JIT Code Generation for ARM CPU

Steven Henry
B.A.I. Engineering
Final Year Project  April 2014
Supervisor: Dr. Jeremy Jones

School of Computer Science and Statistics
O’Reilly Institute, Trinity College, Dublin 2, Ireland
Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

______________________________  April 1, 2014

Steven Henry
Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

April 1, 2014

Steven Henry
I would like to thank my supervisor Dr. Jeremy Jones for always being willing to answer my questions and for his support and encouragement throughout the project.
Abstract

The focus of this project is to investigate 'just-in-time' compiling and the effects of such systems on mobile devices. The implementation aim of the project is to dynamically generate native ARM machine code from the intermediary VCode created by Vivio interactive animations, and then execute this code to run an animation.
# Contents

Acknowledgements iii  
Abstract iv  

## 1 Introduction 1  
1.1 Motivation .............................................................. 1  
1.2 Project Aim .............................................................. 2  
1.3 Outline of Report ......................................................... 3  

## 2 State of the Art 4  
2.1 Online Learning .......................................................... 4  
2.2 History of JIT Compilation ............................................... 6  
2.3 JIT Compilation for Java .................................................. 11  
2.4 JIT Tracing ................................................................. 13  
2.5 Security of JIT Systems ................................................... 16  
2.6 JIT on Mobile Devices ..................................................... 17
3 Implementation

3.1 Vivio JIT Outline ................................................................. 19
3.2 Development Tools ........................................................... 20
3.3 Understanding the VCode .................................................... 21
3.4 ARM Register Set ............................................................... 24
3.5 Memory Management of Generated Code ............................... 26
3.6 Executing the Machine Code ................................................ 27
3.7 Constructing the Instructions ............................................... 28
3.8 Calling Functions from Machine Code .................................... 32

4 Conclusion

4.1 Conclusions ........................................................................... 37
4.2 Future Work ........................................................................... 39

A Appendix

A.1 Source Code ........................................................................... 40

Bibliography ................................................................................ 40
List of Figures

2.1 A number of online learning resources[38] .................................................. 5
2.2 Screenshots of Vivio binary tree and cache coherency protocol animations[29] . 6
2.3 The Java Virtual Machine (JVM) uses both interpreting and JIT compiling to execute programs ................................................................. 12
2.4 Depiction of a hot path within a loop[28] ......................................................... 14
2.5 Pros and cons of larger compilation scope (longer traces) compared to shorter . 15
2.6 A comparison of Samsung products and their processors[6] ............................. 17

3.1 Structure of Vivio JIT system ................................................................. 19
3.2 Qt logo ................................................................................................. 21
3.3 Sample VCode instructions ........................................................................ 22
3.4 State of accumulator and stack before and after SolidBrush function call .... 23
3.5 Parameter loading and calling Rectangle object VCode instructions .......... 23
3.6 State of accumulator and stack before and after Rectangle2 function call .... 24
3.7 Primary register set of ARM processor[1] .................................................. 25
3.8 Using the mmap function to allocate memory as executable .................... 27
3.9 Generating instructions to store and retrieve register values ................. 28
3.10 Outline from the ARM Reference Manual of machine code instructions available
on ARM[1] ................................................................. 29
3.11 Functions for generating machine code operations by bitshifting ............ 30
3.12 Flowchart showing progression of generating corresponding ARM machine code
for VCode instructions .................................................. 31
3.13 Inline assembly loading address of function into register ..................... 32
3.14 Conversion of testTwoVar address to int .................................. 33
3.15 Function to generate machine code instruction for branch to address stored in
register ........................................................................... 33
3.16 State of registers and stack before and after calling testTwoVar function .... 34
3.17 State of registers and stack before and after calling testSixVar function .... 34
3.18 State of registers and stack before and after calling testSixVar function using r0
as accumulator .................................................................. 35
3.19 State of registers and stack before and after calling rectangle2 static constructor 36
4.1 A green rectangle animation rendered using JIT code generation .......... 38
Chapter 1

Introduction

1.1 Motivation

As the capabilities and availability of mobile devices increases, so too does their potential as a learning resource. The spread and advances in mobile technologies such as smart phones and tablets has coincided with a renewed push in the research of 'just-in-time’ (JIT) compilation.

Students are one of the primary user groups of smart phones. While they are using their phones for everything from social networking to video games to getting real time directions, there is an opportunity for students to utilise their mobile devices for 'on-the-go' learning as well.

The Vivio animations[29] are a visual and interactive medium for students to learn and experiment with computer architecture concepts. The animations’ intuitive design helps students reinforce and revise materials from lectures and help complete and understand tutorials. Currently Vivio runs on Windows and Linux systems and is usually accessed by a web browser. Expanding Vivio to a mobile app enables students to access animations quickly and easily from any location. Students can make use of commute time on the bus or train by looking over material before or after a class.

JIT compilation is a technique for improving the speed of interpreted code systems. It is the process of generating corresponding native machine code for the processor at run time.
This native machine code can then be executed much faster than interpreting the intermediary bytecode as would be the case in an interpreting system.

JIT compilation can be seen as a technique which possesses the positive aspects of both static compilation and bytecode interpretation. As well as the generated native machine code executing substantially faster than interpreting intermediary code, JIT maintains the portable capabilities of an interpreted system. Unlike static compilation of native code which requires a different version of the code for each CPU it is to be run on, JIT systems remain portable and can then be optimised for specific processors.

With people expecting their mobile devices to keep up in comparison with the speed and techniques of home computers, there is a demand for JIT systems on mobile devices. JIT systems on mobile devices can help achieve faster program execution on the more limited mobile processor, however they must be tailored to avoid spending too long optimising and converting to machine code, affecting the user experience.

This compilation time versus execution speed comparison is especially important with interactive animations like Vivio. If the program gets caught optimising during an animation, it can cause unwanted pauses and a lagging effect. Balancing between optimisation time and execution speed leads to varying ways of determining which code should be converted and which interpreted.

This project piqued my interest as it gives me the opportunity to learn and understand the ‘under-the-bonnet’ workings of JIT systems as they become increasingly widespread.

1.2 Project Aim

The aim of the project is to apply Just-in-time compilation techniques to Dr. Jones’ Vivio animations. The target processor of the project is the ARM processor which is present on Android devices such as Samsung tablets and smart phones.

