generated after the trace has been translated and compiled. It uses information gathered during translation to generate the chaining cells and the logic to branch to the necessary chaining cell based on the identifier returned by the function.

4.1.4 Summary

The result of these subsystems is the production of native code that can perform that same task as the original trace, but with a faster execution time compared to it being interpreted. Figure 4.2 gives an impression of what the compiled trace might look like in memory and how it fits together.

4.2 Translator

The translator is the component responsible for translating a bytecode trace into an LLVM function. The LLVM API is used to build a function that can act as a translated trace by translating each

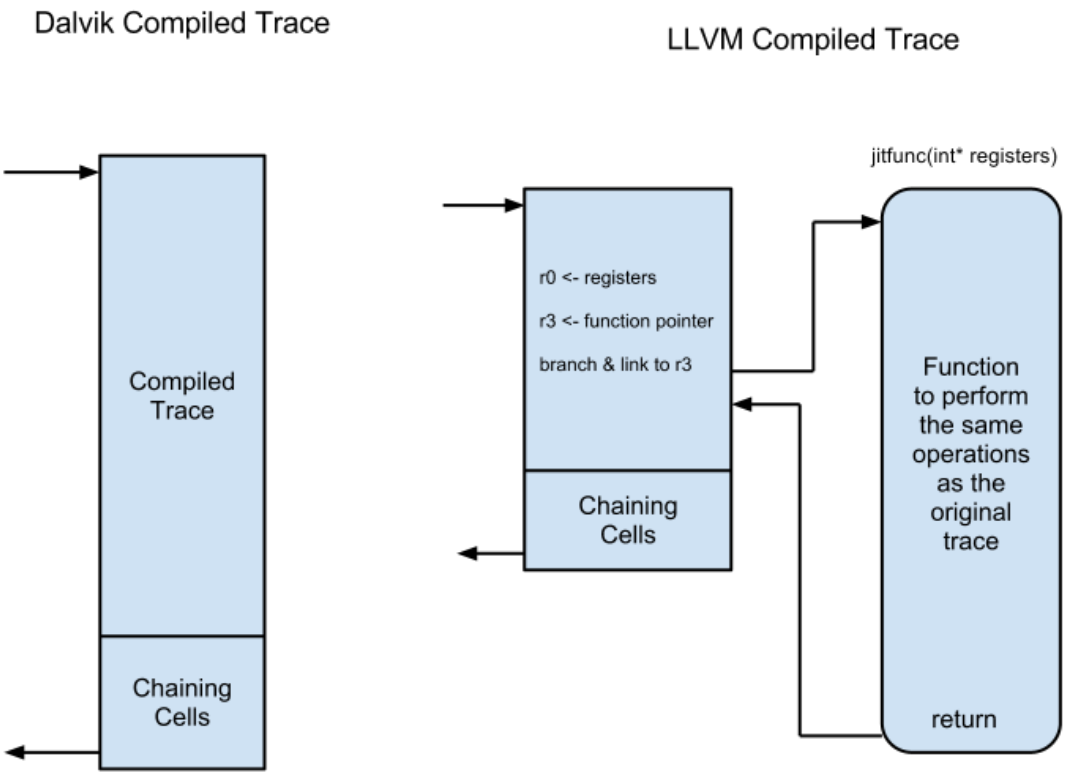

Figure 4.2: What the compiled trace looks like

instruction into the corresponding LLVM bitcode.

```
1 int jitfunc(int* registers)
```

Figure 4.3: JIT Function Prototype

The function prototype described in Figure 4.3 is a function that takes a single parameter, a pointer, which points to the memory location that contains the software registers used by the application. The return value of the function is a chaining cell identification number. This value is later used to branch to the correct chaining cell when the trace-function has completed execution. The first stage of translation is initialisation, which is to use the LLVM API to create the function prototype as described above. In order to create the function prototype, a module must first be defined. The module contains information pertaining to the compilation stage, such as the target platform and the data layout. A new LLVM function is created and added to the module. The final step of the initialisation is to create the first basic block to hold the first necessary instruction, an array allocation instruction to permit access to the register bank.

```
1   %registers = alloca i32*
2   store i32* %regs, 132** %registers
```

Figure 4.4: Preparing register access

This snippet prepares the %registers variables so that it can be used to access the register array later in the function.

The next stage of translation is the main loop, which iterates through the trace. The provided trace is still in a pseudo-bytecode format, and so each instruction must be parsed and decoded into a more meaningful and useful format. The decoded information is stored using the MIR representation provided by the existing JIT compiler. As some instructions have similar implementations when in bitcode format, some opcodes have been grouped together in the translator. For example some of the basic integer operations such as add-int, sub-int and mul-int have identical bitcode representations except for the instruction that performs the arithmetic, otherwise they all use the same array access to load and store the values. By grouping these kinds of instructions together the translator was simpler to implement and test. Due to Dalvik's registers being stored in memory, the first step of most translations is to generate bitcode to fetch the registers from memory. The operands of the bytecode instructions are integers referencing elements in the register array. The first bitcode instructions to be generated store the register number in a constant variable, for each operand in the bytecode instruction. This is achieved with the following bitcode instructions figure

```
1   %vA = alloca i32
2   store i32 0, i32* %vA
```

Figure 4.5: Creating a constant variable to hold the number 0

This snippet stores the register number referenced by the first operand in the Dalvik instruction.

4.2.1 Arithmetic Instructions

Arithmetic instructions are all implemented in a similar fashion, this section will look at the implementation of a simple add Dalvik instruction.

```
1  add-int v0, v1, v2
```

Figure 4.6: Format of the add-int DEX instruction

This Dalvik instruction wants to store the result of adding registers v1 and v2 into v0. This in bitcode would look like:

```
1    %vA = alloca i32
2     store i32 0, i32* %vA
3    %vB = alloca i32
4     store i32 1, i32* %vB
5    %vC = alloca i32
6     store i32 2, i32* %vC
```

Figure 4.7: Creating constant variables to hold register numbers

This snippet prepares the variables %vA, %vB, and %vC to hold the register numbers referenced in the Dalvik instruction. With the indexes of the registers used, the current values of the registers v1 and v2 can be loaded from the array:

```
1    %vBptr = getelementptr i32* %registers, i32 %vB
2    %opB = load i32* vBptr
3    %vCptr = getelementptr i32* %registers, i32 %vC
4    %opC = load i32* vCptr
```

Figure 4.8: Loading the values of two registers

Looking at line 1: the getelementptr bitcode instruction returns a pointer to the element index stored in %vB from the array referenced by

```
1 %result = add %opB, %opC
```

Figure 4.9: Performing arithmetic

This bitcode instruction performs an add operation on %opB and %opC, storing the result in %result.

```
1 %vAptr = getelementptr i32* %registers, i32 %vA
2  store %result, %vAptr
```

Figure 4.10: Storing a value in a register

This last code snipped fetches a pointer to the register references by vA (in this case v0), and stores the result of the add instruction in register v0.

4.2.2 Array Access Instructions

Accessing values from an array are done in a similar fashion to accessing values from the register bank itself. The reference to the array is loaded from the second operand register. This reference is then used in a getelementptr bitcode instruction using the index supplied in the third operand register as the offset:

```
1 %vBptr = getelementptr i32* %registers, i32 %vB
2 %array = load i32* vBptr
3
4 %vCptr = getelementptr i32* %registers, i32 %vC
5   %offset = load i32* vCptr
6
7 %elementptr = getelementptr i32* %array, i32 %offset
```

Figure 4.11: Getting a pointer to an array element

The %elementptr variable can now be used to load a value from the array or to store a value in the array.

```
1 %element = load i32* %elementptr
2
3  store %value, %element
```

Figure 4.12: Storing a value in an array

4.2.3 Flow Control

Flow control instructions require more than just a straightforward translation. If-statement instructions have a branch target if their condition is met. In bytecode this is represented as an offset relating to the current position in the Dalvik bytecode. If these branches are referencing a location outside of the current trace, a chaining cell needs to be generated by Dalvik. In this case the branch target information is saved into a data structure, this information includes the target offset, the control flow type as well as a pointer to the MIR instruction used. The if-statement in bitcode uses a conditional branch to choose between two new basic blocks, each basic block will store a different constant in the return value, depending on the path taken.

```
1   %4 = icmp slt i32 %3, 0
2     br i1 %4, label %TRUE, label %FALSE
3
4
5   ; <label>:TRUE
6     store i32 0, i32* %1
7     br label %RETURN
8
9
10  ; <label>:FALSE
11    store i32 1, i32* %1
12    br label %RETURN
13
14
15  ; <label>:RETURN
16    %8 = load i32* %1
17    ret i32 %8
```

Figure 4.13: Bitcode representation of a if-statement, the return value determines which chaining cell is followed

4.2.4 Return Instructions

Return instructions are handled by saving the register to be returned into the same structure used for the generation of chaining cells at a later stage. Rather than generating a chaining cell, a 'common return' is instead generated by the Dalvik code generator and compiled along with the trampolining block.