My primary objective is to implement the structure of a JIT system that can convert the VCode
instructions of a Vivio animation to ARM machine code at runtime.

I also wished to explore the developments and potential of JIT systems on mobile devices.

1.3 Outline of Report

Chapter 2 - State of the Art
This chapter looks into the current state of online learning and just-in-time systems. It also features a brief history of how JIT systems have developed up to this point.

Chapter 3 - Implementation
This chapter discusses the methods involved in implementing the JIT system for the Vivio animations, and problems encountered during the process and the techniques used in resolving them.

Chapter 4 - Conclusion
This chapter summarises the conclusions taken from the project and looks at potential future works expanding on it.
Chapter 2

State of the Art

2.1 Online Learning

As computers become more and more widely available, their use as a learning tool is increasing. The vast expanse of knowledge available on the internet is an invaluable tool to students at all different levels of education. Students now access the majority of their university notes online on college platforms such as Blackboard. These systems serve as a convenient and efficient way for lecturers and students to exchange lecture notes, assignment information and submit projects.

But as well as their own educational institution’s online resources, students are tapping into the online resources of other universities and other learning websites. Khan Academy[2] is non-profit educational website with over five thousand tutorial videos. About 10 million students a month watch micro lectures on a wide range of subjects, in particular, physics and mathematics. Khan Academy has made excellent use of video sharing site Youtube to expand its audience. As of March 2014, the Khan Academy Youtube channel has over 1.7 million subscribers.[3]

Students around the world make use of Khan Academy’s ”free, world-class education for anyone, anywhere”, and it isn’t the only such site. MIT OpenCourseWare[4] is an immense online resource, featuring material on almost every course taught at the prestigious Boston college.
Online learning is also helping students who could not physically attend college at all. In the two years since it was launched, massive open online course platform Coursera has a user base of an estimated six million worldwide.\cite{24} The platform enables users to enrol in one of over 400 modules from 85 universities in 16 countries completely remotely.

For many potential students, online course enrolment is the most convenient way for them to pursue a degree in higher education. There is a lot of appeal in the flexible, self-paced learning schedule. Students may be working to pay their way through college or battling a serious illness. To these students, the possibility of completing assignments and reviewing lecture notes at their own convenience is vital.

![Figure 2.1: A number of online learning resources\cite{38}](image)

Online learning is especially beneficial in part-time masters level courses. Many people choose to go back and complete masters programs after having been in the workforce for a few years. The majority of these people choose to do their masters course part time, so as not to remove themselves from the salary ladder for a year. These students then have hectic schedules for the two or three years that they are completing their masters while still working a full time job. As
well as the actual time spent both at their workplace and university, the travel time in between adds up very quickly, digging into the precious total hours in a week. Some of these students may be studying at a university far from their workplace, potentially in a different city or even country.

Online learning gives these students the opportunity to spend more of their time actually working on their course or job.

The Vivio animations\[29\] which are the target of this project are interactive animations which aid students in understanding computer architecture concepts within their course. The animations can play forwards and backwards and at greatly varying speeds. They work well as an visual and intuitive way to reinforce material covered in lectures.

![Figure 2.2: Screenshots of Vivio binary tree and cache coherency protocol animations][29]

### 2.2 History of JIT Compilation

JIT compilation defines any system in which code is dynamically converted at program runtime. In this sense, software systems have been using just-in-time compilation techniques from as early as the 1960s. Many would be surprised to learn that JIT has been around for so long. JIT compilers have only started to become widespread in the last few years, with the name popularised upon JITs inclusion within the Java Virtual Machine (JVM).

JIT is considered to be something that was somewhat reinvented. The earliest published
work on JIT compilation is considered to be McCarthy's LISP paper[31] in which he mentions compilation of functions into machine language. McCarthy theorised that this process was fast enough that the compilers output did not need to be saved. This can be seen as an inevitable result of having programs and data share the same notation[32].

Another early published reference to JIT compilation dates back to 1966. The University of Michigan Executive System for the IBM 7090 explicitly notes that the assembler and loader can be used to translate and load during execution. The manual for the system notes in its preface that most sections were written before August 1965, suggesting this instance of dynamic compiling dates back further.

In the late 1960s, Thompson[39] discussed compiling regular expressions into IBM 7094 code in an ad hoc fashion, code which was then executed to perform matching.

LC2[33] was primarily an experimental language designed for interactive programming. It might otherwise be consigned to the depths of history, if not for certain techniques used by Mitchell in its implementation[34]. These techniques later influenced JIT systems for Smalltalk and Self. Mitchell observed that by storing the actions performed during interpretation, compiled code could be derived from an interpreter at runtime. This would only work for code that had been executed, however, he gave the example of an if-statement. Since only one part is executed, code is generated for the unexecuted part as well. The interpreter is then reinvoked should the other code ever be executed.

Smalltalk source code is compiled into virtual machine code whenever a new method is added to a class[23]. The performance of initial Smalltalk implementations was less than satisfactory. To combat this, Deutsch and Schiffman[20] made key optimisations in software. The observation behind this was that so long as any conversion between representations happened automatically and transparently to the user, they could pick the most efficient representation for information. JIT conversion of virtual machine code to native code was one of the optimisation techniques they used. Procedures were compiled to native code and when execution of a procedure began, the native code would be cached for later use. Their system was linked to memory management in that native code would never be paged out, instead being discarded and regenerated later if
necessary.

Deutsch and Schiffman credited the idea to include dynamic conversion to Rau’s 1978 paper[36]. Rau was envisioning universal host machines which would have the ability to execute a number of high-level languages well. He proposed dynamic translation to microcode to the degree of single virtual machine instructions. The dynamic translation buffer, a hardware cache, would store completed translations. If a cache miss occurred it would signify that there was a translation missing, and the system would have to go back to a translation routine to convert the necessary high level code.

The Self programming language[41][37] is primarily a means of research. Self has an unusual design which makes it difficult to efficiently implement. This resulted in more ambitious and aggressive development of JIT compilation and optimisation than had be done up until that point. Similar to Smalltalk, Self is a pure object-oriented language in which everything is an object. However Self abstains from classes in favour of prototypes. In Self, every action is dynamic, even simple operations such as local variable access requires the use of a method. Self is also dynamically typed, this means that the types of identifiers are not known until execution and runtime.

The Self group noted three distinct generations of compiler[25] which will be detailed here. As in Deutsch and Schiffmans[20] Smalltalk system, the compiler was invoked dynamically whenever a method was invoked.

First Generation
Most of optimisation techniques employed by Self compilers dealt with type information with only a few techniques having a direct relationship with JIT compilation. However a major technique used in the first generation of the compiler was customisation[18][15][14]. Instead of dynamically compiling a method into native code for all circumstances of a methods execution, the compiler produced a version of the method that was customized to that particular context. There was significantly more type information available to the JIT compiler compared to static compilation. The generated code could be made notably more efficient by making use of this fact. The issue with this approach was that overcustomization could consume large amounts
of memory at runtime.

Second Generation

The second generation Self compiler extended upon the techniques used by its predecessor and computed much better type information for loops[16][14]. The second iteration of the compilers output could be executed faster than the original, however the compiler itself ran 15 to 35 times slower on benchmarks.[16][17] Many users steered clear of the new version of the compiler altogether.

Adjustments were made to the algorithms responsible for the slowed compiling. The most notable of these in JIT terms was called deferred compilation of uncommon cases. This is essentially the same technique used in Mitchell’s 1970 paper[34] but for different reasons.

The compiler is made aware that certain events are unlikely to occur in the program. This being the case, code does not have to be generated for these uncommon cases. Instead a stub is placed in the code which will invoke the compiler again if needed. This means that the code for some sections of the program does not need to be converted during the initial compilation, saving the compiler a significant amount of time. This vastly speeds up the overall code generation process by reducing the amount it is required to do.

Third Generation

The Self compiler was part of an interactive, graphical programming environment making it an interesting case to examine in relation to the Vivio animations. As such, Self dealt with issues with noticeable pausing due to compilation occurring on the fly. Holzle[25] argued that judging the performance of JIT compilation by the amount of time the compiler took to run was deceptive, and not representative of the users experience. Two invocations of the compiler with some program execution in between would be perceived by the user as one long pause. Holzle compensated by considering an implementation with temporally related groups of pauses rather than individual compilation pauses.

The third generation of the Self compiler dealt with the issue of slow compilation at a more fundamental level. Compilation time was reduced by using adaptive optimisation. Initial method
compilation was performed by a fast, non-optimizing compiler. Then frequency-of-invocation counters were kept for each method to determine when recompilation should occur.\[25\][26][27]

Holzle noted that the best candidate for recompilation may not be the method whose counter triggered the recompilation. Object-oriented programming style generally encourages short methods. This suggests that it may be more beneficial to optimize the caller of the method and incorporate the regularly called method inline.\[27\] Adaptive optimisation of code that is generated 'on the fly' has the complication that a modified method may already be executing and have information on the stack based on the version of the method from before it was optimised.\[25\]

The Self compilers JIT optimisation also made use of type feedback.\[25\][26] As a program was run, type information was gathered and would then be available if and when recompilation occurred. This enabled more aggressive optimisation.\[9\][7]

A general problem encountered with software distribution is the greatly varying computing environment in which software runs. Different computer architectures require programs to be in different binary executables. Even within a line of backwards compatible processors, a program statically compiled for the lowest common denominator of processor might not make full use of the processor it is actually being run on.

Franz tackled these obstacles using slim binaries.\[21\] A slim binary contains a machine-independent, high-level representation of a program module. When a module is loaded, executable code is generated for it just-in-time. This just-in-time generated code can then be tailored to the run-time environment.

It was claimed by Franz\[21\], and later Kistler\[30\], that generating code for an entire module at once was often superior to converting a single method at a time as in Smalltalk and Self. They claimed that entire module generation resulted in improved code performance.

Fast code generation was critical to the slim binary approach. In order for the slim binary approach to work, data structures had to be carefully arranged to facilitate fast code generation. Generated code could be reused later in execution if needed, rather than being regenerated as
Java is implemented using the Java Virtual Machine (JVM). Java source code is statically compiled into intermediary bytecode instructions. The JVM then interprets this bytecode for whatever processor the program is being run on. This approach is very portable, but considered at times painfully slow.

\begin{quotation}
Java isn’t just slow, it’s really slow, surprisingly slow. - Tyma\cite{40}
\end{quotation}

Java had to speed up, and the main avenue explored to do this was JIT compilation of the bytecode. The need for faster Java implementations spurred a surge in JIT research. It was Java that made the term just-in-time widespread.

Sun Microsystems [2001] began implementing a JVM which could convert bytecode into a register based representation of the program. Several papers at the end of the 20th century seemed to agree that simply converting the bytecode to native machine code was not satisfactory, and that code optimisation was necessary as well.\cite{12,13,42,11}

However it was also accepted that traditional optimisation methods were expensive. The goal became to find a happy middle ground between speed of conversion algorithms and the speed of the resulting code.

A strategy adopted by Agesen\cite{8} was to translate Java bytecode into Self code. This allowed optimisations already within the Self compiler to be utilised.

Azevedo\cite{10} used annotations to allow the bulk of code optimisation be done before runtime. The information required for effective JIT optimisation was calculated beforehand and added to the bytecode in the form of annotations. These annotations could then be used to assist the JIT system.
Chapter 2. State of the Art

The idea of continuous compilation for Java was proposed and evaluated by Plexbert and Cytron[35]. The aim was for a compiler and interpreter to execute concurrently, ideally each on a separate processor. This may have been an early indicator of current JVM implementations which utilise both interpreting and JIT code conversion.

Most current implementations of the Java Virtual Machine (JVM) utilise both interpreting and JIT compiling. Java source code is compiled by Java compiler into system independent bytecode. This bytecode can then be interpreted by JVM to run on most CPUs. However JVM also uses JIT compilation to converts bytecode to native, more efficient, machine code that can be executed by that specific CPU. This process is illustrated in Figure 2.3.

Figure 2.3: The Java Virtual Machine (JVM) uses both interpreting and JIT compiling to execute programs
Different JVM implementations use different approaches for determining which code should be converted to native code at runtime. An increasingly common method is to use JIT tracing techniques.

2.4 JIT Tracing

Tracing, in terms of just-in-time compilation, is a technique used for determining which source code to optimise and convert to machine code. The trace system for finding the most efficient code to convert is to find hot loops within the code at execution time. These hot loops are loops which are iterated through many times during execution of the code. Tracing works on the basis that most programs spend the majority of execution time within certain key loops. This is opposed to method-compiling JIT systems which compile regularly used methods within the code.