4.3 Optimisations & Compiler

Once the trace has been translated into bitcode format, the generated function can now be verified and optimised.

The LLVM verifier is invoked to ensure that the generated code is consistent and logical, a useful tool included with LLVM to prevent 'bad' code from being compiled. Type-checking is also performed by the verifier to ensure that all operations performed on all variables are done so with the correct types. Another verification test performed by the verifier is to ensure that that code is in correct Single Static Assignment form.

If the function is verified to be ok, the next stage is to perform LLVM optimisations on the bitcode. These optimisations are known as transformation passes and are included with LLVM. The passes performed are detailed in section [opt]

Once the optimisation passes have been completed, the bitcode is finally ready for compilation. An instance of the LLVM Execution Engine is created to handle the JIT compilation of the bitcode. The engine is created and the module is passed to it. A pointer to the JIT function is requested and

then the engine will attempt to compile the function. If it is successful a pointer to the installed code is returned. If not then a null pointer will be returned. The function pointer is placed inside a data structure passed in with the trace, and the translator and compiler return true to indicate that compilation was successful.

4.4 Trampolining

The purpose of the trampolining code is to act as an entry block to the LLVM-compiled function, and to handle the returned value of the function. This block is generated in an assembly format called ArmLIR, a representaion of ARM assembly.

As the LLVM function will use r0 as it's parameter, r0 must be loaded with the address of the software registers. The register rFP holds this address and so a mov instruction is used to copy the address.

The next step is to load the address of the LLVM function into a register. The address of the LLVM function is stored as a constant. Register r3 is used to store the function address.

Once the address of the JIT compiled function is stored in a register, a branch and link can be used to call the function.

Upon completion of the function, r0 will now hold the identifier of the chaining cell that should be followed. This is handled by a short series of conditional branches, for each chaining cell identification number.

Chaining information saved during the translation stage is used to generate the required chaining cells, with their addresses being hard coded into the assembly as the branch targets.

First of all, the conditional branches to each chaining cell are created using the ArmLIR representation used for creating the entry block. The target information for each branch isn't known at this stage, so a pointer to each ArmLIR branch instruction is saved for completion later.

First, the conditional branches are generated, but with the targets left blank. The chaining cells are then generated. when a chaining cell is generated, an ArmLIR pointer is returned. These pointers are stored in a list. Once the cells have been generated, the conditional branches have their targets set to the corresponding chaining cell.

Chapter 5

Experimental Setup

5.1 Introduction

This chapter details the experimental setup used to generate results from the new LLVM-based just-in-time compiler. The goal of the experiments is to gather run-time statistics on the performance of both just-in-time compilers. This is achieved by performing benchmarks and recording information such as execution time, compilation time, and memory usage. The experimental information will provide insight into the benefits and hindrances of the new LLVM-based just-in-time compiler compared to the existing just-in-time compiler.

5.2 Hardware & Operating System

The device used for running the experiments is the Google Nexus 7 tablet. The Nexus 7 is an ARM-based device with a 1.2GHz quad-core Cortex-A9 processor, 1GB of DDR3L RAM and the Nvidia Tegra 3 System-on-a-Chip (SoC). The operating system used for the experiments is Android Version 4.1.2, Jellybean. This was the latest version of android available at the start of the project. As the Android Open Source Project is designed with the Nexus devices in mind, the Android source code will compile and run on the Nexus 7 without modification.

5.3 Benchmarks

Several benchmarks were performed to test the abilities of the new JIT compiler. The same benchmarks are also executed using the standard JIT compiler and with JIT compilation switched off to act as a baseline. Each of the benchmarks specified in this section were included in the application source three times, each with a modified method name with the tag 'LLVM', 'JIT or 'SLOW' appended. The part of dalvik that handles compilation work orders was modified to detect the benchmarking application, so that statistics about the benchmarks could be recorded. When the benchmarking application is detected, further checks are performed on the method name of each trace. Traces from methods marked SLOW have JIT completely disabled. This is to measure the execution time if the the entire method is handled by the interpreter. Traces with the tag 'JIT'

are passed to Dalviks JIT compiler while traces with the tag 'LLVM' are left to be compiled by the new JIT compiler but with statistic recording enabled.

5.3.1 Factorial

Factorial (Figure 5.1) was the first arithmetical benchmark to be written. It computes the factorial of a number by multiplying the supplied integer with every preceding interger down to the number 1.

```
1     private int Factorial(int n) {
2         int res = 1;
3         int i = n;
4         while(i>=2) {
5                 res *= i;
6                 i -= 1;
7         }
8         return res;
9     }
```

Figure 5.1: Factorial Implementation

```
1     private void inlineFactorial()
2     {
3         for(int i = 0; i < RUNS; i++) {
4                 int res = 1;
5                 int j = i%FACTORIAL_MAX;
6                 while(j>1) {
7                         res *= j;
8                         j -= 1;
9                 }
10
11        }
12    }
```

Figure 5.2: Factorial Inline Implementation

The benchmark uses the Factorial code in Figure 5.1 but removes the method call. The complete benchmark code is listed in figure 5.2. The purpose of this benchmarks is to test arithmetic combined with variable iteration counts. In all benchmarks *RUNS* is set at 1000000.

5.3.2 Fibonacci

Fibonacci is another arithmetic-based benchmark. The Fibonacci benchmark calculates the Fibonacci number at position indicated by the parameter. This benchmark is executed 1000000

times, varying the input number on each iteration. Figure 5.3 shows the implementation.

```
1  private int Fibonacci(int t){
2      int a = 0;
3      int b = 1;
4      int c = a+b;
5      if(t==0){return a;}
6      if(t==1){return b;}
7      for(int i = 1; i < t; i++) {
8          c = a + b;
9          a = b;
10         b = c;
11     }
12     return c;
13 }
```

Figure 5.3: Fibonacci Implementation

5.4 Data Capture

The run-time statistics of the just-in-time compilers were recorded using *logcat*. *Logcat* is Android's debug logging tool which can be accessed using the Android Debug Bridge (adb). To record the needed run-time statistics, a combination of these tools and the command line tool *grep* were used to filter the logs to only output the run-times from the Application, the compilation times from Dalvik, and the memory usages from Dalvik.

```
1  double starttime = System.nanoTime();
2
3  /* perform benchmark */
4
5  double endtime = System.nanoTime();
6  double milleseconds = (endtime - starttime)/1000000;
```

Figure 5.4: Calculating benchmark run time

The run time of each benchmark is calculated using Java's system time as shown in Figure 5.4. The runtime is then printed to the debug log with the 'BENCHMARK' tag along with a tag to identify the method and the run-type ('LLVM', 'JIT' or 'SLOW'). Execution time is measured in milleseconds as the benchmarks tended to execute in the range of a few hundred milleseconds to just over 1 second.

Calculating compilation time is done in a similar fashion. When a compiler work order is dequeued, the system time is recorded as the start time. When either the standard JITC or the LLVM JITC completes, the system time is recorded as the end time. The time since the work

started is calculated and the result is printed to the debug log with the tags 'COMPILE', the method name stored within the trace structure, which JITC was used and the PC of the trace head. Figure 5.5 details how the time was calculated. The compilation time is calculated by summing the times recorded for each JITC type, using the total time taken to compile traces within the benchmark method as the timing metric.

```
1  bool dvmCompilerDoWork(CompilerWorkOrder *work)
2  {
3      time_t start, end;
4      start = clock();
5      /* carry out work order */
6
7      end = clock();
8      float timediff = ( (float)end - (float)start ) / 1000.0f;
9      /* print out compile time, which compiler was used,
10             method name, pc value */
11
12     /* rest of DoWork function */
13 }
```

Figure 5.5: Calculating compilation time

Memory usage by each of the JITC types is recorded with messages printed to the debug log. When a trace is compiled the standard Dalvik compiler stores the size of the installed code in the trace structure. This size is printed out to the debug log with the 'COMPILE' tag along with the method name and the PC value. For the LLVM-based JIT the size of the trampolining block is printed out, and the size of the LLVM function is found inside an information structure generated by LLVM when the function is compiled.