The first implantation of JIT tracing was Dynamo. Dynamo transparently improved the performance of a system at runtime by interpreting the code until a hot instruction was found. It would then generate, cache and execute an optimised version of this instruction.

A more recent implementation, HotpathVM was the first tracing JIT compiler for high level languages. The virtual machine is designed to dynamically identify regularly executed bytecode instructions and converting them to machine code. The aim of HotpathVM was to create an efficient Java Virtual Machine for mobile devices and their reduced resources.

TraceMonkey is a JIT compiler within Mozillas SpiderMonkey javascript engine which uses JIT tracing. TraceMonkey compiles frequently executed loops at runtime and optimises using dynamic types occurring on each path.

Tracing takes place in a number of stages, all occurring during runtime.

Profiling Phase

Trace system identifies hot loops by counting the number of consecutive iterations of a single
loop. Once the iteration counter goes above a certain threshold, the loop is deemed a hot loop and the tracing phase is entered.

**Tracing Phase**

As the entire tracing system takes place at runtime, the code within the hot loop continues to execute during the tracing phase. Each operation within the hot loop is recorded into a trace. The trace is usually stored in a form of intermediary representation.

Since traces follow a single execution path of the loop in question, later executions could diverge from the trace path, for example at 'if' statements. To deal with this, guard instructions are inserted. Guard instructions are a quick check to determine if the same conditions hold on the current execution as in the trace case. If the current execution differs from the trace, the guard will abort the trace and return to the source code.

Since tracing occurs during program execution, runtime information is available to it. This means that the type information can be stored in the trace. This type information can be used later for optimising the code.

![Figure 2.4: Depiction of a hot path within a loop][28]
Optimisation Phase

After the operations of the hot loop have been recorded into the trace, the code within the trace is optimised. This optimisation is generally simple to complete as the trace represents a single execution path so there is no control flow to deal with.

Generation Phase

After optimisation, equivalent native machine code is generated for the trace. This is relatively straightforward, again because the trace represents a single execution path.

Execution

Once the generation phase has been completed, the program can now execute the generated machine code on subsequent iterations of the loop until the guard fails. This generated machine code will executed far quicker.

Inoue [2012][28] proposed an adaptive multi-level compilation system as a practical way to balance the startup time and steady-state performance in a trace-JIT.

The system begins with small compilation scope and a low optimisation level, allowing the program to start running quickly. Then, using timer-based profiling, “really” hot paths are identified. Longer traces that capture these particularly hot paths are then traced and recompiled with a high optimisation level to improve the peak performance. The key aspect of this re-optimisation is selecting long traces that effectively capture the entire hot paths, leading to a higher steady state performance.

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Better steady-state performance</td>
<td>Hurts start-up performance</td>
</tr>
<tr>
<td>More optimisation opportunities for compilers</td>
<td>Longer compilation time</td>
</tr>
<tr>
<td>Smaller trace transitioning overhead</td>
<td>More duplicated code among traces</td>
</tr>
</tbody>
</table>

Figure 2.5: Pros and cons of larger compilation scope (longer traces) compared to shorter
2.5 Security of JIT Systems

The Write-XOR-Execute (WˆX) security feature is employed in most modern operating systems to avoid the danger of code injection attacks. WˆX is a memory protection policy which disallows code execution from writable memory, by never allowing a section of memory to be both writeable and executable simultaneously.

Unfortunately though, JIT compilation often clashes with the use of WˆX since it relies on writable memory to place the dynamically generated code. When a JIT compiler generates executable code, it needs to be stored in memory so that it can be executed later in the program. But since WˆX states that a page in memory can be either writable or executable, but not both, the executable code deliberately generated by the compiler will be not be able to run while WˆX is in effect. This means that programmers of JIT systems often code around the WˆX countermeasure, leaving their programs open to code injection attacks.

This behaviour gives an attacker the chance to have the JIT compiler insert malicious code in the executable memory. Then, when the JIT generated machine code is being executed, the attackers code will be as well. This process is known as JIT spraying. A number of techniques have emerged to deal with this issue.

JITSec\cite{19} is a countermeasure designed specifically for programs that use JIT compilation techniques. JITSec decouples sensitive code from non-sensitive code and blocks the execution of sensitive from writeable memory pages. The distinction in based on the separation of system calls from normal function calls within the address space of the process.

This countermeasure gives no protection against the process of code injection itself, instead protecting against execution of system calls originating from the injected code. JITSec can work in conjunction with other countermeasures. The system works on a kernel level so all applications will be protected immediately upon deployment and is completely transparent to the operating system. The process causes a slight overhead, however this is under 2% and generally negligible. The countermeasure serves its purpose of adding an extra line of defence in cases where code execution from writeable pages is required and so implementing WˆX is
troublesome.

### 2.6 JIT on Mobile Devices

The approach to JIT compilation system on resource constrained mobile devices differs slightly. Given the widespread use of mobile devices such as tablets and smartphones, there is an expectancy that mobile devices make use of JIT systems as they develop. JIT systems on mobile devices can achieve faster program execution on the more limited mobile resources, however the system must be tailored for the constrained resources available. As much as mobile processors have advanced in recent years, so too have all processors. To expect the same systems to function similarly on such varying CPUs is unreasonable.

Figure 2.6 shows the processors on current Samsung mobile and laptop devices. Even on the very light ATIV book, the Intel i5 processor has more computing power available than the ARM processors on the similarly recent Galaxy S4 phone and Galaxy 3 tablet.

![Samsung Galaxy S4 Phone](image1) ![Samsung Galaxy Tablet 3](image2) ![Samsung ATIV Book 9](image3)

**Samsung Galaxy S4 Phone**
1.2 GHz quad-core ARM Cortex-A7

**Samsung Galaxy Tablet 3**
1.5 GHz dual-core Samsung Exynos 4212 (ARM based)

**Samsung ATIV Book 9**
Intel® i5 Processor 4200U (up to 2.6GHz)

Figure 2.6: A comparison of Samsung products and their processors[6]

The main difference in approach to JIT systems on mobile devices is avoiding spending too long optimising and converting to machine code. Similarly long traces to those that might be used in a program on a laptop or desktop could cause pauses that are noticeable enough to affect the user experience on the less powerful phone.
This potential generation time delay is particularly crucial with visual animations like Vivio. If the program has to stop and optimise during an animation, the resulting lagging effect can be greatly detrimental to the user experience. Balancing between optimisation time and the execution speed of the code is a major research point in terms of JIT tracing and method based JIT compilers.

HotpathVM[22] claims to be a Java VM small enough to function correctly on resource-constrained mobile devices. The system identifies traces of frequently executed bytecode instructions, potentially across several methods, and then compiles them via Static Single Assignment (SSA) construction.

The novel use of SSA form in this way enables the system to hoist instructions across trace side-exits without the need for expensive compensation code in off-trace paths. The overall code and data memory consumption of the system is only 150 kBytes. The light-weight system performs comparatively very positively in benchmarks against heavy-weight JIT compilers. In some cases the speed-up caused by the JIT compilation even matches the heavy-weight systems.
Chapter 3

Implementation

3.1 Vivio JIT Outline

The proposed structure of the Vivio App containing the JIT Compiler can be seen in Figure 3.1. The Vivio source code is written in the Vivio 6.0 development environment. The Vivio code can be compiled by the Vivio compiler within the Vivio IDE. Here the source code is compiled into intermediary language Vcode. VCode files containing the VCode instructions for a Vivio animation can be uploaded online. The Vivio app running on an Android phone or tablet can then download this VCode file when it starts up.
In current implementations of Vivio, this intermediary VCode is interpreted directly for whichever processor it is running on, similarly to how the JVM interprets intermediary bytecode to run Java applications on different processors.

The aim for this project is to apply just-in-time compiling techniques and generate ARM machine code for each Vcode instruction in the program and then execute this machine code correctly. This JIT compiler stage occurs within the Vivio app.

The app can access the VCode and take each VCode instruction and convert it into corresponding native ARM machine code. The ARM CPU in the tablet can process this native code much faster than interpreting the VCode directly. This JIT Compiler is the primary section I implemented in the system for the project.

3.2 Development Tools

Qt 5.2.1 for Android was used to develop the JIT compiler for Android. Qt is a cross-platform application framework, often used for applications with a graphical user interface. Qt primarily uses standard C++.

To compile the program for Android, I also needed to include:

- The Java Development Kit (JDK)
- Android Software Development Kit (SDK)
- Android Native Development Kit (NDK)
- Apache Ant Java library and command line tool

Environment path variables had to be set for Java and the Android SDK.

I also had to use the Android SDK to download the appropriate API for the Android version the VivioApp would run on. The app should run on anything from Android 2.3.3 (API 10) onwards.
3.3 Understanding the VCode

The development environment used for the project was Qt Creator 2.8, the IDE developed for use with the Qt cross-platform framework.

![Qt logo](image)

Figure 3.2: Qt logo

The development environment used for the project was Qt Creator 2.8, the IDE developed for use with the Qt cross-platform framework.

3.3 Understanding the VCode

In order to implement a JIT compiler which could generate corresponding ARM machine code for Vcode instructions, it was necessary to understand how the Vcode operated.

The Vcode uses an accumulator. All the values loaded in VCode are loaded into the accumulator. In order to push a value onto the stack, it is loaded into the accumulator and then pushed to the stack. When a variable is popped off the stack, it is popped into the accumulator.

Vcode instructions sizes are in 16 bit multiples. Each operation is 16 bits in size, and then it may be followed by a number of 16 bit values containing values relevant to the instruction. For example, in Figure 3.3 below, the LD instruction is made up of three 16 bit values. The first is 04ff. 04ff designates that the instruction is to load a 32 bit value into the accumulator. The following two 16 bit values combine to make this 32 bit value which is to be loaded. Also viewable with the VCode is a human readable representation of each of the instructions to the
right of the figure. This representation can be seen in the .dec files of the Vivio animation projects.

```c

greenbrush = SolidBrush(rgb(0, 255, 0));
!! 00000000 04ff ff00 ff00  LD  0xff00ff00
!! 00000003 04fc  SOLIDBRUSH
!! 00000004 0115  ST  G[1]
```

Figure 3.3: Sample VCode instructions

Since a LD operation, which is loading a 32 bit value, will always be followed by two more 16 bit values which are used executing the instruction, the program will move on 3 spaces in memory to find the next instruction.

The next instruction is a function call. Vivio functions can be called from the Vcode and these will then be interpreted directly for the processor.

In this case the function being called is a SolidBrush function. The SolidBrush function is used to create a brush object which is then used to fill in shapes for the animation. The only parameter of the SolidBrush is a 32 bit value denoting the colour of the brush.

When a function is called by the Vcode, it will take in the value in the accumulator as the first parameter of the function. In the SolidBrush example, the only parameter is the colour, which has been loaded into the accumulator as shown. This means that when the function is called, it will take the value 0xff00ff00 and use it as the parameter of the function. The result of the function is then returned in the accumulator. In the case of the SolidBrush, the result is the address in memory of the SolidBrush object created, as shown in Figure 3.4.

The VCode instruction at memory location 4 is an instruction to store a value in a global variable. The VCode maintains a number of global variables, which one the value is stored in is denoted in the 2nd most significant byte of the instructions 16 bit value, in this case 1. The value that will be stored is the value in the accumulator, in this case the address of the greenBrush object. This is so that the greenBrush can be called later to fill in shapes in the animation.
3.3. Understanding the VCode

Figure 3.4: State of accumulator and stack before and after SolidBrush function call

The greenBrush is used in the example shown in Figure 3.4. Figure 3.5 shows VCode which calls the Rectangle2 function. The Rectangle2 function has significantly more variables than the SolidBrush function above. The VCode instructions leading up to Rectangle2 call are to load the parameters for the function.

\[ r = \text{Rectangle2}(0, 2, 0, \text{greenbrush}, 0, 0, 400, 300, 0); \]

```
!! 00000005 03bc
!! 00000006 00f8 4396
!! 00000008 00f8 43c8
!! 0000000a 02bf
!! 0000000b 0109
!! 0000000c 00c0 0002
!! 0000000e 0009
!! 0000000f 90fc
!! 00000010 0215
```

LDZx3 0, 0, 0
PLRH3 0x43960000
PLRH3 0x43c80000
PLZx2 0, 0
PL G[1]
PL1x3 0, 2, 0
PL G[0]
RECTANGLE2
ST G[2]

Figure 3.5: Parameter loading and calling Rectangle object VCode instructions

In the case of functions with more than one parameter, the first parameter is taken from the accumulator, and the remaining ones from the stack in descending order from the top. This means that the values must be loaded into the accumulator one at a time and then pushed to the stack in reverse order to how they appear in the Vivio function call.