5.5 Bitcode Optimisations

The optimisations chosen to be performed on the translated code are all optimisations designed to optimise basic blocks. The right set of optimisations would allow the translated code to be executed faster. However performing too many optimisation passes would result in the compilation and optimisation stage taking longer and being more processor intensive.

5.5.1 Dead Instruction Elimination & Dead Code Elimination

Dead instruction elimination is a very useful optimisation to perform on the translated code. This optimisation analyses the code to determine which instructions are 'dead' and are never performed or have no effect. This is a useful pass as it will help reduce the final code size. The dead code elimination is a secondary pass performed after dead instruction removal. For example, the dead instruction elimination pass may mark a conditional branch to be removed due to the condition

never being fufilled (for example comparing two constants). The dead code elimination pass would detect the removal and realise that the branch target is now never executed and would mark the whole block for removal too.

5.5.2 Dead Store Elimination

Dead store elimination is an optimisation pass that attempts to remove uneeded memory store operations. This optimisation can be very beneficial as memory operations can be considered time-expensive compared to register based operations. An example of a dead store elimination could be a block were a memory location is written to a value and is immediately replaced with a different value.

5.5.3 Combine Redundant Instructions

Combine redundant instructions is an optimisation pass that attempts to combine instructions where possible, for example:

```
1 %b = add %a, 1
2 %c = add %b, 1
```

Can be combined to:

```
1 %c = add %a, 2
```

(This is assuming that %b is never used again) This kind of optimisation can be very beneficial in arithmetic and logical intensive sections of code, as many simple instructions can be condensed down into a few more complex instructions.

5.5.4 Promote Memory to Register

This optimisation promotes memory references to register references. This optimisation should be beneficial to JIT compiled traces, as all software register reads and writes involve memory access and are generally a series of loads and stores. This optimisation can remove unecessary stores such as storing a software register value and immediately restoring it to a different value.

5.5.5 Constant Merging

This optimisation looks at constant values, and attempts to convert as many values to constants as possible. This optimisation will also attempt to 'kill' basic blocks that are unreachable due to conditional branches on constant values.

5.5.6 CFG Simplification

Simplifies the code by merging basic blocks and control flow. It removes basic blocks with no predecessors(blocks that are never executed), merges each basic block with it's predecessor if it only has one predecessor. This optimisation also attempts to eliminate PHI nodes and eliminates blocks that only contain a single unconditional branch. The purpose of this optimisation is to make the LLVM IR as simple as possible before compilation, this is very useful in this project as

it allows the translation stage to be more abstract. One example of this is implementing bytecode instructions into their own basic block, and then using the CFG pass to combine them all into as few blocks as possible.

Chapter 6

Results & Evaluation

This chapter describes the experiments performed and discusses the results of the experiments. The results include compilation times, execution times, and resulting code sizes. These stats have been graphed to show comparisons in performance in each experiment.

6.1 Factorial

Two experiments were run using the Factorial benchmark. The first experiment uses the code shown in Figure 5.1. The benchmark is as follows: As the benchmark is a method within an Android application (invoked through a button widget), the application needs to be closed to reset the JIT cache. To ensure the JIT cache is reset, the whole application is terminated by calling *finish();* when the quit button is pressed. Each run of the benchmark is concluded by pressing the quit button and relaucnhing the application. The results for this experiment are all done in this fashion. Every run has an execution time, compile time, and code size for both types of JIT compiler as well as the time taken to execute the benchmark with JIT turned off.

Figure 6.2 compares the time taken to compile a trace. It compares the time taken to compile the trace(s) by both JIT compilers. The LLVM-based JIT takes significantly longer to compile the trace compared to the fast reacting Dalvik JIT. This is due to Dalvik being designed for very quick compilation, and the LLVM JIT attempting many optimisations on the code.

The execution time results shown in Figure 6.3 compare the execution times of the benchmark runs. Each run has the benchmark handled by the two different JIT compilers. In these results the benchmark being handled by the LLVM-based JIT has a longer execution time compared the the same benchmark being handled by the Dalvik JIT. But these times also include time spent

```
1       private void runFactorial() {
2           for(int i = 0; i < RUNS; i++) {
3               Factorial(i % FACTORIAL_MAX)
4           }
5       }
```

Figure 6.1: Factorial Benchmark

compiling the code. Figure 6.4 shows steady state execution times calculated by subtracting the compile time from the execution time in each run.

The average times from the experiment are shows in Figure 6.6. It shows the differences in compilation time between the two JIT compilers.This graph also shows the differences in execution times between the benchmarks for both JIT types and the average benchmark execution time when JIT is turned off.

Figure 6.7 shows a comparison of the average size of the native code produced by both JIT compilers.

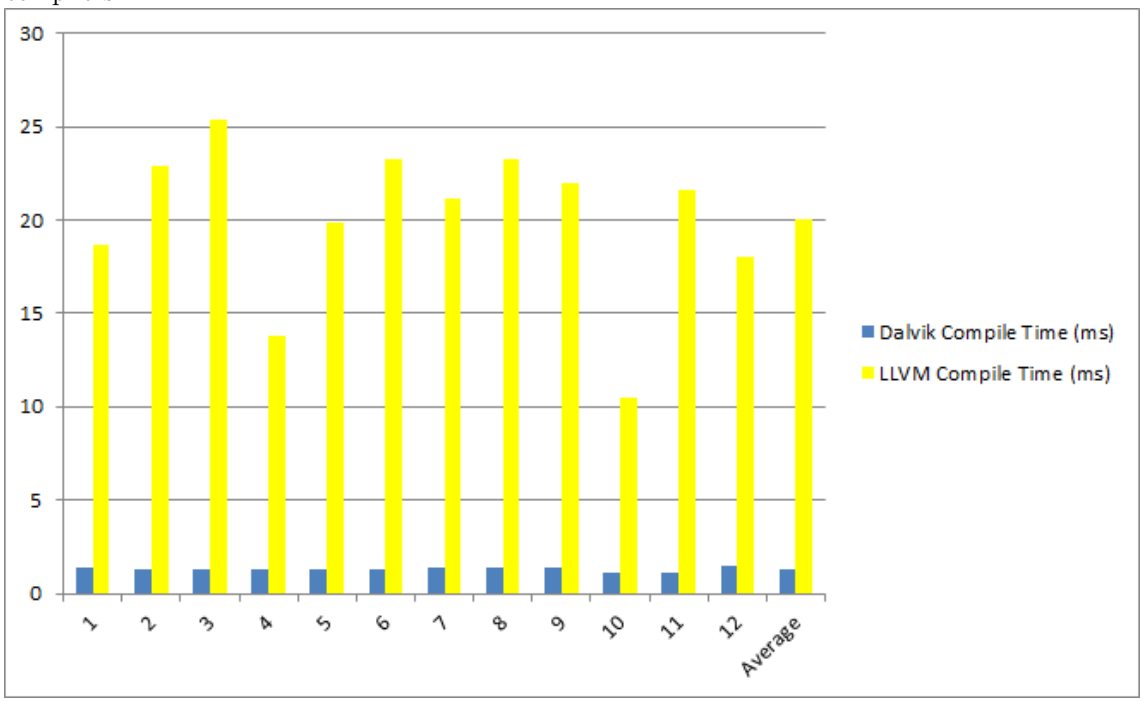

Figure 6.2: Factorial: Compile Times

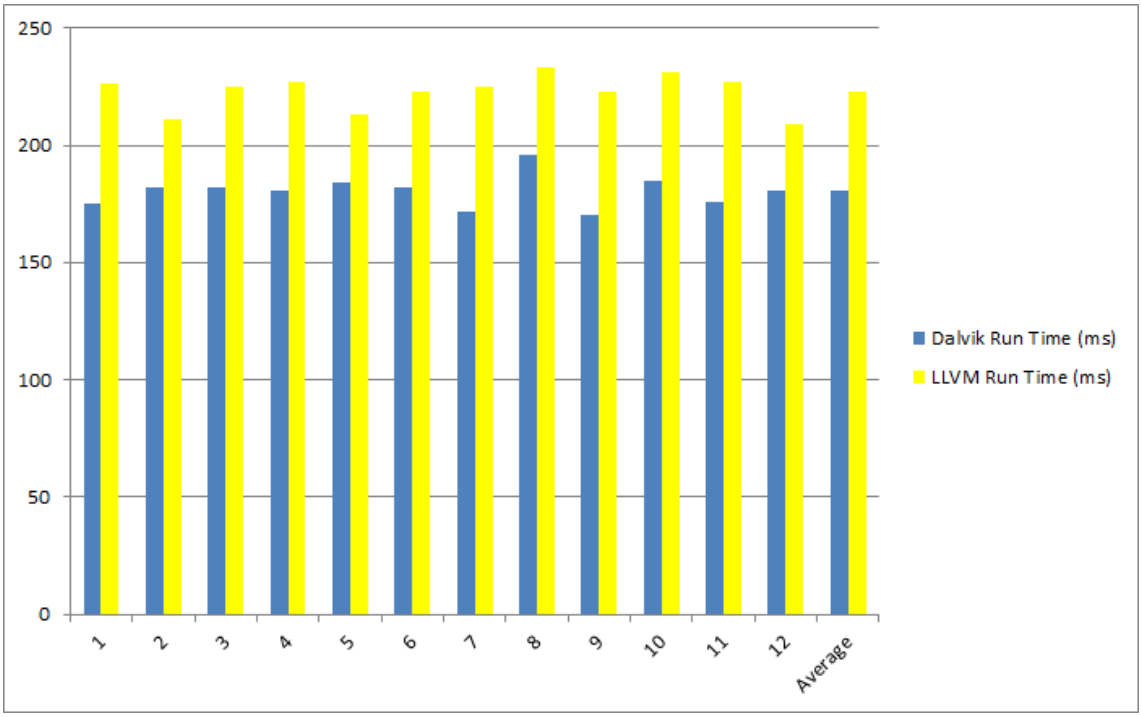

Figure 6.3: Factorial: Execution Times

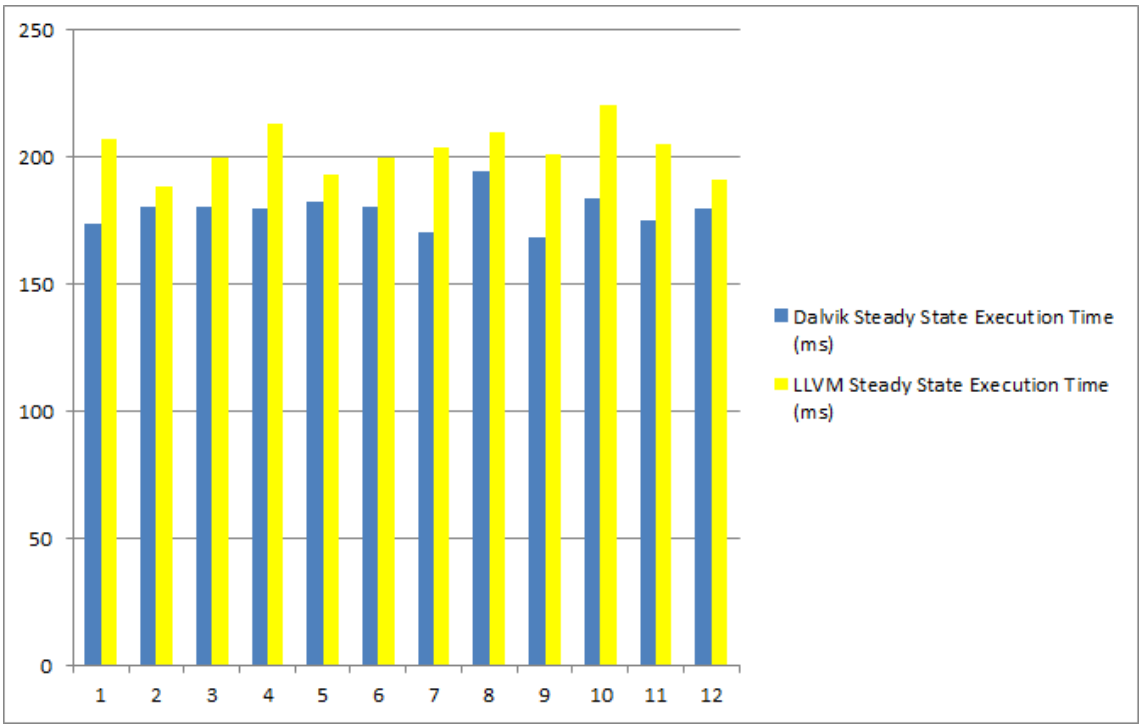

Figure 6.4: Factorial: Steady State Execution Times

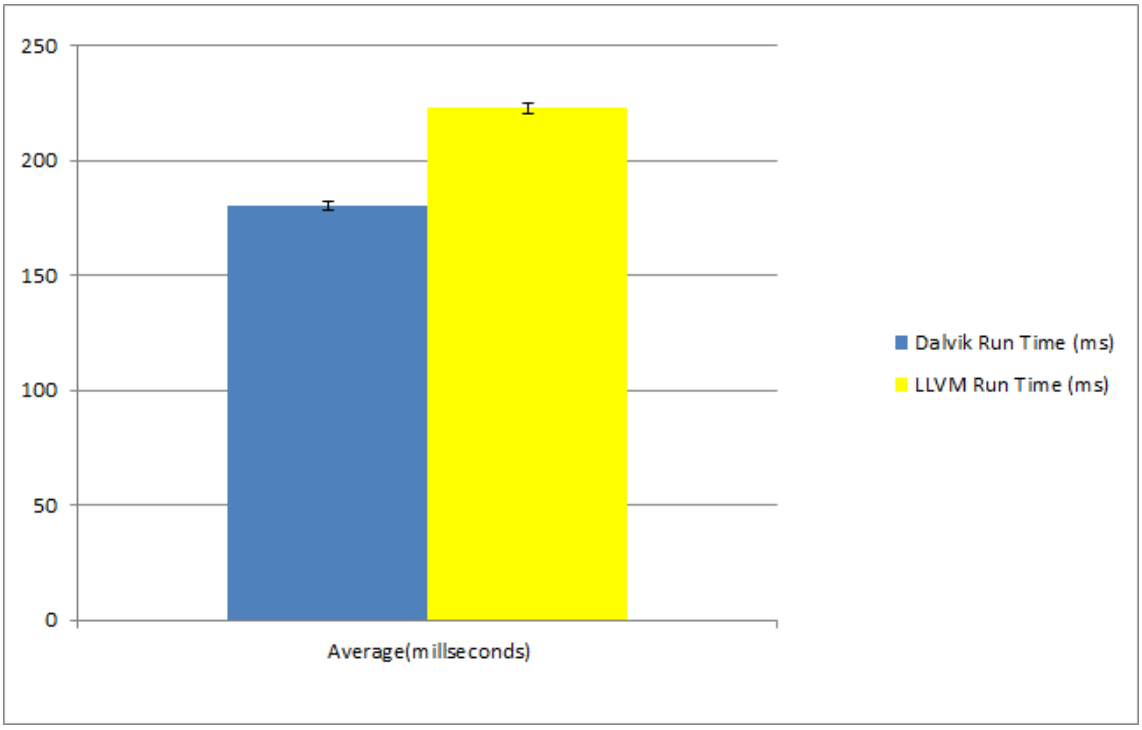

Figure 6.5: Factorial: Average Execution Time

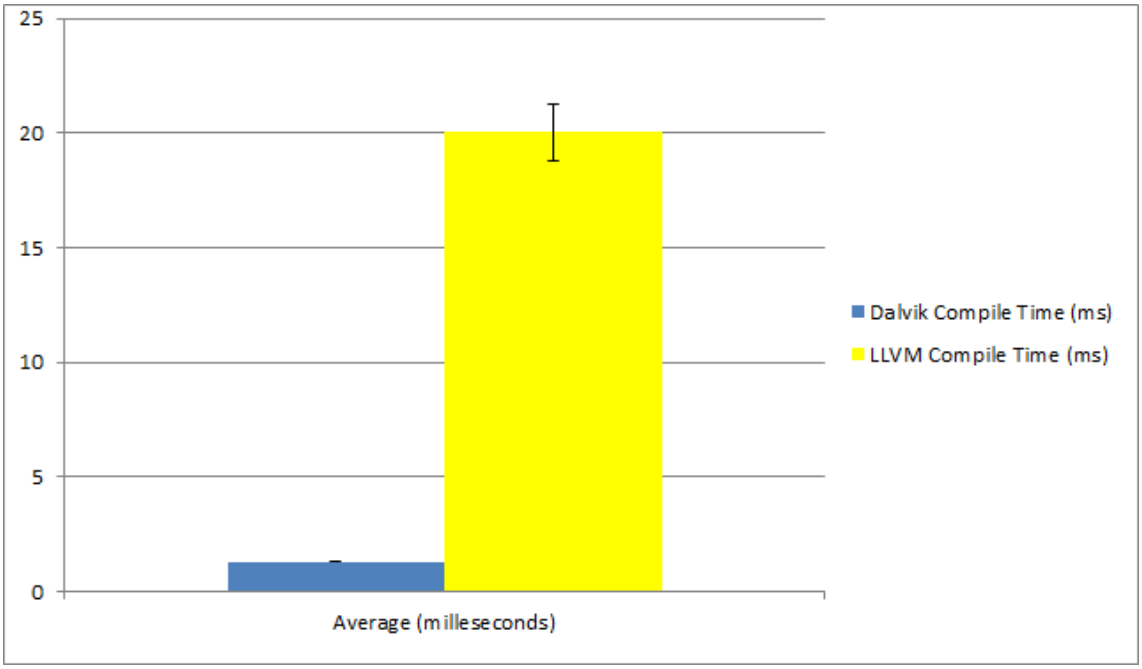

Figure 6.6: Factorial: Average Compilation Time

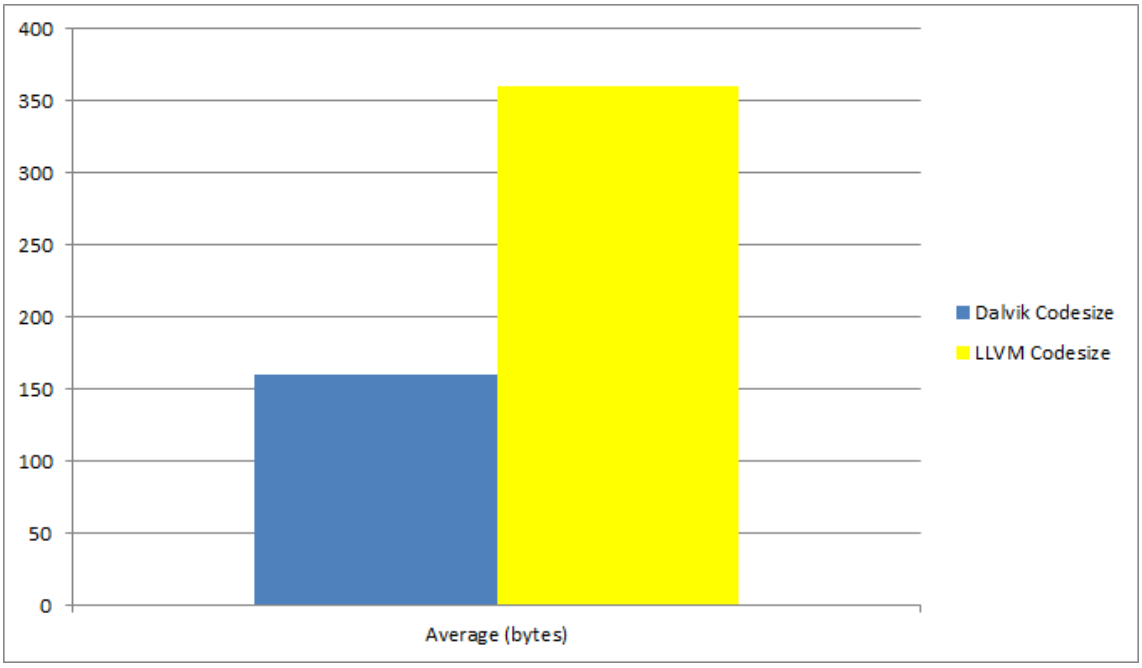

Figure 6.7: Factorial: code size

6.2 Factorial - Inline

The inlined Factorial benchmark was used in the second experiment. The source code for this benchmark is detailed in Figure 5.2. Because the method has been inlined in this benchmark, a trace can span the entire loop. This experiment was run in a similar fashion to the first Factorial experiment, with the application being terminated after each benchmark run. As with the first experiment, all of the results are from a 'cold' start with an empty JIT cache.

Figure 6.8 shows a comparison of compilation times. The Dalvik JIT compiler reacts quickly with this benchmark, with the LLVM JIT compiler taking much longer to compile the trace.

However, the execution time makes up for the long compilation time. The steady state execution times shown in Figure 6.12 show the native code produced by LLVM having a faster execution time in most cases.