The instructions seen in Figure reffig:vcode3 leading up to Rectangle2 are loading and pushing the parameter values. LDZx3 loads and pushes two zeroes and then loads one to the accumulator. PLRH3 pushes the accumulator to the stack and then loads a 16 bit value to the top half of bits in the accumulator. PL loads the global variable 1 into the accumulator (again 1
designated by second most significant bit), PL1x3 pushes and loads three immediate values.

By the time the Rectangle2 instruction is reached, the parameters should be aligned correctly in order in the accumulator and on the stack as shown in Figure 3.6. The Rectangle2 function can then be called and correctly take in the nine parameters required to create and initialise a rectangle object. The address of the newly created rectangle is then returned in the accumulator. The next instruction stores this address in a global variable so that the created rectangle can be drawn at render time.

### 3.4 ARM Register Set

Writing a JIT compiler to generate correct machine code instructions would require a working knowledge of the ARM register set. Figure 3.7 shows the primary register set of the ARM processor.
3.4. ARM Register Set

<table>
<thead>
<tr>
<th>Register</th>
<th>Synonym</th>
<th>Special</th>
<th>Role in the procedure call standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>r15</td>
<td>PC</td>
<td></td>
<td>The Program Counter.</td>
</tr>
<tr>
<td>r14</td>
<td>LR</td>
<td></td>
<td>The Link Register.</td>
</tr>
<tr>
<td>r13</td>
<td>SP</td>
<td></td>
<td>The Stack Pointer.</td>
</tr>
<tr>
<td>r12</td>
<td>IP</td>
<td></td>
<td>The Intra-Procedure-call scratch register.</td>
</tr>
<tr>
<td>r11</td>
<td>v8</td>
<td>FP</td>
<td>ARM-state variable-register 8. ARM-state frame pointer.</td>
</tr>
<tr>
<td>r10</td>
<td>v7</td>
<td>SL</td>
<td>ARM-state variable-register 7. Stack Limit pointer in stack-checked variants.</td>
</tr>
<tr>
<td>r9</td>
<td>v6</td>
<td>SB</td>
<td>ARM-state v-register 6. Static Base in PID/re-entrant/shared-library variants</td>
</tr>
<tr>
<td>r8</td>
<td>v5</td>
<td></td>
<td>ARM-state variable-register 5.</td>
</tr>
<tr>
<td>r7</td>
<td>v4</td>
<td>WR</td>
<td>Variable register (v-register) 4. Thumb-state Work Register.</td>
</tr>
<tr>
<td>r6</td>
<td>v3</td>
<td></td>
<td>Variable register (v-register) 3.</td>
</tr>
<tr>
<td>r5</td>
<td>v2</td>
<td></td>
<td>Variable register (v-register) 2.</td>
</tr>
<tr>
<td>r4</td>
<td>v1</td>
<td></td>
<td>Variable register (v-register) 1.</td>
</tr>
<tr>
<td>r3</td>
<td>a4</td>
<td></td>
<td>Argument/result/scratch register 4.</td>
</tr>
<tr>
<td>r2</td>
<td>a3</td>
<td></td>
<td>Argument/result/scratch register 3.</td>
</tr>
<tr>
<td>r1</td>
<td>a2</td>
<td></td>
<td>Argument/result/scratch register 2.</td>
</tr>
<tr>
<td>r0</td>
<td>a1</td>
<td></td>
<td>Argument/result/scratch register 1.</td>
</tr>
</tbody>
</table>

Figure 3.7: Primary register set of ARM processor[1]

Significant registers for the project:

- r0, r1, r2 and r3, the first four registers, are used as function parameters. The first four parameters of a called function (should it have that many) are taken from these registers in order. This meant that the machine code would have to manipulate the parameter variables of a function into these registers before the function was called. The fifth and onwards parameters are taken off the stack.

- The returned results of functions are returned in r0. This made it the primary candidate register for simulating the accumulator employed by Vivio. For this reason, and because it is the first parameter register for functions, I used r0 to simulate the accumulator.

- r15 is the ARM program counter (PC). The PC contains the address of the next instruction to be executed.
• r14 is the ARM link register (LR). This is used for storing the program address before entering a function. The stored address could then be popped into the program counter to return at the end of a function. The LR would have to be stored when nested functions were needed.

• r13 is the ARM stack pointer (SP) so I used it when accessing variables and objects on the stack.

• r8 is the fifth variable register. It is not used as a parameter for functions nor in C++ functions, so I made use of r8 for storing addresses of functions that the program is to branch to.

Each of the registers in the ARM contains 32 bits.

3.5 Memory Management of Generated Code

Before the generating and executing of the machine code instructions, there needed to be somewhere to store them. On the target ARM processor all the instructions are 32 bits in length. This meant I was able to store the instructions as an array of integers.

I used a counter variable to keep track of how many instructions had been generated. This allowed me to step through the VCode, generating appropriate ARM machine code instructions for each VCode instruction and stored in the array.

After the code has all been generated, the next step is branching to the start of the array in memory, and executing each machine code instruction by progressing through the array. However, in order to do this, the memory the array is stored in would have to be executable.

The function initially found to do this on the Linux based Android system was mmap. The mmap function allowed for a number of consecutive bytes of memory to be allocated for the array and the executable protection removed. The function being used can be seen in Figure 3.8.
3.6 Executing the Machine Code

Before executing the machine code, I had to make sure that the executed machine code would not interfere with the rest of the program. This worked much like writing a well behaved subroutine in ARM assembly language.

The first instructions generated were at the start and end of the array. Before execution of the rest of the array, the values in the four utilised registers, r0-r3, were stored on the stack. They could then be popped off at the end of the machine code, and restored to their registers as they were beforehand.

To return from the memory location of the machine code, the address to branch back to had to be stored in the link register (LR). This occurred at the beginning of the array. Then, at

```c
//allocating memory for array to execute
unsigned int codeBytes = 4096;
void * virtualCodeAddress = mmap(
    NULL,
    codeBytes,
    PROT_READ | PROT_WRITE | PROT_EXEC,
    MAP_ANONYMOUS | MAP_PRIVATE,
    0,
    0);

int * machineCode = (int*) (virtualCodeAddress);
int machCodeNum = 0;
```

Figure 3.8: Using the mmap function to allocate memory as executable

This memory allocation and protection settings functioned as hoped. I was able to make the program branch to the beginning of the array in memory and execute the contained integers as machine code instructions. More detail on this in the next section.