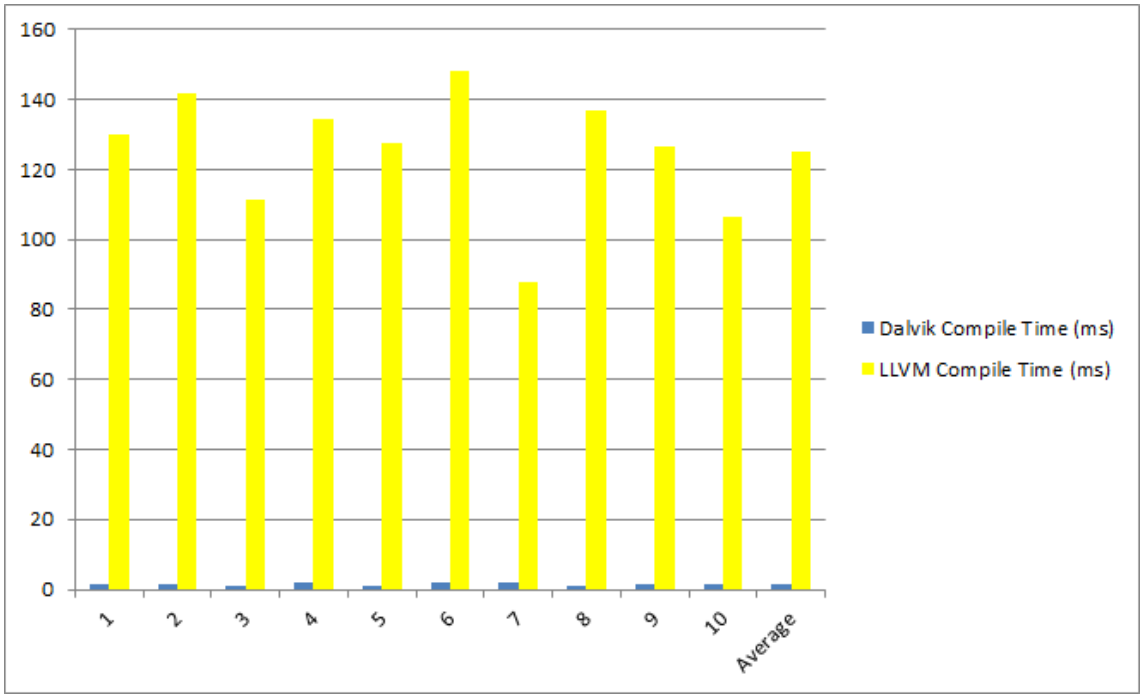

Figure 6.8: Factorial Inline: Compile Times

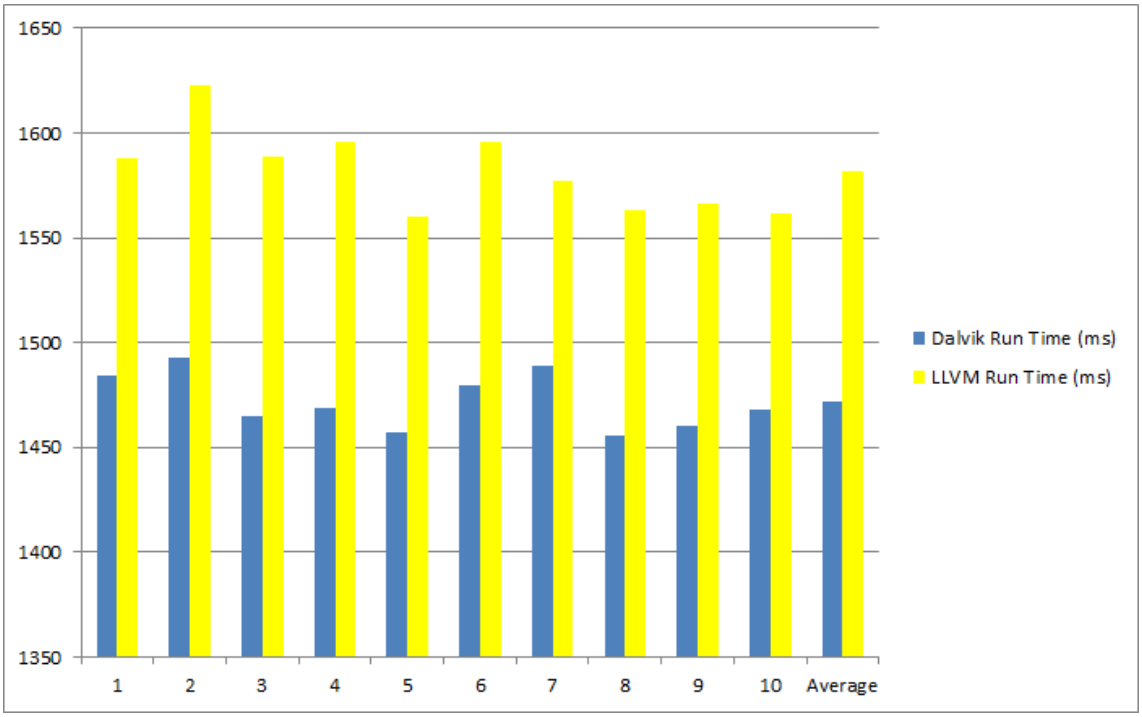

Figure 6.9: Factorial Inline: Execution Times

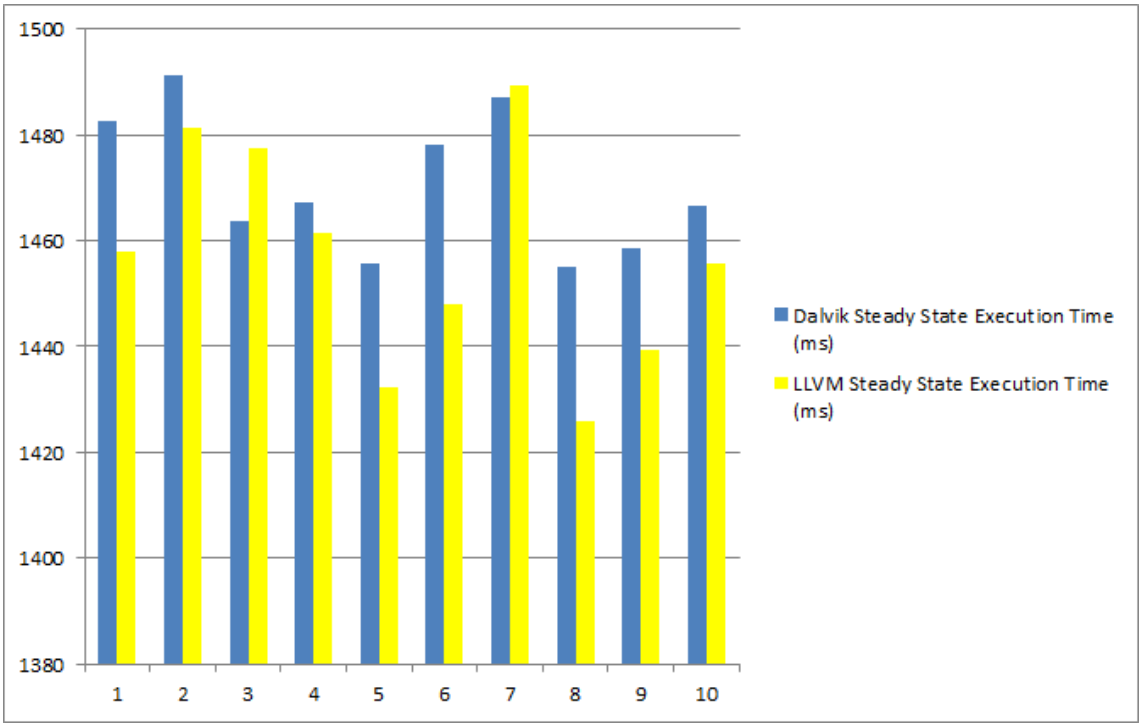

Figure 6.10: Factorial Inline: Steady State Execution Times

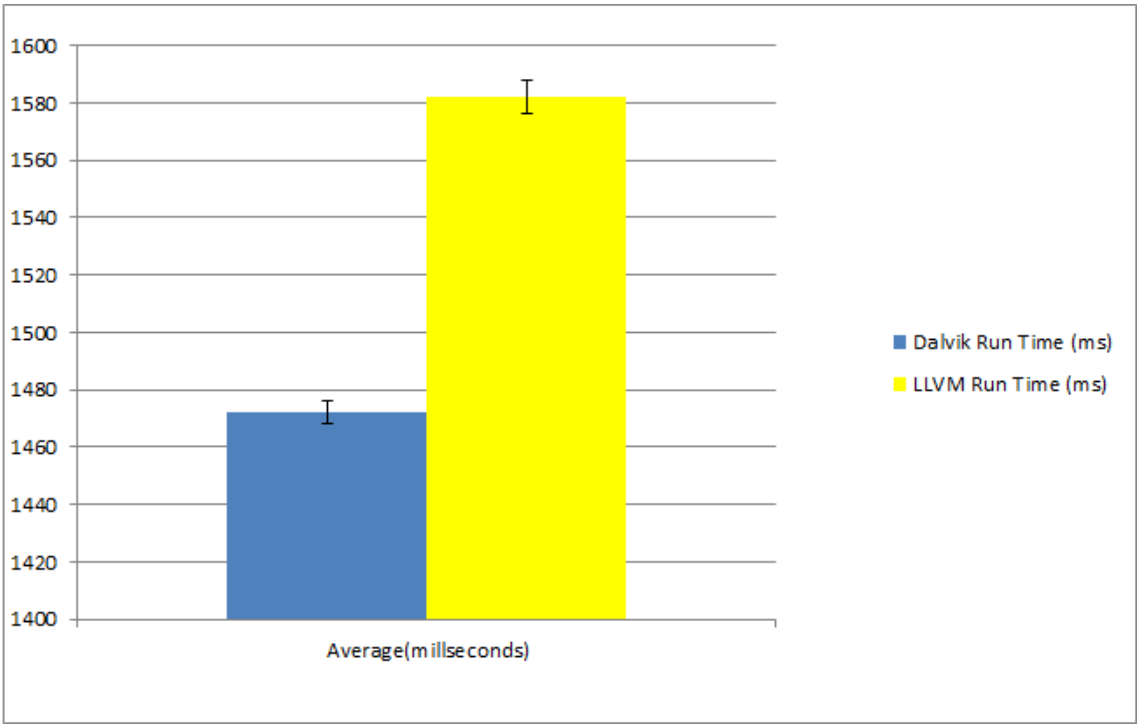

Figure 6.11: Factorial Inline: Average Execution Times

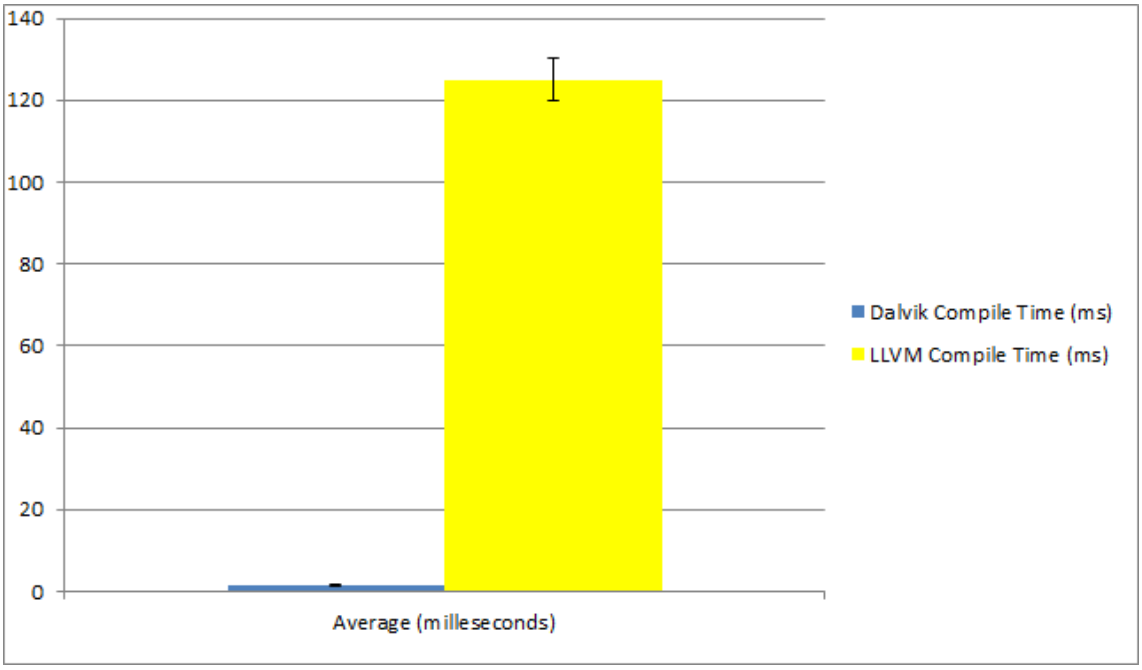

Figure 6.12: Factorial Inline: Average Compilation Times

6.3 Fibonacci

The Fibonacci benchmark was run as the third experiment. The source code for the benchmark is detailed in Figure 5.3. This experiment was run slightly differently, running the benchmark once to 'warm up' by allowing both JIT compilers to produce code and then running the benchmark multiple times to get a set of execution times all from their respective JIT compiled code.