However, research on the security aspects of JIT compilers reveals that it is generally bad practice to have memory be both writeable and executable simultaneously. This is part of the W^X (write xor executable) memory protection policy, which is a security feature employed to prevent code injection attacks. A future extension to this project would include security features to combat this, potentially utilising the mprotect function.
the end of the array, after the machine instructions for the animation have been executed, this value from LR is popped into the program counter (PC). This makes the program return from inside the array of generated code. This approach of storing the LR is particularly necessary in cases where the machine code calls functions as this will lead to nested link values. All of these link addresses must be stored correctly so that they may be returned to.

These two necessary procedures at the beginning and end of the array can be done together in single instructions, since they both involve pushing and popping a number of registers. The machine code instructions being generated can be seen in Figure 3.9. The operations used are STMFD (store multiple full descending) and LDMFD (load multiple full descending). The second of these instructions is generated just before the array is executed.

### 3.7 Constructing the Instructions

Each arm instruction is 32 bits long. There is a general overall structure of how each is formed, though it varies for certain operations.

The condition byte at the start of the instruction declares if the instruction should only be executed if certain conditions are met. ARM has four condition flags which are tracked in the Current Program Status Register (CPSR). They are overflow, carry, negative and zero.

The next two bytes together determine the operation itself, and what inputs it has. For example, bit 25 being set for data processing determines whether the sources are registers or immediates, as seen in Figure 3.10.
### 3.7. Constructing the Instructions

![Table of ARM Machine Code Instructions](image)

**Figure 3.10:** Outline from the ARM Reference Manual of machine code instructions available on ARM[1]
Chapter 3. Implementation

The destination register (Rd) and two sources (Rm, Rn, or immediate) mostly remain in the same place. To generate instructions to perform specific instructions that corresponded to the VCode of the animation, I wrote functions which used bit shifting to line up the source and destinations of each instruction. Examples of some of the arithmetic functions can be seen in Figure 3.11.

```c
int movRegister(int dst, int src)
{
    int machCodeSlot = 0;
    machCodeSlot = machCodeSlot | 0xElA00000;
    dst = dst << 12;
    machCodeSlot = machCodeSlot | dst;
    machCodeSlot = machCodeSlot | src;
    return machCodeSlot;
}

int addRegisters(int dstReg, int in1Reg, int in2Reg)
{
    int machCodeSlot = 0;
    machCodeSlot = machCodeSlot | 0xE0800000;  //add operation
    dstReg = dstReg << 12;
    machCodeSlot = machCodeSlot | dstReg;
    in1Reg = in1Reg << 16;
    machCodeSlot = machCodeSlot | in1Reg;
    machCodeSlot = machCodeSlot | in2Reg;
    return machCodeSlot;
}

int addImmToReg(int dstReg, int in1Reg, int imm)
{
    int machCodeSlot = 0;
    machCodeSlot = machCodeSlot | 0xE2800000;  //add operation
    dstReg = dstReg << 12;
    machCodeSlot = machCodeSlot | dstReg;
    in1Reg = in1Reg << 16;
    machCodeSlot = machCodeSlot | in1Reg;
    machCodeSlot = machCodeSlot | imm;
    return machCodeSlot;
}
```

Figure 3.11: Functions for generating machine code operations by bitshifting

The function called determines which operation will be generated and what sources it will have. The parameters passed to the functions contain the register or immediate values which will be used in the instruction. Through bitshifting these values will be inserted into the 32 bit instruction at the correct bit locations. The functions return the 32 bit instruction as an integer which can then be stored in the array of instructions to be executed later.
3.7 Constructing the Instructions

Figure 3.12 shows the flow of the compiler as it generates native ARM code for the Vivio animation. The process begins with the case statement reading the next VCode instruction. Most of the time, this instruction will need machine code to be generated for it. The case statement calls the correct function to generate the corresponding machine code for the VCode instruction. The program then returns to the case statement.

In the case of an exit VCode instruction, the case statement will call the function which branches the program to the beginning of the array. The function will then execute the array of stored instructions sequentially. After all of the machine code has been successfully executed, the program will render the animation.

Figure 3.12: Flowchart showing progression of generating corresponding ARM machine code for VCode instructions
3.8 Calling Functions from Machine Code

One of the major tasks of implementing the compiler was successfully calling existing Vivio and C++ functions from the machine code.

At the machine code level, the process of a function call being made is that the address of the function in memory is put in the PC. Since the PC determines the next instruction in the program to be executed, the program will then jump to the function and being executing it. This is done using a branch instruction.

There are two main variants of branch instruction on the ARM. The branch to a location offset from the current PC, or branch to the address stored in a register. The branch to offset instruction is generally used for loops and conditional statements. For calling the functions I used the branch to address in register instruction. To put the address of a function in a register I used the inbuilt inline assembler in the GNU Compiler Collection (GCC). I wrote a function in which the address of the function was converted to an integer and then this integer was placed in r8. r8 was used as it is the ARM states fifth variable register, but is not used within C++ functions or as a parameter input register. The inline assembly line for loading the address into r8 can be seen in Figure 3.13.

```
asm("MOV r8, %[v]" : [v] "=r" (addr)); //put addr of func into r8
```

Figure 3.13: Inline assembly loading address of function into register

The conversion of the addresses to integers is not particularly common practice in programming as with high level programs any interactions with a function are handled already and the addresses are rarely needed as the 32 bit values which they are. As such, doing so caused the fpermissive flag to occur and throw up an error. To deal with this and be able to compile the project I added fpermissive to the flags to be set as warnings in the projects Qmake file. This downgraded the fpermissive flag from an error to a warning, meaning the Vivio app could then compile with the conversion to int occurring.
3.8. Calling Functions from Machine Code

With the address correctly loaded into r8, the machine code instruction BLX r8 must be executed. The generation of this branch instruction is quite straightforward compared to other instructions due to the lack of potential options or variables in the instruction. As can be seen Figure 3.14 above, most of the bits for the instruction do not change, only the condition bits and register being used need to be set. The simple function I wrote for generating these branch instructions can be seen in Figure 3.15 below.

```c
int branchToAddrInReg (int reg)
{
    int machCodeSlot = 0;
    machCodeSlot = machCodeSlot | 0xE12FFFF0; //branch operation
    machCodeSlot = machCodeSlot | reg;
    return machCodeSlot;
}
```

Figure 3.15: Function to generate machine code instruction for branch to address stored in register

When a function is called from higher level languages, the parameters used in calling the function are placed in registers or on the stack in accordance to the processors architecture. At machine code level, the instructions to place the parameters in the correct locations are separate instructions that must be generated.