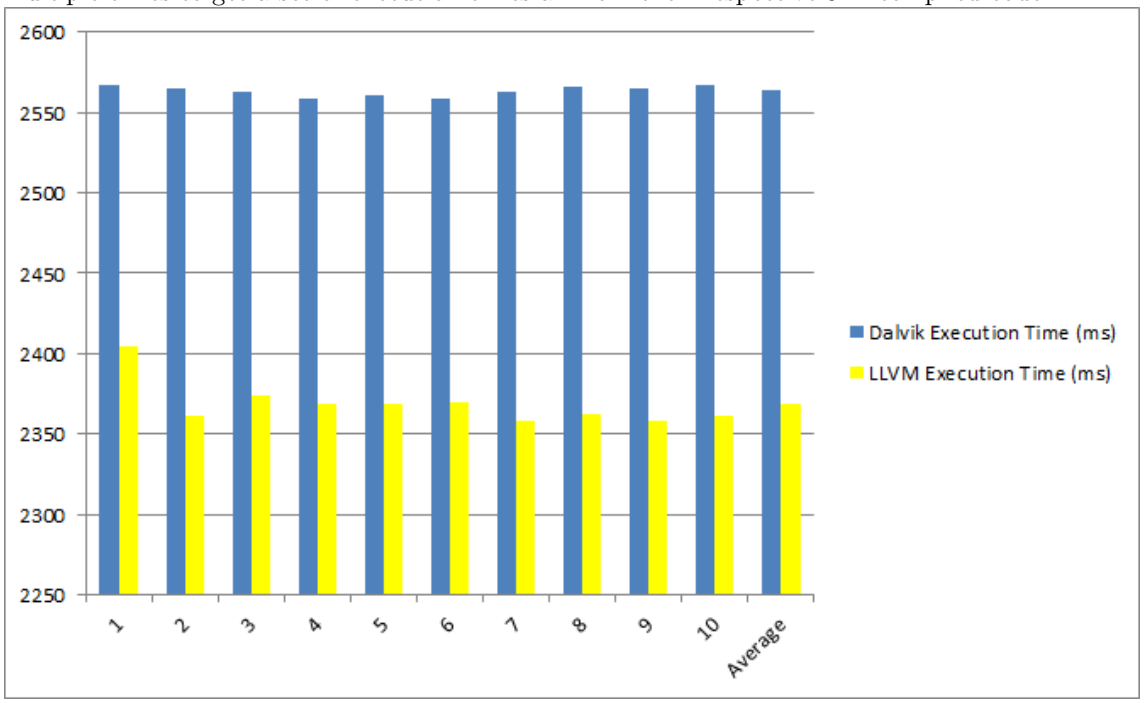

Figure 6.13: Fibonacci: Execution Times

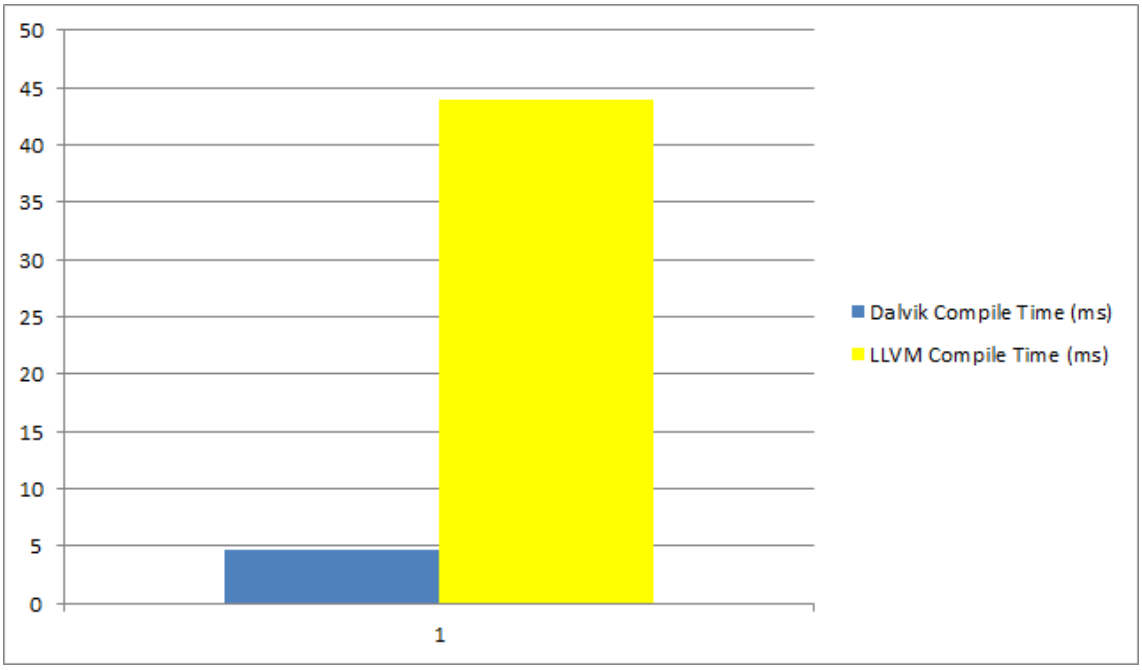

Figure 6.14: Fibonacci: Compile Times

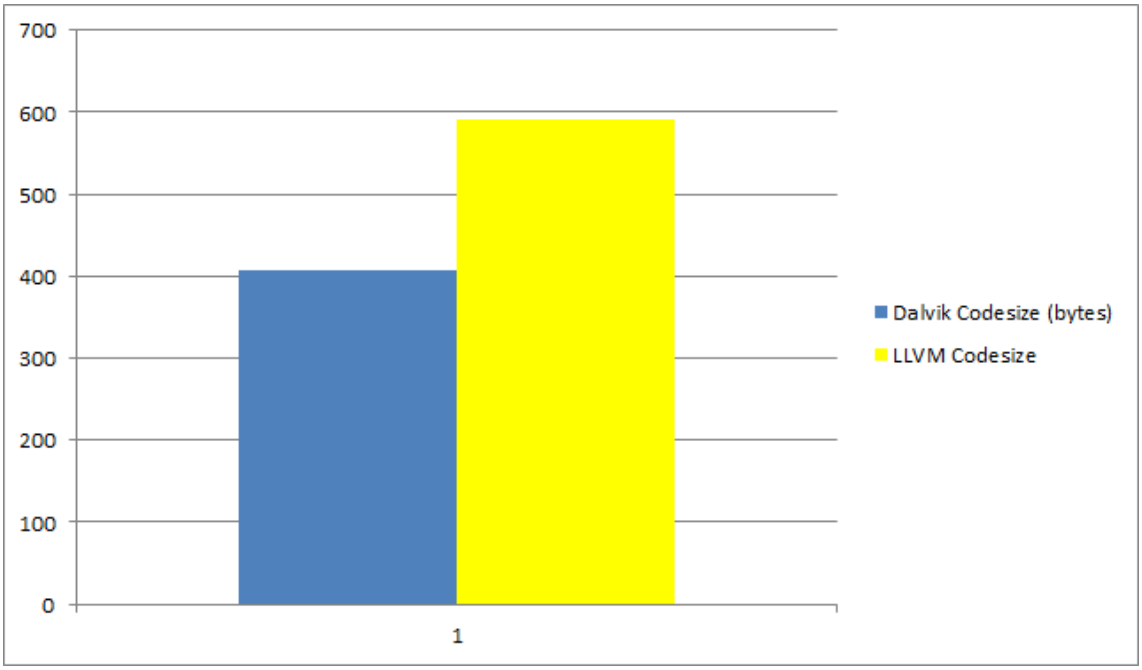

Figure 6.15: Fibonacci: code size

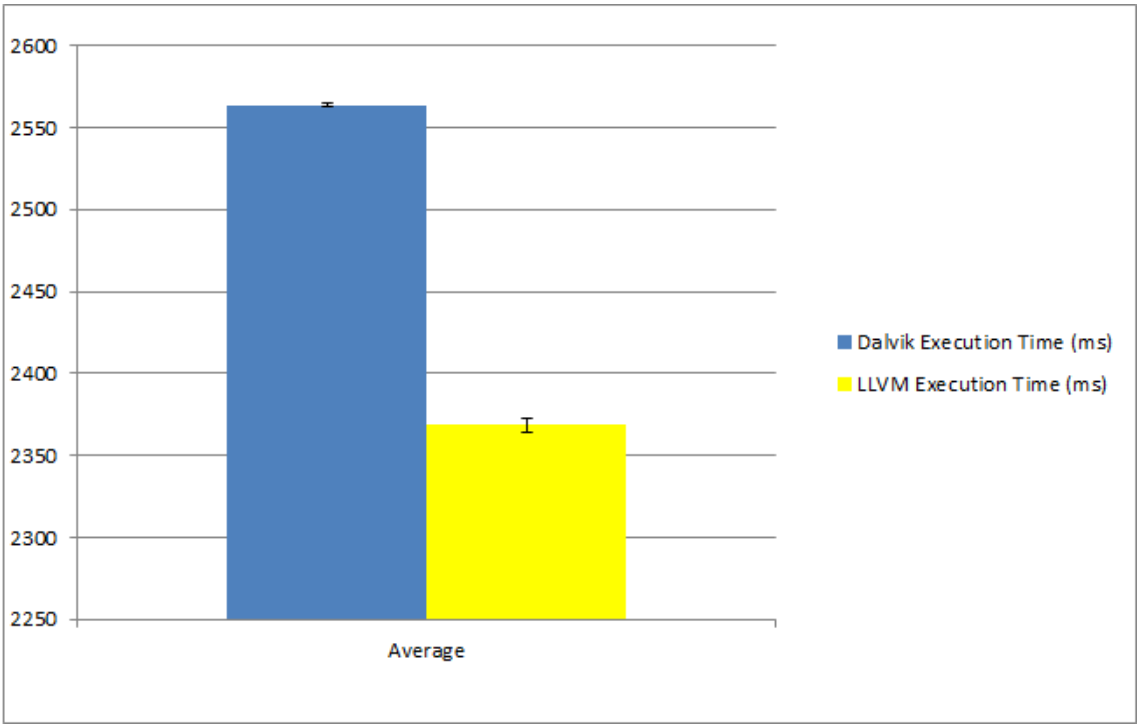

Figure 6.16: Fibonacci: Average Execution Time

Figure 6.13 shows the execution times for the experiment. The LLVM-based JIT appears to produce the fastest code, completing the benchmark approximately 200ms faster than the code produced by the Dalvik JIT. Figure 6.16 shows the average execution time with standard error.

The code sizes of this benchmark are shown in Figure 6.15. The total memory used by the LLVM-based JIT compiler is almost 1.5x the size of the code produced by the Dalvik JIT compiler.

The compile times shown in Figure 6.14 show Dalvik compiling very quickly, while the LLVM-based JIT compiler takes significantly longer to compile its trace. However the benefit of the benchmark running 200ms faster makes up for the extra 40ms spent compiling.

6.4 Summary

The experimental results show that the new LLVM just-in-time compiler can produce faster code, at the cost of increased compilation time. The execution time results of the Factorial-Inline (Figure 6.12) shows an improvement in runtime compared to the existing JIT compiler. The results of the Fibonacci experiment (Figure 6.13) shows an approximate 7% speedup when using the LLVM-based JIT compiler.

Chapter 7

Conclusion

This project has shown that it is possible to improve the performance of just-in-time compiled code within Android. By using LLVM's powerful optimisations a trace can be simplified before it is compiled to produce native code that executes faster than the native code produced by the stock just-in-time compiler. While the stock just-in-time compiler produces faster native code for simple traces, the new LLVM-based one can produce faster native code for more complex traces.

7.1 Critical Analysis

The just-in-time compiler implemented during this project can produce native code from a trace provided by Dalvik. As shown in the Factorial-Inline experiment, the produced code can execute faster than code produced by Dalviks JIT compiler, but the large compilation time means that Dalvik does a better job overall in this case. In the Fibonacci experiment however, the average execution time is around 200ms lower than the execution time of code compiled by the Dalvik JIT. The Dalvik JIT still reacts faster than the LLVM-based JIT, and produces better native code for shorter traces. But the LLVM-based JIT can produce better code in cases with longer and more complex traces. This is due to there being little scope for optimisation with shorter traces, whereas longer and more complex traces have a much greater scope for optimisations to be applied.

7.2 Future work

This section details further work that could be done to the project, to add to its functionality or to improve its performace. This section also describes issues with the current system and bugs that need to be fixed.

7.2.1 Bugs

There currently exist a few bugs within the LLVM-based JIT. One of the bugs is an issue with the implementation of the *GOTO* bytecode instruction. In order to accomodate the *GOTO* bytecode instruction, every other bytecode instruction was implemented into its own basic block. When a

GOTO instruction was encountered, a list of basic blocks is used to work out which block should be branched to. For example:

```
1 add-int v0, v1, v2
2 sub-int v3, v1, v2
3 goto -2
```

Figure 7.1: *GOTO* Example in DEX

In this code, the *goto* at the end indicates to move back two instructions, back to the *add-int*. In bitcode this would ideally translate to:

```
1 label ADD-INT0:
2     /* perform add */
3     br label SUB-INT0
4 label SUB-INT0:
5     /* perform sub */
6     br label GOTO0
7 label GOTO0:
8     br label ADD-INT0
```

Figure 7.2: *GOTO* Example in bitcode, the labels and branches between labels are automatically generated due to each bytecode instruciton being implemented in its own basic block

However, attempting this would cause LLVM to crash while the branch bitcode instruction was being appended to the GOTO block. After several attempts to fix this problem, the *GOTO* instruction implementation was changed to generate a chaining cell instead, which produced working native code at the cost of having a trace chain to itself rather than quickly branching to the correct place within the native trace.

Another bug that caused an unknown LLVM crash was one of the *CONST* bytecode instructions. This particular bytecode instruction encoded the destination register (in the virtual machine) in the first 4 bits of the second byte and the value to be stored in the lower 4 bits. The bitcode to extract these values worked as expected on (x86) linux, but would cause LLVM to crash when generated on the device. Again, this crash could occur during generation before the bitcode could be verified.

Finally, there is a bug in the implementation of the array access bytecode instructions. The bitcode generated and was verified, but a runtime crash would appear to occur randomly. Running a test trace with array access instructions would be accepted, compiled, and would run without issue but running the same test trace again would result in a crash the second time around.

7.2.2 Implementaion

Further work to the system itself could involve improvements to the way chaining cells are generated and used. Currently the function produced by LLVM returns an identification number, indicating

which chaining cell should be used. One way of improving the efficiency and reducing codesize would be to hardcode the address of the chaining cells direcly into the LLVM function, returning an address rather than an identifier.

Improvements could be made to the trace building part of Android, as traces are restricted to within a method. Allowing traces to span multiple methods would provide longer traces and more room to perform optimisations.

Other further work would include implementation of missing instructions, such as exception handling and arithmetiic with 64-bit values.

7.3 Summary

This project has produced an LLVM-based just-in-time compiler for the Dalvik virtual machine. It uses more aggressive optimisations than the stock just-in-time compiler in an effort to produce faster native code. The experimental results show a decrease in execution time for traces compiled by the new just-in-time compiler.

Bibliography

[1] Android 2.2 and developers goodies.

[2] Zipalign documentation.

[3] Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. Adaptive multi-level compilation in a trace-based java jit compiler. *SIGPLAN Not.*, 47(10):179–194, October 2012.

[4] John Koetsier. Android captured almost 70% global smartphone market share in 2012, apple just under 20%. *Venture Beat.*

[5] Guillermo A. Perez, Chung-Min Kao, Yeh-Ching Chung, and Wei-Chung Hsu. A hybrid just-in-time compiler for android: comparing jit types and the result of cooperation. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '12, pages 41–50, New York, NY, USA, 2012. ACM.

[6] 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. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, CASES '11, pages 15–24, New York, NY, USA, 2011. ACM.

[7] Yuan Zhang, Min Yang, Bo Zhou, Zhemin Yang, Weihua Zhang, and Binyu Zang. Swift: a register-based jit compiler for embedded jvms. *SIGPLAN Not.*, 47(7):63–74, March 2012.