To emulate the VCode interpretation, the parameters of a function had to be loaded into registers zero to three and the remainder pushed to the stack. The parameters must be in the correct order or the function will take them in incorrectly.

Figure 3.16 shows a simple function with two parameters. In order for the function to use the correct values for the parameters, they must be in r0 and r1 in the correct order when the machine code branches to the functions location in memory.
Since there are only two parameters in the testTwoVar function, the contents of r2 and r3 are irrelevant when the function is called.

For functions with more than four parameters, some parameters will be called from the stack. The example shown in Figure 3.17 shows a function with six parameters. For this function to be called using the correct parameters, variables a, b, c and d must be in their corresponding registers, and e and f on the stack. The first parameter on the stack will be taken from where the SP is pointing, and the remaining ones in order descending from there.
However the VCode only has one accumulator and then the stack, as opposed to the four variable registers the ARM has. Since r0 is being used to simulate the accumulator, in reality, all the parameter variables are loaded into r0 and then pushed to the stack, in reverse order, with the exception of the final variable loaded. This means that to place the second, third and fourth variable in their corresponding registers, they are actually popped off the stack into those registers.

This is not the optimal way for these parameters to be loaded, however it is the only one possibly while reading in a single VCode instruction at a time. Potential future optimisations of the compiler could include looking ahead in the VCode instructions for function calls requiring variables in r1, r2 and/or r3. This would mean these variables could be loaded into the registers when they are first loaded, and not first loaded to the accumulator register and pushed to the stack only to be popped off again.

This can be seen in action in Figure 3.18.

![Diagram showing state of registers and stack before and after calling testSixVar function using r0 as accumulator](image)

Another variation on calling functions is the case of member functions. Since an object can be created dynamically, I could not convert the addresses in memory of the member functions to ints in the same way as with non-member functions. Some of the objects within Vivio, such as the SolidBrush object, have static constructors which create and initialise the values
of an object. These static constructors could be reached by converting the address, storing it a register and branching to it. So, based on this approach, I created static constructors for Vivio objects which did not already have static constructors, such as the rectangle object seen in Figure 3.19.

In the ARM processor architecture, the address of the object which a function is a member of is passed to the function in r0. In the case of the creating new Vivio objects for the animations, this meant the address of the Vivio player had to be passed in r0. For these constructor functions to be able to use the address of the player, the static constructors also had to have the address as a parameter.

This meant that when branching to a static constructor function, the variable that was in the accumulator register (r0) needs to be moved to r1. And then, only two parameters will be popped off the stack, into registers 2 and 3, as opposed to 3. The address of the player is then loaded into the accumulator. Now when the PC branches to the static constructor, the parameters will all be passed correctly.

![Figure 3.19: State of registers and stack before and after calling rectangle2 static constructor](image-url)
Chapter 4

Conclusion

4.1 Conclusions

Mobile devices are becoming increasingly advanced and widespread. The potential of smartphones and tablets as a source of learning is vast and tangible. With online learning a growing educational resource, it makes sense for this expansion to include mobile devices as well.

People expect their mobile devices to keep with the latest cutting edge advances. Java has sparked a surge in the research of 'just-in-time' compilation systems. These techniques are enabling processors to execute programs faster and more efficiently. JIT systems are capable of maximising the capabilities of the more limited mobile processors, as long as they are tailored for them correctly.

The aim of this project was investigate the use JIT compilation systems with the Vivio interactive learning animations. The exploration done throughout the project shows the potential for JIT implementations with animations on mobile devices. An important aspect is balancing the time spent generating the code with the increased speed of execution, to create the optimal user experience.

I set out looking to generate machine code that could be executed on the ARM processors of Android devices. I am pleased to have implemented the plumbing functions to do this and to
have created the structure of a JIT compilation system. A subset of Vcode instructions is now available for JIT code generation for the ARM. This subset of instructions allows simple Vivio drawing mechanisms to work by executing the machine code generated at runtime.

At the end of this project, I feel that I have gained a greater understanding of how software functions at a machine code level. As JIT becomes more widely used and continues to evolve, I am glad to have seized this opportunity to learn how it works 'under the bonnet'.

I have learned a lot throughout this project, often the most when faced with obstacles. Encountering and overcoming the tasks related to making the generated machine code executable and calling functions from the machine code has lead to me exploring and learning in areas of memory allocation and processor architecture I had not yet been exposed to.

The successful implementation of these simple animations shows that utilising JIT techniques with online learning animations has untapped potential. This shows that there is scope to expand this JIT system to incorporate the more elaborate Vivio animations. The Vivio animations can be a tool for students which incorporates modern JIT approaches to keep pace with the ever advancing speed of technology.
4.2 Future Work

This section identifies areas where work on the project could be continued.

The first step would be to expand upon the subset of VCode instructions available for JIT native code generation. There are 520 Vcode instructions and each would require code to be written which can generate appropriate machine code if that instruction is encountered. However, with the structure of a Vivio JIT compiler for ARM in place, this could now be done quite mechanically. This is especially the case in many of the instructions which are very similar to existing ones, such as calling static constructors of other drawn objects.

The expansion of the Vcode instructions available to the point where benchmark programs such as the sieve or ackerman functions would be beneficial for evaluating more discretely how the JIT code generation improves the performance of the Vivio animations.

The current implementation performs as much native code generation as possible. As discussed in earlier sections, this is not always ideal, especially for animations and mobile devices. It would be beneficial to employ JIT tracing or a method based JIT system which can assess the need for re-optimisation and code generation, and best balance the code generation time with the faster execution of the machine code.

The possible improvements outlined in this section could lead to the Vivio JIT compilation system reaching its potential as a high-speed and modern optimisation tool on mobile devices.
Appendix A

A.1 Source Code

Please find on the CD attached to this project:

• source code of Vivio App with JIT Code Generation

• source code of Test App used while developing and experimenting
Bibliography


[38] TeachThought. 50 top sources of free elearning courses.


